

# 263-3855-00: Cloud Computing Architecture

## Section 10: Systems for AI

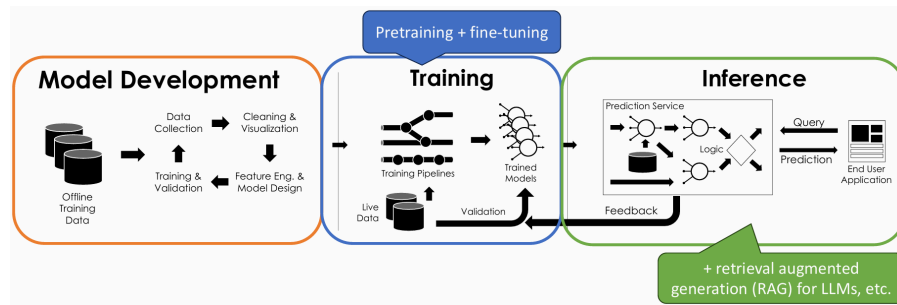
Swiss Federal Institute of Technology Zurich (ETH Zürich)

Last Edit Date: 05/31/2025

### **Disclaimer and Term of Use:**

1. We do not guarantee the accuracy and completeness of the summary content. Some of the course material may not be included, and some of the content in the summary may not be correct. You should use this file properly and legally. We are not responsible for any results from using this file.
2. This personal note is adapted (derived) from the slides and codes from *Professor Ana Klimovic and people from ETH Zurich Systems Group*. Please [contact us](#) to delete this file if you think your rights have been violated.
3. This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

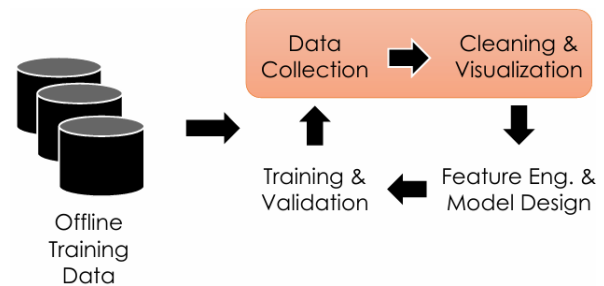
# I. End-to-end machine learning



Only a fraction of real-world ML systems consists of “ML code.” The required surrounding infrastructure is vast and complex.

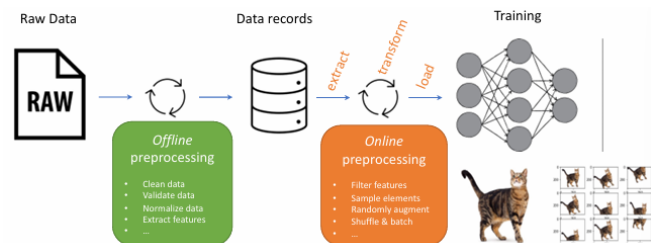
## 1. Model development - data part

- Identify potential sources of data
- Join data from multiple sources
- Clean missing values and outliers
- Plot trends to identify anomalies



### Data collection and cleaning

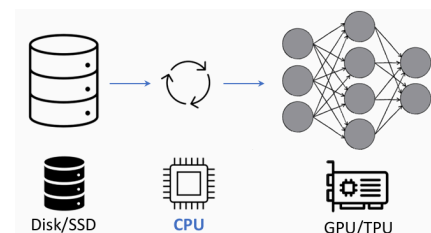
- Goal: gather training data and maximize data quality
- Common challenges may include:
  - Collecting sufficient data
  - Labeling sufficient data
  - Detecting / filtering outliers
  - Too much data; need to pick the most relevant data to train on



- Machine learning models do not train on raw data
- Some preprocessing operations are done offline, others on-the-fly during training

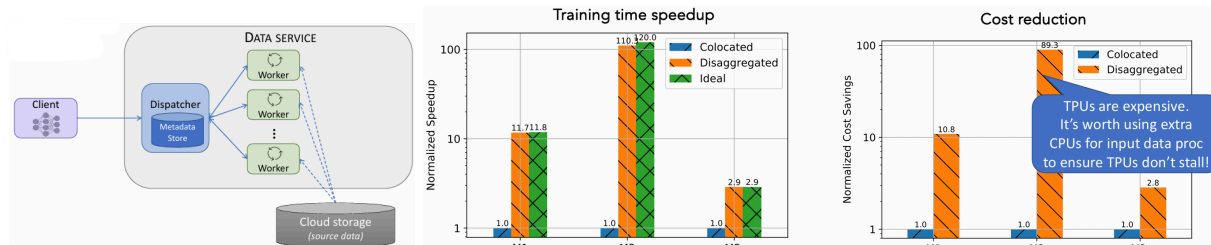
### Why care about ML data preprocessing?

- Impacts end-to-end training time and cost
- Consumes significant CPU resources and power. Sometimes data preprocessing can even consume more power than model training itself.



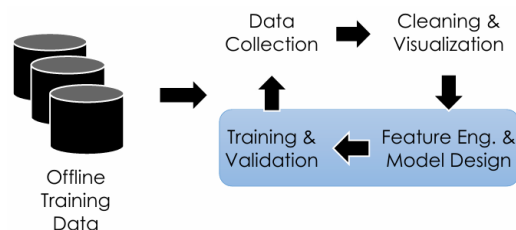
### Disaggregated ML data processing

- Idea: decouple training client (with GPU / TPUs) from data processing workers (on CPU machines). Scale training clients and data workers independently, adjust scale for each job
- This approach can remove input bottlenecks up to 110x speedup, 89x cost reduction



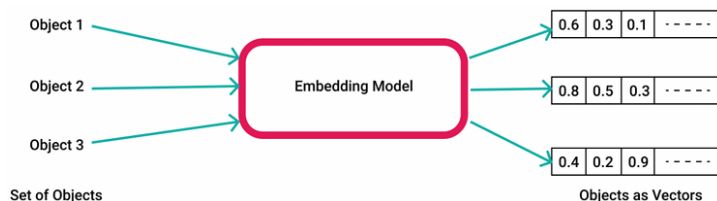
## 2. Model development - model part

- Build informative feature functions
- Design new model architectures
- Tune hyperparameters
- Validate prediction accuracy



### Feature extraction

- Derive values (“features”) from data to feed to the model during training
- Often involves reducing the dimensionality of data
- “Embed” high-dimensional data into lower dimensional representation, can even use neural networks to learn embeddings

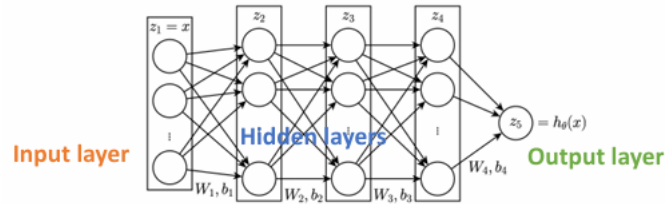


### Model architecture and parameter search

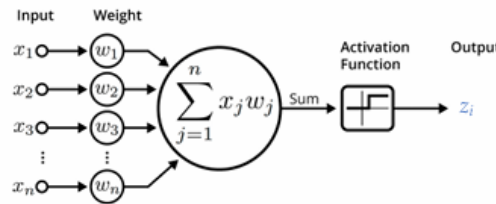
- Model search:
  - Explore different model architectures to find a suitable design. For example, recurrent neural networks, transformers, etc.
  - Neural Architecture Search (NAS) automates model search, often using ML approaches like reinforcement learning
- Hyperparameter tuning:
  - Adjust hyperparameters to maximize model training accuracy. For example, batch size, learning rate, number of layers, activation function, etc.
  - Large search space, can be parallelized, but still expensive

### Deep Neural Network Training

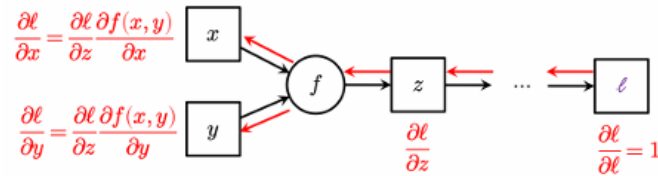
- Deep neural networks are neural networks with multiple hidden layers
- To train a DNN model, iteratively adjust model weights (e.g., using stochastic gradient descent) to minimize a loss function



