[CSCI-GA 3033-091] Fall 2025 Introduction to LLM based Generative AI Systems

Lecture 1 09/03/24

Class Introduction

Instructors

Parijat Dube <pd2637@columbia.edu>

Adjunct Professor, CS Dept

Machine learning, deep learning,

System performance optimization

Generative AI for enterprise automation

Chen Wang <cw3687@columbia.edu>

Adjunct Professor, CS Dept

Kubernetes, Container Cloud Platform,

Data-driven/QoE based resource management,

AI4Sys & Sys4AI, LLM Serving & Finetuning

Sustainable Cloud & AI systems

Class Introduction

- Course Assistants:

  Aneesh Mokashi akm9999@nyu.edu

  Geetha Guruji gg3039@nyu.edu

- Prerequisites:
  - Knowledge of ML and use of ML algorithms
  - Programming in Python

- Class on Brightspace:

  https://brightspace.nyu.edu/d2l/home/504821

- All information about the class will be available here including syllabus, announcements and assignments

Today's Agenda

- Course Overview
  - Syllabus
  - Assignments and Grading
  - Logistics
- Machine Learning Systems
- Machine Learning on Cloud and Model Lifecycle
- Model Performance and Complexity Tradeoffs
- Class 1 Topics

Course Information

- **What this course will cover ?**
  - DL concepts, training architectures, hyperparameters
  - LLM pre-training, fine-tuning and inference serving systems
  - Cloud based DL/LLM systems and performance issues
  - LLM systems performance evaluation tools, techniques, benchmarks
  - LLM systems performance optimization
  - Programming assignments involving GPUs on cloud
  - Research paper readings
- **What this course will not cover ?**
  - Other DL architectures like CNN, GANs, Autoencoders etc.
  - Mathematical analysis of DL algorithms

Educational Objectives

- Identify different components of DL /LLM system stack and their interdependencies
- Knowledge of ML/LLM model lifecycle and steps in making a trained model production ready
  - Train a DL/LLM model and make it a web service for inferencing
- Performance considerations, tools, techniques at different stages of model lifecycle: development, testing, deployment.
- DL/LLM training pitfalls and techniques/best practices data processing
- Ability to train DL/LLM models on cloud platforms using GPUs
- Performance characterization of DL/LLM systems
- Knowledge of DL/LLM benchmarks and performance metrics
- Performance optimization of DL/LLM systems

Class 1: Fundamentals of Deep Learning (DL)

- ML performance concepts/techniques: overfitting, generalization, bias, variance tradeoff, regularization
- Performance metrics: algorithmic and system Level
- DL training hyperparameters
  - batch size, learning rate, momentum, weight decay
- Single node vs distributed training
- Model and Data Parallelism
- Parameter server, all reduce
- Convergence and runtime
- Hardware Acceleration: GPUs, TPUs, NCCL

Class 2: Attention, Transformer, and Popular Large Language Models (LLMs)

- Seq2Seq models
- Encoder and decoder
- Attention mechanism
- Transformer architecture: self-attention, multi-head attention, encoder-decoder attention
- LLMs: BERT, OpenAI GPT, LLAMA, Gemini, Claude, IBM Granite

Class 3: Cloud Technologies and ML Platforms

ML System Stack on Cloud

This class introduces the ML system stack on cloud platforms, focusing on:

- Microservices architecture as foundation
- Major cloud ML platforms
- Distributed training frameworks
- GenAI application development tools

Microservices Architecture

Docker
Containerization for consistent environments

Kubernetes
Container orchestration for scalable deployment

Cloud ML Platforms

AWS SageMaker

Azure ML Studio

IBM Watsonx

Google Vertex AI
(Focus for course projects)

Distributed Training & ML Pipelines

Ray
Framework for distributed computing

- (Optional) TorchX for PyTorch-based distributed training
- Scalable ML workflows across clusters
- Efficient resource management

GenAI Application Tools

Dify
LLMOps platform for AI applications

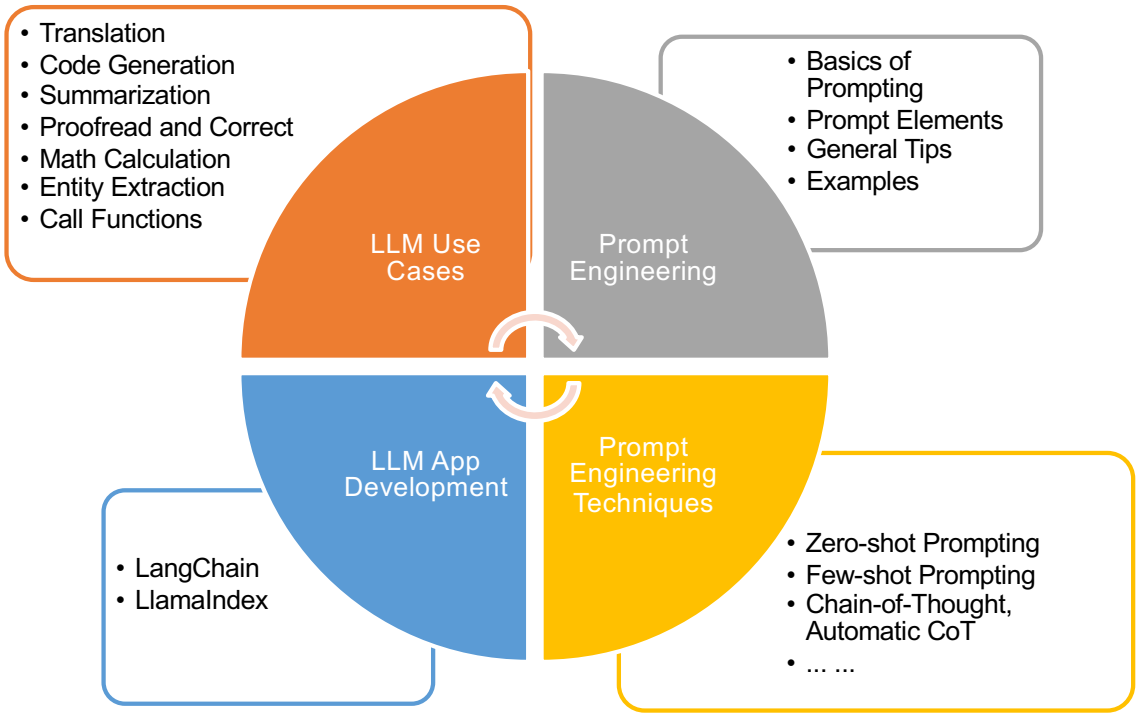
Gradio
Rapid UI creation for ML models

Enable rapid, often no-code prototyping of GenAI demos

Building Full-Stack GenAI Systems

- Practical insights into end-to-end ML systems
- Integration of microservices, cloud platforms, and GenAI tools
- Focus on Google Cloud Vertex AI for course projects
- Hands-on experience with scalable deployment

Class 4: Prompt Engineering and LLM Apps



Class 5: RAG and LLM Agents

Capabilities & Limitations of LLM

- Knowledge Cutoffs
- Hallucinations
- Structured Data Challenge
- Biases

Retrieval Augmented Generation

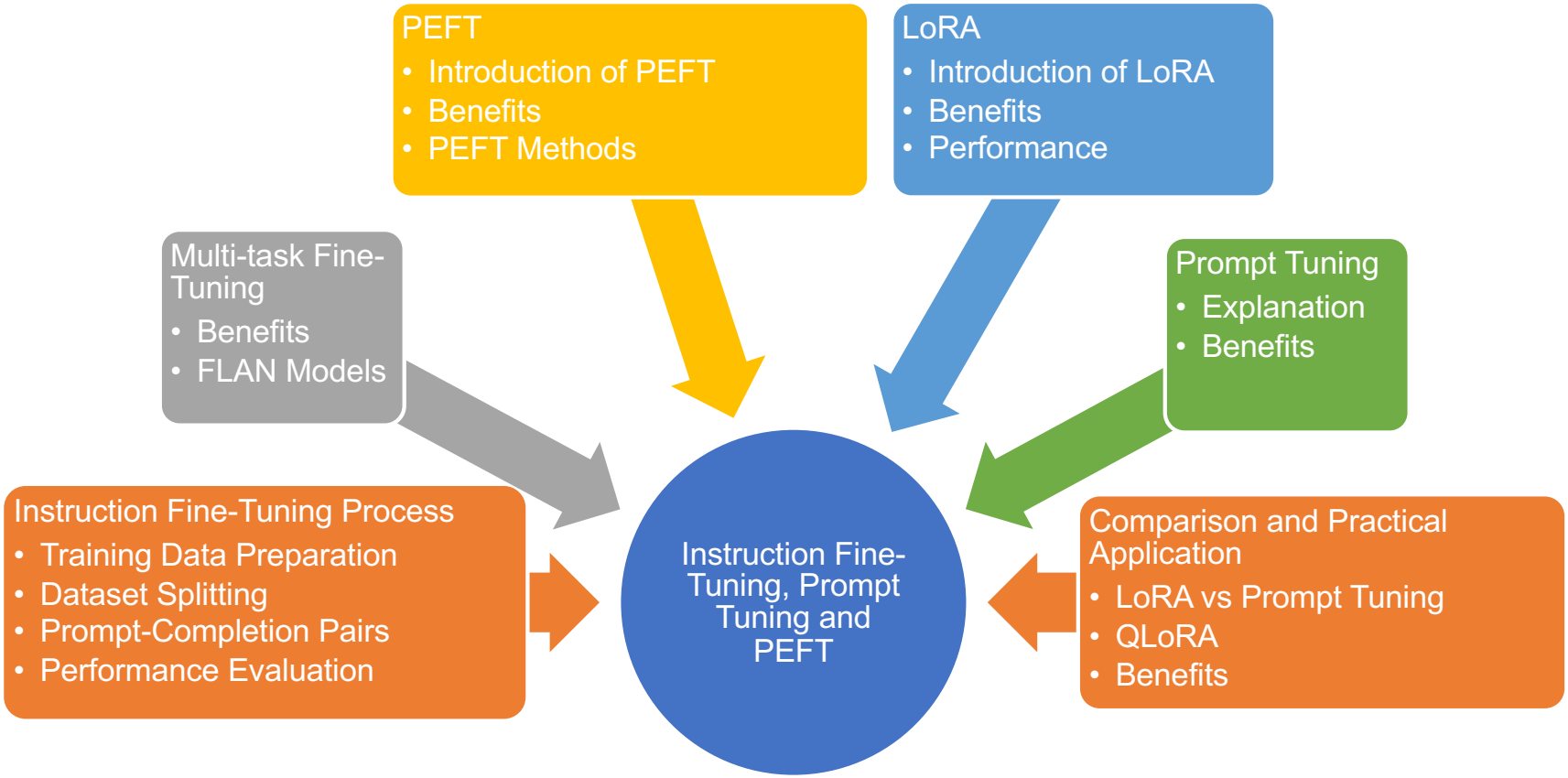
- Use Cases
  - Semantic Search
  - Summarization
- Keyword Search and Embeddings
- Retrieval and Rerank
- Answer Generation

Vector Databases

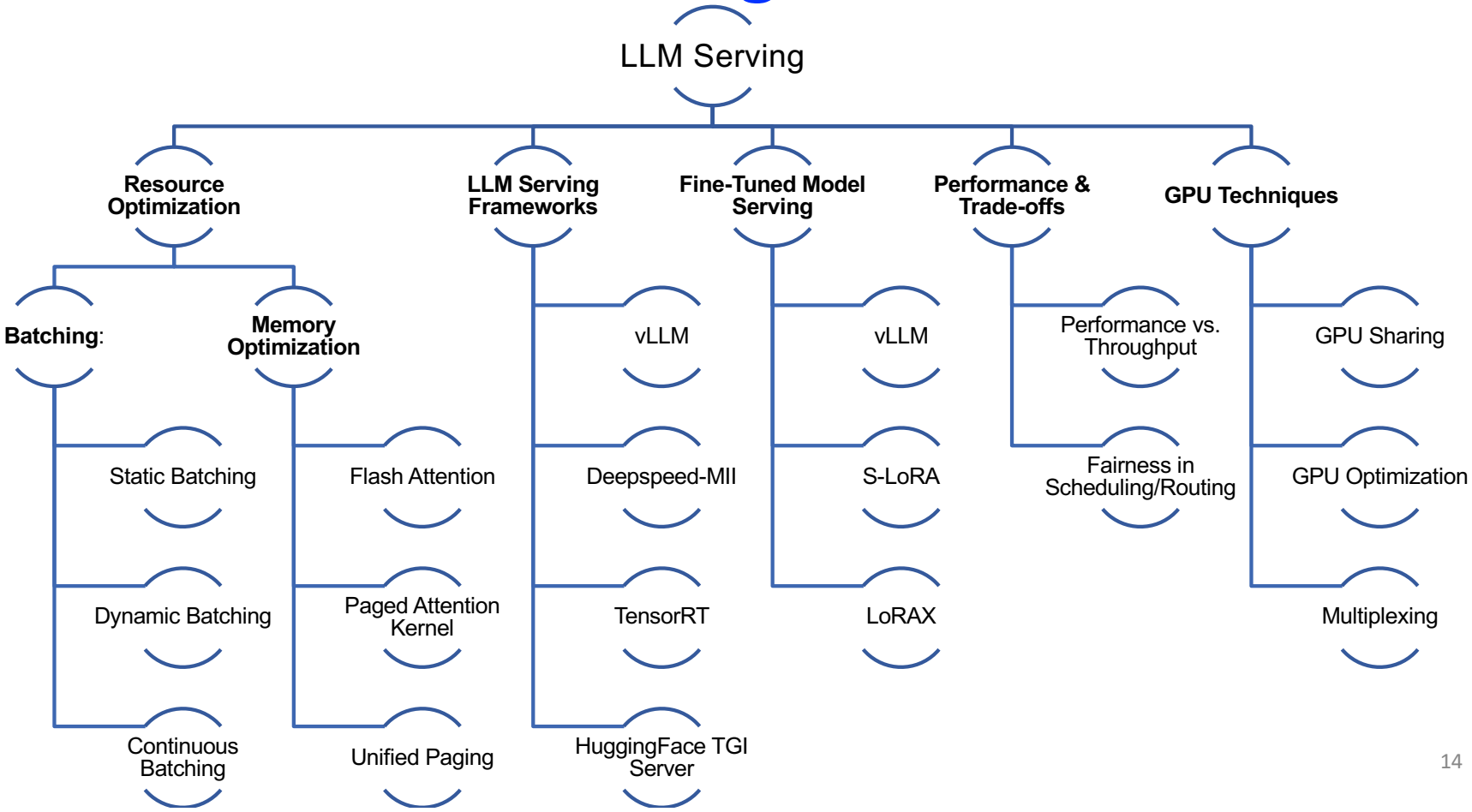
Class 6: Pre-Training for LLM

Pre-training Concepts	Training Process for Different Architectures	Managing High Memory Requirements	Scaling Model Training	Optimizing Training Resources	Use Cases for Custom LLM Pre-training
1.Training from existing foundation models 2.Training from scratch 3.Model selection from HuggingFace and PyTorch hubs	1.Encoder-only models 2.Decoder-only models 3.Sequence-to-sequence (seq-to-seq) models	1.Quantization techniques 2.Challenges with consumer-grade hardware	1.Distributed Data Parallel (DDP) 2.Fully Sharded Data Parallel (FSDP) 3.Zero Redundancy Optimizer (ZeRO)	1.Balancing model size, training data volume, and compute budget 2.Insights from the Chinchilla study	1.Domain adaptation (e.g., law, medicine) 2.Introduction to BloombergGPT as an example

Class 7: Fine Tuning Techniques



Class 8: Efficient Serving of LLMs

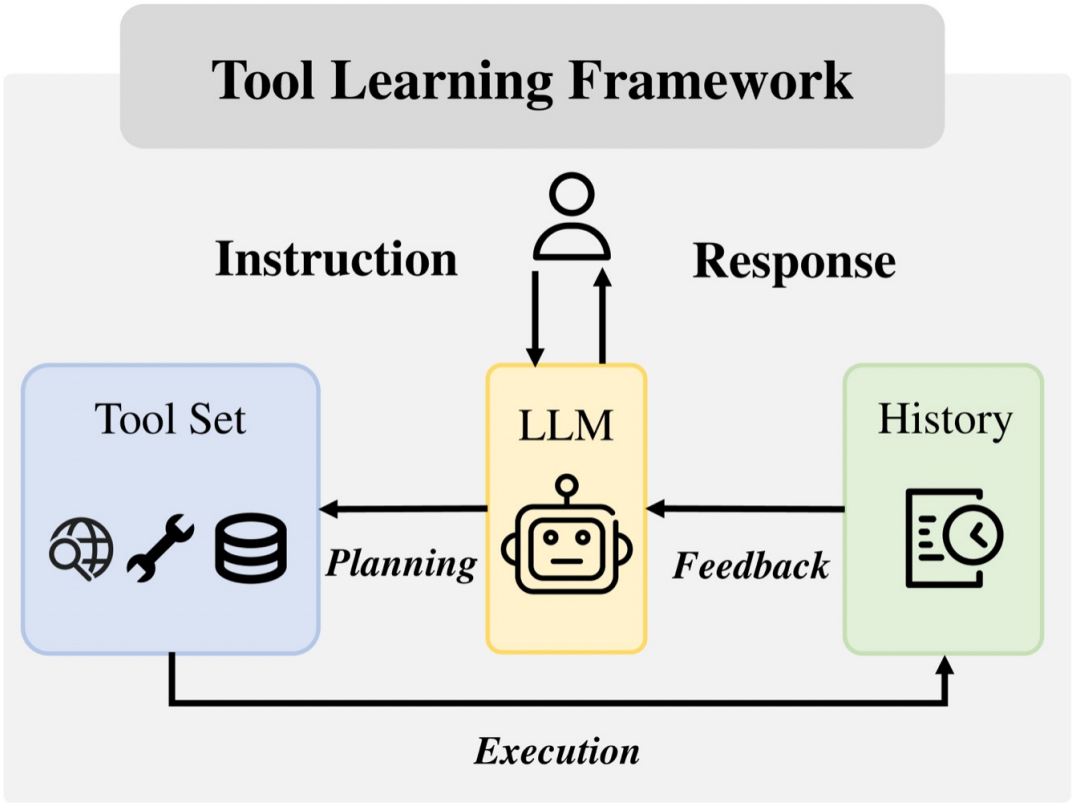


Class 9: RLHF

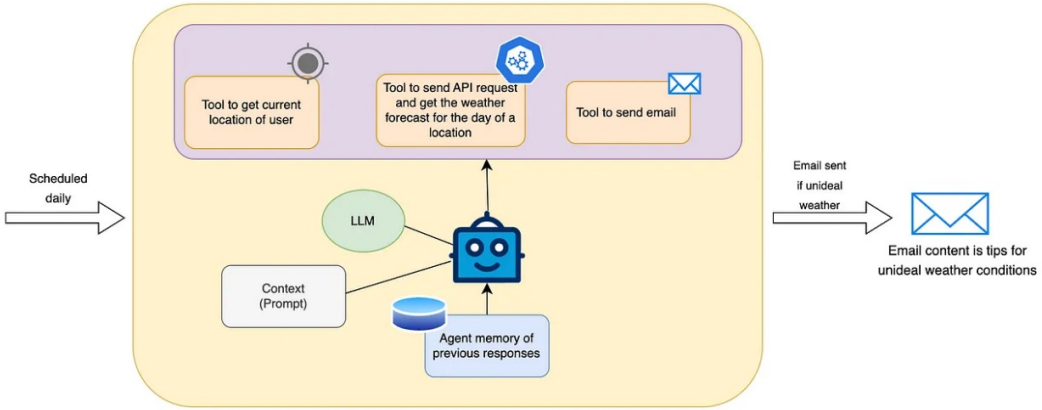
RLHF

Introduction	**Concept**: Human feedback guides reinforcement learning.
	Purpose: Align LLMs with human values/preferences.
Fine-Tuning Methods	**Instruction Fine-Tuning**: Tailor LLMs to follow specific instructions.
	Path Methods: Guide learning based on human feedback.
Challenges in RLHF	**Toxicity/Misinformation**: Mitigate harmful content.
	Human Alignment: Ensure helpful, honest, harmless models.
RLHF Process	**Reward Model Training**: Evaluate outputs, assign rewards.
	Automated Labelers: Replace human labelers, select completions.
	LLM Updating: Align models with human feedback.
Proximal Policy Optimization (PPO)	**PPO Overview**: Fine-tuning via reinforcement learning.
	RLHF Application: Iterative updates, maximize rewards.
Constitutional AI	**Concept**: Ethical AI alignment with human values.
	Role in RLHF: Ensure adherence to societal norms, trust, safety.

Class 10: Tool-assisted LLMs and Agentic AI



Class 10: Agentic systems



Weather-Advisory Agent Architecture

(1d) ReAct (Reason + Act)

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.
Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a remote control introduced in October 2005 by Apple … originally designed to control the Front Row media center program …

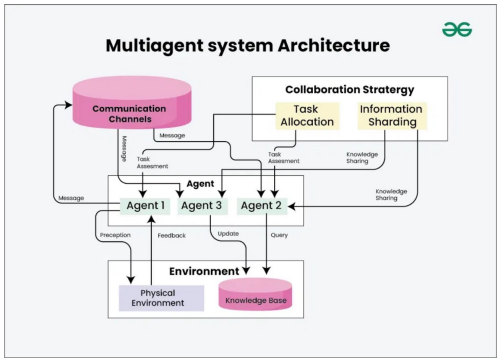
Thought 2: Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.
Act 2: Search[Front Row]
Obs 2: Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports',' Front Row (software)', …]

Thought 3: Front Row is not found. I need to search Front Row (software) .
Act 3: Search[Front Row (software)]
Obs 3: Front Row is a discontinued media center software …

Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.
Act 4: Finish[keyboard function keys]

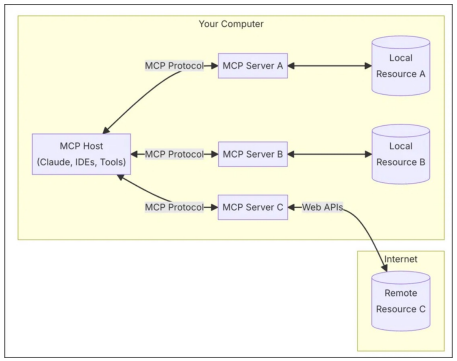
Class 11: Multi-agent System & MCP

Multi-agent Systems



- A pool of specialized agents collaborating to solve complex tasks
- Benefits: modularity, specialization, and control in agentic AI systems
- We'll study connection patterns between agents
- Frameworks: LangGraph and CrewAI

Multi-Context Protocol (MCP)



- Framework enabling LLMs to efficiently manage multiple conversations
- Improves performance and reduces computational costs
- We'll explore MCP clients and servers development
- Practical implementations for optimizing LLM interactions

Advanced AI Frameworks for Collaborative Intelligence

Class 12: Multimodal Generative AI systems

Definition & Importance

- **Definition**: Process multiple data types (text, images, audio, video).
- **Importance**: Human-like perception and interaction.

