ECE-GY 9143

Introduction to High Performance Machine Learning

Lecture 12

Parijat Dube

Performance Modeling and Benchmarking of DNNs

Why Performance modeling ?

- Helps in studying sensitivity of system performance to different workload related parameters (algorithmic and platform)
- Study tradeoffs between different choices (compute units, hyperparameters etc.)
- Early estimate for making correct design choices
- Helps identify bottlenecks and what-if analysis

Performance Modeling Techniques

- White Box modeling
  - Model the internal flow and dynamics of the system
  - Analytical models (equation based) : computation graph, queueing models
  - Make assumptions for tractability which might not be true in practice
  - Easier to understand
- Black box modeling
  - Model the system as a black box by only observing the input and system's response
  - Machine learning models
  - Linear and non-linear regression models
  - Lacks explainablity
- Grey box modeling
  - In between white and black box in terms of modeling of system internals
  - Partial observability of system

Performance Estimation Challenges

- Both algorithmic and system related challenges
- Performance depends on different software and hardware factors
  - Interdependent, e.g., batch size governed by GPU type, network type
  - Distributed training performance tied to learner communication topology
- DL workloads are domain specific and lack standard representations.
  - Convolutional (computer vision) vs Recurrent neural networks (NLP)
  - User specified format in Tensorflow, Pytorch, and Keras
  - Space of DL frameworks and hardware resources is continuously evolving
- DL on cloud:
  - Cloud providers support different frameworks and hardware resources
  - Need a generic methodology to works across different cloud-based DL platform services

DL Resource Requirement on Mobile Platforms

- Resource requirement of the forward path of CNNs
- Deployed several Caffe based CNN models mobile platforms
- NVIDIA TK1 and TX1 developer kits
- TX1 is more powerful than TK1 (256 cuda cores Maxwell vs 192 cuda cores Kepler)

compute bound

memory bound

Table 2: Timing benchmarks on AlexNet

Platform		Layerwise Pass (ms)					Total (ms)	Forward Pass (ms)
		CONV	POOL	LRN	ReLU	FC		
TK1	CPU	318.7±0.2	6.1±0.1	103.8±0.0	4.6±0.0	186.3±0.1	619.8±0.2	619.5±0.2
		51.42%	0.99%	16.74%	0.75%	30.05%		
	GPU	24.6±3.5	2.3±0.6	2.4±0.5	5.2±1.2	35.1±5.9	73.3±10.7	54.7±2.4
		33.53%	3.15%	3.22%	7.11%	**47.95%**		
TX1	CPU	66.9±5.3	7.6±0.0	172.4±0.3	2.4±0.0	644.7±5.3	894.3±4.8	892.7±2.3
		7.48%	0.85%	19.28%	0.27%	**72.09%**		
	GPU	24.2±8.3	1.3±2.6	2.7±3.0	5.9±5.9	15.2±4.7	52.8±15.7	29.3±6.5
		45.79%	2.51%	5.12%	11.23%	28.76%		
FLOPs		666M	1M	2M	0.7M	59M	729M	
		91.36%	0.14%	0.27%	0.10%	8.09%		

Lu et al. Modeling the Resource Requirements of Convolutional Neural Networks on Mobile Devices. 2017

Layerwise time addition does not work

Table 6: Timing benchmarks on ResNet

Platform		Layerwise Pass (ms)							Total (ms)	Forward Pass (ms)
		CONV	POOL	BatchNorm	ReLU	Scale	Eltwise	FC		
TK1	CPU	1830.4±0.4	8.8±0.0	97.1±0.1	64.0±0.1	42.0±0.1	24.8±0.1	5.4±0.0	2072.7±0.4	2072.2±0.3
		88.31%	0.42%	4.68%	3.09%	2.03%	1.20%	0.26%		
	GPU	245.8±16.3	5.5±0.6	249.5±11.6	38.7±2.0	76.0±3.3	47.0±2.7	3.9±0.1	673.0±33.4	149.4±4.9
		36.53%	0.81%	**37.08%**	5.75%	11.29%	6.98%	0.58%		
TX1	CPU	362.3±5.4	13.7±0.2	83.5±0.3	33.2±0.1	31.9±3.6	20.4±4.2	22.2±0.1	567.6±7.6	566.8±9.7
		63.83%	2.41%	14.7%	5.86%	5.62%	3.59%	3.92%		
	GPU	279.4±42.6	3.0±2.7	198.1±36.8	63.6±31.3	79.8±24.2	34.8±12.9	1.8±2.4	664.7±116.5	104.4±14.0
		42.03%	0.45%	29.80%	9.57%	12.01%	5.24%	0.27%		
FLOPs		3866M	2M	32M	9M	11M	6M	2M	3922M	
		98.59%	0.05%	0.81%	0.23%	0.27%	0.14%	0.05%		

For GPUs summation of layerwise pass time is much higher than the actual forward pass time. For CPUs layerwise addition is a good estimate.

Lu et al. Modeling the Resource Requirements of Convolutional Neural Networks on Mobile Devices. 2017

Observations

- Overhead of time measurement on GPUs is very high
- CUDA supports asynchronous kernel execution
- For timing measurement explicit synchronization is required
  - Call to cudaDeviceSynchronize API to make sure that all cores have finished their tasks

Deep Learning for Predicting Execution Time

- Predicting the execution time for a deep learning network through the use of deep learning.
  - Capture non-linear dependency between different features derived from the computational resource used, the network being trained, and the data used for training
- Approach:
  - Predict execution time for commonly used layers in deep neural networks
    - Execution time required for one training step (forward and backward pass) for processing an individual batch of data.
    - Overall execution time can then be calculated as the time per batch multiplied by the number of batches

Justus et al. Predicting the Computational Cost of Deep Learning Models. 2018

Features of a Training Job

- Layer Specific Features
  - Multi-Layer Perceptron
    - Number of inputs
    - Number of neurons
  - Convolutional features
    - Input size and depth
    - Kernel size and output depth
    - Stride size and input padding
  - Pooling features
    - Kernel size
    - Stride size
    - Input padding

Features of a Training Job

- Layer Features
  - Activation function
  - Optimizer
  - Batch size
- Hardware Features
  - GPU type
  - GPU count
  - GPU memory
  - GPU clock speed
  - GPU memory bandwidth
  - GPU core count
  - GPU peak performance
  - Card connectivity

Layerwise Performance Modeling

- Benchmark execution time of individual layer operations (atomic operations)
- Use the feature set for an atomic operation along with the execution timings to train a fully connected feed forward network. This network can then be used for predicting the execution time for a new operation based just on the feature set.
- Once a prediction has been made for an individual operation these can be combined together and across layers in order to provide a prediction for the overall performance of the deep learning network.
- Working with atomic operations:
  - Reduces the computational time to train whilst also maximizing the range of layer types that can be predicted

Full Model Prediction

- Per batch time

$$T_b = \sum_{i=0}^{l} b_{M(i)}$$

l is the number of layers in the deep neural network

$b_{M(i)}$ is the batch execution time estimate

$M(i)$ type of layer i

- Execution time of a single epoch

$$E = pT_b$$

p is the number of batches required to process the data

Experimental Setup

Name	Provisioning	Cuda cores	Clock (boost)	Memory	GPU memory bandwidth	Bus
V100	Local	5120	1455 MHz	16 GB HBM2	900 GB/s	NVLink
P100	Cloud	3584	1303 MHz	16 GB HBM2	732 GB/s	PCIe
GTX1080Ti	Local	3584	1582 MHz	11 GB GDDR5X	484 GB/s	PCIe
M60	Cloud	4096	1178 MHz	16 GB GDDR5	320 GB/s	PCIe
K80	Cloud	2496	875 MHz	12 GB GDDR5	240 GB/s	PCIe
K40	Local	2880	875 MHz	12 GB GDDR5	288 GB/s	PCIe