- Repeat as iterate through the training dataset
  1. **Forward pass:** compute activations  $z_i$  and loss
    - a. At each layer, apply the activation function on the weighted sum of inputs at that layer
    - b. At final layer, compute the loss



2. **Backward pass:** compute gradients



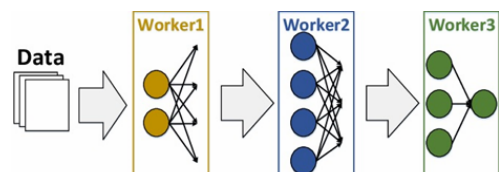
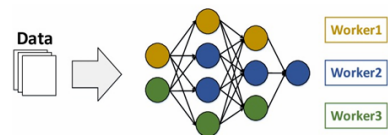
3. **Update model weights** to minimize loss

### Characteristics of DNN training

- **Computationally intensive:** lots of matrix multiplications and convolutions - we can accelerate this in hardware
- **Large model state and datasets:** does not fit on single machine, need distributed computation and efficient communication frameworks
- **Error tolerance:** lead to interesting trade-offs on accuracy, speed, and cost, can trade-off some accuracy for large benefits in cost and / or throughput, e.g., pruning, quantization, asynchrony

### Distributed training

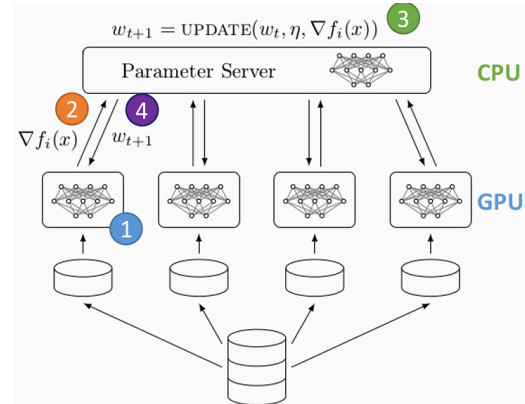
- Data parallelism:
  - Partition the data and run multiple copies of the model, which synchronize to exchange model weight updates
- Model parallelism:
  - Partition the model across multiple nodes



- Tensor parallelism: divide per-layer state across nodes
- Pipeline parallelism: divide model layers across nodes

### Data parallelism: parameter server

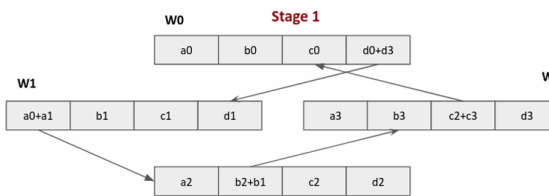
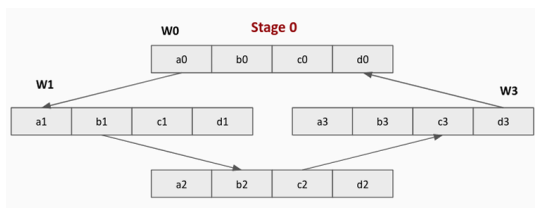
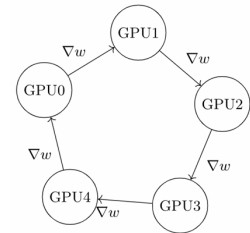
- Each worker has a copy of the full model and trains on a shard of the dataset
  - Step 1: workers compute forward / backward
  - Step 2: workers send gradients to parameter server
  - Step 3: Parameter server aggregates gradients and computes new model weights
  - Step 4: Parameter server sends new model weights to workers



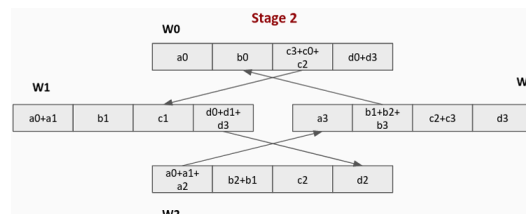
- Repeat the above steps
- Data parallelism enables training models faster by leveraging multiple workers in parallel

### Data parallelism: AllReduce

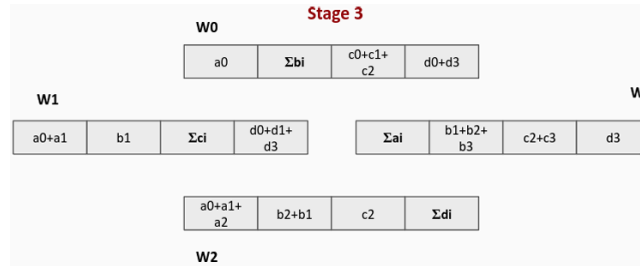
- Alternative way to implement data parallelism, with no parameter server
- Idea: GPU workers exchange updates directly via All Reduce collective and gradient aggregation is done across workers vs. centralized in parameter server



- Step 2: reduce-scatter



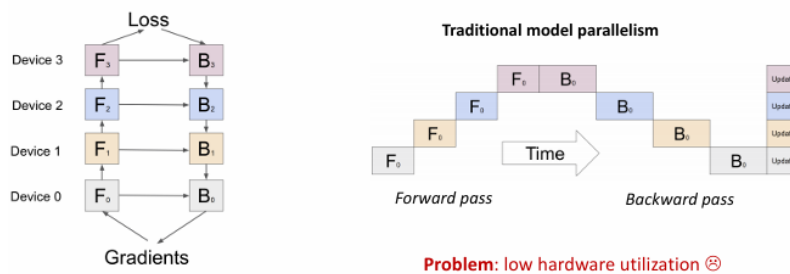
- Step 3: all-gather



- Repeat the above steps
- Due to fast GPU-GPU interconnection, AllReduce is commonly used today

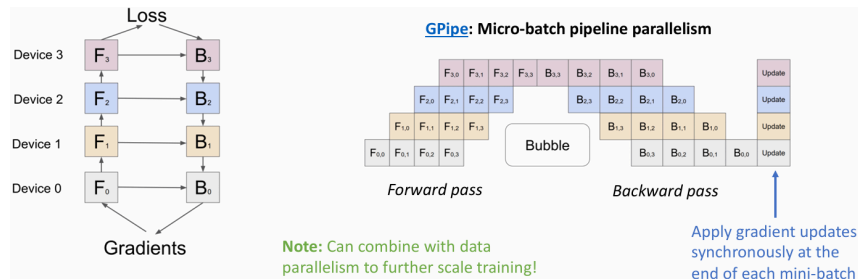
## Model parallelism

What if the model cannot fit on one device? Need to partition the model



## Pipeline parallelism

To improve hardware utilization, pipeline micro-batches

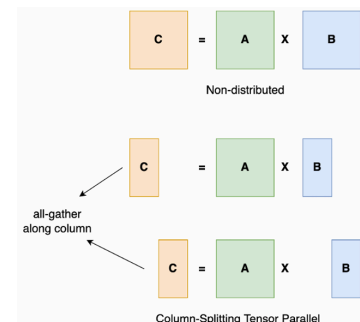


## Tensor parallelism

- Split tensors (weights, gradients, activations) within a model layer across nodes
- Useful for very large models
- Even if a model fits on a single GPU, tensor parallelism can still be useful, can improve inference latency for burst of requests

## Distributed training: sync or async?

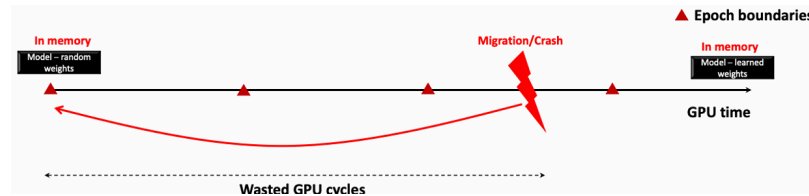
- **Synchronous training:** workers operate in lockstep to update the model weights at each iteration
- **Asynchronous training:** each worker independently updates model weights