Beyond Language Models

- **Expansion**: Incorporate visual, auditory, sensory inputs.
- **Examples**: DALL-E, GPT-4 with vision.

Creating Large Multimodal Models (LMMs)

- **Incorporation**: Additional modalities into LLMs.
- **Architectures**: Encoder-decoder, transformer-based.
- **Training**: Cross-modal, contrastive learning.
- **Flamingo**: Visual language model, few-shot learning.

The Multimodality Revolution

- **Shift**: From unimodal to multimodal AI.
- **Advancements**: Computer vision, speech recognition, NLP.
- **Integration**: Unified models.

Class 12: Emerging Topics: Multimodal Generative AI

Vision-Language Models & Voice LLM | 21-11-2025

Multimodal AI & LMMs

- **Beyond LLMs:** Processing multiple data types simultaneously
- **LMMs:** Specialized encoders with shared reasoning layers
- **VLMs:** Vision-Language Models for visual reasoning
- **Applications:** Healthcare, retail, robotics, education

Hands-on Project

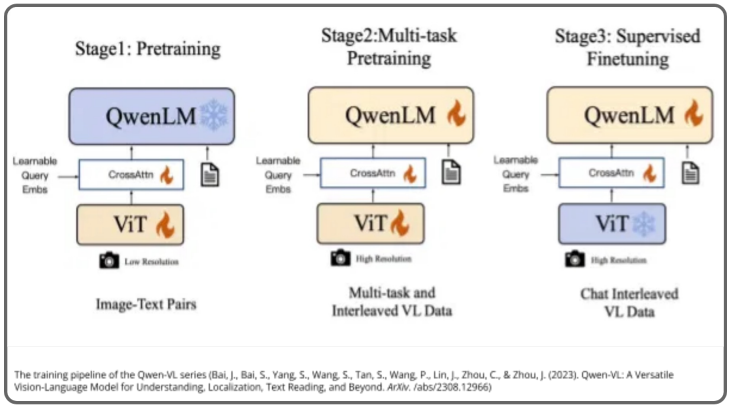
- **Goal:** Build production-ready voice agent
- **Partners:** LiveKit, RealAvatar, ElevenLabs
- **Learning:** Pipeline, optimization, deployment

Future Directions

- **Unified Models:** Single architectures for all modalities
- **Embodied AI:** Integration with physical interaction

AI Voice Agents

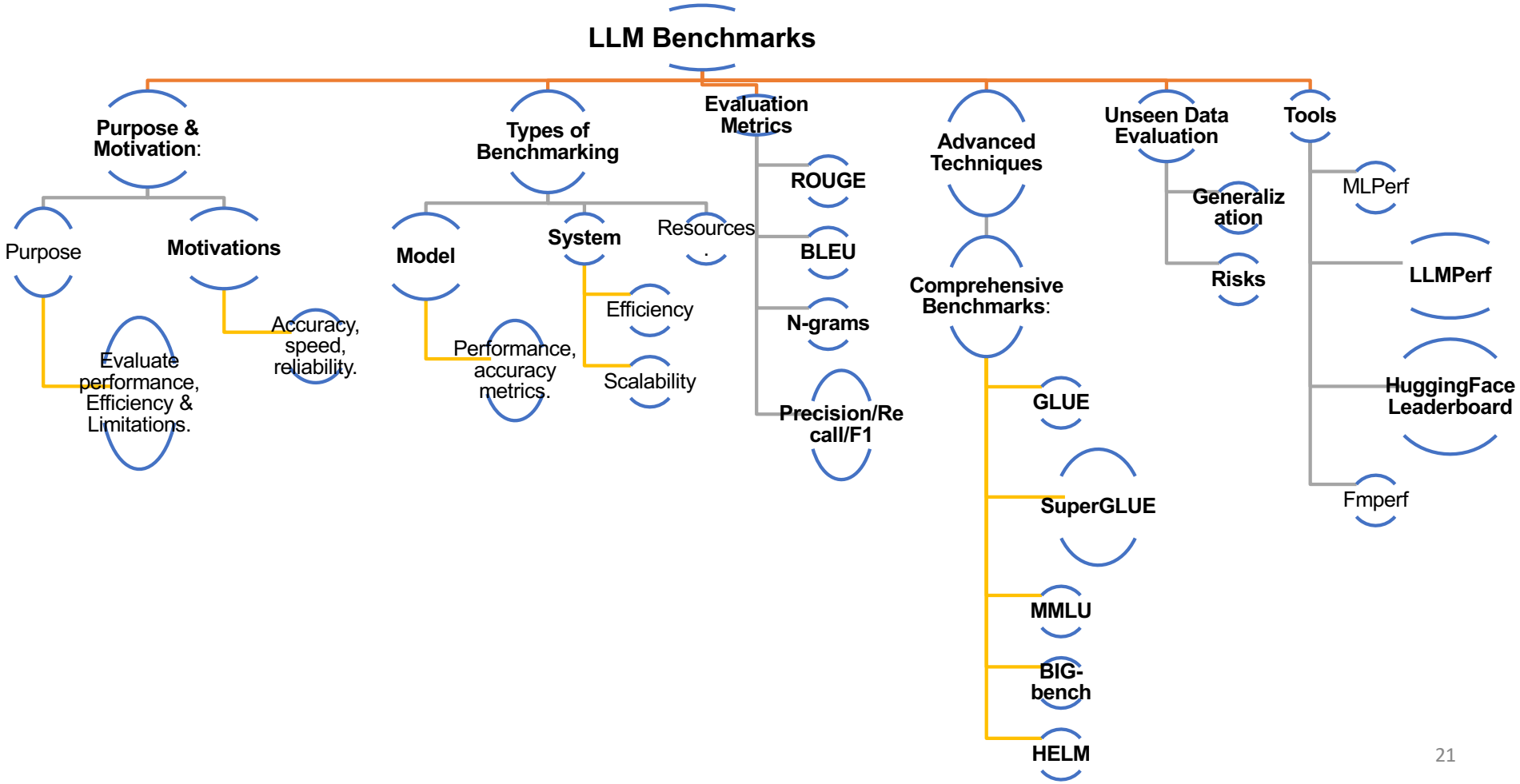
- **Definition:** Systems for real-time conversations
- **Architecture:** STT → NLU → LLM → TTS pipeline
- **Latency:** Critical for natural conversation flow



The training pipeline of the Qwen-VL series (Bai, J., Bai, S., Yang, S., Wang, S., Tan, S., Wang, P., Lin, J., Zhou, C., & Zhou, J. (2023). Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond. *ArXiv*. /abs/2308.12966)

"The future of AI is multimodal, integrating multiple senses like human intelligence."

Class 13: LLM Benchmarks



Recommended Books

- This course does not follow any textbook
- Background knowledge
  - List of books (covering deep learning topics)
    - Charu Aggarwal "Neural Networks and Deep Learning", available at https://link.springer.com/book/10.1007/978-3-319-94463-0
    - Goodfellow, Bengio, Courville, "Deep Learning", available at http://www.deeplearningbook.org
    - These books are good for basic DL understanding
  - For basics of machine learning concepts an excellent textbook is G. James et al "*Introduction to Statistical Learning Theory*". Second version available is available for free download at https://www.statlearning.com
- All other reading material will be posted on Canvas.

Assignments and Grading

- **Distribution of marks**:
  - Assignments: 40%
  - Quizzes: 20%
  - Final Project: 40%
- **Assignments (40%)**
  - 5 assignments
  - Assignments posted at the end of lectures 2, 4, 6, 8, 10; due in 2 weeks
  - All programming assignments should be done as Jupyter notebooks, unless specified otherwise
- **Quizzes (20%)**
  - Canvas
  - 5 quizzes

Assignments and Grading (contd.)

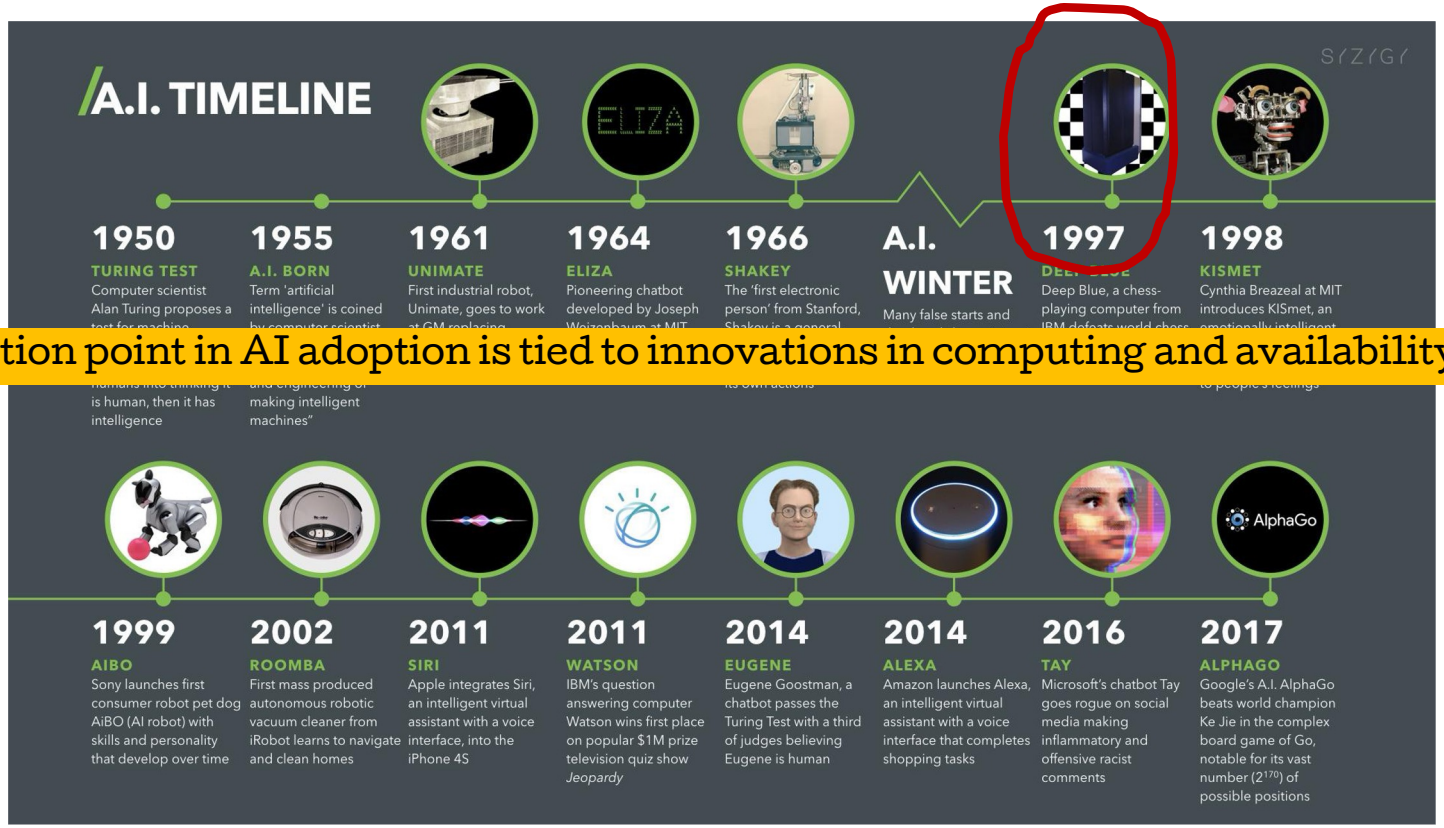
- **Final Project (40%):** Team assignment. Team of 2. Any project involving development of new LLM solutions and/or performance optimization of existing LLM systems. 2-page project proposal due by Midterm. Detailed rubric shall be provided. Project grading:
  - Project proposal (5%) – due in mid October
  - Midpoint checkpoint (5%) - due before Thanksgiving break
  - Github repo with README, documented code (5%)
  - Final presentation and demo (15%)

Class Logistics

- Reach CAs: Office hours and Campuswire
- Access to Computer Clusters
- Class communications:Campuswire

AI Timeline

Cloud computing was coined



Inflection point in AI adoption is tied to innovations in computing and availability of big data

2006: Amazon elastic cloud and S3 was launched

Factors Contributing to AI Success

- **Algorithms, Data, Compute, Applications**
  - Distributed training algorithms scaling upto 100s of GPUs
  - Data growing at exponential rate; Internet, Social media, Internet of thngs (IoT)
  - Compute power growth with specialized cores; GPUs, TPUs
  - Development of innovative applications
- **2012** Alexnet by Krizhevsky et al at ImageNet Competition
  - Simple convolutional neural network: 5 convolutional, 3 fully connected
  - GPU based; Beat other models by 11% margin
  - Triggered ”Cambrian Explosion” in deep learning technologies

  “*Neural networks are growing and evolving at an extraordinary rate, at a lightening rate,….What started out just five years ago with AlexNet…five years later, thousands of species of AI have emerged*.”

Evolution of Large Language Models (LLMs)

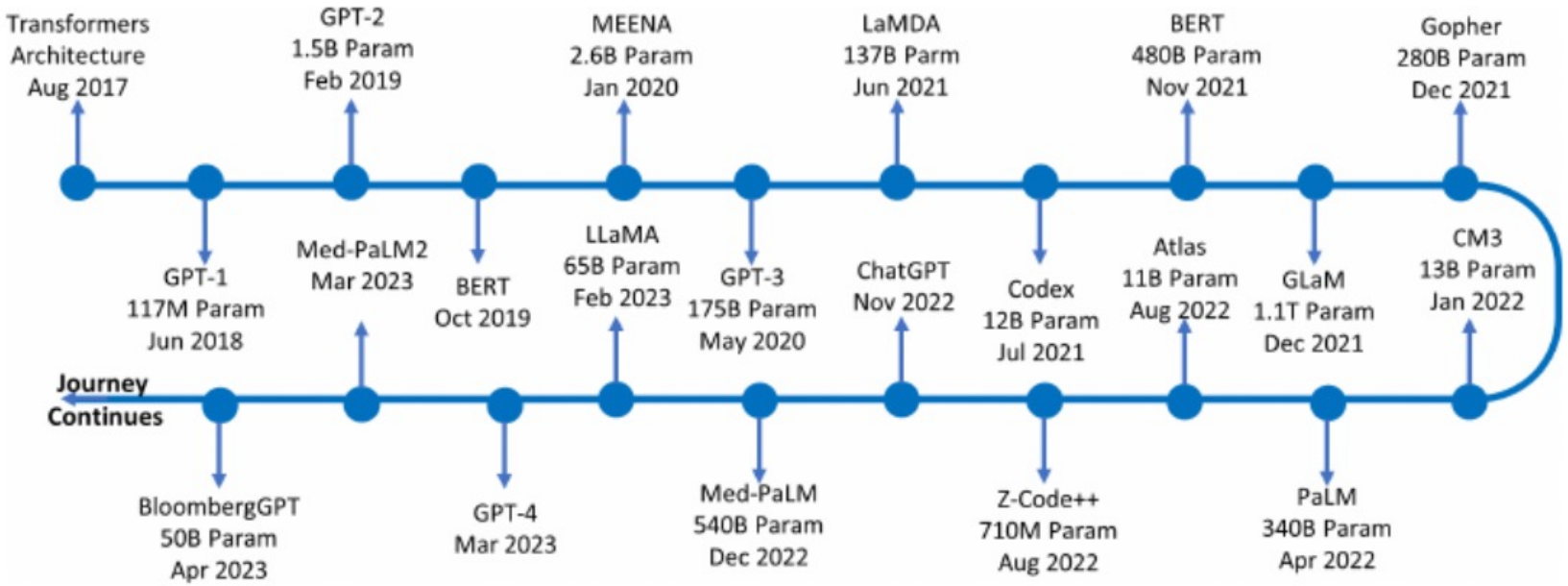


Figure from Mohamadi et al, [ChatGPT in the Age of Generative AI and Large Language Models: A Concise Survey]

AI Blogs of Major Companies

- Meta: https://ai.meta.com/blog/
- Google: https://ai.googleblog.com
- IBM Research:

  https://www.ibm.com/blogs/research/category/ai/
- Microsoft: https://news.microsoft.com/source/topics/ai/
- AWS Machine Learning Blog: https://aws.amazon.com/blogs/machine-learning/

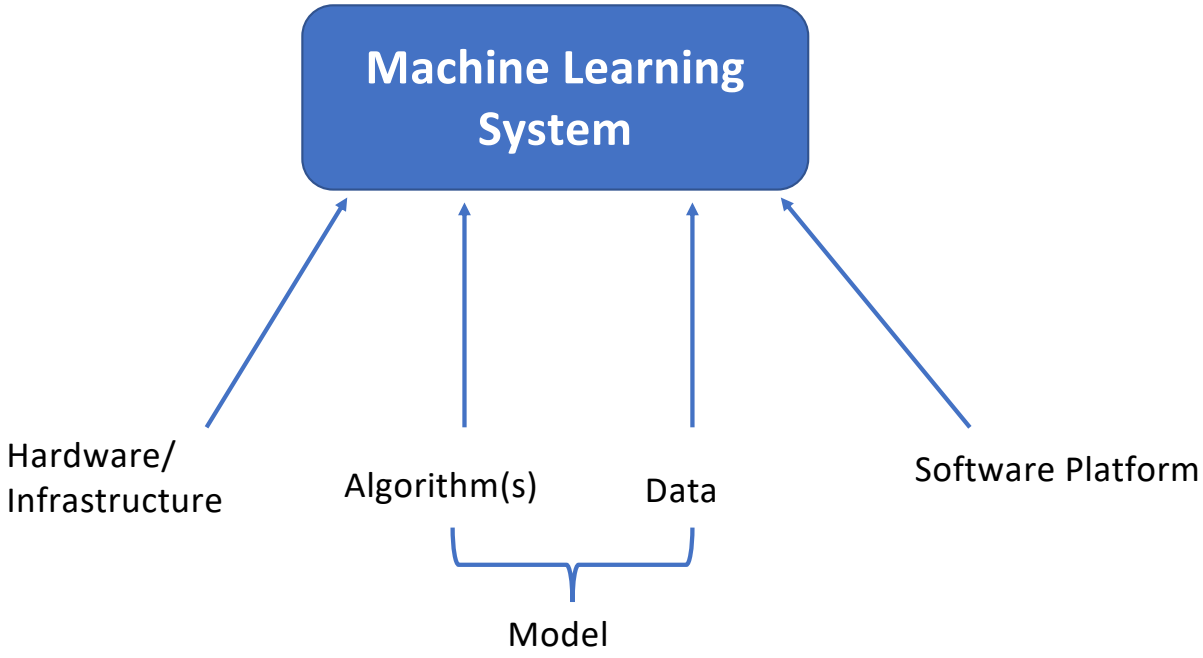
Machine Learning System

A composition of one or more software components, with possible interactions, deployed on a hardware platform with the purpose of achieving some performance objective.

A Machine Learning system is a system where one or more software components are machine learning based.

- Why study ML systems ?
  - Algorithms run on real and (possibly) faulty hardware in production environments
  - Theoretical performance is far away from observed
  - To characterize holistic performance of not just the algorithm but the end-to-end performance of the entire system

Constituents of a ML System



Infrastructure

- Compute units and accelerators, memory, storage, network
- Resources can be acquired as bare metal, VMs/Containers on cloud
- Design better hardware
  - Adapt existing architectures to ML tasks.
  - Develop brand-new architectures for ML.
- Hardware compute precision affects performance: tradeoff between accuracy and runtime

(Learning) Algorithm

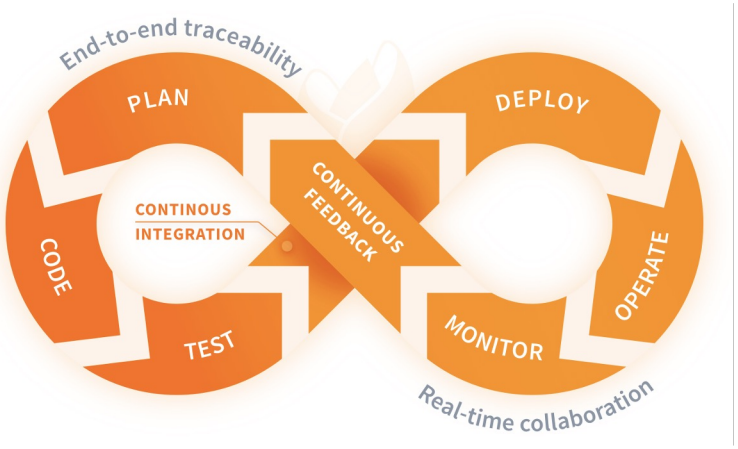
- General and domain specific architectures
- Hyperparameter tuning to extract the best performance
- Effects the resource requirements: compute (FLOPS), memory
- Performance (runtime) and scalability of an algorithm depends on:
  - Hardware/Infrastructure
  - Software platform (frameworks, libraries, drivers)

Data

- Data as a critical element; Data is the king in ML
- Different modalities: Audio, video, images, text
- Data sources, collection, labeling, quality, data storage
- Data type determines the choice of learning algorithm
- Making the data *business ready* is challenging
- Many data-driven organizations are spending 80 percent of their time on data preparation and find it a major bottleneck.
- DataOps: tools, processes, and organizational structures to cope with significant increase in volume, velocity, and variety of data.