TABLE I
SPECIFICATION OF HARDWARE TESTED

- Training space is very large (~10^9 combinations for fully connected and 10^12 combinations for convolutional neural network)
  - Random subsampling of training space to generate experimental data
- Fully connected layer modeling
  - *tensorflow.layers.dense* to generate fully connected layers
  - 25K randomly chosen training data points
- Convolutional layer modeling
  - *tensorflow.layers.conv2d*
  - 50K randomly chosen training data points

Convolution layer runtime estimation

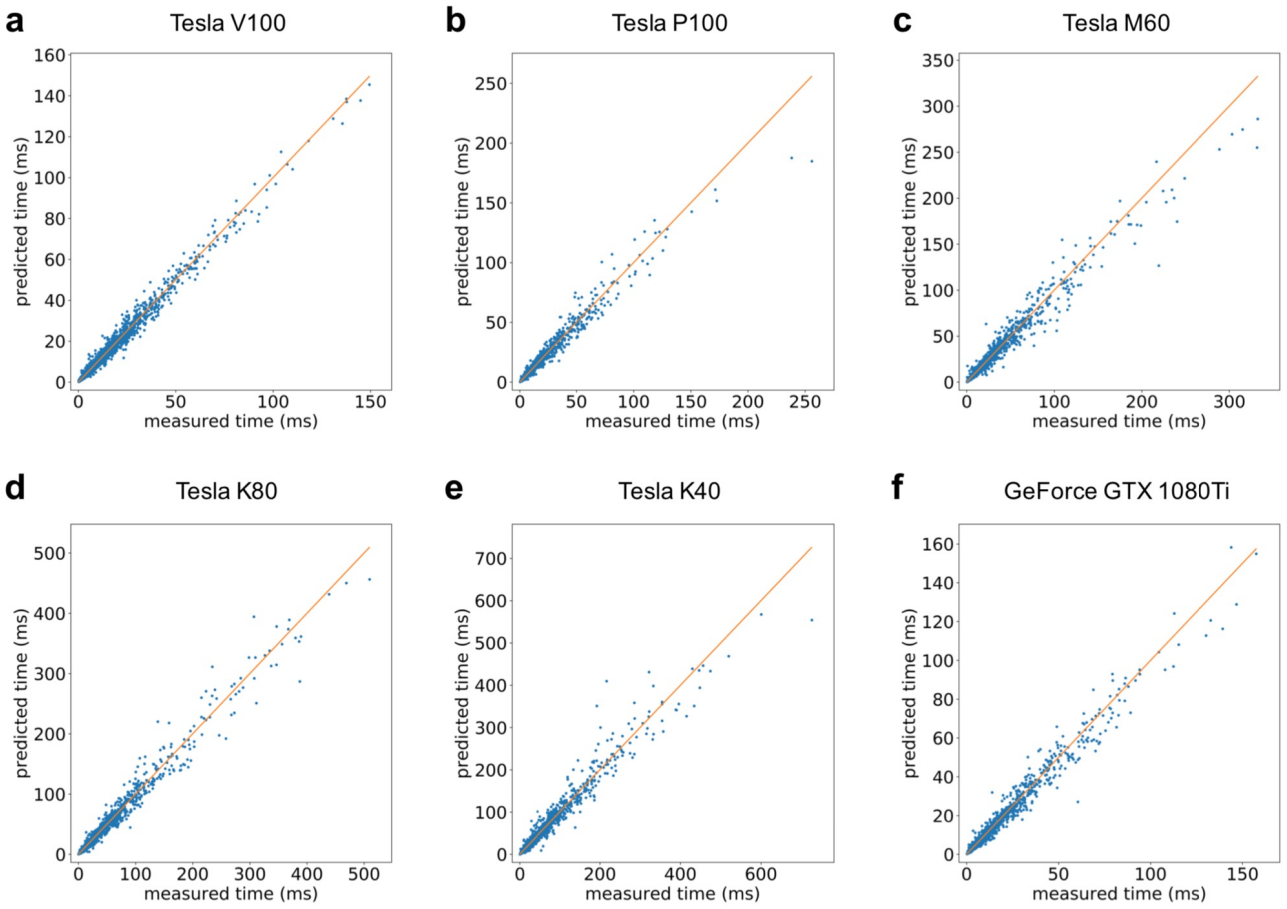


Fig. 3. Predicted time for convolutions versus measured time. **a)** Tesla V100, RMSE of prediction 1.73 ms. **b)** Tesla P100, RMSE 3.32 ms. **c)** Tesla M60, RMSE 6.07 ms. **d)** Tesla K80, RMSE 7.84 ms. **e)** Tesla K40, RMSE 11.90 ms. **f)** Geforce GTX1080Ti, RMSE 2.55 ms

Why RMSE is higher for older GPUs ?

- Separate model for each GPU type
- Execution time prediction without accounting for hardware features

Justus et al. Predicting the Computational Cost of Deep Learning Models. 2018

How good are linear regression models ?