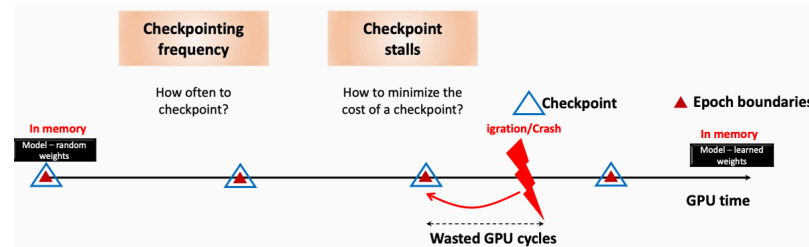
- Synchronous training achieves better model quality, but lower throughput
  - Accuracy vs. throughput trade-off
  - Useful metric: time to train to a target accuracy

## Fault tolerance

- The more machines involved, the more likely a failure is to occur
- Any interruption can wipe out the model parameters learned so far in memory, restarting the expensive training process



- Solution: checkpoint and persist model parameters periodically during training



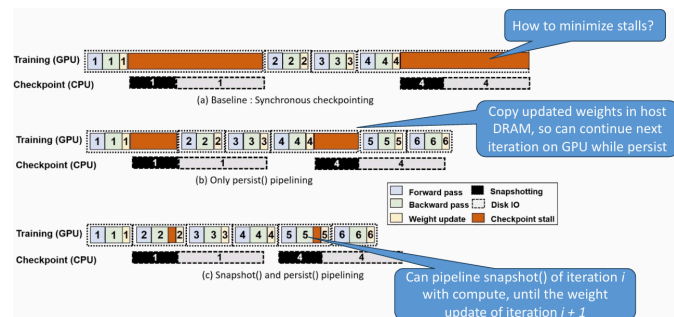
## Checkpointing challenges

How often to checkpoint?

- Depends on
  - Model size
  - Storage bandwidth
  - Training iteration time

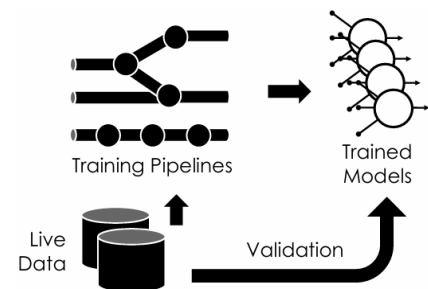
How to minimize the cost of a checkpoint?

- Pipeline 2 phases to minimize GPU calls
  - Snapshot from GPU -> hosted DRAM
  - Persists from DRAM -> storage



## 3. Training

- Train models at scale on live data
- Retain on new data
- Automatically validate prediction accuracy
- Manage model versioning
- Requires minimal expertise in machine learning

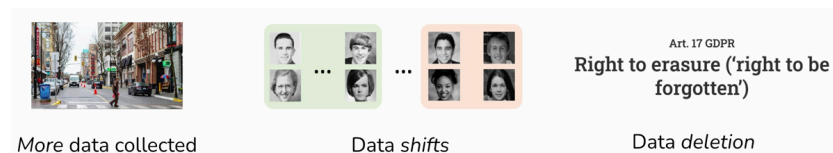


## Pretraining, fine tuning & alignment

- Models are large and expensive to train. It is wasteful to train models from scratch each time.
- Large-scale ML training is commonly divided into 2 or 3 phases:
  - Pretraining: train large model on extremely large dataset, only done by large organizations with large computing infrastructure and datasets
  - Fine tuning: continue training model on smaller dataset specialized to the “downstream” task for which the model will be used
  - Alignment: continue training on a preference dataset, typically with the goal of making the model more helpful, safe, and reliable

## Continual training on dynamic datasets

- Real training data is often dynamic:



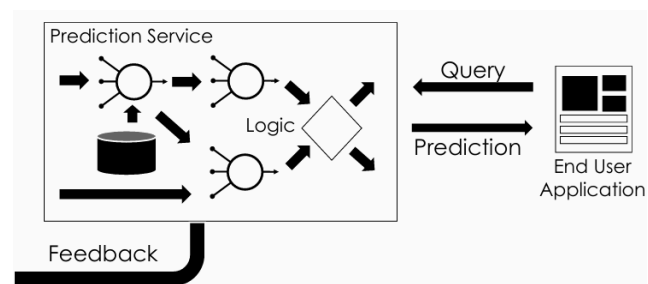
- Model need to adapt to dynamic data, need to re-train
- Cost of ML training is proportional to
  - How often re-train model
  - How much data the model is trained on each time

## Modyn: platform for training on dynamic data

- Users plug policies for
  - Triggering training
  - Data selection
- Modyn orchestrates training jobs based on triggering policy and manages metadata to implement data selection policies
- Optimizes for fast data ingestion while implementing data selection policies

## 4. Inference

- Serve models with real data, in real-time
- Embed model serving inside the end-user application
- Optimize for low latency predictions
- May quantize model weights to improve memory efficiency



## DNN inference vs. training

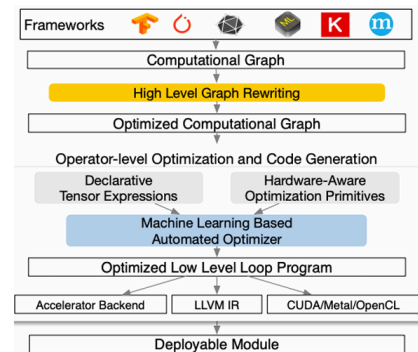
- **Training:** Iterate through dataset, compute forward & backward pass on each batch of data to update model parameters. Care about high throughput and fast convergence to high accuracy
- **Inference:** Compute forward pass on a (small) batch of data. Care about low latency, high throughput, and high accuracy

### Where does inference happen?

- In the cloud
  - ChatGPT model is served in OpenAI clusters in Azure Cloud
  - Good for serving large models at high throughput
  - Challenge: latency
- On the edge
  - Apple Siri speech recognition inference happens on the phone
  - Good for low-latency and preserving privacy
  - Challenges:
    - Energy consumption
    - Many heterogeneous devices, how to make one model run efficiently on all devices?

### ML compilers

- Goal: efficiently serve a trained model on different kinds of devices
- ML compiler applies multiple rounds of optimizations on computation graph, e.g., vectorization, loop tiling, op fusion
- Many ML compilers use ML to do some of the optimizations, e.g., TVM



### ML inference in the cloud

- How to serve requests to many models efficiently in a sharded cluster?
  - Examples of inference systems: Clipper, Clockwork
- Common techniques
  - Batch requests
  - Cache results of common queries
  - Compress / quantize models (e.g., GPTQ)
  - Select model to serve based on latency target and accuracy requirement
    - “Model-less” inference, user does not decide which model to query, the platform picks

### Cluster management for ML jobs

Challenges:

- Jobs require heterogeneous resources (GPUs, CPUs, interconnects). Each job has a unique mix of resource requirements
- Jobs consume many expensive resources (GPUs). Therefore, ensuring jobs make good use of the hardware is critical for cost-efficiency

### How to allocate GPU resources fairly to AI jobs?

- AI jobs have unique characteristics
  - Consist of long-lived tasks that need to be gang-scheduled (job does not proceed until all tasks receive resource allocations)
  - AI training involves frequent synchronization, thus sensitive to GPU placement (e.g., GPU in same machine >> GPU in other rack)
- Idea:
  - Themis scheduler introduces new metric “finish time fairness”: ratio of execution time in shared cluster with N apps vs. execution time alone in cluster of size 1 / N
  - Themis’s goal: minimize the max finish time fairness across all ML apps while efficiently utilizing GPUs in the cluster

## II. Optimizations for ML / DNN

- We can define a deep neural network as a function of input features and model parameters (weights).
- At inference (serving) time, we output for a given input. At training time, we iterate over a set of inputs and compute updates to model parameters.
- At training time, we iterate over a set of inputs and compute updates to model parameters
- The training objective is to minimize the loss by optimizing the model parameters (weights)
- The loss consists of a loss function (e.g., classification loss, regression loss) evaluated over a training dataset for a set of parameters

*Neural network as a function:*

$$f(x, w): \mathbb{R}^I \times \mathbb{R}^P \rightarrow \mathbb{R}^O$$

$$f(x, w) = \hat{y}$$

- $\hat{y}$ : “Prediction”
- $\mathbb{R}^I$ : “Feature/Input space”
- $\mathbb{R}^P$ : “Parameter/Weight space”
- $\mathbb{R}^O$ : “Output/Prediction space”