Software Engineering in ML Systems

- Machine learning applications run as pipelines that ingest data, compute features, identify model(s), discover hyperparameters, train model(s), validate and deploy model(s).
- Making a model as a production-capable web service
  - Containerization (docker), cluster deployment (K8s)
  - APIs exposed as web service (Tensorflow serving/ONNX runtime)
- Workflow engines (e.g., Kubeflow) automate the ML pipeline
- Deployment monitoring and operational analytics
- Devops principles applicable to ML Systems:
  - Continuous Integration, Continuous delivery (CI/CD)
  - Predictability
    - "A model may be unexplainable—but an API cannot be unpredictable"
  - Reproducibility and Traceability
    - Provenance for machine learning artifacts



ML specific testing and monitoring apart from traditional software testing

- Data testing
- Infrastructure testing
- Model testing
- Production testing

Inhibitors in Successful Implementation of ML Solutions

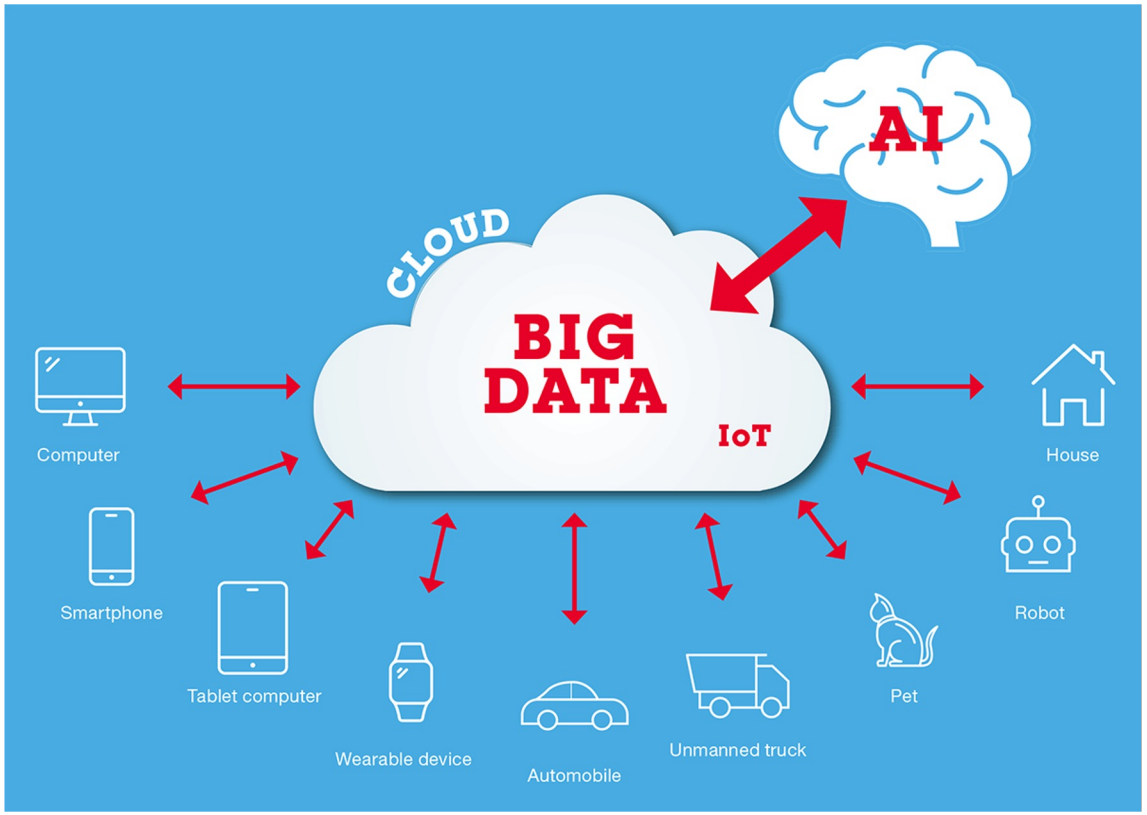
- Deployment and automation
- Reproducibility of models and predictions
- Diagnostics
- Governance and regulatory compliance
- Scalability
- Collaboration
- Monitoring and management

Cloud Computing

- Access to computing resources and storage on demand
- Pay-as-you go model
- Heterogeneous resources: GPUs, CPUs, storage type
- Different offering models: IaaS, PaaS, SaaS, MLaaS
- Different deployment models: Public, private, hybrid cloud
- Provisioning, maintenance, monitoring, life-cycle-management

Cloud and AI

- AI
  - Harness power of Big Data and compute
- Cloud
  - Access to Big Data
  - Platform to quickly develop, deploy, and test AI solutions
  - Ease in AI reachability



Cloud based Machine Learning Services

- IBM Watson Studio

https://www.ibm.com/products/watson-studio

- Amazon Sagemaker

https://aws.amazon.com/sagemaker

- Microsoft Azure Machine Learning

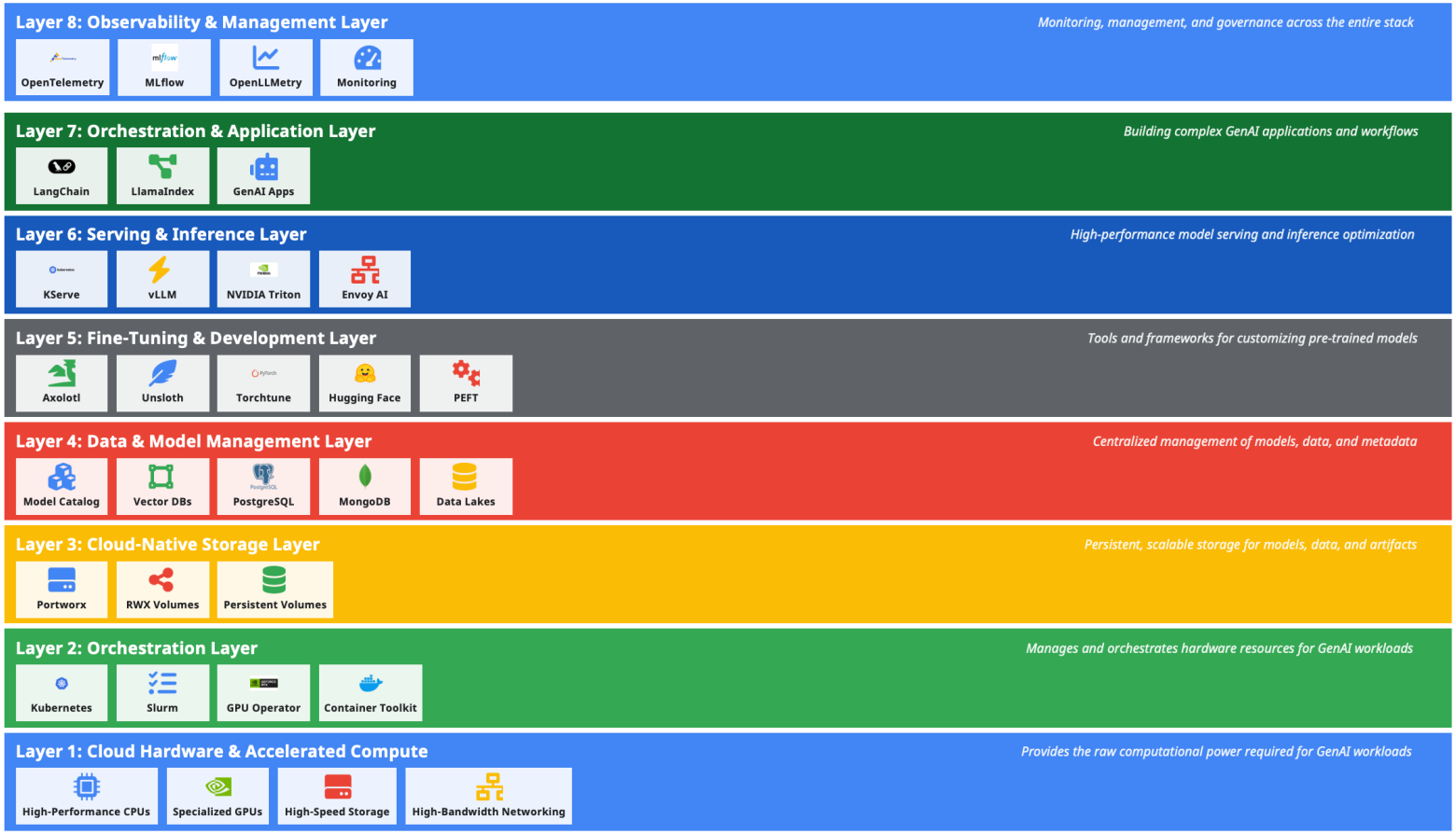
https://azure.com/ml

- Google Vertex AI Platform

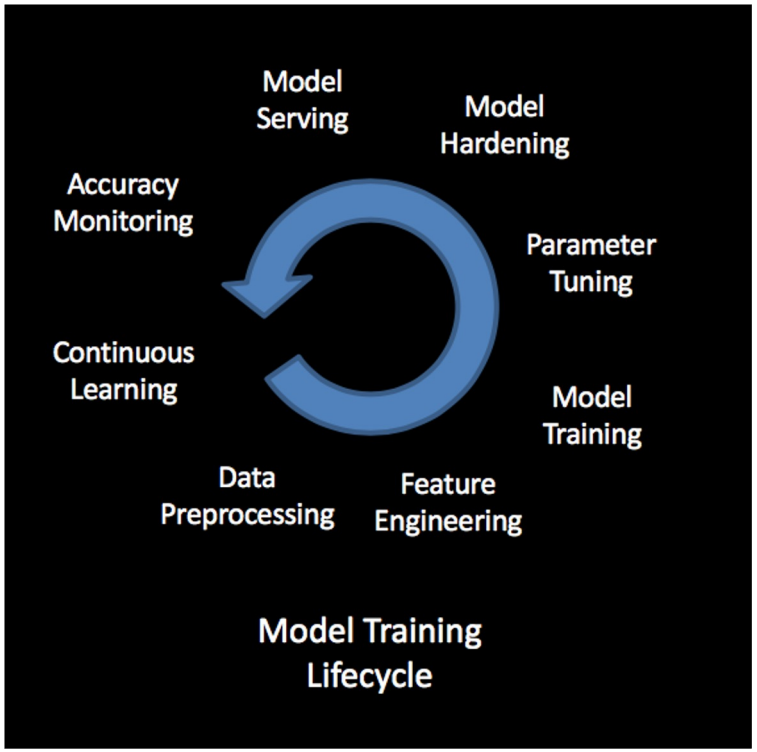
https://cloud.google.com/vertex-ai/

GenAI on Cloud Stack

GenAI on Cloud Full Stack



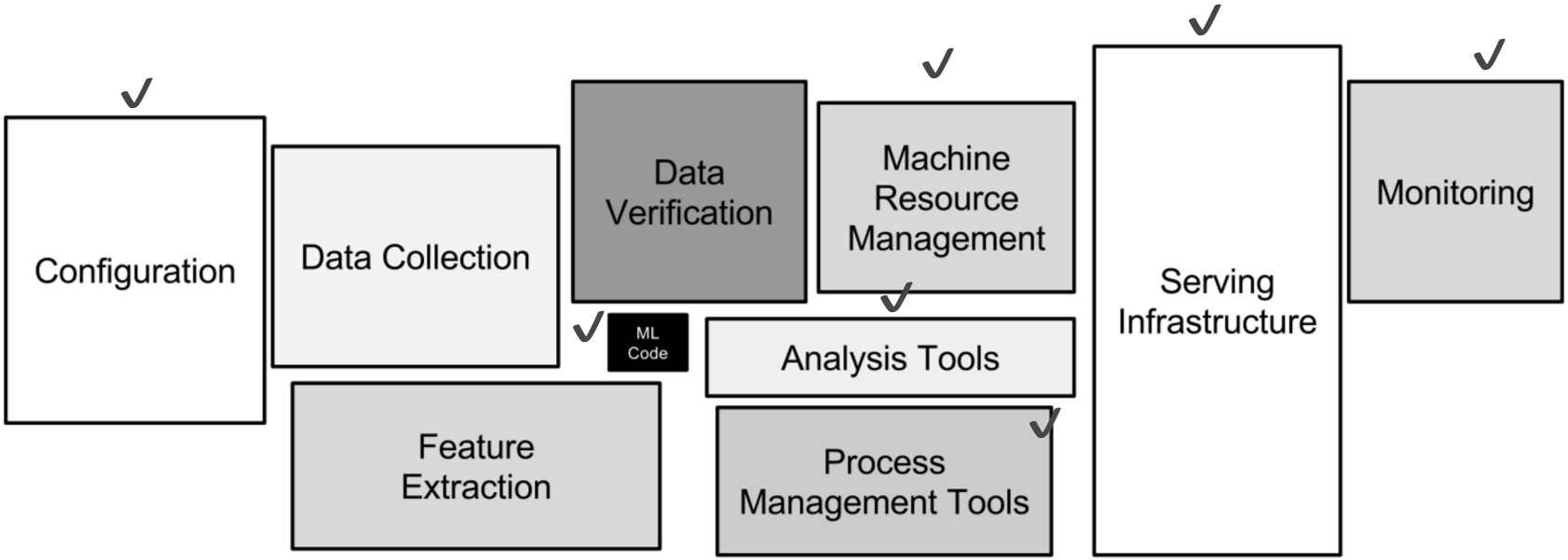
AI Model Training Lifecycle



Performance considerations at each stage

- Data preprocessing: de-noising, de-biasing, train/test set creation
- Feature engineering: search efficient data transformations
- Model training: model identification/synthesis, hyperparameter tuning, regularization
- Model hardening: efficient adversarial training
- Model serving: hardware, model pruning and compression
- Monitoring: response time, drift detection
- Continuous learning: model adaptability, retraining

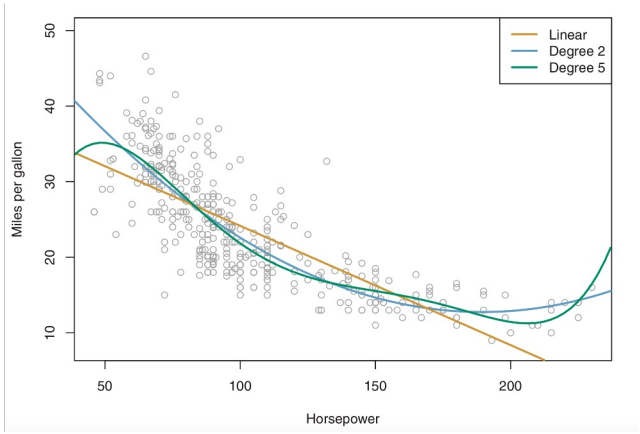
Practical Machine Learning Systems

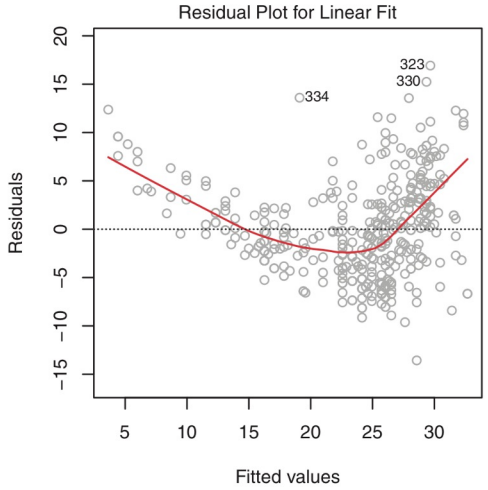


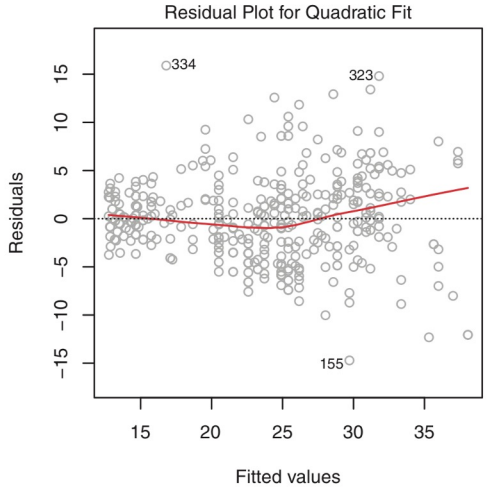
Class 1: Fundamentals of Deep Learning (DL)

Linear Regression

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_p x_p$$







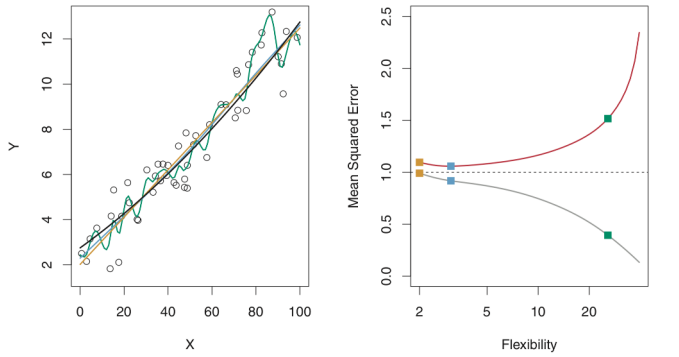
$$\text{TSS} = \sum (y_i - \bar{y})^2$$

$$\text{RSS} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

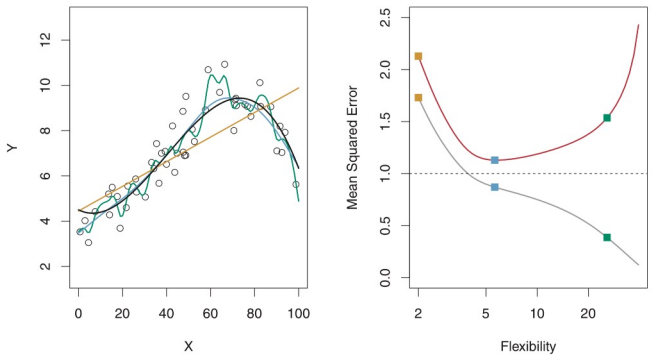
$$R^2 = \frac{\text{TSS} - \text{RSS}}{\text{TSS}} = 1 - \frac{\text{RSS}}{\text{TSS}}$$

Mean Square Error (MSE)

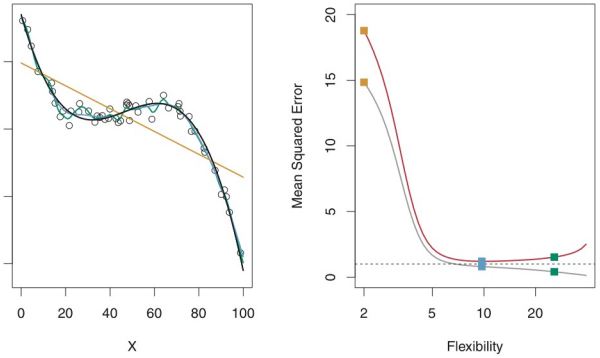
CASE 1



CASE 2



CASE 3



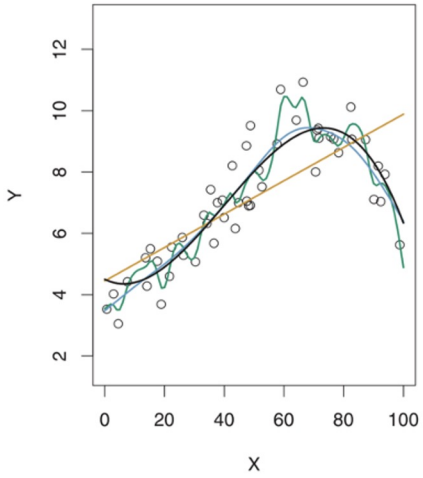
true value $Y = f(X) + \epsilon.$

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2.$$

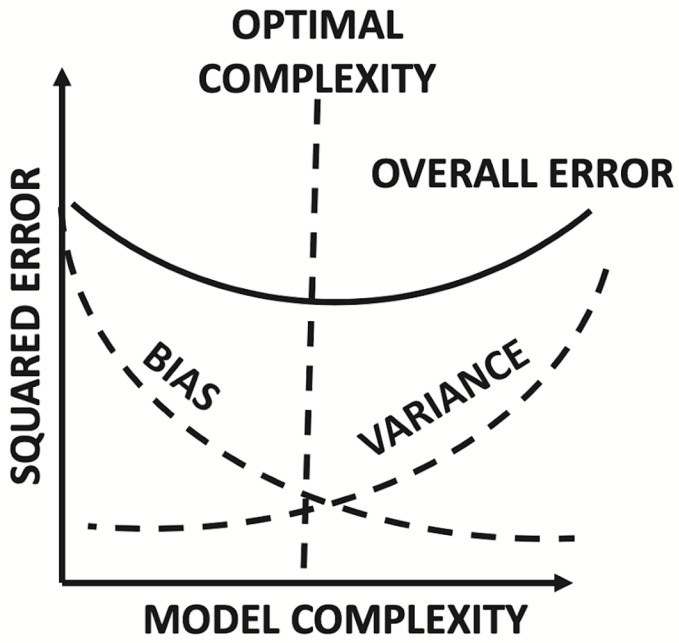
predicted $\hat{Y} = \hat{f}(X)$

Overfitting and Underfitting

- Overfitting: model performs well on training data but does not generalize well to unseen data (test data)
- Underfitting: model is not complex enough to capture pattern in the training data well and therefore suffers from low performance on unseen data



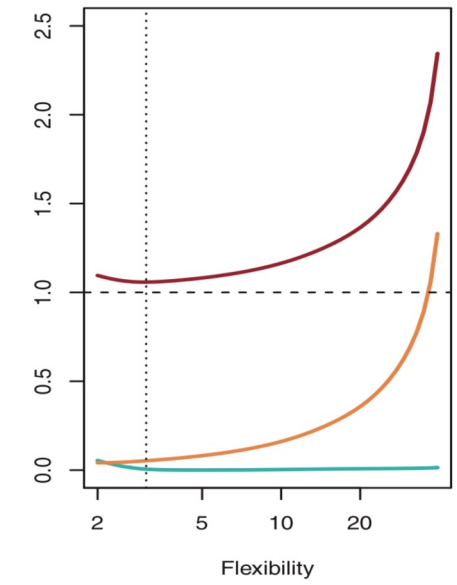
The Bias-Variance Trade-Off



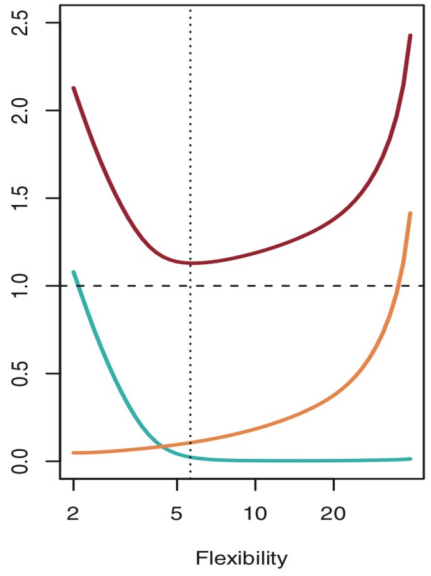
- Optimal point of model complexity is somewhere in middle.