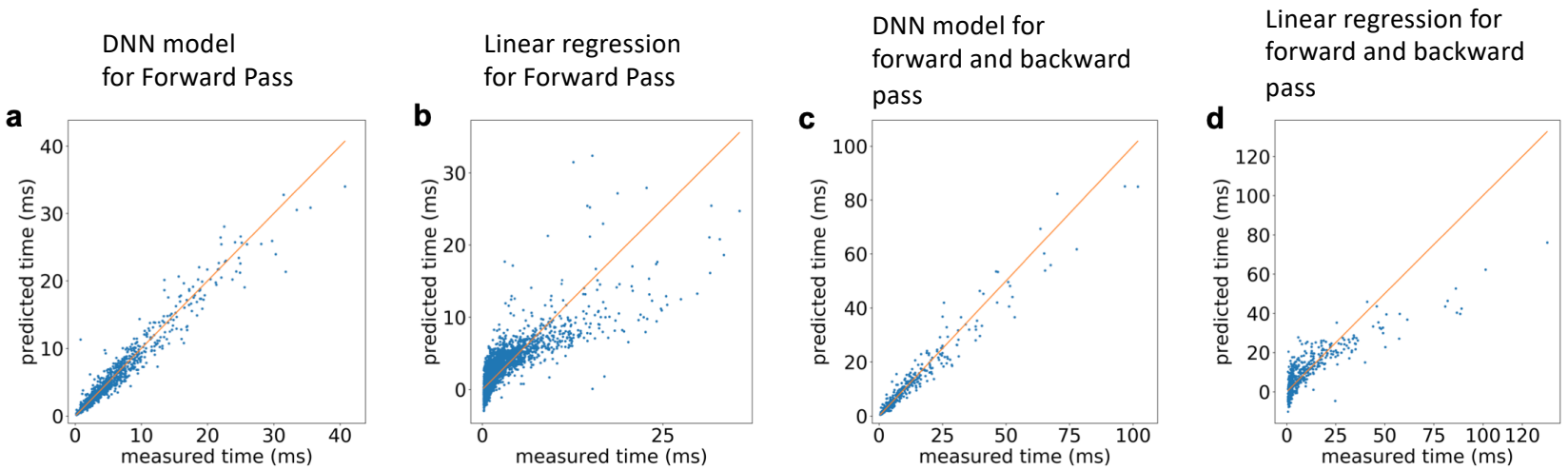


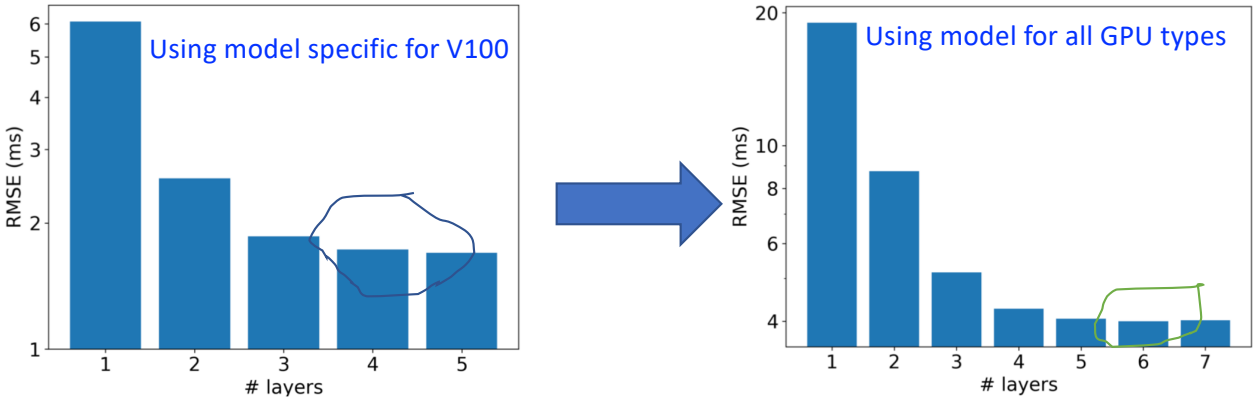
Fig. 4. Predicted execution time for convolutions on a Tesla V100 versus the measured time. **a)** Deep neural network model for a forward pass only, RMSE 0.83 ms. **b)** Linear regression model for a forward pass only, RMSE 2.42 ms. **c)** Deep neural network model for a forward and backward pass with stochastic gradient descent, RMSE 3.18 ms. **d)** Linear regression model for a forward and backward pass with stochastic gradient descent, RMSE 8.95 ms.

- Linear regression method becomes progressively worse as the actual execution time increases
- Linear regression approach is modelling for majority of observed data points and missing the non-linear effects of longer run- times

Single performance model for all GPU types

- Features of the model were expanded to include GPU memory bandwidth, GPU clock frequency and the number of CUDA cores.
- More features ➔ more complex prediction model
- Merge data from all different GPU types → bigger size training data
- Can be used to predict performance on a new, unseen hardware

V100 convolution layer time prediction time



Justus et al. Predicting the Computational Cost of Deep Learning Models. 2018

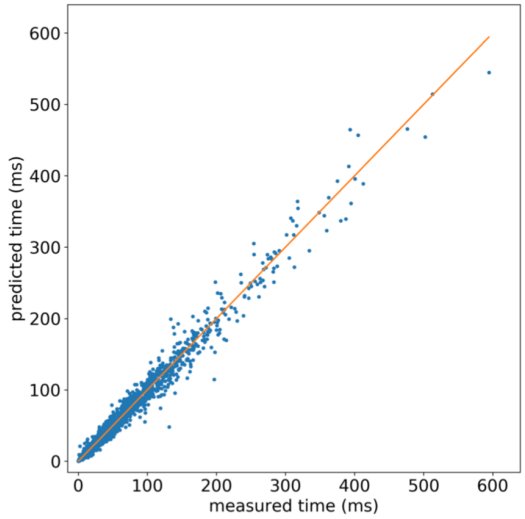
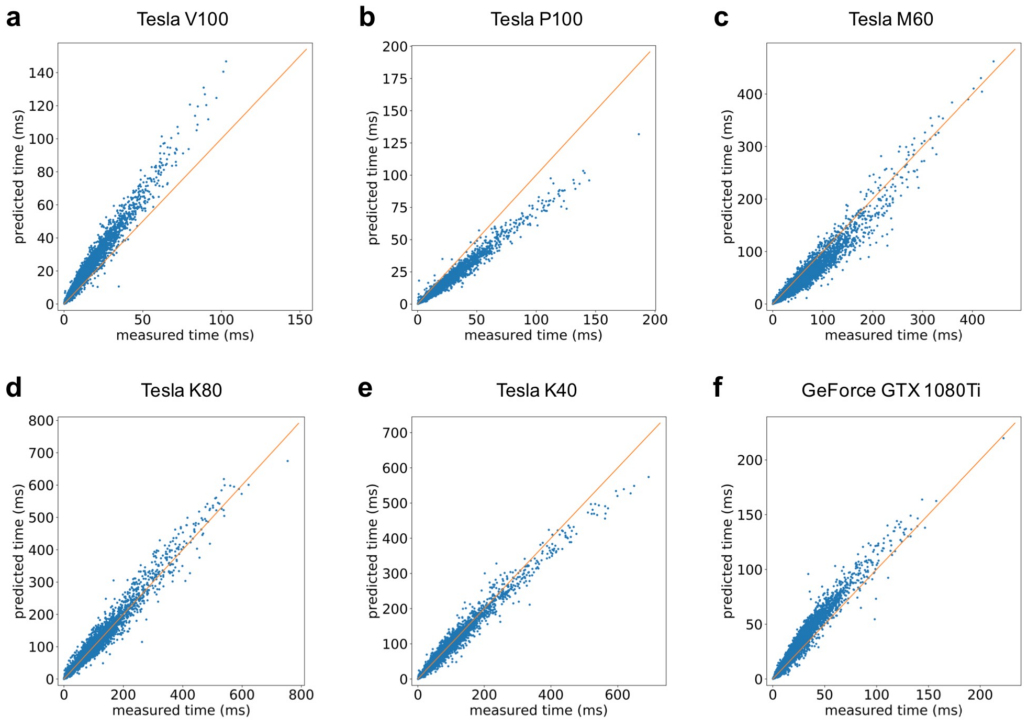


Fig. 6. Predicted time for convolutions versus measured time for a model that has been trained on data from all available GPUs. The RMSE is 3.88 ms.

Single performance model for all GPU types : How generalizable is this?



- Predicted time for convolutions plotted against measured time for a model that has been trained on data from a set of GPUs excluding the one tested
- The uncertainty of predictions can be expected to be particularly large for new GPUs with specifications that don't fall into the range of known hardware.

Justus et al. Predicting the Computational Cost of Deep Learning Models. 2018

Batch Execution Time Prediction

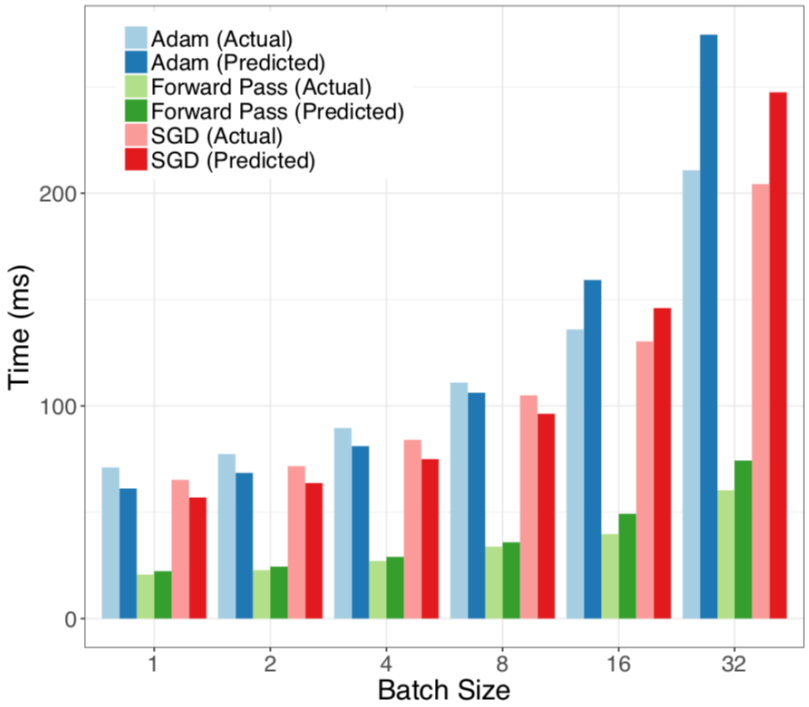


Fig. 9. Predicted versus actual execution time for VGG-16.