*Loss function:*

$$l(\hat{y}, y): \mathbb{R}^O \times \mathbb{R}^O \rightarrow [0, 1]$$

$$l(\hat{y}, y) = l(f(x, w), y)$$

*Example loss functions:*

- 1-0 loss:  $l(\hat{y}, y) = \begin{cases} 0, & \text{if } \hat{y} = y \\ 1, & \text{otherwise} \end{cases}$
- MSE:  $l(\hat{y}, y) = \|\hat{y} - y\|_2^2$

*Loss:*

$$L(w): \mathbb{R}^P \rightarrow [0, 1]$$

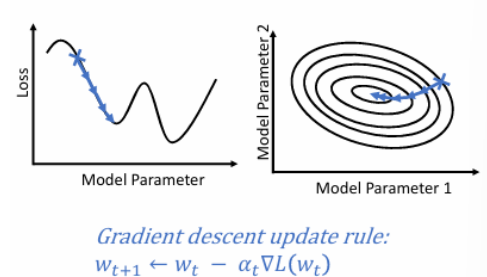
$$L(w) = \frac{1}{n} \sum_{i=1}^n l(f(x_i, w), y_i)$$

- n: number of sample

### 1. Gradient descent

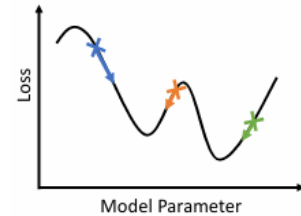
- The loss is a function returning a single number for a fixed set of parameters
- The gradient of the loss measures its sensitivity

- The negative gradient represents the steepest downhill direction
- Gradient descent (GD) follows the negative gradient for multiple steps (scaled by a positive learning rate)
- Guaranteed to reach a local minimum
- Computing the gradient for a complex neural network and all training samples is demanding



### Where to start?

- Different strategies: e.g., random or zero initialization
- Fine-tuning: Initialize (parts) of the weights based on a pretrained network. The train from there (all or subset of the weights)



### GD optimization on a single GPU

- The gradient of a neural network can be computed efficiently using back propagation (chain rule)
- We can approximate the gradient using a subset of the full dataset (randomly stochastically sampled)

Remember:

$$L(w) = \frac{1}{n} \sum_{i=1}^n l(f(x_i, w), y_i)$$

Then

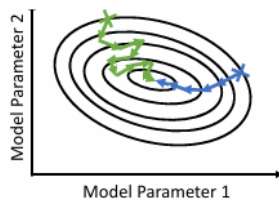
$$\nabla L(w) = \frac{1}{n} \sum_{i=1}^n \nabla l(f(x_i, w), y_i)$$

Chain rule:

- if  $f(w) = g(h(w))$  then  $f'(w) = g'(h(w))h'(w)$

### Stochastic gradient descent

- Stochastic gradient descent (SGD) uses a single sample
- Converges to the same solution in expectation but slower (i.e., needs more iterations)
- Faster computation of gradient for single sample

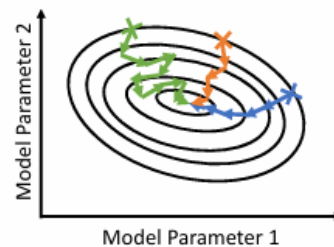


SGD update rule:

$$w_{t+1} \leftarrow w_t - \alpha_t \nabla l(f(x_i, w_t), y_i) \text{ for } i \sim [1, n]$$

Unbiased estimator of full gradient:

$$\mathbb{E}_i[\nabla l(f(x_i, w), y_i)] = \frac{1}{n} \sum_{i=1}^n \nabla l(f(x_i, w), y_i)$$



(Mini-)Batch SGD update rule:

$$w_{t+1} \leftarrow w_t - \alpha_t \frac{1}{m} \sum_{i=1}^m \nabla l(f(x_i, w), y_i),$$

with  $i \sim [1, n]$  for  $1 \leq i \leq m$

### Mini-batch SGD tradeoffs

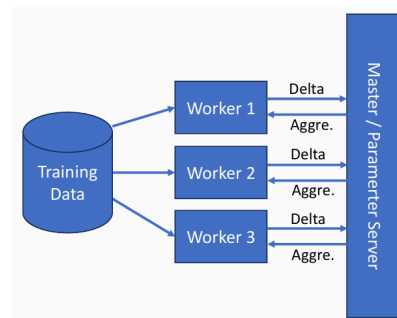
- Speedup convergence by taking multiple samples: Mini-batch SGD
- How big should the mini-batch (batch size) be?
- Tradeoff convergence speed vs. computation speed
- In practice, choose biggest batch-size fitting the GPU memory

## Approximate model memory calculator

- **Inference:** model parameters + inference batch size (typically small) + overhead (e.g., 20%)
  - $1.2 * (2 * P + 2 * I * B)$  bytes, with 2 bytes per number and B is inference batch size
- **Training:** model parameters + gradient memory + activations (batch size dependent) + (optimizer state, e.g., for momentum SGD) + overhead (e.g., 20%)
  - $1.2 * (2 * P + 2 * P + 2 * (I + L * D) * B)$  bytes, with 2 bytes per number and L is number of hidden layers, D is dimension of hidden layer outputs, B is training batch size

## Data-parallel distributed training

- If your GPU does not have enough memory for entire mini-batch (but model and batch size 1 fits): scale out to multiple GPUs
- Data-parallel SGD computes the gradient (or model updates) over multiple nodes
- Access entire dataset randomly to keep convergence
- (Strong / weak) scaling of batch vs. cluster size to balance convergence vs. speed



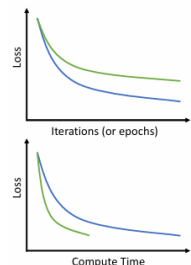
## On model vs. gradient averaging

For standard SGD (e.g., not momentum) model averaging is equal to gradient averaging. Can be relevant for distributed or privacy preserving communication (i.e., Federated Learning)

$$\begin{aligned} w_{t+1} &\leftarrow w_t - \alpha_t \frac{1}{m} \sum_{i=1}^m \nabla l(f(x_i, w), y_i) \\ &= \frac{1}{m} \left( m w_t - \sum_{i=1}^m \alpha_t \nabla l(f(x_i, w), y_i) \right) \\ &= \frac{1}{m} \sum_{i=1}^m (w_t - \alpha_t \nabla l(f(x_i, w), y_i)) \end{aligned}$$

## 2. Relaxation

- Can change convergence behavior (correctness), typically need more iterations to converge
- Single iteration can be faster to compute (higher throughput or lower latency)
- Tradeoff: convergence slowdown vs. computation speedup
- Relevant metric - “time to accuracy”



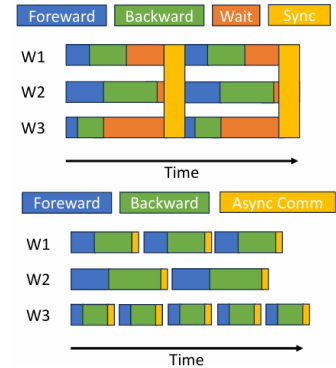
### Relaxation 1: data access

- Problem: Remote (or random) data access (i.e., data movement) is expensive

- Relax constraint by distributing data prior training and locally sample on each node
- Changes convergence behavior
- Not an issue in practice if large number of data samples vs. number of nodes ratio (and non-grouped samples)

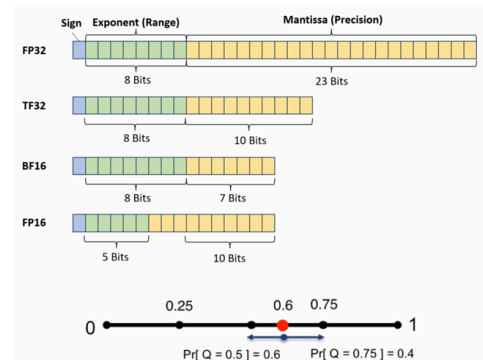
### Relaxation 2: asynchronous communication

- Problem: Synchronization (barriers) can be time consuming
- Relax constraint by allowing for asynchronous updates
- Again: change convergence behavior
- Not an issue in practice if not fully asynchronous, but bounded stateless (at most fix delay)
- This is used in PipeDream