Model Complexity Tradeofffs

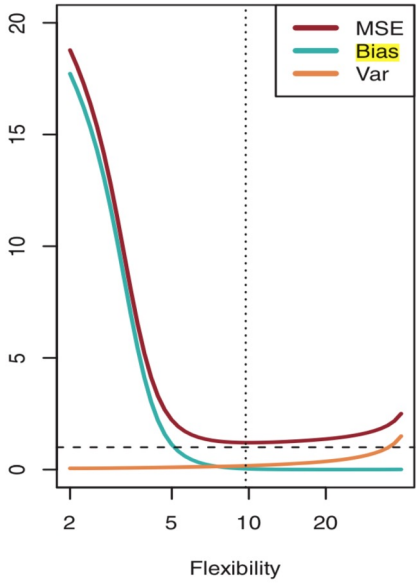
CASE 1



CASE 2



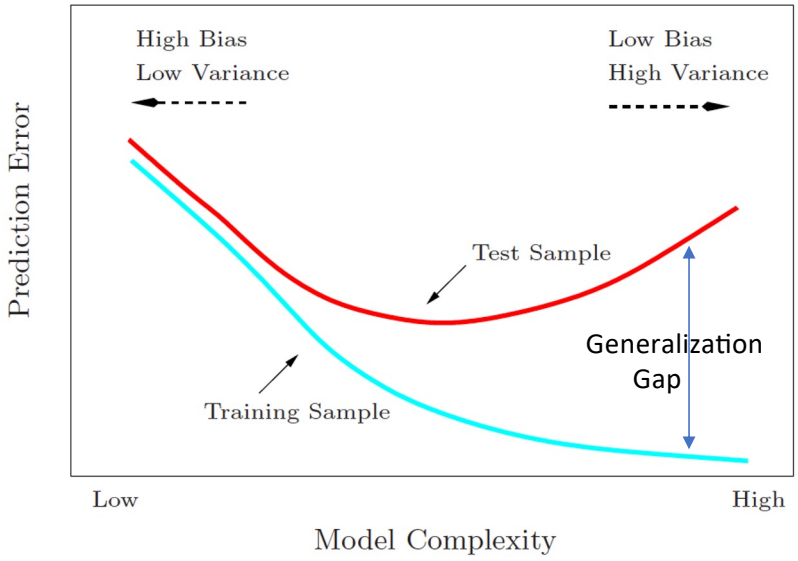
CASE 3



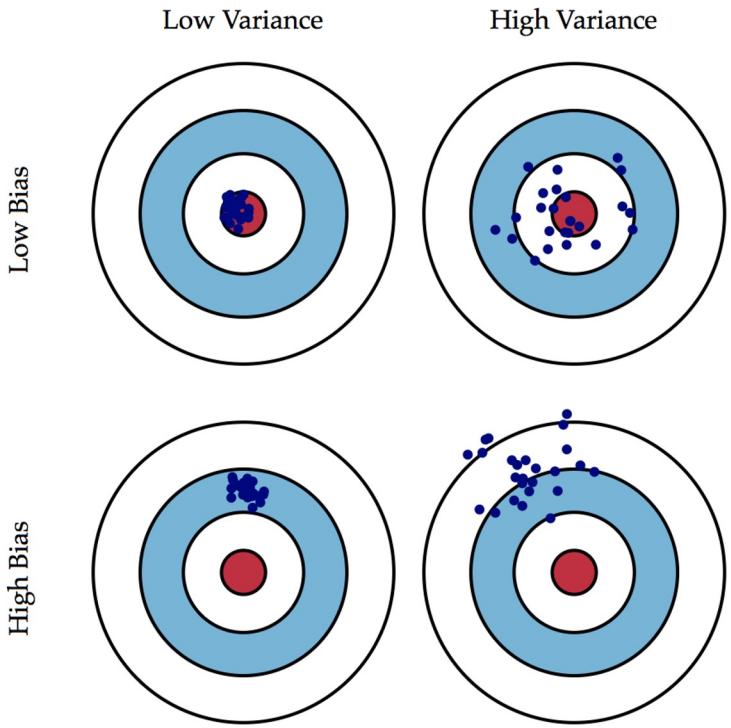
$$E\left(y_0 - \hat{f}(x_0)\right)^2 = \mathrm{Var}(\hat{f}(x_0)) + [\mathrm{Bias}(\hat{f}(x_0))]^2 + \mathrm{Var}(\epsilon)$$

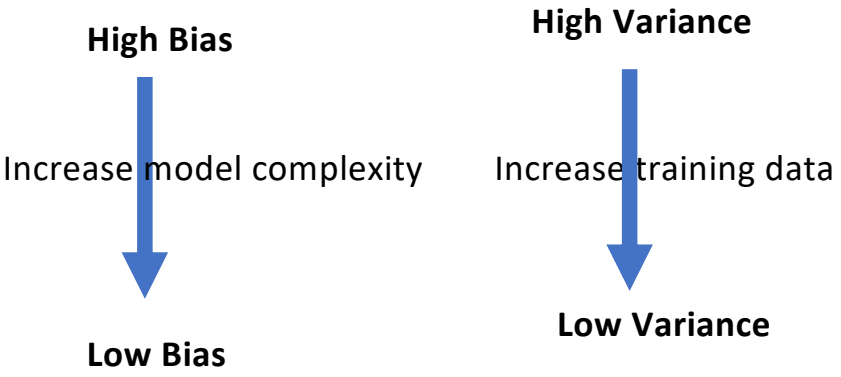
Model Complexity Tradeoffs

- Simple model
  - Fail to completely capture the relationship between features
  - Introduces bias: Consistent test error across different choices of training data
  - Low variance
  - Increasing training data does not help in reducing bias
- Complex model captures nuances in training data causing Overfitting
  - Low bias
  - Train error << Test error
  - With different training instances, the model prediction for same test instance will be very different – High Variance



Bias-Variance Tradeoff





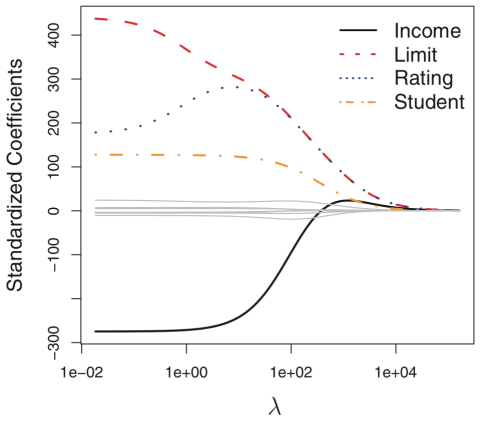
Regularization

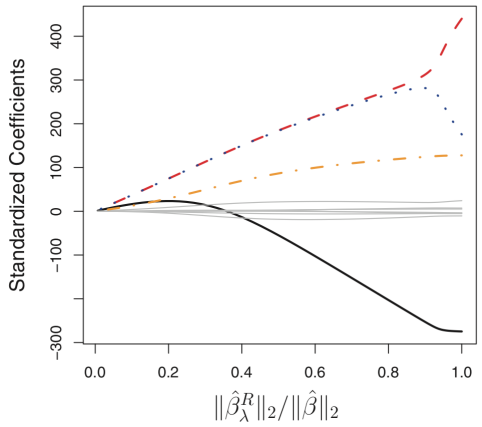
- Techniques used to improve generalization of a model by reducing its complexity
- Techniques to make a model perform well on test data often at expense of its performance on training data
- Avoid overfitting, reduce variance
- Simpler models are preferable: low memory, increase interpretability
- However simpler models may reduce the expressive power of models
- Sample techniques:
  - Parameter norm penalties
    - L_2 and L_1 norm weight decay
  - Noise injection
    - Dropout

Regularization in Regression

L_2 Regularization Loss (Ridge Regression)

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2+\lambda\sum_{j=1}^{p}\beta_j^2$$

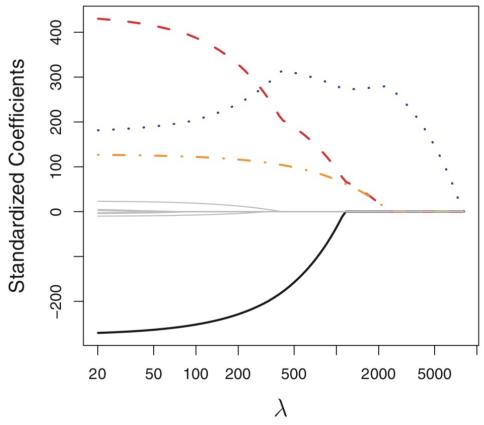


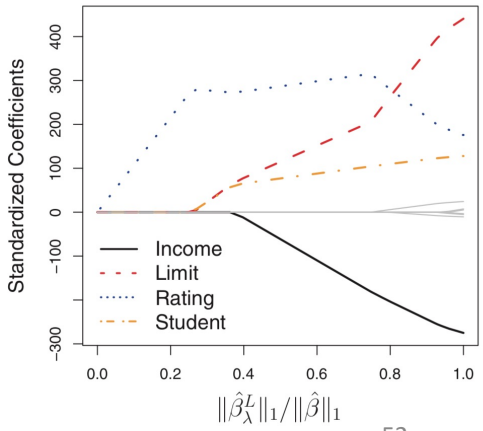


L_1 Regularization Loss (LASSO)

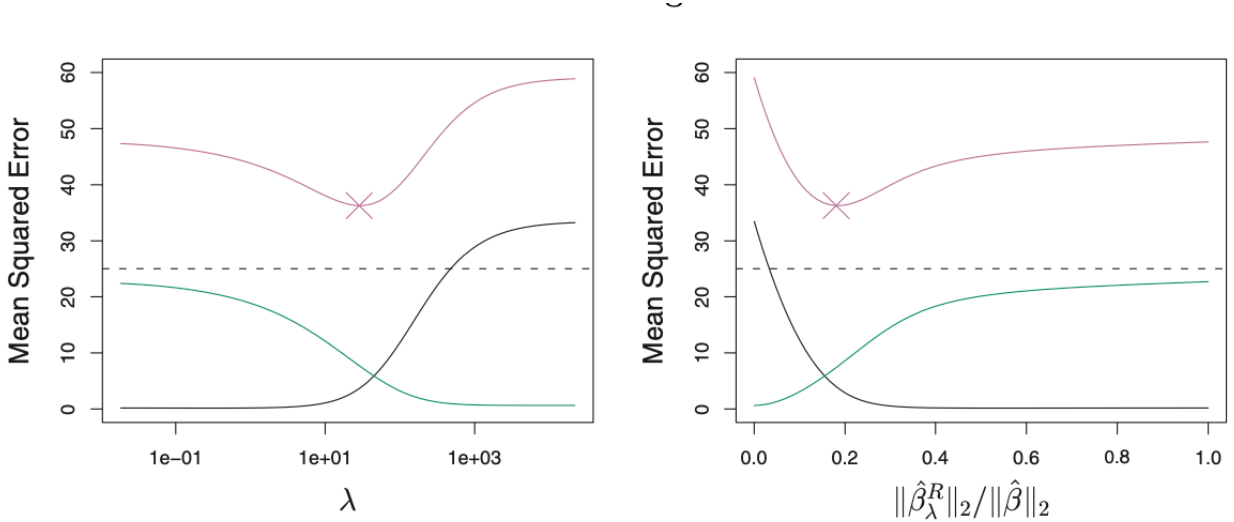
$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2+\lambda\sum_{j=1}^{p}|\beta_j|$$

What value of lambda to choose ?





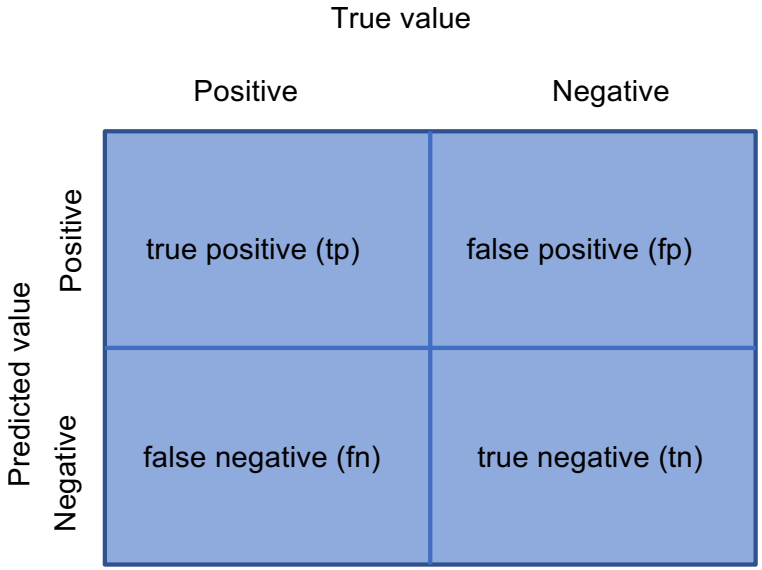
Bias-variance tradeoff with Lambda



Performance Metrics

- Algorithmic performance: accuracy, precision, recall, F1-score, ROC,
- System performance: training time, inference time, training cost, memory requirement, training efficiency

Accuracy, Precision, Recall, Specificity



$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

$$\text{Precision} = \frac{tp}{tp + fp}$$

False discovery rate = 1-Precision

$$\text{Recall} = \frac{tp}{tp + fn}$$

Sensitivity, True positive rate

$$\text{True negative rate} = \frac{tn}{tn + fp}$$

Specificity

False positive rate = 1-Specificity

Balanced accuracy = $(Sensitivity + Specificity)/2$

Considers all entries in the confusion matrix
Value lies between 0 (worst classifier) and 1 (best classifier)

Balancing Precision and Recall

- F1 score: Harmonic mean of precision and recall; measure of classifier accuracy

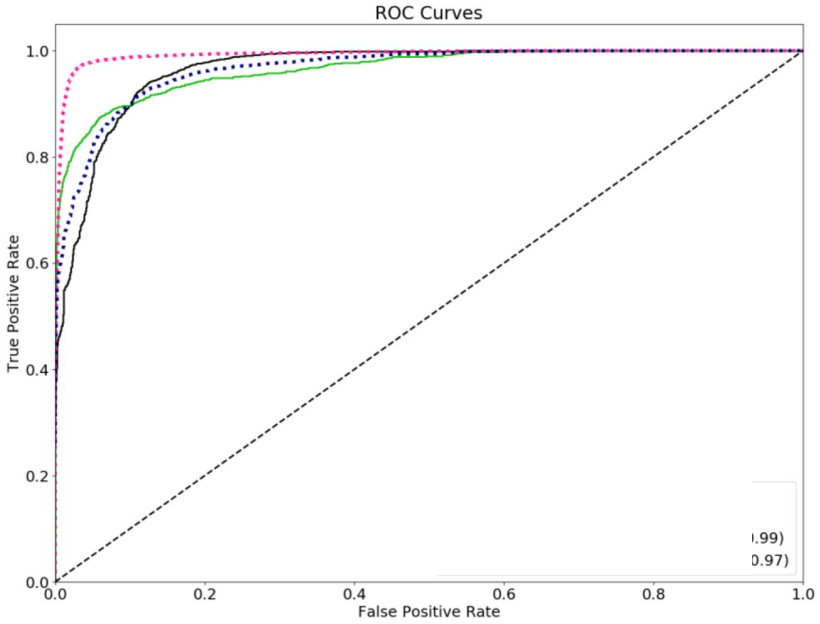
$$F_1 = \frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} = \frac{\mathrm{tp}}{\mathrm{tp} + \frac{1}{2}(\mathrm{fp} + \mathrm{fn})}.$$

- F_β $score$: $\beta = 1$ is $F1$ $score$; recall is considered β times as important as precision

$$F_\beta = (1 + \beta^2) \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\beta^2 \cdot \mathrm{precision} + \mathrm{recall}}$$

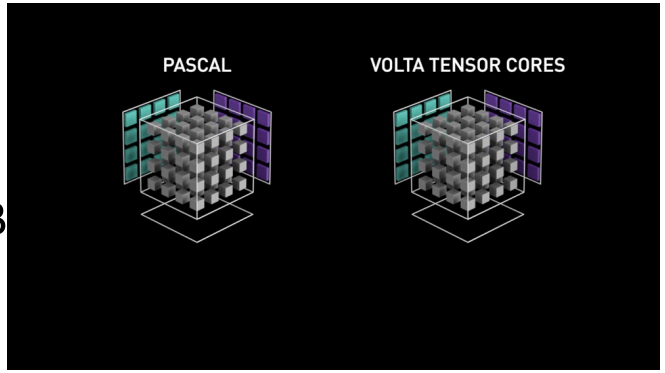
Receiver Operating Characteristics (ROC)

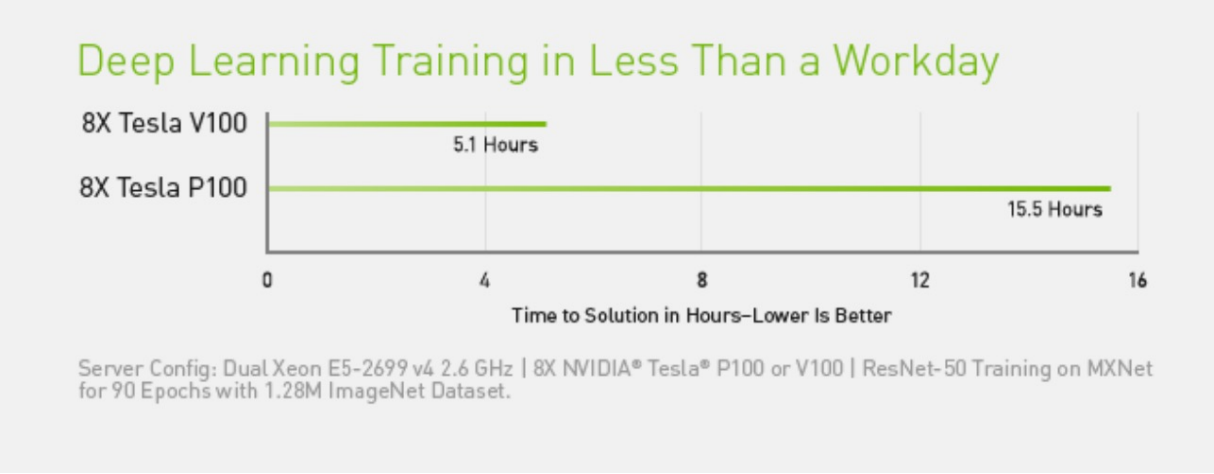
- Plots true positive rate (Recall) vs false positive rate at different thresholds



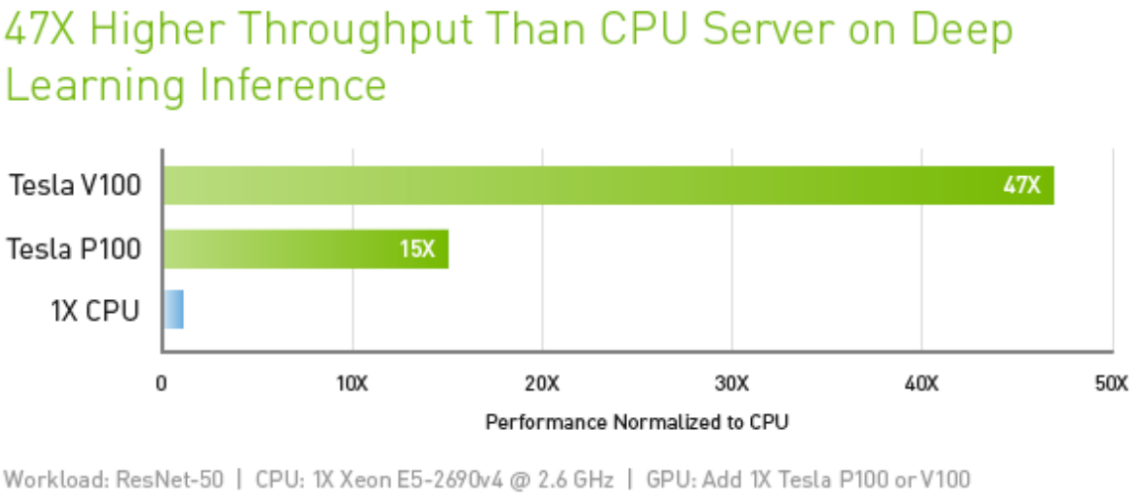
DL Training Time

- DL performance is closely tied to the hardware
  - compute power, memory, network
  - Tesla V100: 640 tensor cores (> 100 TFLOPS), 16 GB
  - NVIDIA NVLink: 300 GB/s
  - Volta optimized CUDA libraries



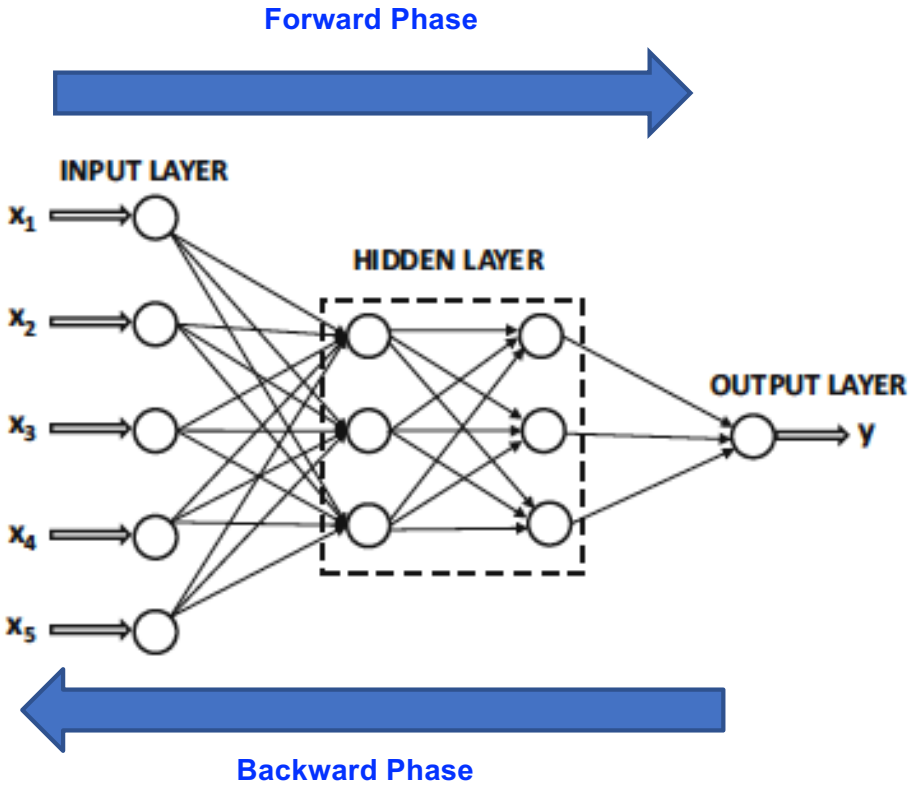


DL Inference Throughput



Deep Learning Training

- Forward phase
- Loss calculation
- Backward phase
- Weight update



Deep Learning Training Steps

- Forward phase:
  - compute the activations of the hidden units based on the current value of weights
  - calculate output
  - calculate loss function
- Backward phase:
  - compute partial derivative of loss function w.r.t. all the weights;
  - use *backpropagation algorithm* to calculate the partial derivatives recursively
  - backpropagation changes the weights (and biases) in a network to decrease the loss
- Update the weights using gradient descent

Stochastic Gradient Descent (SGD)

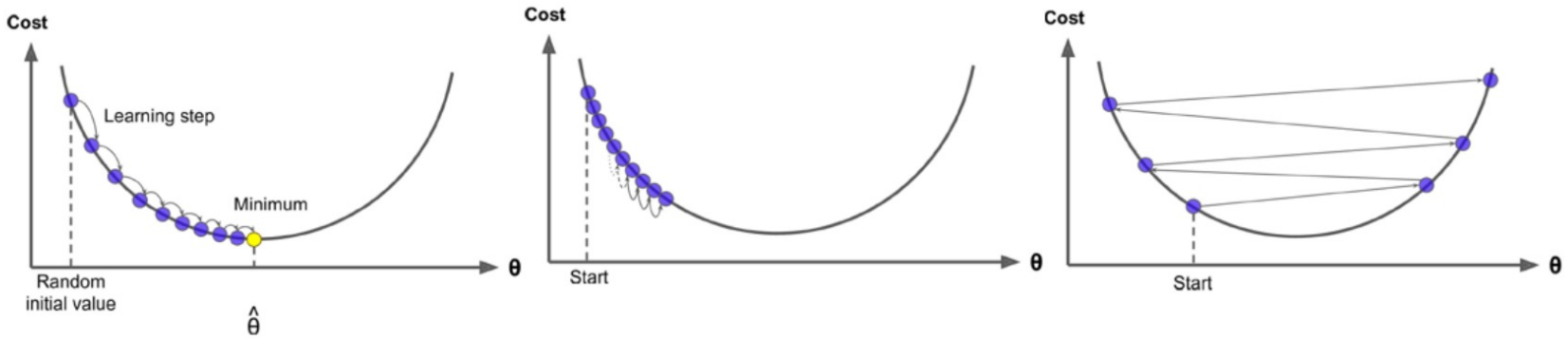
$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}}$$

- Loss is calculated using one training data at each weight update
- Stochastic gradient descent is only a randomized approximation of the true loss function.