Predictive performance modeling of DL training and inference time is promising

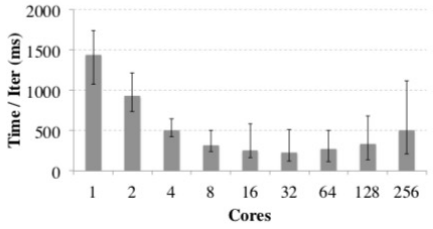
- Estimate training time of an epoch and hence the total training time execution
- Study effect of hardware, optimizers etc. on runtime
- Runtime translates to cost
- Cost-benefit tradeoff

Justus et al. Predicting the Computational Cost of Deep Learning Models. 2018

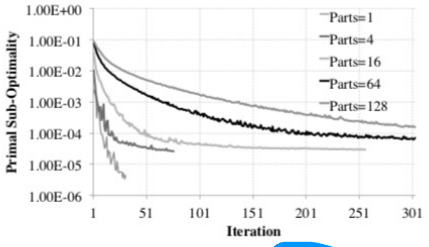
Distributed ML on Compute Clusters

- Choose the right algorithm
- Choose the cluster size (how many nodes) and node configurations
- Nodes can be virtual servers or bare metal servers (dedicated machines with no resource sharing; no resource virtualization)

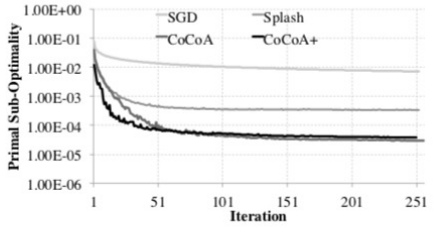
Distributed ML Performance and Cluster Size



(a) Time per iteration as we vary the degree of parallelism used. The plot shows the mean across 50 iterations and the error bars show the 5th and 95th percentile



(b) Convergence of CoCoA as we vary the degree of parallelism used.



(c) Comparison of convergence rate of CoCoA, CoCoA+, SGD and Splash when using 16 cores.

M. Jaggi, V. Smith, et al. Communication-efficient distributed dual coordinate ascent. In *NIPS*, 2014

- Performance (time to converge to certain error threshold) depends on the cluster set up used for training
  - Data parallelism and strong scaling: compute time decreases with increase in the cluster size but communication time increases; per iteration time may first decrease but then increase
- Convergence rates (number of iterations to reach a certain error threshold) of algorithms is dependent on the size of the cluster use
  - With more nodes, batch size per node becomes smaller, the gradient estimates are noisier and it takes more iterations to converge

Pan et al. Hemingway: Modeling Distributed Optimization Algorithms. NIPS 2016

Challenges with Distributed ML

- **Changing degree of parallelism affects performance and convergence**
- Computation vs. communication balance and convergence rates differ across algorithms (e.g., first-order methods vs. second-order methods)
  - Hard to predict which algorithm will be the most appropriate for a given cluster setup
- Convergence rate of an algorithm is also data dependent
- Choosing an appropriate algorithm and cluster size for a given task (dataset+problem) is challenging
  - Automated ML systems help in identifying the best algorithm (optimizes some algorithmic performance score) from a candidate list for a given task
  - Choosing the right cluster configuration to run an algorithm is difficult

Hemingway

- **Goal**: Develop an interface where users can specify an end-to- end requirement and the system will automatically select the appropriate algorithm and degree of parallelism using predictive performance models.
- **Use cases**:
  - Given a relative error goal of ε, choose the fastest algorithm and configuration
  - Given a target latency of t seconds choose an algorithm that will achieve the minimum training loss
- **Approach:** A black-box modeling applicable to different distributed ML algorithms

Pan et al. Hemingway: Modeling Distributed Optimization Algorithms. NIPS 2016

Hemingway: Performance Models

- Build two models:
  - **System model**: captures the system level characteristics of how computation, communication change as we increase cluster sizes
  - **Convergence model**: captures how convergence rates change with cluster sizes
- System model: $f(m) = \theta_0 + \theta_1 \times (size/m) + \theta_2 \times \log(m) + \theta_3 \times m$

  iteration time with m node cluster

  compute time

  communication time
- Convergence model: $g(i,m) = \boxed{\sum_{j=1}^{k} \lambda_j \phi_j(i,m)}$

  score after i iterations with m node cluster

  Regression with non-linear features; Coefficients λ_j determined by standard techniques like least square, LASSO etc.
- Combined model: $h(t,m) = g(t/f(m), m)$

Pan et al. Hemingway: Modeling Distributed Optimization Algorithms. NIPS 2016

Model decomposition helps

- Allows to train the two models independently and reuse them based on changes.
- Retrain the system model and reuse the convergence model
  - New machine types or networking hardware changes in a datacenter
- Retrain the convergence model and reuse the system model
  - Changes to the algorithm in terms of parameter tuning
- What about changes to batch size ? Will it require retraining both the models ?

Hemingway in Action

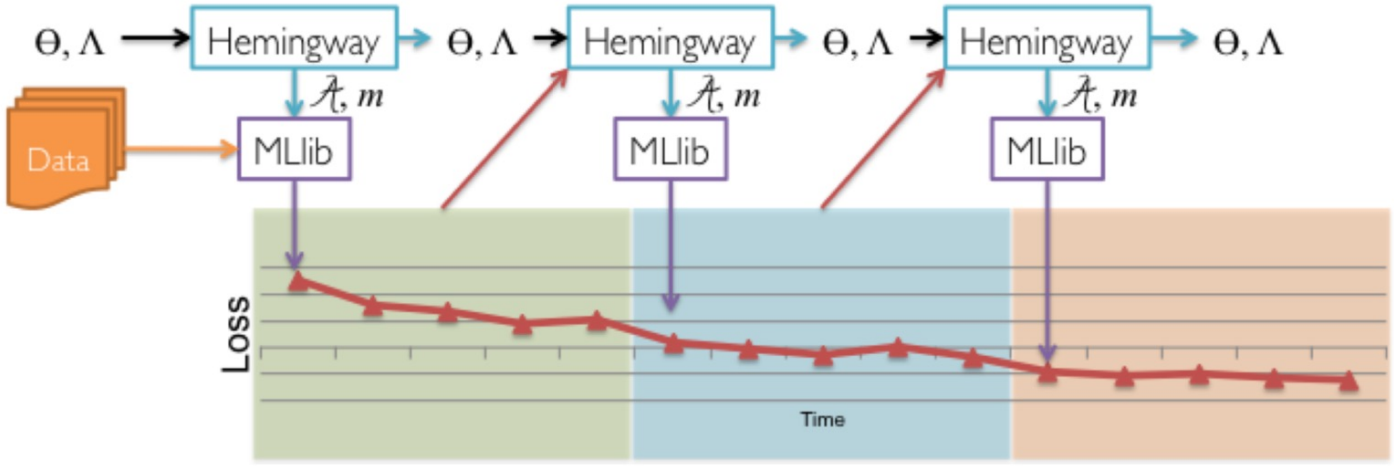


Figure 2: Idealized Example of using Hemingway. For each time frame, Hemingway takes as input the current estimated system model Θ and convergence model Λ, and suggests the best algorithm $\mathcal{A}$ and number of machines m to use for the next time frame. These are then fed into a machine learning framework, e.g. MLlib, which executes the convex optimation algorithm. Resultant losses are provided as input into Hemingway to update Θ and Λ.