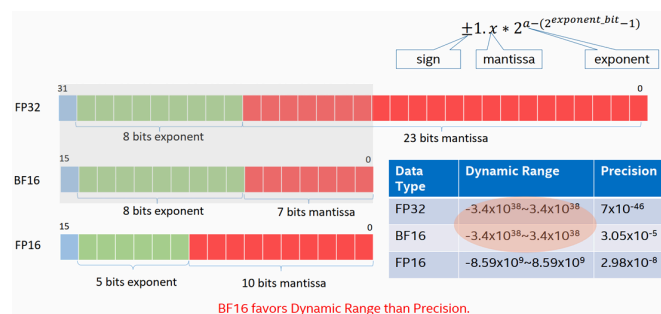
### Relaxation 3: low precision training

- Problem: Communicating the full model or gradient can be costly
- Relax constraint by communicating updates in low (or mixed) precision, or compressing it (e.g., using quantization)
- This changes the convergence behavior
- Quantization for communication, especially on slow interconnects has become standard in practice (gradient remains unbiased)



### Low precision representation and serving

- Problem: Storing model, gradient and activations is costly
- Solution: Represent parts or everything in low or mixed precision
- This changes the performance of your model, not only convergence
- Representing models using mixed precision (e.g., bfloat16) has become standard in practice during training
- Post-training, models and inputs are then sometimes further compressed, e.g., using quantization



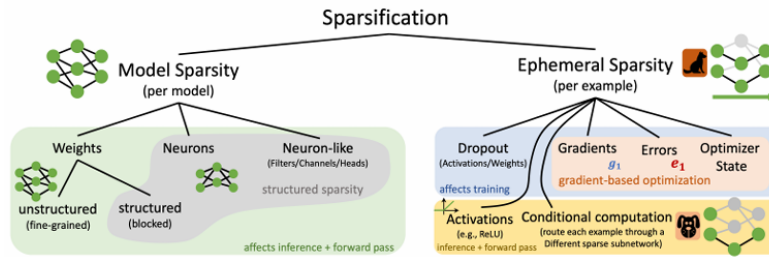
### Updated model memory calculator

- **Inference:** model parameters + inference batch size (typically small) + overhead (e.g., 20%)

- $1.2 * (Q_p * P + Q_I * I * B)$  bytes, with  $Q_p$  and  $Q_I$  is the number of bits to represent the model and input
- **Training:** model parameters + gradient memory + activations (batch size dependent) + (optimizer state, e.g., for momentum SGD) + overhead (e.g., 20%)
  - $1.2 * (Q_p * P + Q_G * P + Q_I * (I + L * D) * B)$  bytes, with  $Q_G$  is the number of bits to represent in the gradient

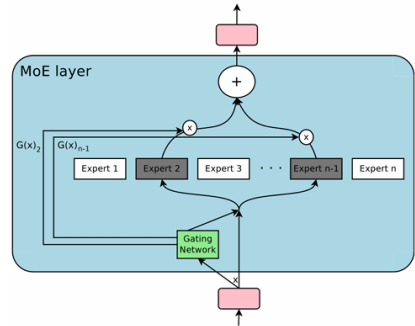
### 3. Sparsity

Sparsity in DL can speed up computation, slow down convergence but also affects generalization



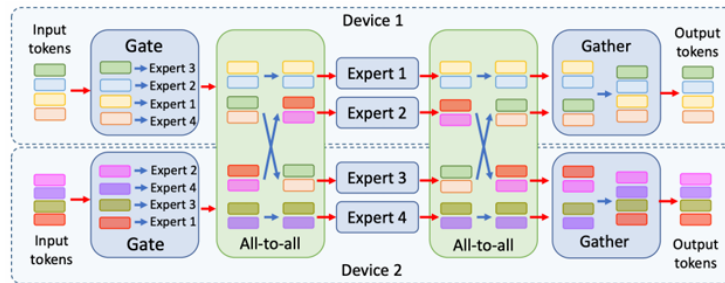
#### Mixture-of-Expert (MoE)

- Traditionally DNNs are dense, i.e., each input is processed by the entire model
- Mixture-of-Expert (MoE) introduce sparsity during inference
- Experts can be placed on different devices (requires some load-balancing)



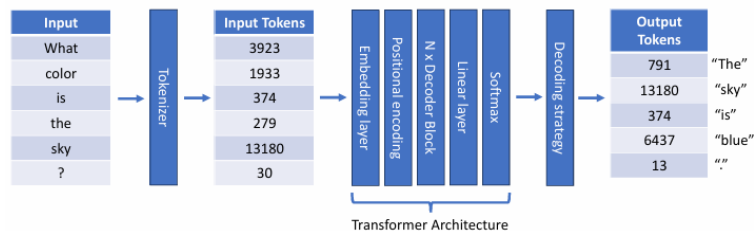
#### Challenges training MoEs

- Expert balancing: Non-uniform assignments can increase latency on “busy” experts. Also, accelerators require static tensor shapes
- Expert placement on multiple devices introduces a lot of communication overhead.



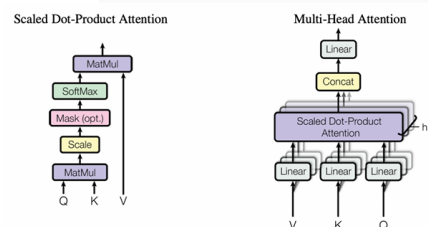
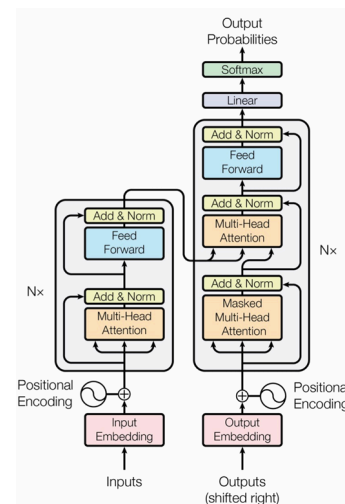
## III. Optimization for LLMs

### 1. Large language models / transformers



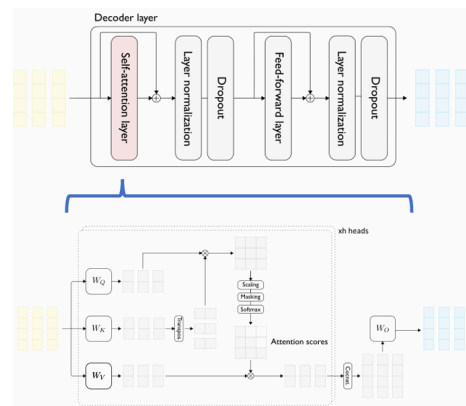
## What is attention?

- The core operations is transformers
- Inspired by the idea of tokens “paying attention” to other tokens
- Advantages over convolutional (CNN) or recurrent NN (RNN):
  - More flexible over time (vs. RNN) think vanishing gradients
  - More flexible over space (vs. CNN) only local influence
  - Easily parallelizable (good for GPUs)
- $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$



## Matrix visualizations

- We can represent each input token (embedding) as vector (yellow)
- The output matrix correspond to the blue vectors (same dimensions)
- Each head as a linear transformation corresponding to matrices ( $W_Q$ ,  $W_V$ , and  $W_K$ )
- Masking (lower diagonal) to prevent forward attention



## Number of parameters for GPT-2

- Some constants first (for GPT2)
  - Vocabulary size  $V = 50257$
  - Embedding size  $E = 768$  (also often marked as  $d_{model}$ )
  - Max sequence length  $P = 1024$
  - Number of decoder layers  $n = 12$
  - Number of heads  $h = 12$
  - Dimension of attention head  $d_{head} = E/h$
- Embedding layer:  $V \times E$