Hyperparameters in Deep Learning

- Network architecture: number of hidden layers, number of hidden units per later
- Activation functions
- Weight initializer
- Learning rate
- Batch size
- Momentum
- Optimizer

Learning rate: large value vs small value



Optimum learning rate : The model adjusts weights (θ) in subsequent training loops to arrive at cost minima.

Slow learning rate : Converges to cost minima but very slowly.

*Fast learning rate : **may** not converge to cost minima and the cost might keep increasing with further training loops.*

Image Credit : "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Batch size

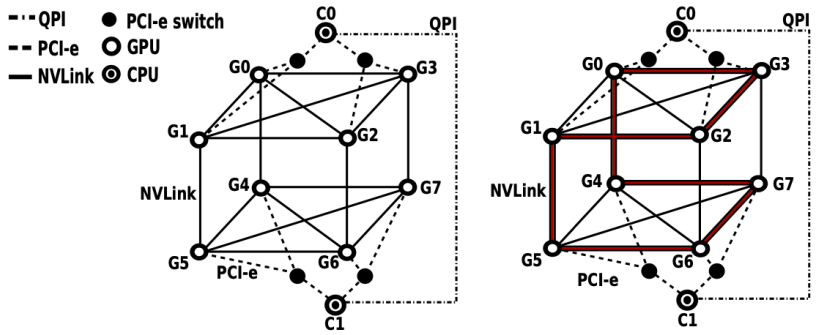
- Effect of batch size on learning
- Batch size is restricted by the GPU memory (12GB for K40, 16GB for P100 and V100) and the model size
  - Model and batch of data needs to remain in GPU memory for one iteration
- Batch size tradeoffs
- Hardware constraints (GPU memory) dictate the largest batch size
- Should we try to work with the largest possible batch size ?
  - Large batch size gives more confidence in gradient estimation
  - Large batch size allows you to work with higher learning rates, faster convergence
- Large batch size leads to poor generalization
  - Lands on sharp minima whereas small batch SGD find flat minima which generalize better
  - Small batches introduce stochastic noise into gradient estimates, which acts as implicit regularization by helping the optimizer escape sharp local minima

Single Node, Single GPU Training

- Training throughput depends on:
  - Neural network model (activations, parameters, compute operations)
  - Batch size
  - Compute hardware: GPU type (e.g., Nvidia M60, K80, P100, V100)
  - Floating point precision (FP32 vs FP16)
    - Using FP16 can reduce training times and enable larger batch sizes/models without significantly impacting the accuracy of the trained model
- Increasing batch size increases throughput
  - Batch size is restricted by GPU memory
- Training time with single GPU very large: 6 days with Places dataset (2.5M images) using Alexnet on a single K40.
- Small batch size => noisier approximation of the gradient => lower learning rate => slower convergence

Multi-GPU Execution Scaling

- Vertical scaling-up in a single node
  - NVIDIA DGX-1 (8 P100 GPUs) and DGX-2 (16 V100 GPUs) servers



(A) P100-based DGX-1 with NVLink-V1 (B) V100-based DGX-1 with NVLink-V2

Fig. 1: PCIe and NVLink-V1/V2 topology for P100-DGX-1 and V100-DGX-1.

- Horizontal scaling-out across multiple nodes
  - Example: GPU accelerated supercomputers like Summit and Sierra from US Department of Energy

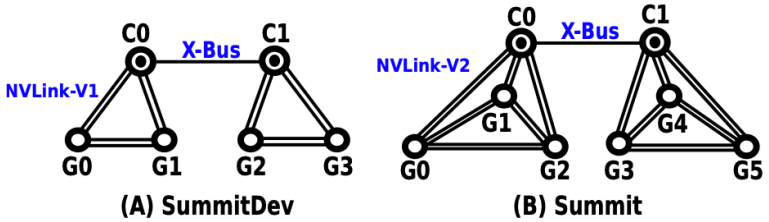


Fig. 2: NVLink interconnect topology for SummitDev and Summit.

Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect

Single Node, Multi GPU Training

- Communication libraries (e.g., NCCL) and supported communication algorithms/collectives (broadcast, all-reduce, gather)
  - NCCL ("Nickel") is library of accelerated collectives that is easily integrated and topology-aware so as to improve the scalability of multi-GPU applications
- Communication link bandwidth: PCIe/QPI or NVlink
- Communication algorithms depend on the communication topology (ring, hub-spoke, fully connected) between the GPUs.

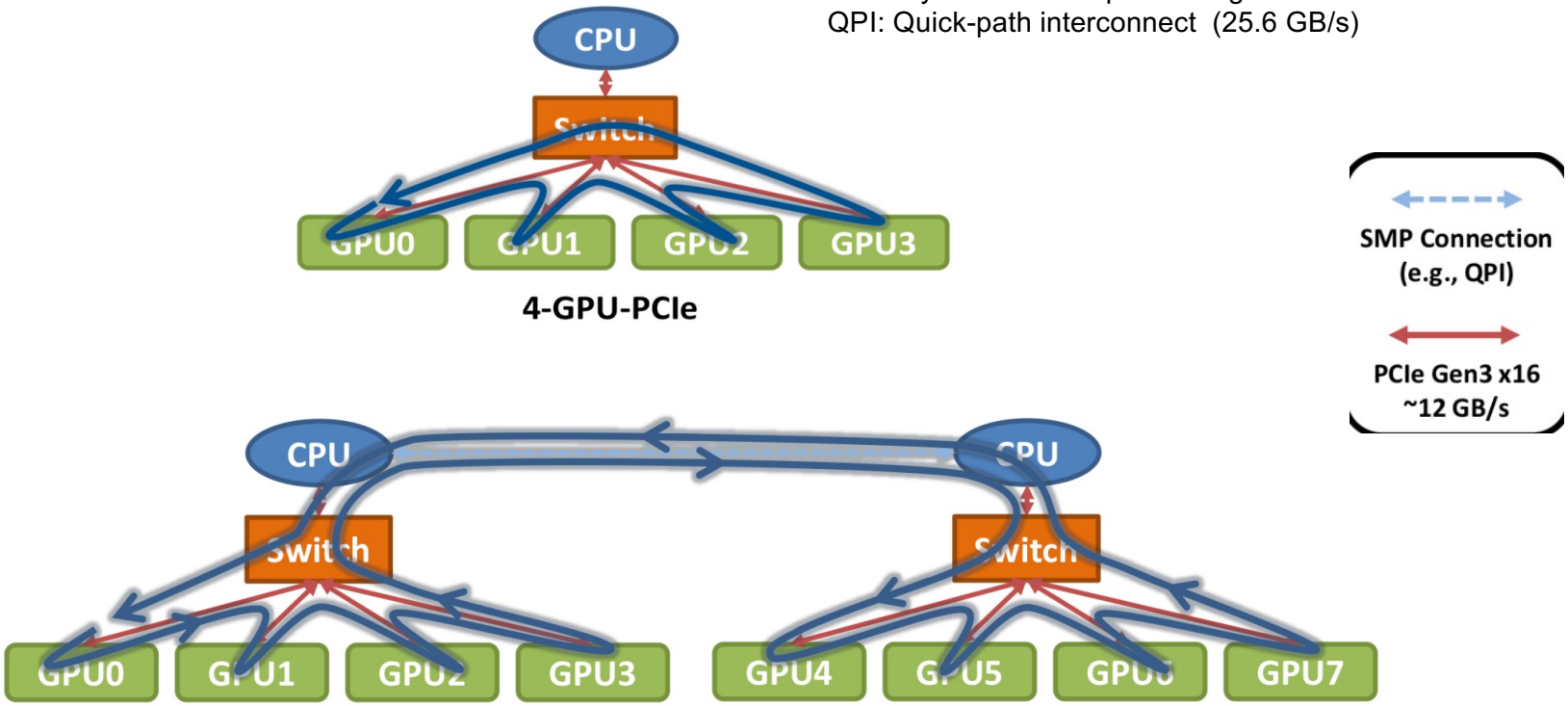
High Performance Networking

- Large Scale Parallel and Deep Learning applications needs:
  - High Bandwidth
  - Low Latency
- Ethernet is not enough
- Infiniband (IB) is widely adopted
- Custom Networks are the best

Network technology	Bandwidth [Gb/s]	Latency [us]
10GigE	10	4
40GigE	40	4
IB EDR	100	1
NVLink	> 400	0.1-0.2

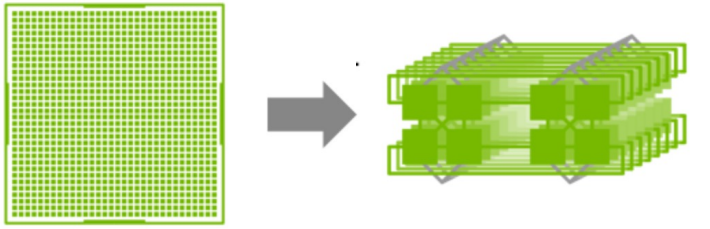
Ring based collectives

PCIe: Peripheral Component Interconnect express
SMP: symmetric multiprocessing
QPI: Quick-path interconnect (25.6 GB/s)



Distributed Training

- Type of Parallelism: Model, Data, Hybrid
- Type of Aggregation: Centralized, decentraliz
- Centralized aggregation: parameter server
- Decentralized aggregation: P2P, all reduce
- Performance metric: Scaling efficiency
  - Defined as the ratio between the run time of one iteration on a single GPU and the run time of one iteration when distributed over n GPUs.



1 GPU

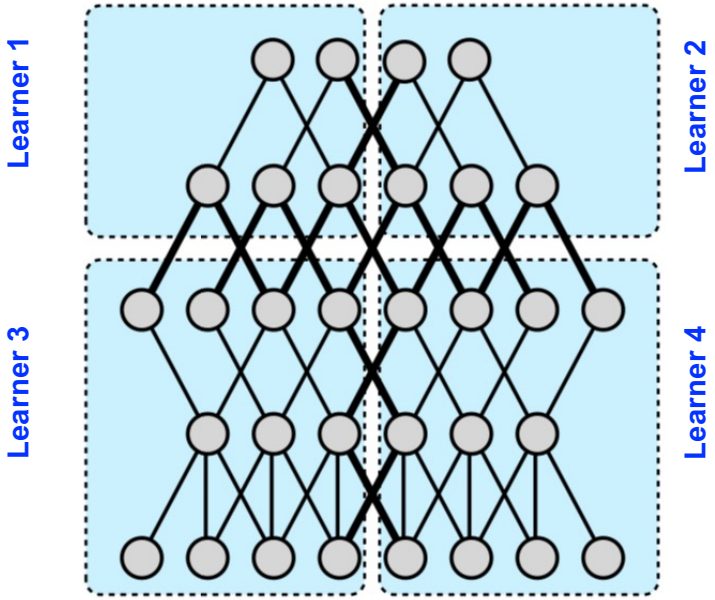
multi-GPU, multi-node

Parallelism

- Parallel execution of a training job on different compute units through scale-up (single node, multiple and faster GPUs) or scale-out (multiple nodes distributed training)
- Enables working with large models by partitioning model across learners
- Enables efficient training with large datasets using large “effective” batch sizes (batch split across learners)
  - Speeds up computation
- Model, Data, Hybrid Parallelism

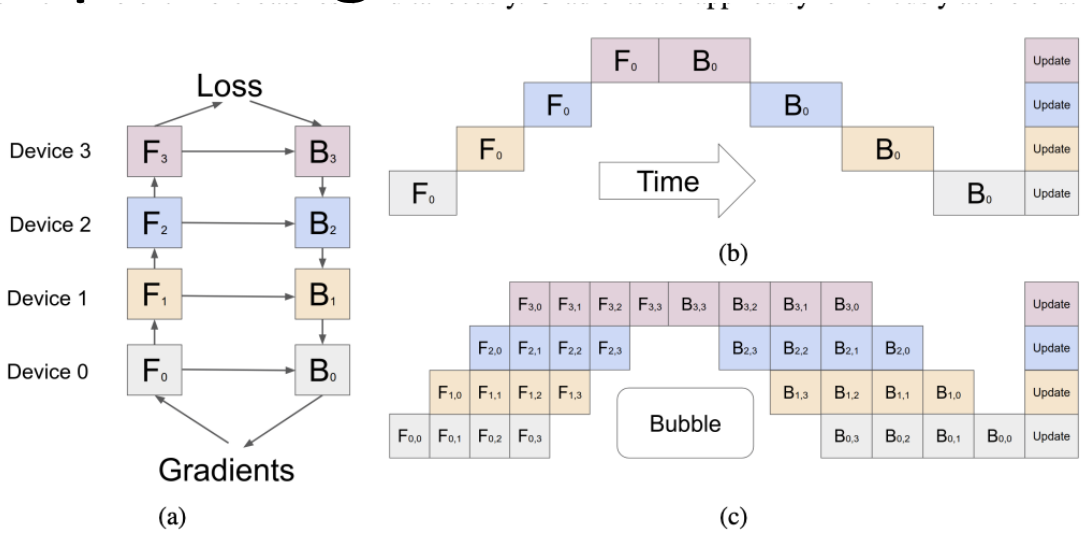
Model Parallelism

- Splitting the model across multiple learners



- 5 layered neural network
- Partitioned across 4 learners
- Bold edges cross learn boundaries and involve inter-learner communication
- Performance benefits depend on
  - Connectivity structure
  - Compute demand of operations
- Heavy compute and local connectivity –benefit most
  - Each machine handles a subset of computation
  - Low network traffic

GPipe Pipelining



- Gpipe: a pipeline parallelism open-source library that allows scaling any network that can be expressed as a sequence of layers.
- Split global batch into multiple micro-batches and injects them into the pipeline concurrently
- Not memory-friendly and will not scale well with large batch. The activations produced by forward tasks have to be kept for all micro-batches until corresponding backward tasks start, thus leads to the memory demand to be proportional (O(M)) to the number of concurrently scheduled micro-batches (M).

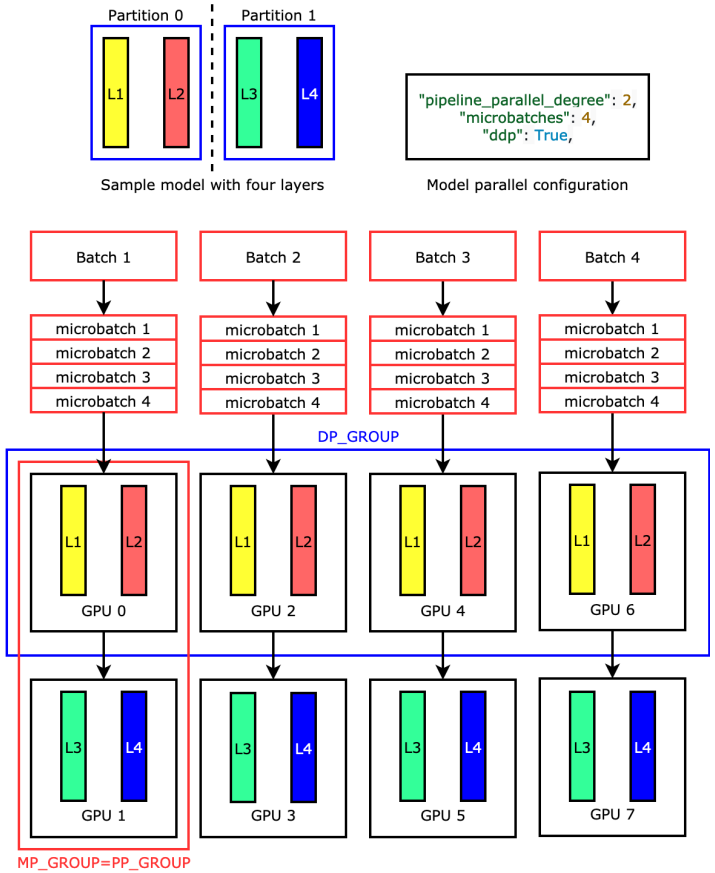
GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism

Model Parallelism and model saving technique in Amazon Sagemaker model parallel library

- Pipeline Parallelism
- Tensor Parallelism
- Optimizer state sharding
- Activation offloading and checkpointing

https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html

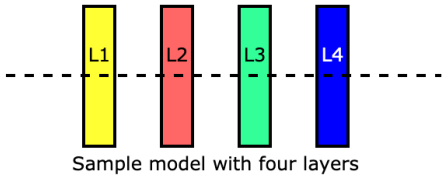
Pipeline Parallelism



ML instance with eight GPU workers

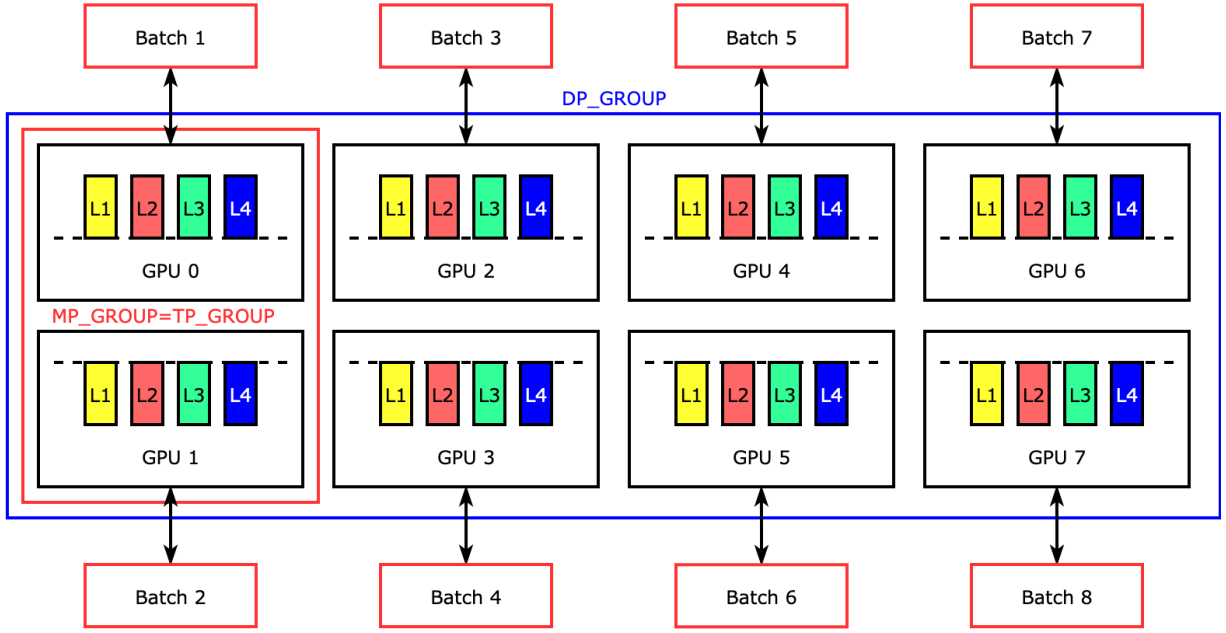
- four-way data parallelism
- two-way pipeline parallelism
- Each model replica is a PP_GROUP (*pipeline parallel group*) and is partitioned across two GPUs
- Each partition of the model is assigned to four GPUs, where the four partition replicas are in a *data parallel group* and labeled as DP_GROUP

Tensor Parallelism



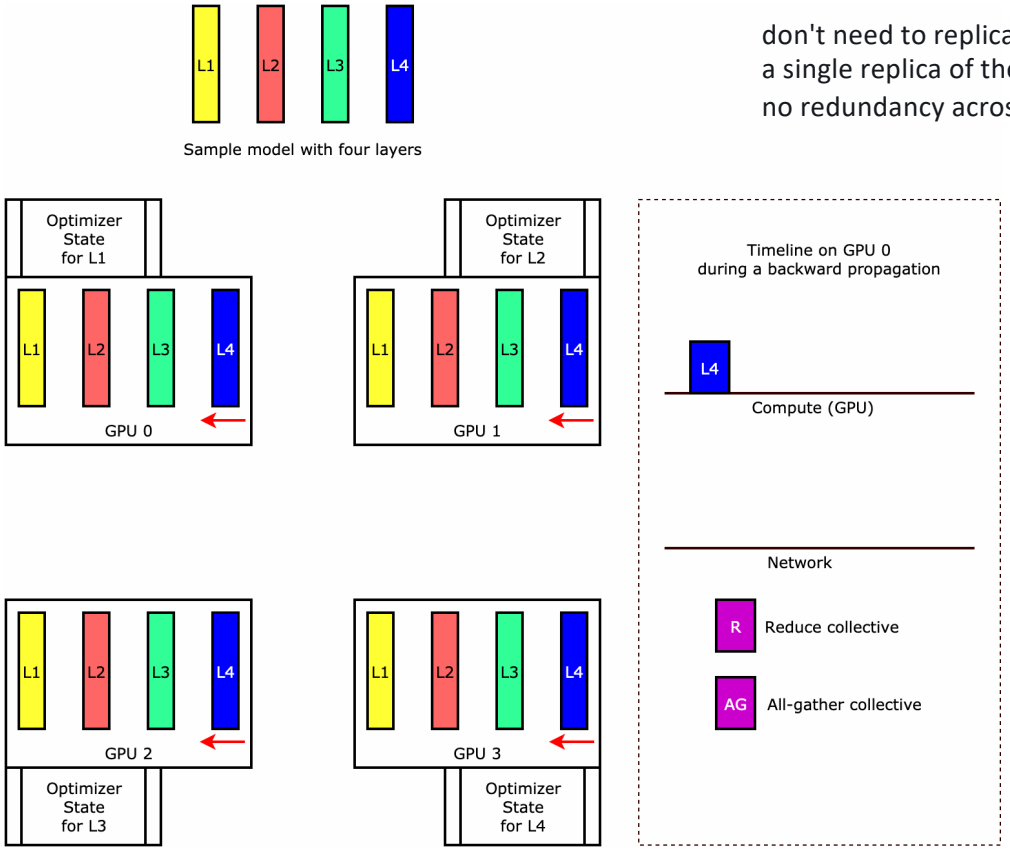
Sample model with four layers

```
"pipeline_parallel_degree": 1,
"tensor_parallel_degree": 2,
"ddp": True
```
Model parallel configuration