PALEO

- An analytical model to estimate the scalability and performance of deep learning systems.
- PALEO decomposes the total execution time into computation time and communication time
- Computation time is calculated from factors including
  - the size of the computation inputs imposed by the network architecture
  - the complexity of the algorithms and operations involved in the network layers
  - the performance of the hardware to be used
- Communication time is estimated based on
  - the computational dependencies imposed by the network
  - the communication bandwidth of the hardware
  - the assumed parallelization schemes
- Once the network architecture and design space choices are fixed, all the key factors in PALEO can be derived, and we can estimate execution time without actually implementing the entire network and/or an underlying software package.
- Paleo code https://github.com/TalwalkarLab/paleo
- Live demo https://talwalkarlab.github.io/paleo/

H. Qi, E. Sparks, and A. Talwalkar. Paleo: A performance model for deep neural networks. ICLR 2017

PALEO Modeling Approach

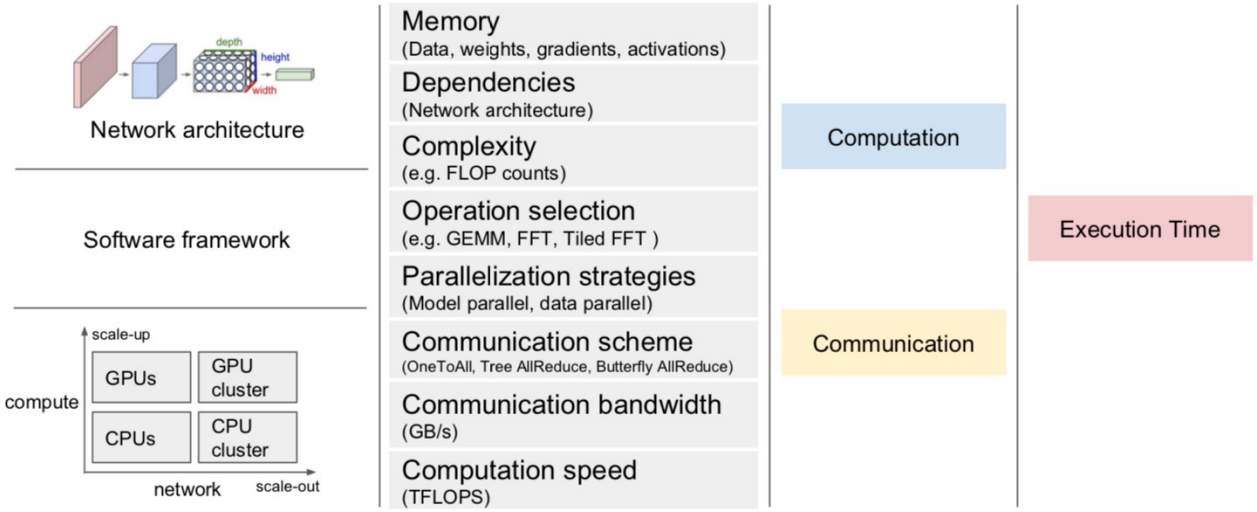


Figure 1: Overview of the PALEO modeling approach. PALEO decomposes execution time into computation time and communication time, which can be derived from various factors implicitly specified by network architectures and hardware configurations.

PALEO Computation Model

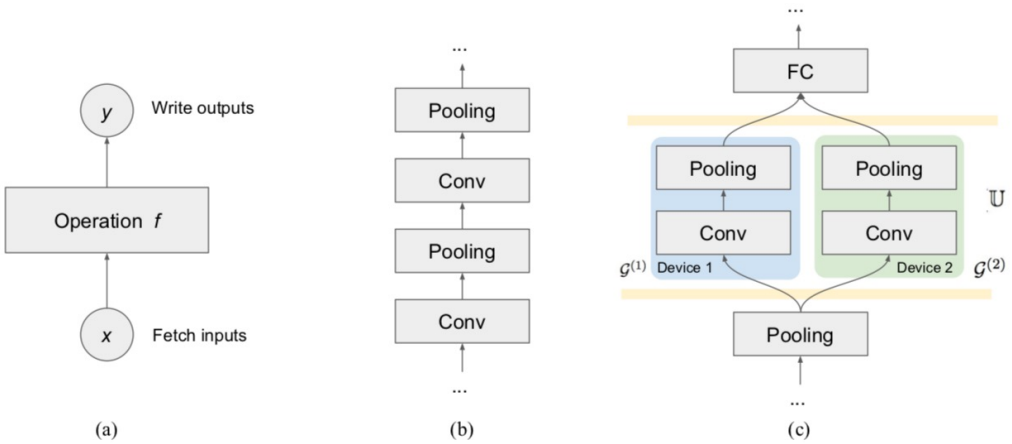


Figure 2: (a) The execution time of a node in the computation graph consists of the time for fetching input, computing results, and writing results to memory. (b) An example of a sequential computation graph segment. (c) An example of a parallel computation graph segment.

Computation Node

- Expressed as a directed graph $\mathcal{N} = \langle \{u^{(i)}\}_{i=1}^{n}, \{(u^{(i)}, u^{(j)})\} \rangle$

- Each node in the graph is associated with an operation and a device on which it is executed

PALEO Computation Model

- Computation time for a single layer

$$T(u) = \mathcal{R}(\mathrm{Pa}(u)) + \mathcal{C}(f, d) + \mathcal{W}(f, d).$$

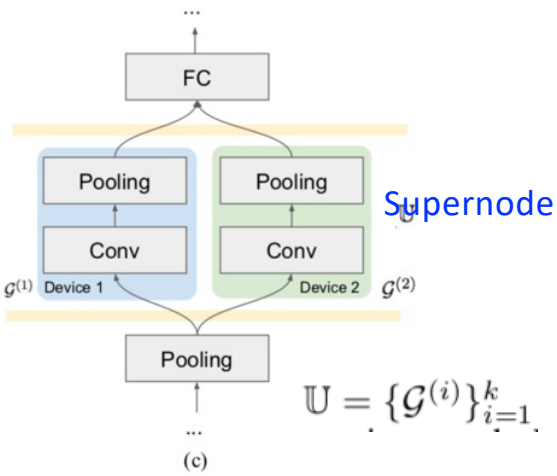
Time to fetch input from parent layer

Time to compute operation f on device d

Time to write output to local memory

$$\mathcal{C}(f, d) = \mathrm{FLOPs}(f)/\mathrm{speed}(d)$$

- Computation time for networks
  - Sequential structures: $T(\mathcal{N}) = \sum_{i=1}^{n} T(u^{(i)})$.
  - Parallel structures: $T(\mathbb{U}) \in [\max_i T(\mathcal{G}^{(i)}), \sum_i T(\mathcal{G}^{(i)})]$
    the lower bound corresponds to perfect parallelization
    the upper bound corresponds to sequential execution
  - Example: Inception module



FLOPs calculation

- Several optimized implementations to perform convolution
- FLOPs depend on the choice of algorithm for convolution
- Choice of algorithm – matrix multiplication or FFT – is problem specific
  - Depends on the filter size, strides, input size of the convolutional layers, and memory workspace.
- Two common approaches are employed in existing DNN software frameworks and libraries to choose between these algorithms:
  1. Using predefined heuristics based on offline benchmarks;
  2. Autotuning to empirically evaluate available algorithms on the given specification

PALEO Communication Model

- PALEO considers three communications schemes: OneToAll, Tree AllReduce, and Butterfly AllReduce.
- Time to transfer |D| data between two workers with a communication bandwidth B

$$T_{\text{comm}} = |D|/B$$

- Depending on the communication scheme the communication time can have different scalability with number of learners (K)