- Position encoding:  $P \times E$
- Multi-headed attentions:
  - Per head:  $3 \times d_{model} \times d_{head}$
  - For  $E = d_{model}$  and  $d_{head} = E / h$  yields total  $3E^2$
- Bias vector  $3E$
- Weighted sum of head with bias  $E^2 + E$
- Total  $n(3E^2 + 3E + E^2 + E) = n(4E^2 + 4E)$
- Feed forward layers
  - Projection into a hidden space of dim  $H$  (with typically  $H = 4E$ ):  $E \times H + H$  with bias
  - Projection back into embedding space:  $H \times E + E$
  - Total:  $n(2 \times H \times E + H + E) = n(8E^2 + 5E)$
- Layer norms:
  - Two per decoder block with  $2E$  parameters each
  - Total:  $n(4E)$
- Putting all together (with one final layer norm for GPT-2)
 
$$V \times E + P \times E + n(4E^2 + 4E + 8E^2 + 5E + 4E) + 2E$$

$$= E(VP) + n(12E^2 + 13E) + 2E$$

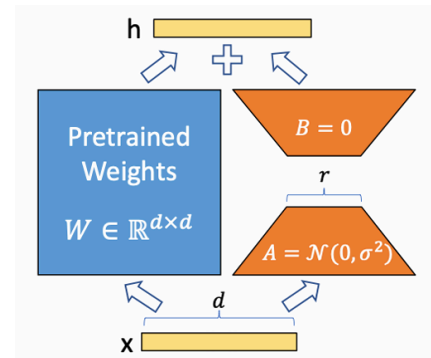
$$= 768(50257 + 1024) + 12(12 \times 768^2 + 13 \times 768) + 2 \times 768 = 124439808$$

## 2. Efficient fine-tuning

- Fine-tuning all these parameters can be very expensive
- Different downstream tasks typically require their own fine-tuning
- Using a distinct set of weights for every task can be costly
  - In terms of storage
  - When switching between tasks
- One solution: Parameter efficient fine-tuning (PEFT)

### LoRA: Low-Rank Adaptation of LLMs

- Only fine-tune LoRA Adaptors
- Typically only for attention blocks ( $W_Q$  and  $W_V$ )
- No additional inference costs if adaptors are merged post training  $h = Wx + BAx = (W + BA)x = W'x$



## Quadratic costs in training

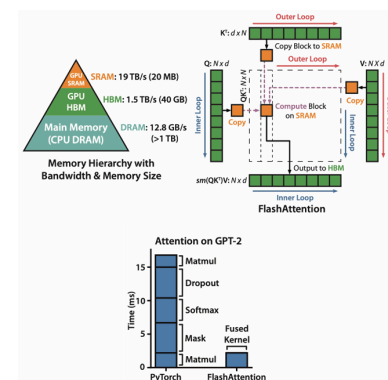
The attention matrix scales quadratically with respect to context window / sequence length

## Longer and longer context

- GPT2 had a context window of 1024 tokens
- This translated into roughly 340 words (assuming 3 tokens per word)
- Modern LLMs use larger vocabularies -> Less token per word in average
- They also support much larger context windows, which is important to put entire chat history or long documents as context
  - GPT3.5 (i.e., ChatGPT) supported 4096 tokens
  - GPT4 started with 8192 tokens but since then increased to 128000 tokens
  - Gemini 1.5 Pro support 2 million token

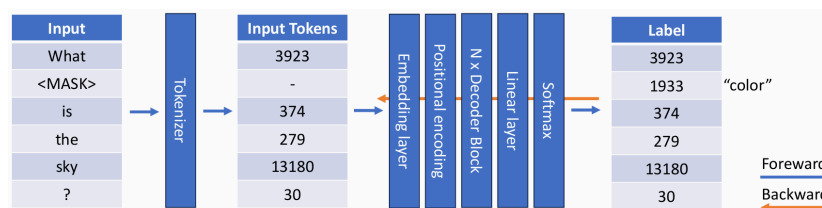
## FlashAttention

- Many approaches to support longer context approximate the computation of the exact attention matrix
- FlashAttention achieves fast exact attention computation using purely systems optimizations
  - Careful caching and memory access (on GPU)
  - Fusing CUDA kernels into a single operation

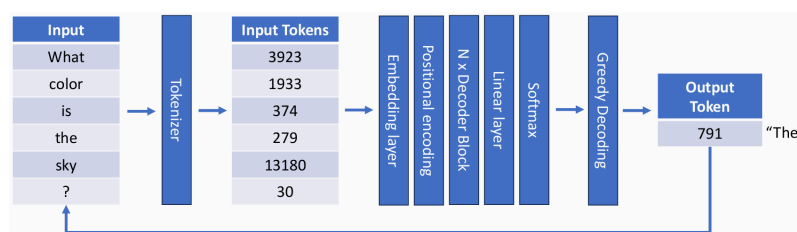


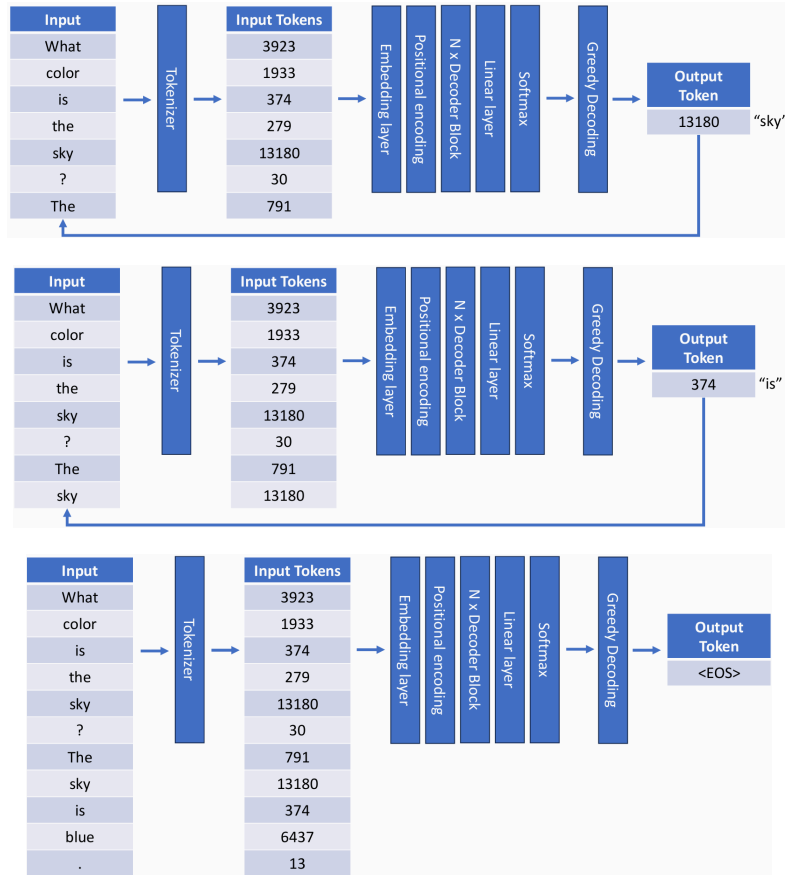
## 3. Optimization in serving

- These models are typically trained using masked word prediction
- Afterwards they are used in an autoregressive manner