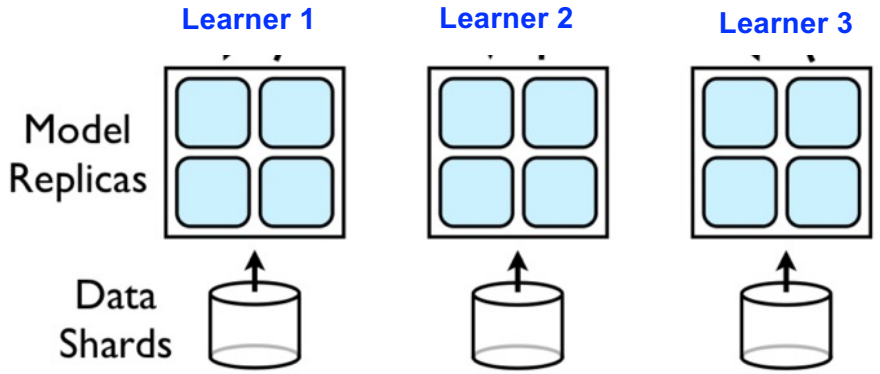
- Splits individual layers, or nn.Modules, across devices, to be run in parallel
- two-way tensor parallelism
- degree of data parallelism is eight

Optimizer state sharding



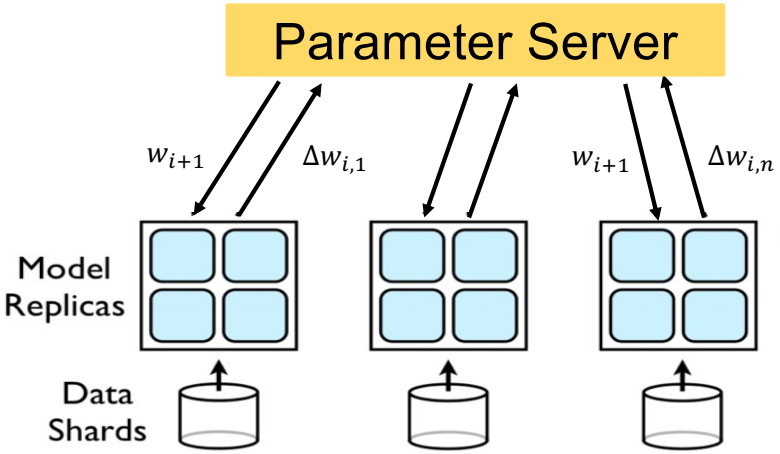
don't need to replicate your optimizer state in all of your GPUs
a single replica of the optimizer state is sharded across data-parallel ranks, with no redundancy across devices

Data Parallelism



- Model is replicated on different learners
- Data is sharded and each learner work on a different partition
- Helps in efficient training with large amount of data
- Parameters (weights, biases, gradients) from different replicas need to be synchronized

Parameter server (PS) based Synchronization

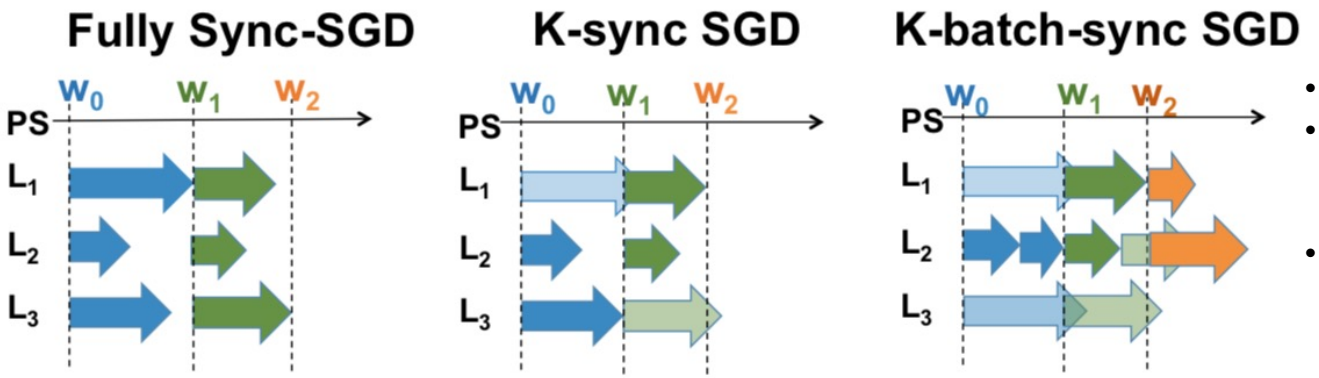


- Each learner executes the entire model
- After each mini-batch processing a learner calculates the gradient and sends it to the parameter server
- The parameter server calculates new value of weights and sends them to the model replicas

Synchronous SGD and the Straggler problem

- PS needs to wait for updated gradients from all the learners before calculating the model parameters
- Even though size of mini-batch processed by each learner is same, updates from different learners may be available at different times at the PS
  - Randomness in compute time at learners
  - Randomness in communication time between learners and PS
- Waiting for slow and straggling learners diminishes the speed-up offered by parallelizing the training

Synchronous SGD Variants



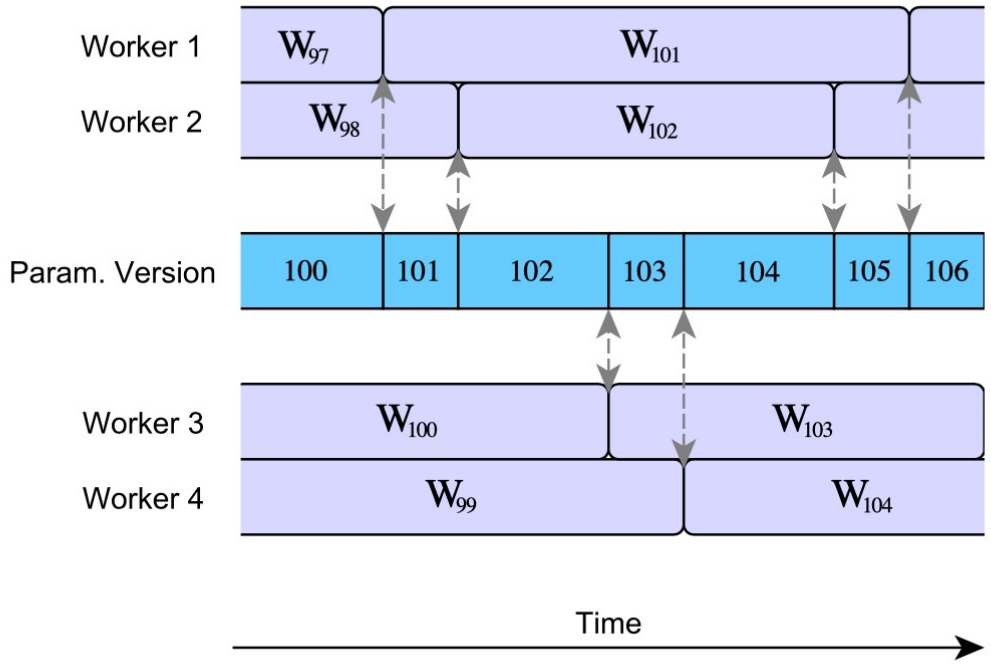
- P: total number of learners
- K: number of learners/mini-batches the PS waits for before updating parameters
- Lightly shaded arrows indicate straggling gradient computations that are canceled.

- K-sync SGD: PS waits for gradients from K *learners* before updating parameters; the remaining learners are canceled
- When K = P , K-sync SGD is same as Fully Sync-SGD
- K-batch sync: PS waits for gradients from K *mini-batches* before updating parameters; **the remaining (unfinished) learners are canceled**
  - Irrespective of which learner the gradients come from
  - Wherever any learner finishes, it pushes its gradient to the PS, fetches current parameter at PS and starts computing gradient on the next mini-batch based on the same local value of the parameters
- Runtime per iteration reduces with K-batch sync; error convergence is same as K-sync

Asynchronous SGD and Stale Gradients

- PS updates happen without waiting for all learners
- Weights that a learner uses to evaluate gradients may be old values of the parameters at PS
  - Parameter server asynchronously updates weights
  - By the time learner gets back to the PS to submit gradient, the weights may have already been updated at the PS (by other learners)
  - Gradients returned by this learner are stale (i.e., were evaluated at an older version of the model)
- Stale gradients can make SGD unstable, slowdown convergence, cause sub-optimal convergence (compared to Sync-SGD)

Stale Gradient Problem in Async-SGD



$$\vdots$$

$$W_{101} = W_{100} - \lambda \Delta W_{97}$$

$$W_{102} = W_{101} - \lambda \Delta W_{98}$$

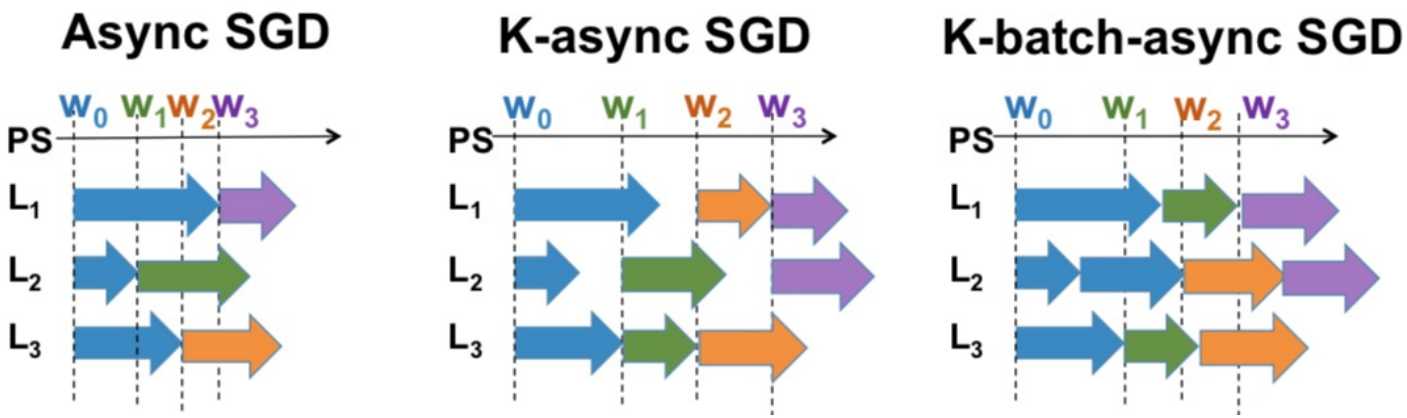
$$W_{103} = W_{102} - \lambda \Delta W_{100}$$

$$W_{104} = W_{103} - \lambda \Delta W_{99}$$

$$\vdots$$

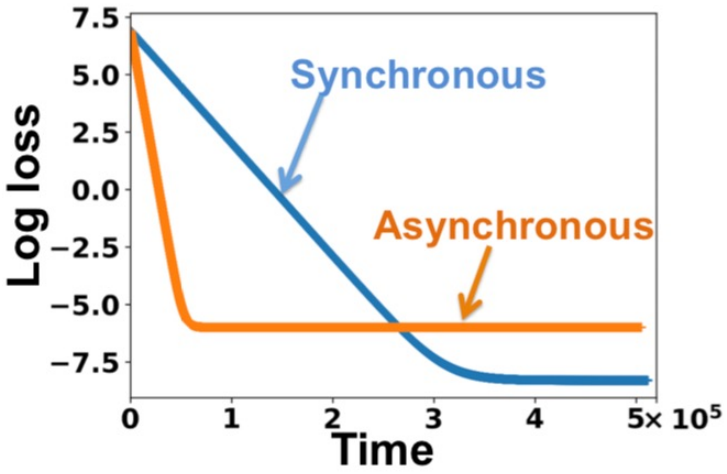
blog.skymind.ai Distrbuted Deep Learning, Part 1: An Introduction to Distributed Training of Neural Networks

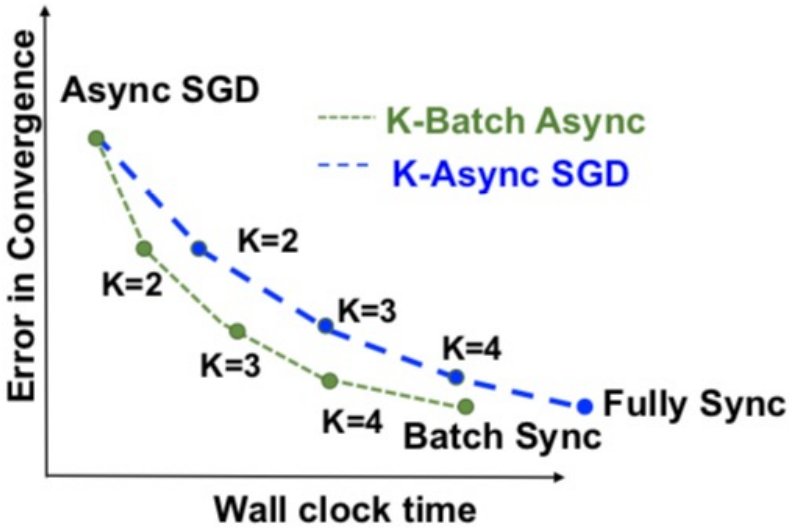
Asynchronous SGD and Variants



- K-async SGD: PS waits for gradients from K *learners* before updating parameters but the remaining learners are not canceled; each learner may be giving a gradient calculated at stale version of the parameters
- When K = 1 , K-async SGD is same as Async-SGD
- K-batch async: PS waits for gradients from K *mini-batches* before updating parameters; **the remaining learners are not canceled**
  - Wherever any learner finishes, it pushes its gradient to the PS, fetches current parameter at PS and starts computing gradient on the next mini-batch based on the current value of the PS
- Runtime per iteration reduces with K-batch async; error convergence is same as K-async

Convergence-Runtime Tradeoff in SGD Variants





- Error-runtime trade-off for Sync and Async-SGD with same learning rate.
- Async-SGD has faster decay with time but a higher error floor.

Dutta et al. Slow and stale gradients can win the race: error-runtime tradeoffs in distributed SGD. 2018

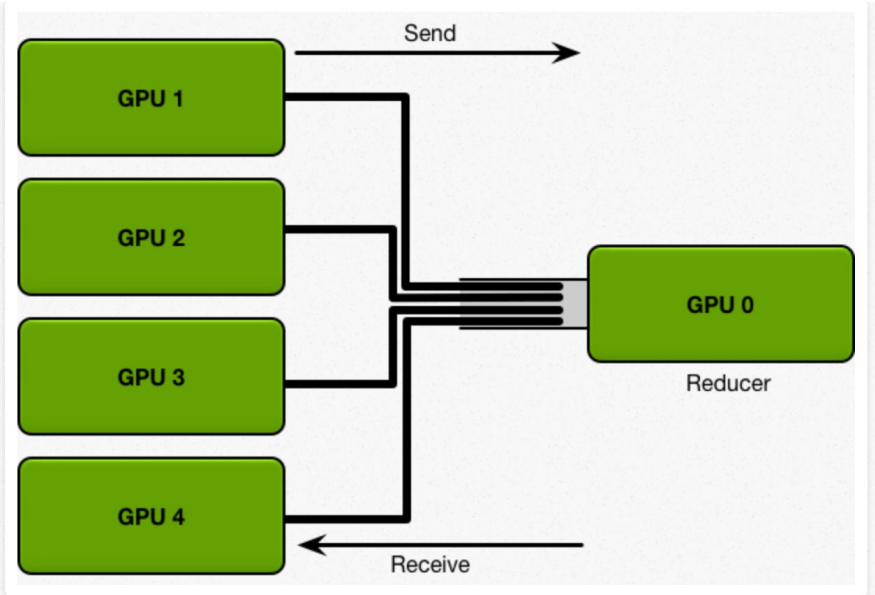
Reduction over Gradients

- To synchronize gradients of N learners, a reduction operation needs to be performed

$$\sum_{j=1}^{N} \Delta w_j$$

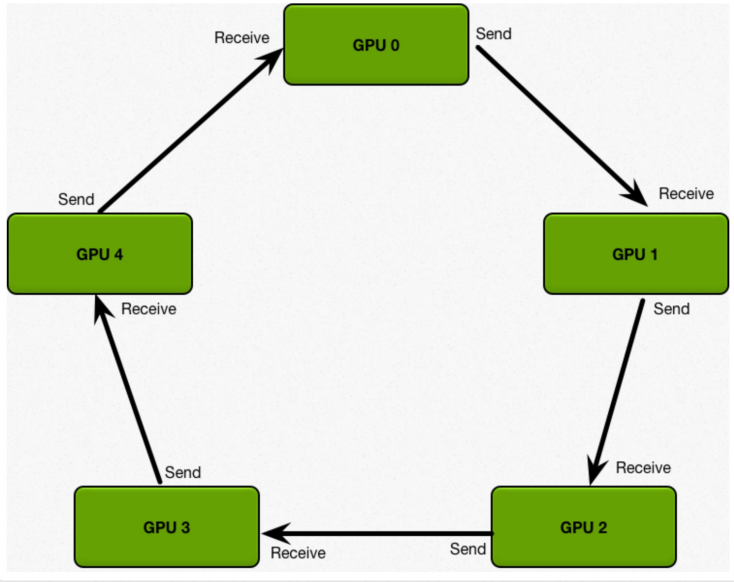
Reduction Topologies

Parameter server: single reducer



SUM (Reduce operation) performed at PS

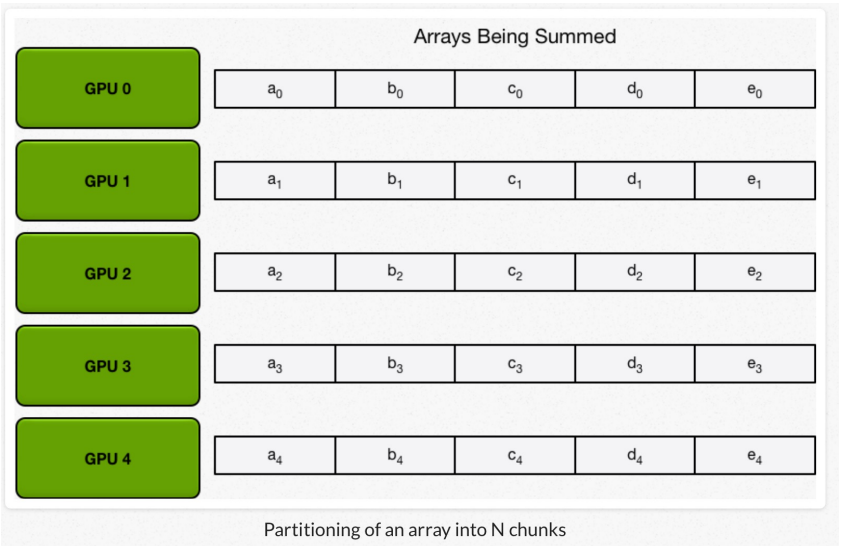
GPUs arranged in a logical Ring (aka bucket) :
all are reducers



SUM (Reduce operation) performed at all nodes

- Each node has a left neighbor and a right neighbor
- Node only sends data to its right neighbor, and only receives data from its left neighbor

All-Reduce

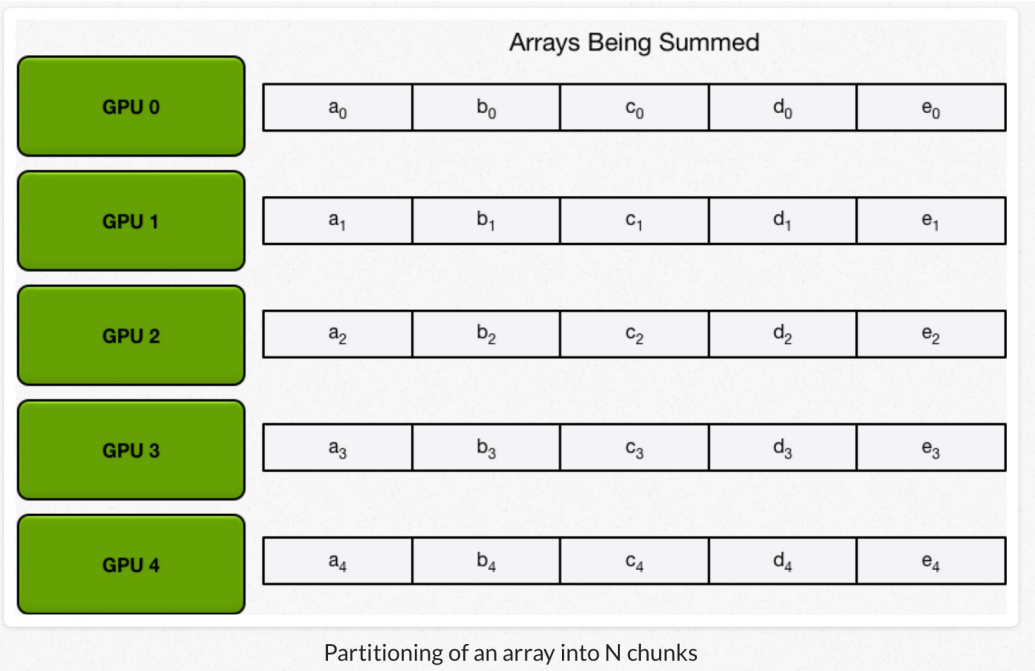


Partitioning of an array into N chunks

Final state after all allgather transfers

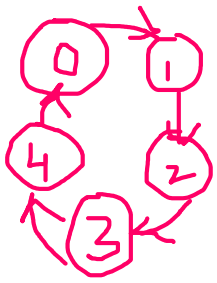
Ring All-Reduce

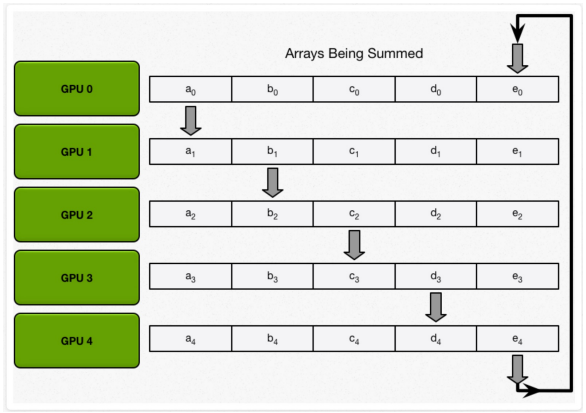
- Two step algorithm:
  - *Scatter-reduce*
    - GPUs exchange data such that every GPU ends up **with a chunk** of the final result
  - *Allgather*
    - GPUs exchange chunks from scatter-reduce such that all GPUs end up with the complete final result.