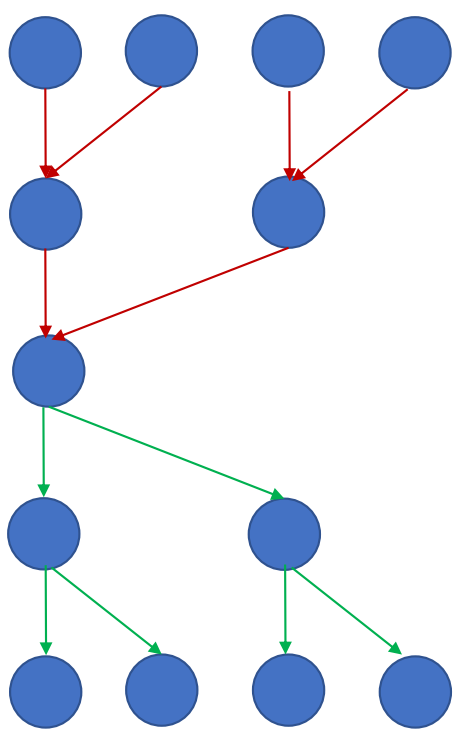
Communication schemes and time

TreeAllReduce

$2log_2K$

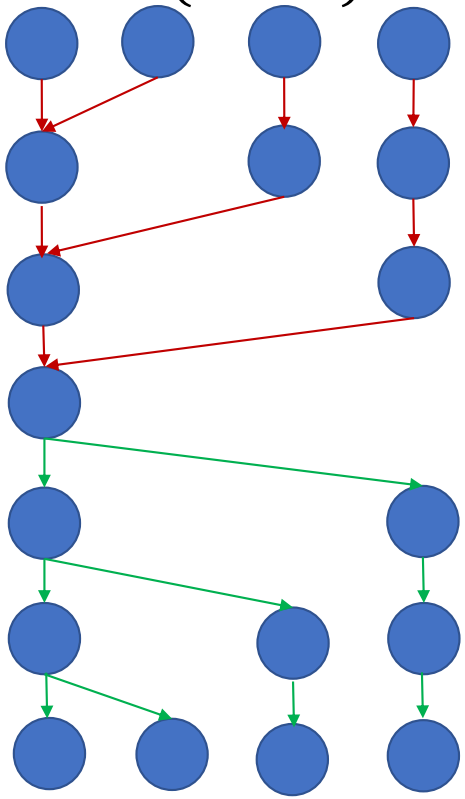
aggregation

broadcast



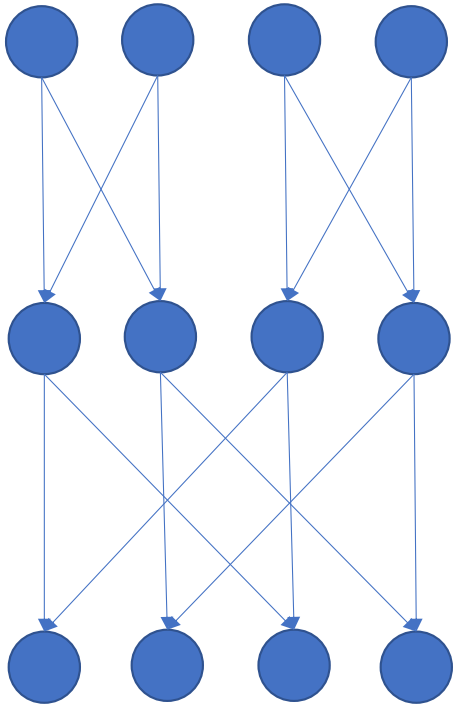
OneToAll

$2(K-1)$



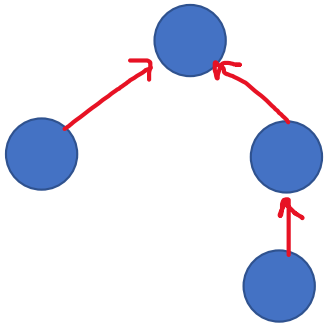
Butterfly AllReduce

log_2K

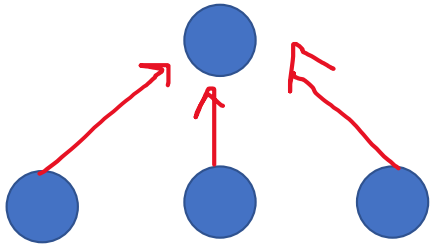


Alternative representation

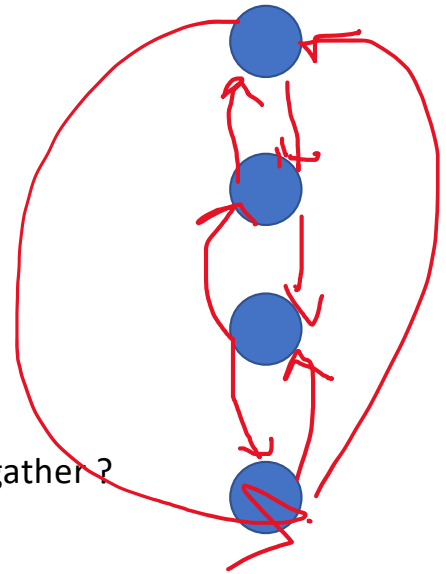
TreeAllReduce



OneToAll



Butterfly AllReduce



- What about number of communication rounds in AllReduce with scatter-gather ?
- Is it same as which scheme on this slide ?
- What about communication time ?
- Can you use any of the above scheme and achieve same communication time scaling with K as AllReduce ?

Platform Percent of Peak (PPP)

- Hardware devices are marked with peak FLOPS
- Achieving peak FLOPS in systems is difficult
  - Requires customized libraries developed by organizations with intimate knowledge of the underlying hardware, e.g., Intel's MKL (Math Kernel Library), NVIDIA cuDNN (CUDA Deep Neural Network)
- PPP captures average relative inefficiency of the platform (hardware+framework) compared to peak theoretical FLOPS
- Calculated using observed total throughput and estimated total throughput on a benchmark consisting of a set of representative DL workloads
- Given observed total throughput and estimated total throughput on this benchmark we fit a scaling constant to estimate a *platform percent of peak* (PPP) parameter
- Calculate PPP for both compute and communication

Optimus

- **Goal**: To learn relation between resource configuration and the time a training job takes to achieve model convergence to make efficient scheduling decisions on Deep Learning cluster
- **Approach**:
  - Estimate number of remaining epochs needed to converge
    - Model training loss as a function of number of epochs
  - Estimate the time to complete per epoch as a function of hardware resources
    - Model time to finish per epoch as a function of number of learners and parameter servers

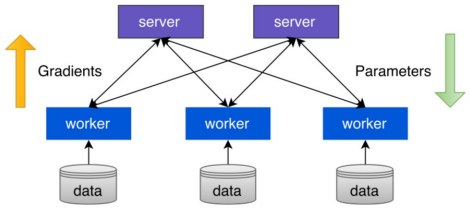


Figure 3: Parameter server architecture

Y. Peng et al. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. Eurosys 2018

Optimus: Training loss modeling

- Training loss model: $l = \frac{1}{\beta_0 \cdot k + \beta_1} + \beta_2$ $\quad l : training\ loss$, $k : number\ of\ epochs$
- Online fitting (using a regression solver) after every epoch using the entire history
- Using the model and a convergence error threshold we can calculate the remaining number of training epochs needed

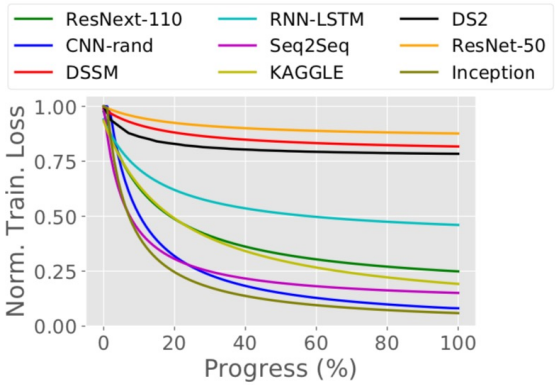


Figure 5: Training loss curves for different DL jobs

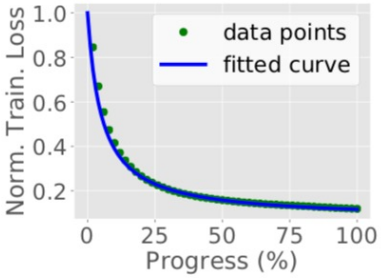


Figure 7: Online model fitting for training Seq2Seq: $\beta_0 = 0.21$, $\beta_1 = 1.07, \beta_2 = 0.07$