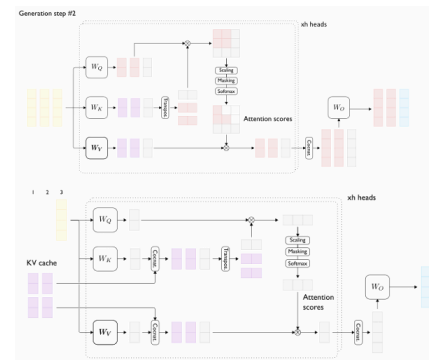
## Autoregressive inference





## Key-value (KV) caching

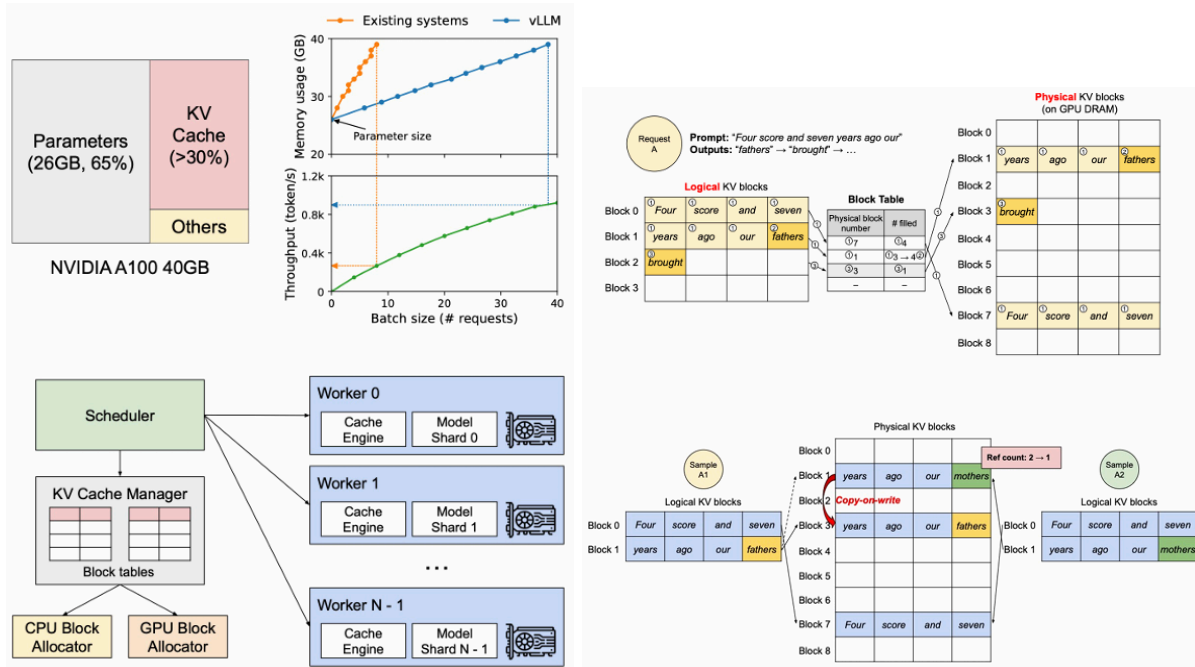
- We don't need the previous token (input or output) to compute new new token representation
- We use a KV cache to store the key and values for each head and layer
- Note that a KV cache is not a KV store, but can be stored linearly
- But it must be stored in memory and can grow quickly



## vLLM and PagedAttention

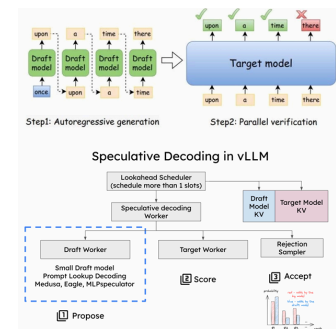
- The KV cache can grow quickly for large context models especially when
  - Serving multiple requests in parallel
  - Deploying more complex decoding strategies
- vLLM is a distributed serving engine using an abstraction over the KV store to make sure that
  - Blocks are efficiently fetched from disk and pushed to GPU or CPU memory
  - Subsequent blocks are stored in the same page on disk (for efficient read / write)

- PagedAttention achieved this by abstracting the KV cache into logical and physical blocks (like virtual memory access in OS)
- By carefully managing the state of physical KV blocks, it can
  - Explore parallel decoding strategies
  - Try complex decoding strategies like beam search
  - Serve concurrent requests in parallel



## Speculative decoding

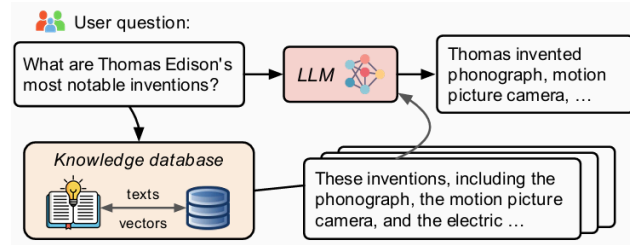
- Even with these optimizations, running inference on a very large LLM (to achieve high accuracy) can be too costly
- A smaller model might be much faster but typically achieves fewer good results
- Speculative decoding uses a target model to verify proposed tokens by small model (much faster)
- Also supported in vLLM (with careful scheduling)



## IV. Retrieval-augmented generation (RAD) & Vector search

### 1. Retrieval-augmented generation (RAG) overview

- Vector database and retrieval play a key role in the pipeline
- Various RAG algorithms of drastically different workload
- Multiple system components on heterogeneous hardware

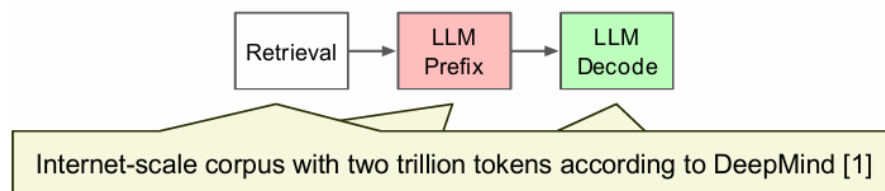


## 2. Various RAG algorithms and pipelines

### Pipeline 1: RAG with internet-scale retrieval

Argument: **Smaller model + hyper-scale retrieval = Larger model**

10x model size saving given similar generation quality [1,2]



### Pipeline 2: RAG for long-context processing

Answering questions of **user-defined long context** in real-time



Divide the document into many passages

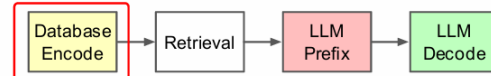
**Encode each passage** into a vector using a BERT-style model

**Naive solution:** include documents in the prompt (e.g., 1M tokens)

Possible but **very costly**, e.g., 60 USD / million token for GPT4

**RAG solution:** retrieve relevant passages

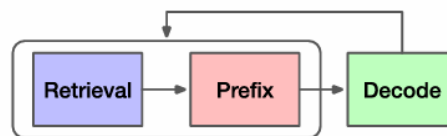
Significant **lower cost with comparable quality** [1,2]



**Small model** (e.g., 100M~1B) + **small databases** (1K~1M vectors)

### Pipeline 3: RAG with iterative retrievals

Multi-hop reasoning style, initiating retrievals during generation [1,2]



**User Query:**

“In what year did the company that invented CUDA go public?”

**Step 1:** “Which company invented CUDA?”

Retrieved answer: “NVIDIA developed and introduced CUDA in 2006.”

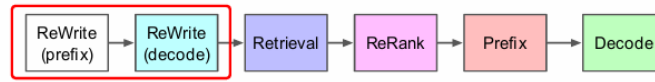
**Step 2:** “When did NVIDIA go public?”

Retrieved answer: “NVIDIA went public in January 1999.”

### Pipeline 4: RAG with rewriter and reranker

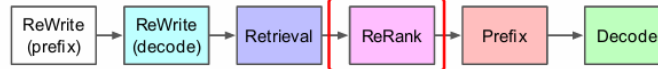
**Query rewriter:** a generative model to rephrase user questions [1]

**Retrieval reranker:** an encoder model to score retrieved documents [2]



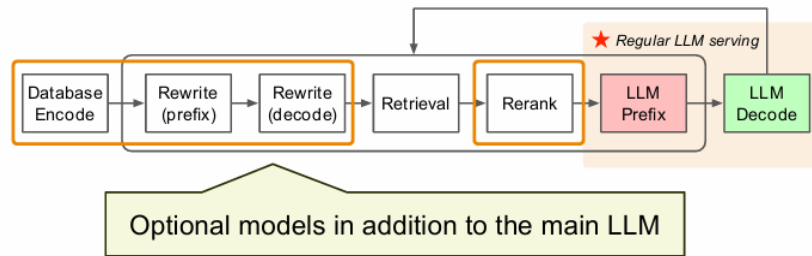
**Original query (conversational):** "Tell me about when NVIDIA started selling shares."