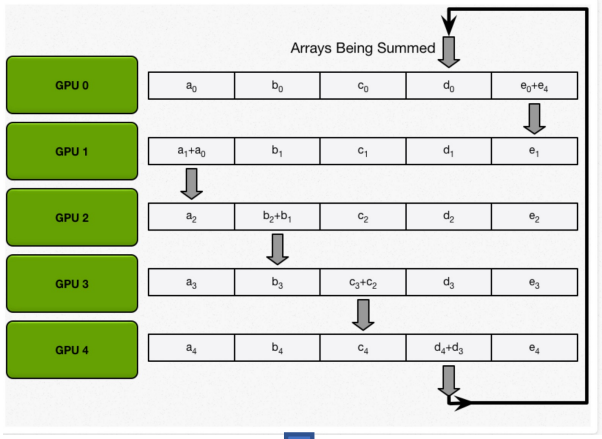


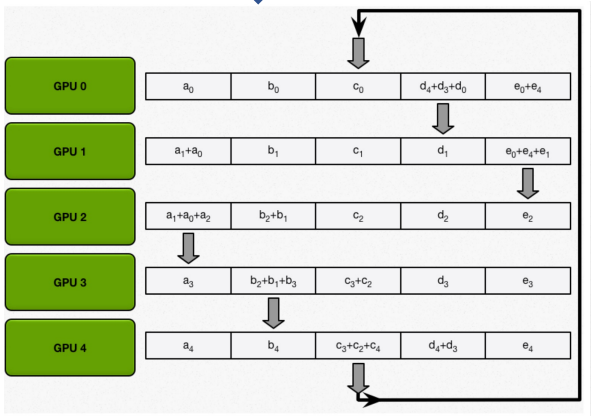
Partitioning of an array into N chunks

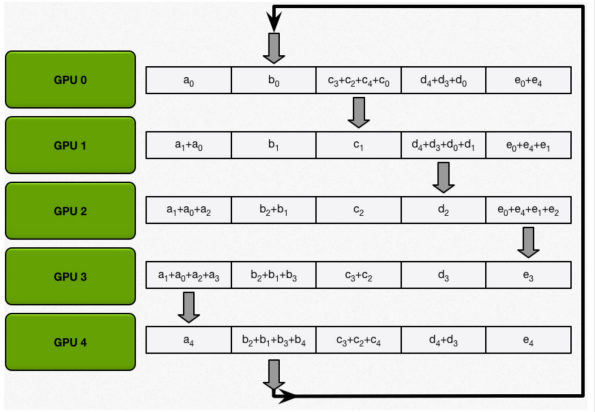
Ring All-Reduce: Scatter-Reduce Step



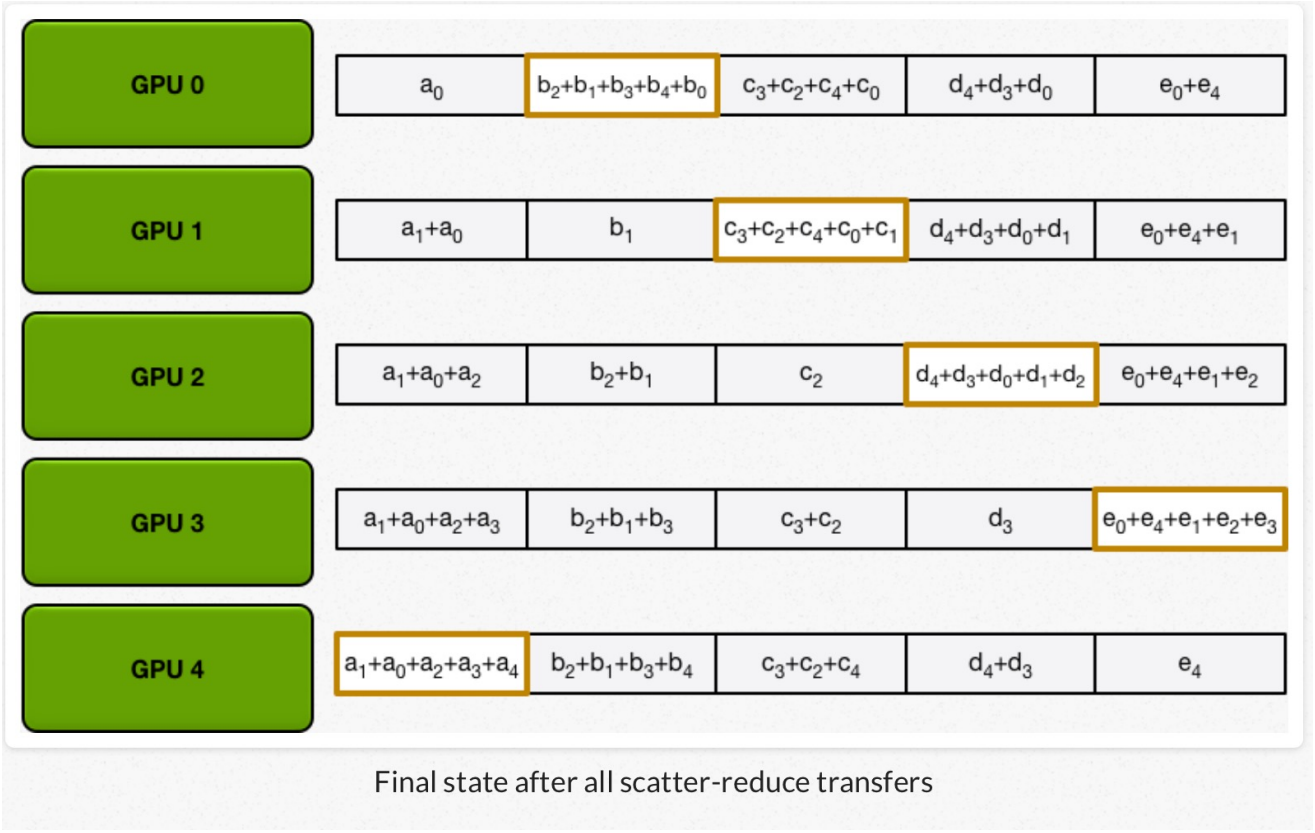








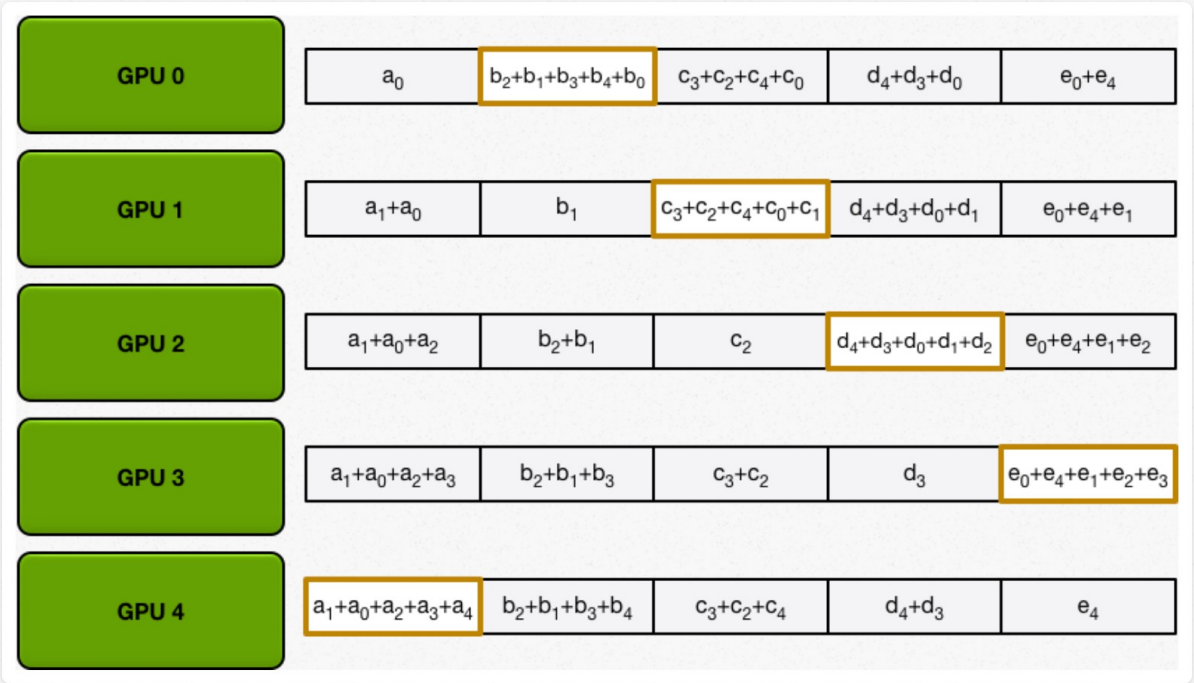
Ring All-Reduce: End of Scatter-Reduce Step



Final state after all scatter-reduce transfers

How many iterations in scatter-reduce step with N GPUs ?

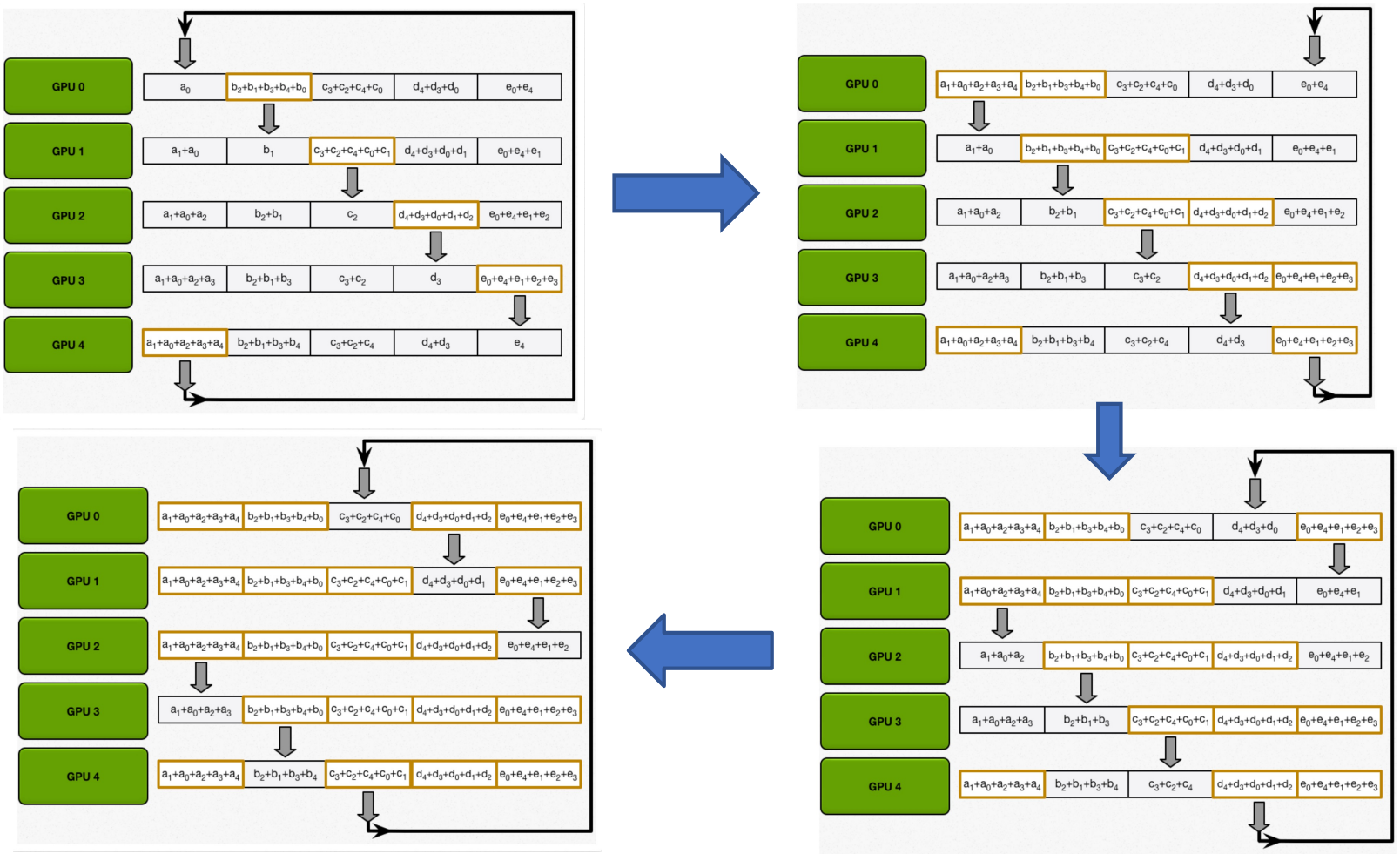
Ring All-Reduce: What to do next ?



Final state after all scatter-reduce transfers

GPU	Send	Receive
0	Chunk 1	Chunk 0
1	Chunk 2	Chunk 1
2	Chunk 3	Chunk 2
3	Chunk 4	Chunk 3
4	Chunk 0	Chunk 4

Ring All-Reduce: AllGather Step



Ring All-Reduce: End of AllGather Step

Final state after all allgather transfers

How many iterations in allgather step with N GPUs ?

Parameter Server (PS) vs Ring All-Reduce: Communication Cost

- P: number of processes N: total number of model parameters
- PS (centralized reduce)
  - Amount of data sent to PS by (P-1) learner processes: N(P-1)
  - After reduce, PS sends back updated parameters to each learner
  - Amount of data sent by PS to learners: N(P-1)
  - Total communication cost at PS process is proportional to 2N(P-1)
- Ring All-Reduce (decentralized reduce)
  - Scatter-reduce: Each process sends N/P amount of data to (P-1) learners
    - Total amount sent (per process): N(P-1)/P
  - AllGather: Each process again sends N/P amount of data to (P-1) learners
  - Total communication cost per process is 2N(P-1)/P
- PS communication cost is proportional to P whereas ring all-reduce cost is practically independent of P for large P (ratio (P-1)/P tends to 1 for large P)
- Which scheme is more bandwidth efficient ?
- Note that both PS and Ring all-reduce involve synchronous parameter updates

All-Reduce applied to Deep Learning

- Backpropagation computes gradients starting from the output layer and moving towards in the input layer
- Gradients for output layers are available earlier than inner layers
- Start all reduce on the output layer parameters while other gradients are being computed
- Overlay of communication and local compute

Distributed Deep Learning Benchmarking Methodology

- Speedup
- Scaling efficiency
- Accuracy and end-to-end training time
- Neural network
- Deep learning framework
- GPU type
- Communication overhead

Speedup (throughput) with n machines = n x Scaling efficiency with n machines

Scaling efficiency

- Scaling efficiency: ratio between the run time of one iteration on a single GPU and the run time of one iteration when distributed over N GPUs. **Why is this ratio a measure of scaling efficiency ?**
- One can satisfy any given scaling efficiency for any neural network by increasing the batch size and reducing communication overhead
- Too big a batch size will result in converging to an unacceptable accuracy or no convergence at all
- A high scaling efficiency without being backed up by convergence to a good accuracy and end to end training time is meaningless

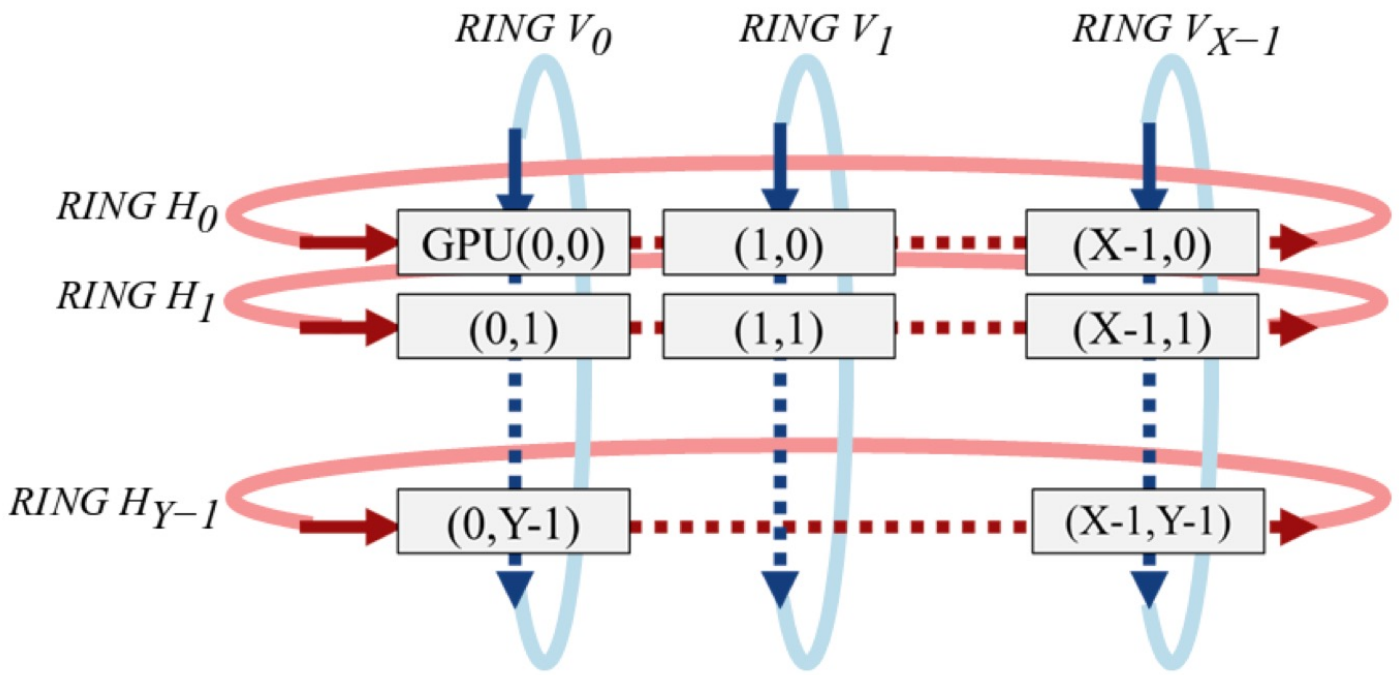
Imagenet1K/ResNet50 Training at Scale

Work	Batch size	Processor	DL Library	Interconnect	Training Time	Top-1 Accuracy	Scaling Efficiency
He et al	256	Tesla P100 x8	Caffe		29 hrs	75.3%	
Goyal et al (Facebook)	8K	Tesla P100 x256	Caffe2	50 Gbit Ethernet	60 mins	76.3%	~90%
Cho et al (IBM)	8K	Tesla P100 x256	Caffe	Infiniband	50 mins	75.01%	95%
Smith et al	8K → 16K	Full TPU Pod	Tensorflow		30 mins	76.1%	
Akiba et al	32K	Tesla P100 x1024	Chainer	Infiniband FDR	15 mins	74.9%	80%
Jia et al	64K	Tesla P40 x2048	Tensorflow	100 Gbit Ethernet	6.6 mins	75.8%	87.9%
Ying et al	32K	TPU v3 x1024	Tensorflow		2.2 mins	76.3%	
Ying et al	64K	TPU v3 x1024	Tensorflow		1.8 mins	75.2%	
Mikami et al	54K	Tesla V100 x3456	NNL	Infiniband EDR x2	2.0 mins	75.29%	84.75%

Cho et al achieved highest scaling efficiency; Goyal et al and Ying et al achieved highest accuracy

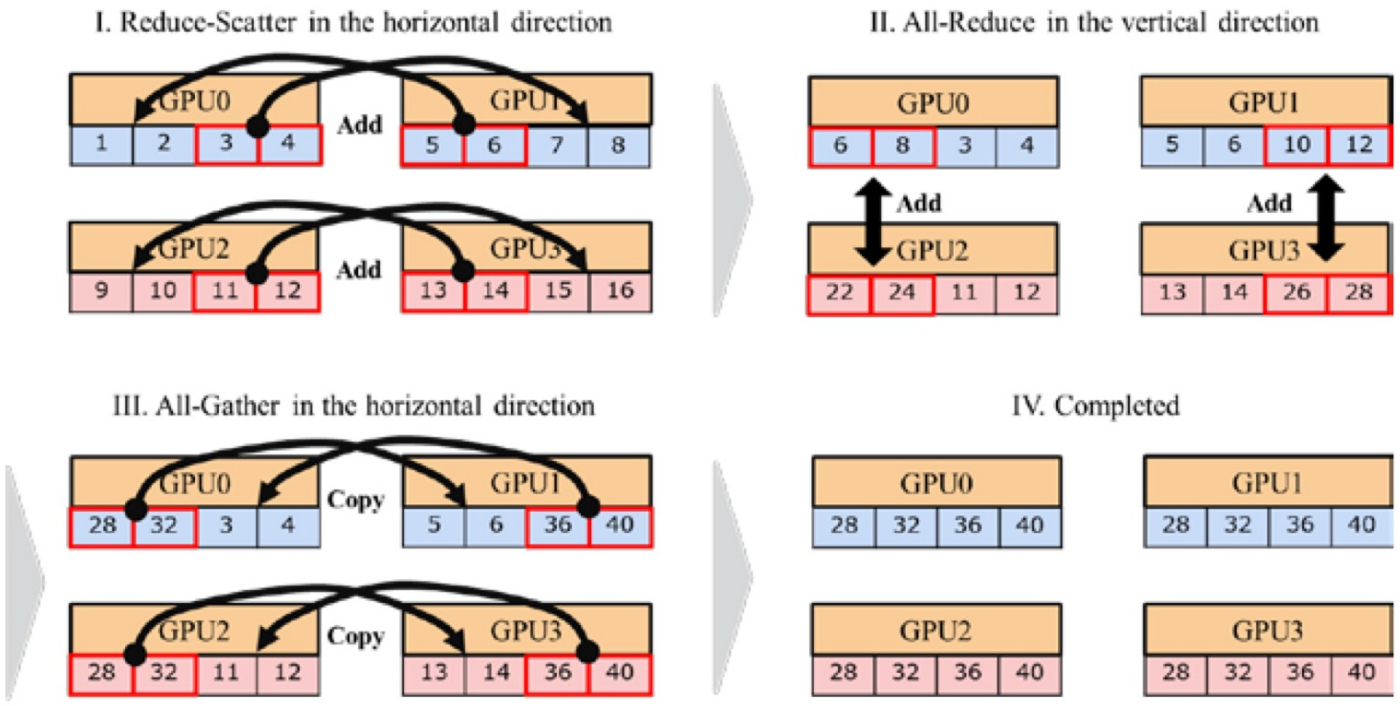
2-D Torus Topology for inter-gpu communication

multiple rings in horizontal and vertical orientations.