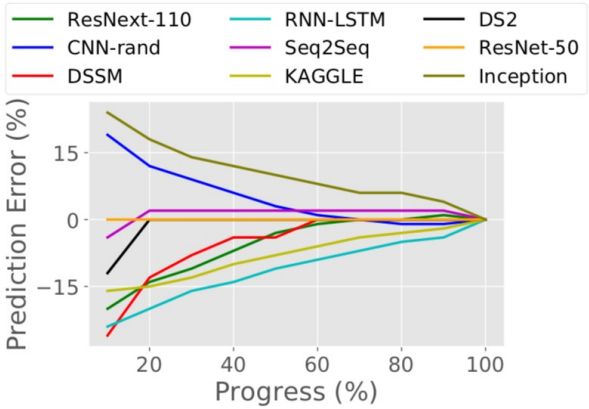


Figure 6: Prediction errors in different DL jobs

Optimus: Resource-speed modeling

Modeling duration of one training step(iteration) on a learner T

$$T_\rho = Gradient\ compute\ at\ learner + Gradient\ comm.time + Gradient\ update\ time\ at\ PS\ \rho + Comm.overhead$$

- Forward pass time mT_f
- Backward pass time T_b
- Gradient data/bw between PS and learner $2\frac{S/P}{B/w'_\rho}$
- Time to perform gradient aggregation at PS
- Connection setting between learner and PS (linear in learners and PS)

$$T_\rho = mT_f + T_b + 2\frac{S/P}{B/w'_\rho} + \frac{T\_update \cdot w'_\rho}{P} + \delta \cdot w + \delta' \cdot P$$

$$T = max_\rho T_\rho$$

- Forward propagation time to process one sample: T_f
- Forward propagation time to process a mini-batch: mT_f
- Backward propagation time: T_b
- Model/gradient size: S
- Size of gradients transferred from a worker to a PS: S/P
- Bandwidth at parameter server (PS): B
- Number of workers that send gradients to a PS concurrently: w'_ρ
- Bandwidth between a PS and a worker: B/ w'_ρ
- Communication time between a PS and a worker: 2 ((S/P)/(B/w'))
- Time to update parameters with size S: T_update
- Parameter update time on PS: (T_update*w')/P
- Communication overhead: $\delta \cdot w + \delta' \cdot P$
- Homogeneous workers; Load balanced parameter servers

Optimus: Resource-speed modeling

Throughput in terms of training steps completed per unit time

- Asynchronous Training:

Where did w' go ?

Throughput = $w \cdot T^{-1}$

w' proportional to w for asynchronous

$$f(p, w) = w \cdot (\theta_0 + \theta_1 \cdot \frac{w}{p} + \theta_2 \cdot w + \theta_3 \cdot p)^{-1}$$

- Synchronous Training:

Throughput = T^{-1}

w' = w for synchronous

$$f(p, w) = (\theta_0 \cdot \frac{M}{w} + \theta_1 + \theta_2 \cdot \frac{w}{p} + \theta_3 \cdot w + \theta_4 \cdot p)^{-1}$$

$$Total\ batch\ size: M$$

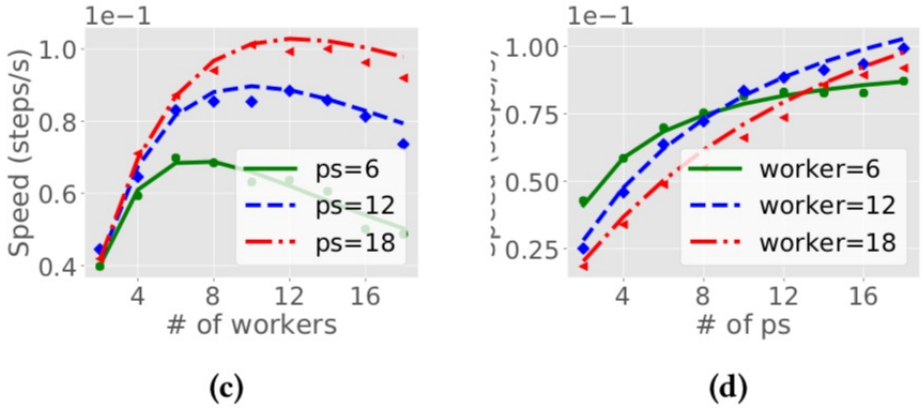
$$Batch\ size\ per\ learner: m = \frac{M}{w}$$

Effective mini-batch size trained by all workers in each step remains the same and does not change with w, guaranteeing convergence to same model

Optimus: Predictive Performance for Strong Scaling

Asynchronous Training

Synchronous Training



(a) due to communication overhead, there is a trend of diminishing return where adding more parameter servers or workers does not improve the training speed much

(b) for synchronous training, more workers may lead to lower training speed.

Table 2: Coefficients in speed functions

	θ_1	θ_2	θ_3	θ_4	θ_5	Residual sum of squares for fitting
Async	2.83	3.92	0.00	0.11	-	0.10
Sync	1.02	2.78	4.92	0.00	0.02	0.00

Less than 10% prediction error

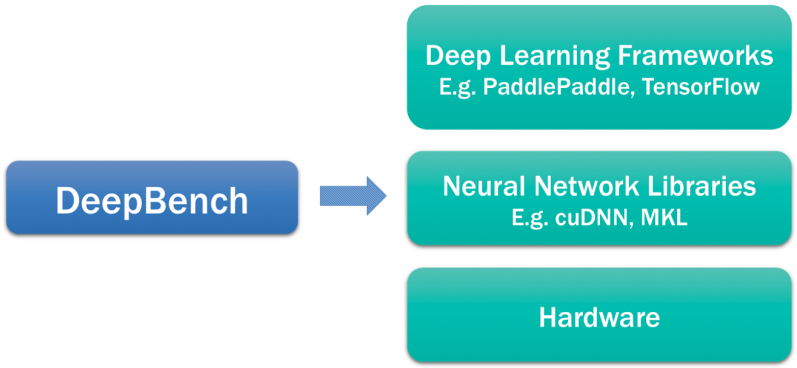
Y. Peng et al. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. Eurosys 2018.

Deep Learning Benchmarks

- DL jobs (training and inference) are computationally expensive
- Research targeted at reducing computation time and cost
  - software systems, training algorithms, communication methods, hardware
- Different optimizations target different performance metrics
- Lack of standard evaluation criteria (datasets and metrics) to objectively study and compare the benefits of different results

DeepBench

- https://github.com/baidu-research/DeepBench
- Released in 2016 by Baidu Research
- Open source benchmarking tool
- Measures performance of basic operations involved in training and inference of deep neural networks
  - Dense matrix multiplication, convolution, recurrent layers (with different activations), communication (AllReduce)



- Uses neural network libraries to benchmark the performance of basic operations on different hardware
- Does not work with deep learning frameworks or deep learning models built for applications
- Can identify bottlenecks in different operations
- Cannot benchmark end-to-end training time or inference time

"Which hardware provides the best performance on the basic operations used for training deep neural networks?"