**Rewritten query for retrieval:** "When did NVIDIA have its IPO?"



**A general RAG system pipeline**

**Model components + Retrieval configurations**



### 3. Vector search and retrieval

- Given
  - A query vector  $q \in R^d$
  - A collection of database vectors  $D = \{x_1, x_2, \dots, x_N\} \subset R^d$
  - A similarity (or distance) function  $\text{sim}: R^d \times R^d \rightarrow R$
- Vector search aims to find the top-k nearest neighbors of  $q$  in  $D$ , defined as  $\arg_{x \in D} \text{top-k sim}(q, x)$
- Common similarity / distance functions

**Cosine similarity:**  $\text{sim}(q, x) = \frac{q \cdot x}{\|q\| \|x\|}$

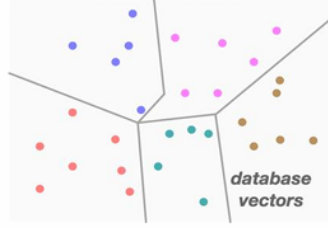
**Inner product:**  $\text{sim}(q, x) = q^\top x$

**Euclidean distance:**  $\text{dist}(q, x) = \|q - x\|_2$

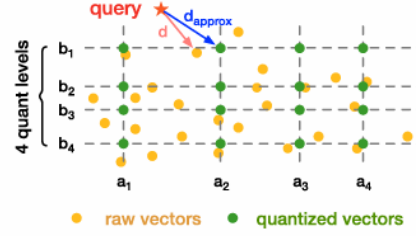
**Vector search is approximately nearest neighbor search**

**IVF-PQ:** a popular vector search algorithm in RAG

**Inverted-file (IVF) index:**  
prune the search space

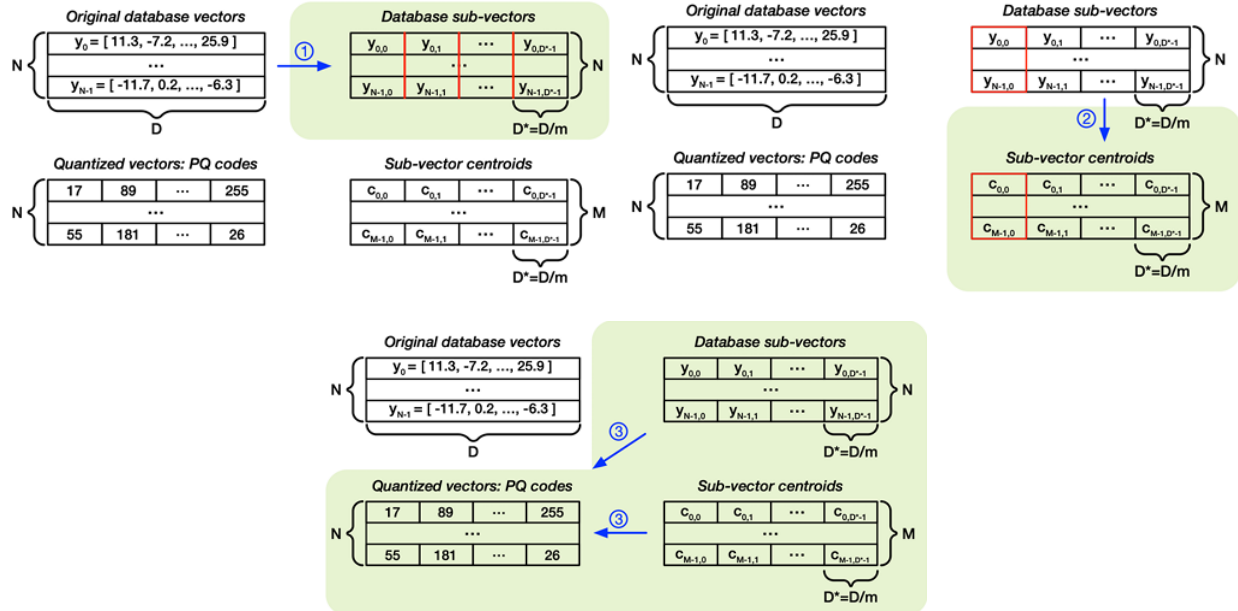


**Product quantization (PQ):**  
lossy compression of vectors

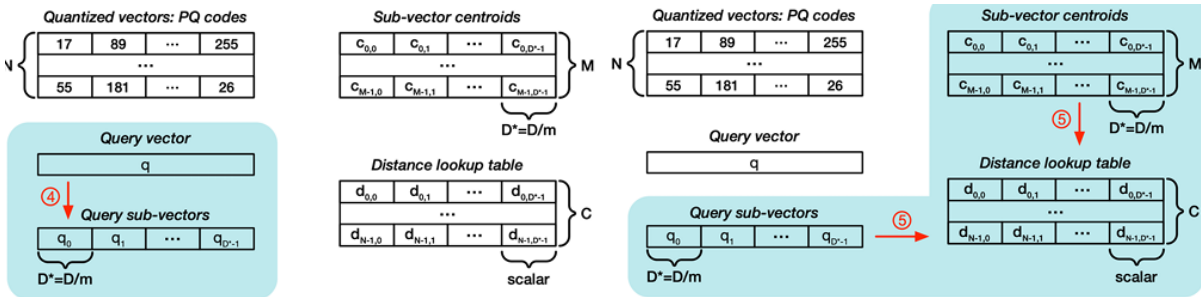


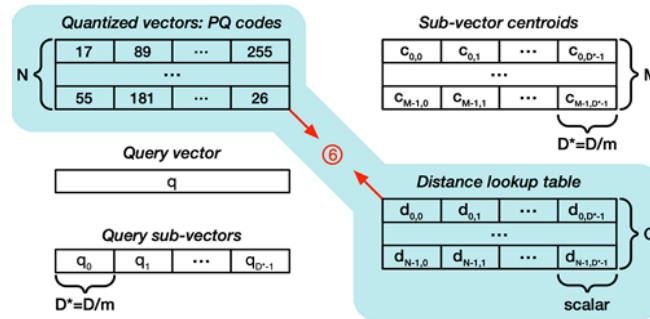
## Product quantization (PQ)

### Training



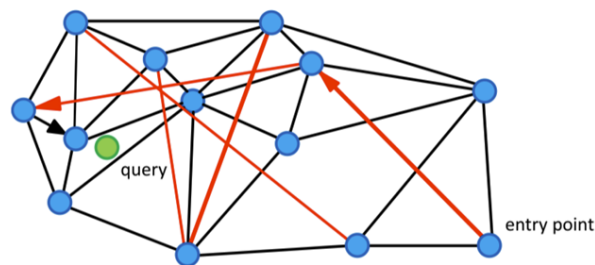
### Searching





## Graph-based vector search

- Key idea: a neighbor of a neighbor is likely to be a neighbor
- Nodes = data points (vectors)
- Edges = links to other “close” nodes



## Graph Construction: KNN-graph + pruning (global construction)

- Compute the exact KNN graph
- Prune edges using heuristics: e.g., keep diverse directions, not just the closest points (e.g., angular separation)
- Advantages:
  - High recall due to global optimization
- Disadvantages:
  - Requires all data up front
  - Can be slow and memory-intensive for large datasets

## Graph Construction: Incremental insertion (online construction)

- For each new point, perform a search
- Connect to the top nearest nodes
- Possibly prune to maintain diversity
- Advantages:
  - Can be done online (supports dynamic updates)
  - More memory-efficient; no need to compute full KNN graph
- Disadvantages:

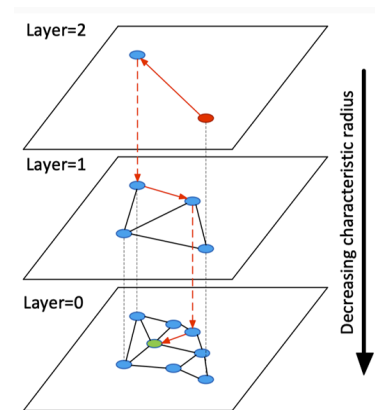
- Quality depends on insertion order and search quality

### Graph Construction: Traversal

- Select an entry point in the graph, e.g., a fixed point or a random selection
- QueueA: candidate nodes to visit
- QueueB: existing nearest neighbors
- Iteratively pop candidates from queue A and insert results to queue B
- End condition
  - Not enough high-quality candidate nodes to visit
  - All the nodes in queue A are further away from the query compared to queue B

### Hierarchical Navigable Small World (HNSW)

- Limitations of a regular graph
  - Search can get stuck in local minima - greedy walks can't escape to better regions
- Slow search in large graphs: More nodes -> longer walks
- Key Idea of HNSW: Add a Hierarchy
  - Upper layers: sparse, long-range links (quick overview of space)
  - Lower layers: denser, short-range links (local precision)
- Randomly assign a maximum layer to each point
- Insert top-down: for each layer, connect the node to nearest neighbors using heuristic rules (e.g., diversity).
- Traverse the upper layer first before moving to the layer below



## V. Related readings

1. (Optional) [Varuna: Scalable, Low-cost Training of Massive Deep Learning Models](#)
2. (Optional) [MODYN: A Platform for Model Training on Dynamic Datasets With Sample-Level Data Selection](#)
3. (Optional) [THEMIS: Fair and Efficient GPU Cluster Scheduling](#)