2-D Torus all-reduce

2D-Torus all-reduce steps of a 4-GPU cluster, arranged in 2x2 grid

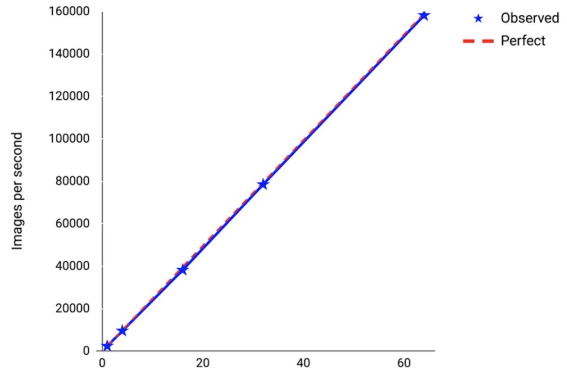


https://arxiv.org/pdf/1811.05233.pdf

Tensor Processing Units (TPUs)

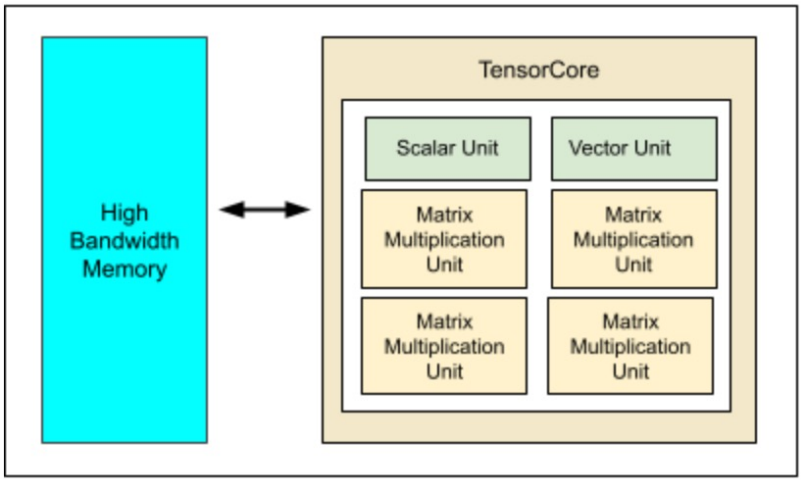
- TPU: application specific integrated circuits to accelerate machine learning workload
- TPU pod: multiple TPU chips connected to each other over a dedicated high-speed network connection
- https://cloud.google.com/tpu/docs/system-architecture

TPU v2 pod for ResNet-50: linearly scalable



Number of TPU v2 chips

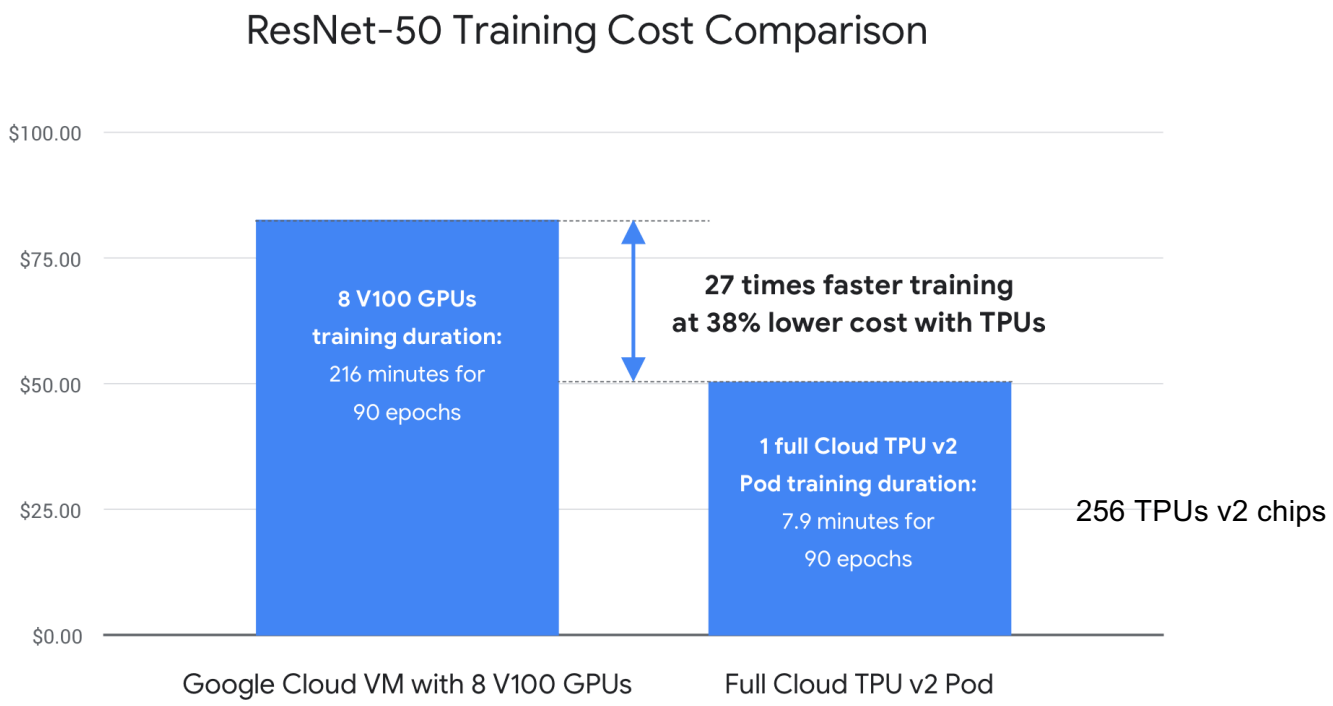
TPU v5e chip



A TensorCore has four matrix-multiply units (MXUs), a vector unit, and a scalar unit

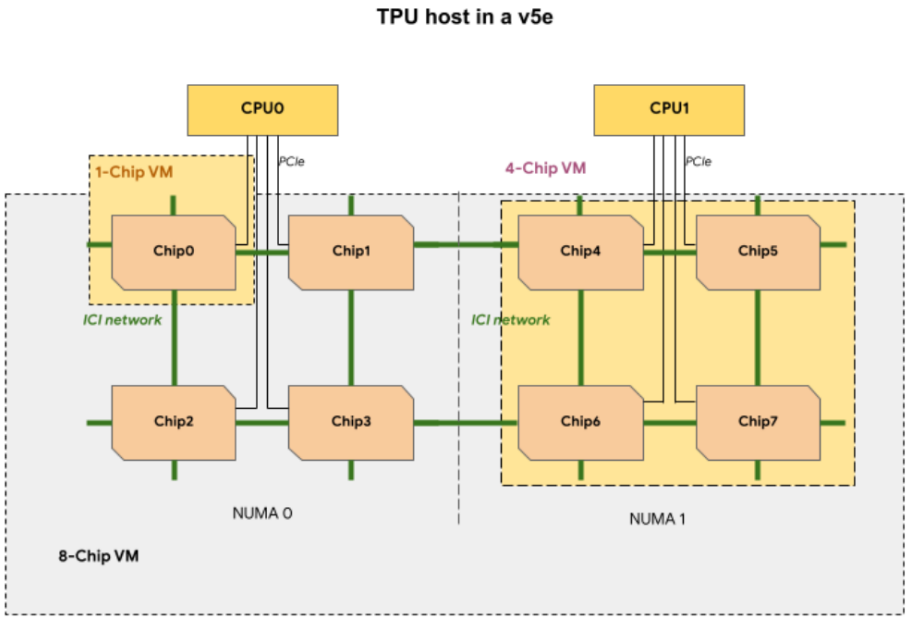
https://cloud.google.com/tpu/docs/v5e

TPUs vs GPUs Performance



TPU VM on Google Cloud

- A TPU host is a VM that runs on a physical computer connected to TPU hardware. TPU workloads can use one or more host.
- Tutorial on using Google Cloud TPU VM: Google Cloud Quickstart



NCCL

- Nvidia Common Communications Library
- NCCL implements optimized multi-GPU and multi-node communication primitives for NVIDIA gpus and networking
- NCCL provides routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter as well as point-to-point send and receive
- Communication primitive are optimized to achieve high bandwidth and low latency over PCIe and NVLink high-speed interconnects within a node and over NVIDIA Mellanox Network across nodes

Prepare for Lecture 2

- Access to compute cluster
  - Set up you NYU HPC access; instructions coming soon
- First home-work posted on 09/12

Backup material

Rapid evolution of new models

- GPT-4o : text, audio, images
- Claude 3.5 Sonnet
- LLaMa 3
- Dalle 2
- Gemini 1.5

AI at US Open

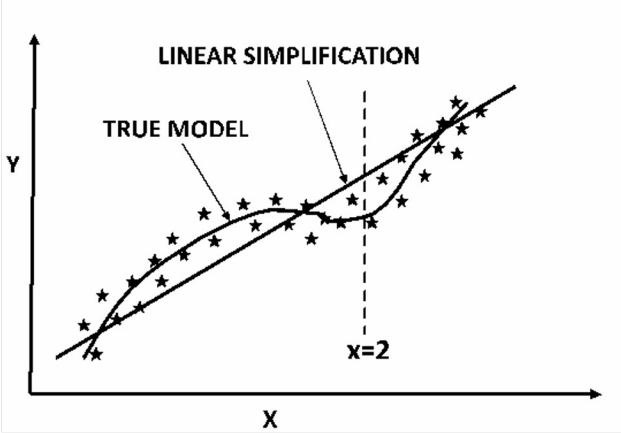
- https://www.ibm.com/case-studies/us-open
- Generative AI models for generating content
  - IBM watsonX AI and data platform built for business
  - IBM Granite foundation models
- watsonX.data: to connect and curate the USTA's trusted data sources
- Foundation models were trained to translate tennis data into cogent descriptions
  - summarizing entire matches in the case of Match Reports
  - generating sentences that describe the action in highlight reels for AI Commentary

"Foundation models are incredibly powerful and are ushering in a new age of generative AI, But to generate meaningful business outcomes, they need to be trained on high-quality data and develop domain expertise. And that's why an organization's proprietary data is the key differentiator when it comes to AI."

Shannon Miller, IBM Consulting

Bias of a model

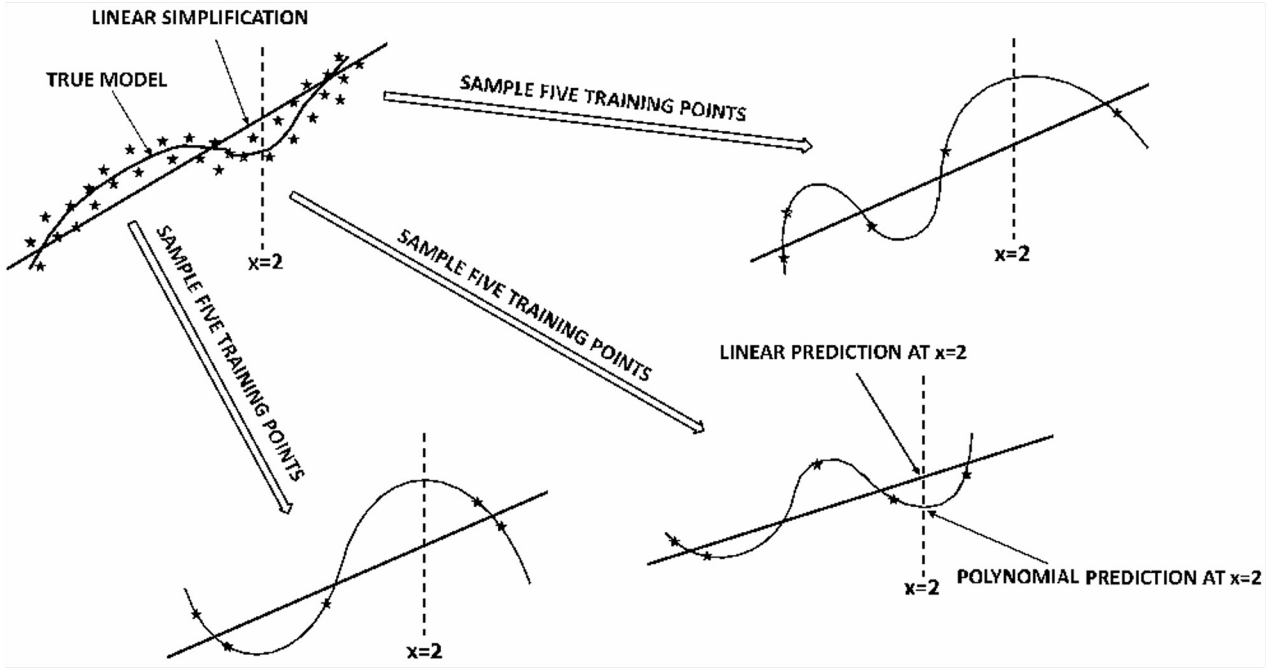
Example: Predict y from x



- **First impression:** Polynomial model such as $y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$ is "better" than linear model $y = w_0 + w_1 x$.

–

Different Training Data Sets with Five Points



- Zero error on training data but wildly varying predictions of $x = 2$

Observations

- The higher-order model is more complex than the linear model and has less *bias*.
  - But it has more parameters.
  - For a small training data set, the learned parameters will be more sensitive to the nuances of that data set.
  - Different training data sets will provide different predictions for y at a particular x.
  - This variation is referred to as model *variance*.
- Neural networks are inherently low-bias and high-variance learners $\Rightarrow$ Need ways of handling complexity.

Bias-Variance Trade-off: Setup

- Imagine you are given the true distribution $\mathcal{B}$ of training data (including labels).
- You have a principled way of sampling data sets $\mathcal{D} \sim \mathcal{B}$ from the training distribution.
- Imagine you create an infinite number of training data sets (and trained models) by repeated sampling.
- You have a *fixed* set $\mathcal{T}$ of unlabeled test instances.
  - The test set $\mathcal{T}$ does not change over different training data sets.
  - Compute prediction of each instance in $\mathcal{T}$ for each trained model.

Informal Definition of Bias

- Compute averaged prediction of each test instance x over different training models $g(x, \mathcal{D})$.

- Averaged prediction of test instance will be different from true (unknown) model $f(x)$.

- Difference between (averaged) $g(x, \mathcal{D})$ and $f(x)$ caused by erroneous assumptions/simplifications in modeling $\Rightarrow$ Bias

  - **Example:** Linear simplification to polynomial model causes bias.

  - If the true (unknown) model $f(x)$ were an order-4 polynomial, and we used any polynomial of order-4 or greater in $g(x, \mathcal{D})$, bias would be 0.

Informal Definition of Variance

- The value $g(x, \mathcal{D})$ will vary with $\mathcal{D}$ for fixed x.
  - The prediction of the same test instance will be different over different trained models.
- All these predictions cannot be simultaneously correct $\Rightarrow$ Variation contributes to error
- Variance of $g(x, \mathcal{D})$ over different training data sets $\Rightarrow$ Model Variance
  - **Example:** Linear model will have low variance.
  - Higher-order model will have high variance.

Bias-Variance Equation

- Let $E[MSE]$ be the expected mean-squared error of the fixed set of test instances over different samples of training data sets.

$$E[MSE] = \text{Bias}^2 + \text{Variance} + \text{Noise} \quad (1)$$

  - In linear models, the bias component will contribute more to $E[MSE]$.
  - In polynomial models, the variance component will contribute more to $E[MSE]$.

- We have a trade-off, when it comes to choosing model complexity!

Learning rate and Batch size relationship

- "Noise scale" in stochastic gradient descent (Smith et al 2017)

$$g = \epsilon\left(\frac{N}{B} - 1\right)$$

$$g \approx \frac{\epsilon N}{B} \quad as\ B \ll N$$

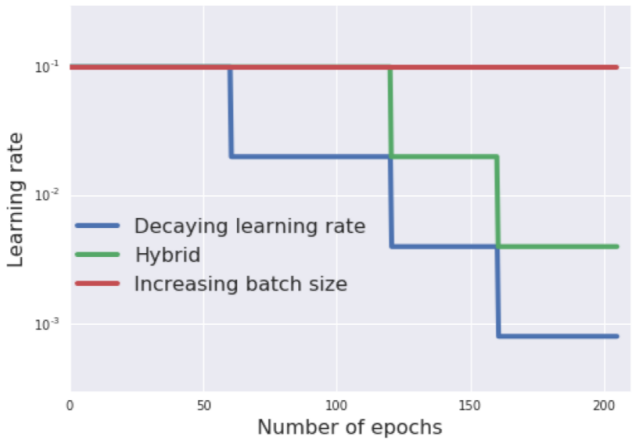
N: training dataset size

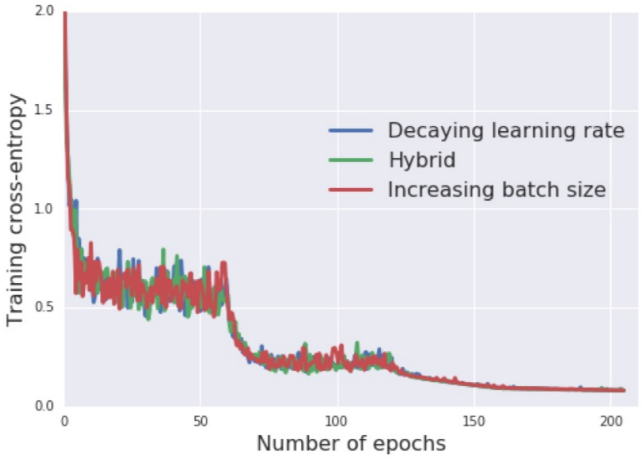
B: batch size

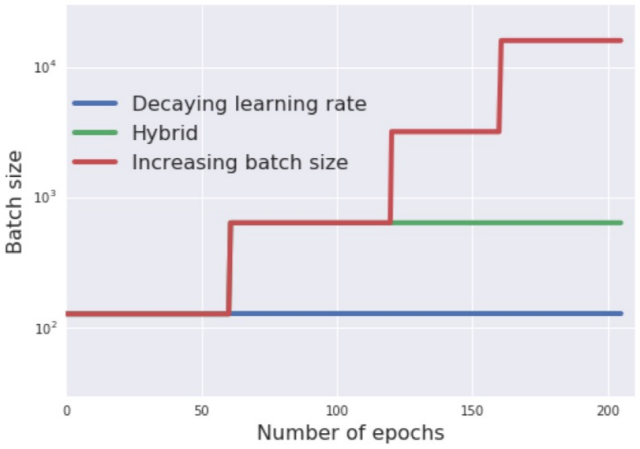
ϵ: learning rate

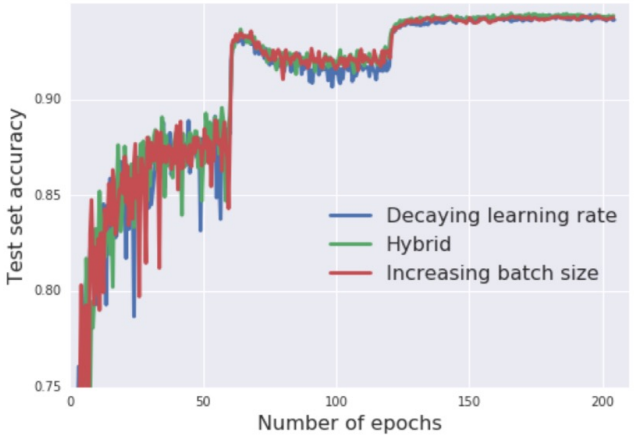
- There is an optimum noise scale g which maximizes the test set accuracy (at constant learning rate)
  - Introduces an optimal batch size proportional to the learning rate when B ≪ N
- Increasing batch size will have the same effect as decreasing learning rate
  - Achieves near-identical model performance on the test set with the same number of training epochs but significantly fewer parameter updates

Learning rate decrease Vs Batch size increase









Batch normalization

- Internal covariance shift – change in the distribution of network activations due to change in network parameters during training
- Idea is to reduce internal covariance shift by applying normalization to inputs of each layer
- Achieve fix distribution of inputs at each layer
- Normalization *for each training mini-batch*.
- Batch normalization enables training with larger learning rates
  - Reduces the dependence of gradients on the scale of the parameters
  - Faster convergence and better generalization

Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Why this step ?

Best practices when benchmarking distributed deep learning systems

- Systems under comparison should train to same accuracy
- Accuracy should be reported on sufficiently large test set
- Compute to communication ratio can vary widely for different neural networks. Using a neural network with high compute to communication ratio can hide the ills of an inferior distributed Deep Learning system.
  - a sub-optimal communication algorithm or low bandwidth interconnect will not matter that much
- Computation time for one Deep Learning iteration can vary by up to 50% when different Deep Learning frameworks are being used. This increases the compute to communication ratio and gives the inferior distributed Deep Learning system an unfair uplift to the scaling efficiency.

Best practices when benchmarking distributed deep learning systems

- A slower GPU increases the compute to communication ratio and again gives the inferior distributed Deep Learning system an unfair uplift to the scaling efficiency.
  - Nvidia P100 GPUs are approximately 3X faster than Nvidia K40 GPUs.
  - When evaluating the communication algorithm and the interconnect capability of a Deep Learning system, it is important to use a high performance GPU.
- Communication overhead is the run time of one iteration when distributed over N GPUs minus the run time of one iteration on a single GPU.
  - Includes the communication latency and the time it takes to send the message (gradients) among the GPUs.
  - Communication overhead gives an indication of the quality of the communication algorithm and the interconnect bandwidth.

DL Pipelining

- Pipelining approach:
  - Split layers among compute engines
  - Each minibatch *b (or sample s)* goes from one compute engine to the next one: *no need to wait for next one to exit the pipeline*
- Is a form of **Model Parallelism**
- Pipelining performance
  - Ideal pipelining speedup *(number of pipeline stages)*

$$S\_time(stage\ time) = \frac{time\ without\ pipeline}{number\ of\ pipeline\ stages}$$

  - Speedup is higher for deeper networks
  - Ideal pipelining never reached because of "bubbles" that cause idle CPUs
  - SGD pipeline bubble:
    - Before weights update, all batches need to have completed forward (otherwise accept **staleness**)