Kernel level and Model level benchmarking

Kernel level
- Hardware vendors
- DeepBench

Application level
- AI system engineers
- DAWNBench, MLPerf

DAWNBench

- Open benchmark for end-to-end deep learning training & inference
- Measures *end-to-end* performance of training (e.g., time, cost) and inference (e.g., latency, cost) at a specified accuracy level
- Developed at Stanford University; predecessor of MLPerf
- DAWNBench evaluates deep learning systems on different tasks based on several objective metrics, using multiple datasets.
  - Allows experimentation of new model architectures and hardware

Tasks	Metrics	Datasets
Image classification	Training time Training cost	ImageNet CIFAR10
Question answering	Inference latency Inference cost	SQuAD

Stanford Question Answering Dataset (SQuA

Table 1: Dimensions evaluated in the first version of DAWNBench. All metrics are for a near-state-of-the-art accuracy.

DAWNBench Metrics Explained

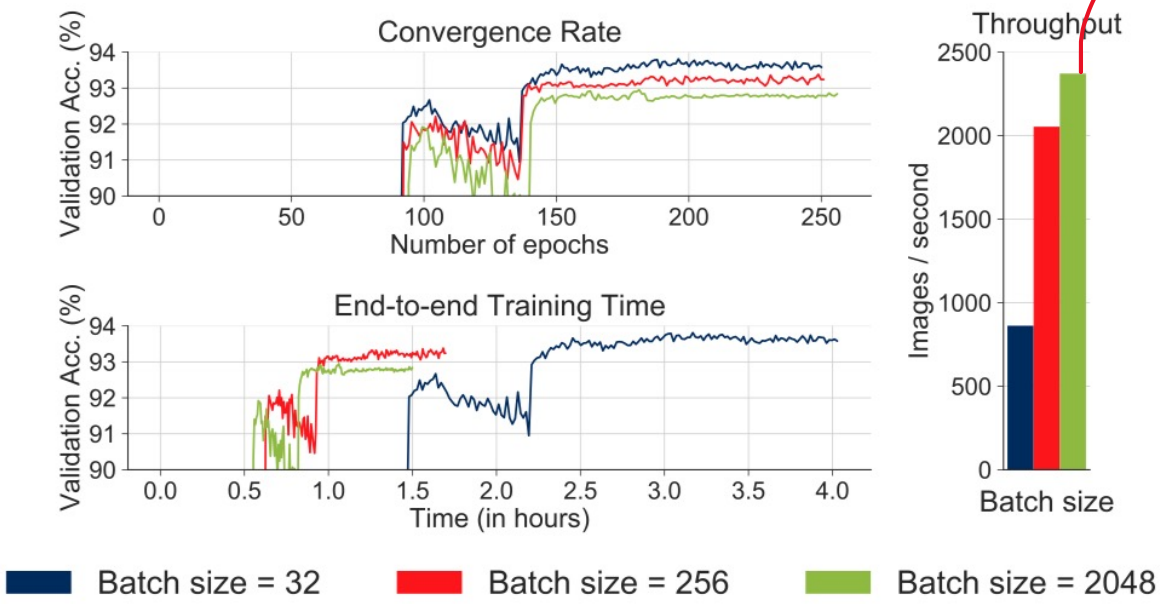
- Image classification with Imagenet1K
  - Training Time: Time taken to train an image classification model to a top-5 validation accuracy of 93% or greater on ImageNet.
  - Training Cost: Total cost of public cloud instances to train an image classification model to a top-5 validation accuracy of 93% or greater on ImageNet.
  - Inference Latency: Latency required to classify one ImageNet image using a model with a top-5 validation accuracy of 93% or greater.
  - Inference Cost: Average cost on public cloud instances to classify 10,000 validation images from ImageNet using of an image classification model with a top-5 validation accuracy of 93% or greater.
- Image classification with CIFAR-10
  - Training Time: Time taken to train an image classification model to a test accuracy of 94% or greater on CIFAR10.
  - Training Cost: Total cost for public cloud instances to train an image classification model to a test accuracy of 94% or greater on CIFAR10.
  - Inference Latency: Latency required to classify one CIFAR10 image using a model with a test accuracy of 94% or greater.
  - Inference Cost: Average cost on public cloud instances to classify 10,000 test images from CIFAR10 using an image classification model with a test accuracy of 94% or greater.
- Question Answering on SQuAD
  - Training Time: Time taken to train a question answering model to a F1 score of 0.75 or greater on the SQuAD development dataset.
  - Training Cost: Total cost for public cloud instances to train a question answering model to a F1 score of 0.75 or greater on the SQuAD development dataset.
  - Inference Time: Latency required to answer one SQuAD question using a model with a F1 score of at least 0.75 on the development dataset.
  - Inference Cost: Average cost on public cloud instances to answer 10,000 questions from the SQuAD development dataset using a question answering model to a dev F1 score of 0.75% or greater.

DAWNBench Vs other DL Benchmarks

- Benchmarks prior to DAWNBench
  - Baidu DeepBench
  - Tensorflow CNN benchmarks
  - Fathom
- DAWNBench focuses on end-to-end performance
  - Tensorflow CNN benchmarks use the time needed to train on a single minibatch of data as the key metric, while disregarding the resulting accuracy of the trained model
  - DeepBench and Fathom focuses on timing individual low-level operations (e.g. convolutions, matrix multiplications) utilized in deep learning computations
- Rolling submissions to DAWNBench ended on **3/27/2020**

TTA vs Minibatch Size

ResNet56 CIFAR10 model on a P100.



large batch size better saturate GPU cores
requires less weight updates

Accuracy-Time tradeoff

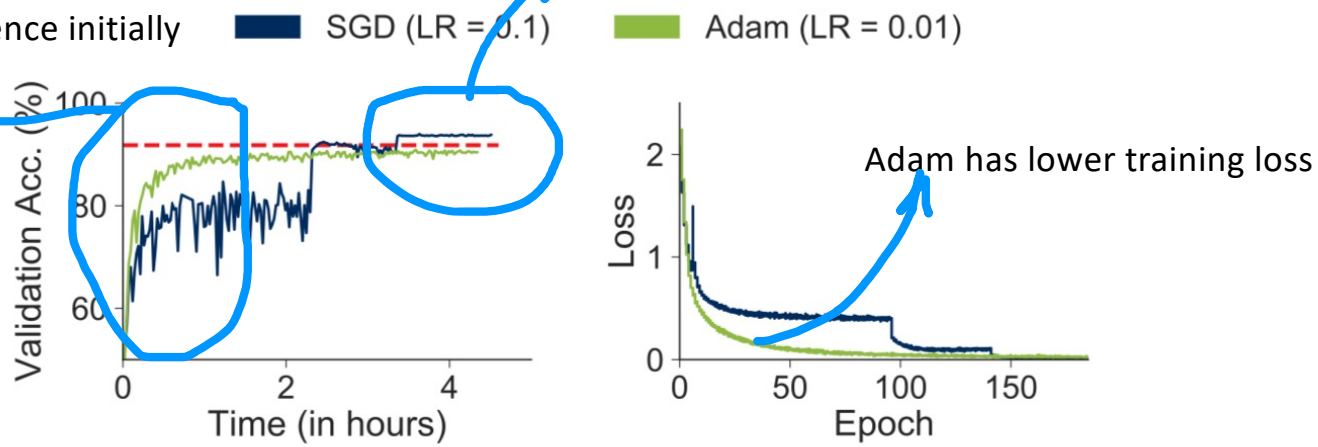
Minibatch size of 32 produces the best convergence rate (least number of epochs to highest accuracy), and a minibatch size of 2048 produces the best throughput (number of images processed divided by total time taken). Minibatch size of 256 represents a reasonable trade-off between convergence rate and throughput. Minibatch size of 256 reaches an accuracy of 93.38%, which is only 0.43% less than the maximum accuracy achieved with a minibatch size of 32, in 1.9x less time

Coleman et al. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. NIPS 2017

Why accuracy and not loss threshold ?

SGD (with momentum) converges to a better accuracy

Adam has faster convergence initially



Adam has lower training loss

Figure 2: Impact of different optimizers while training a ResNet56 model on CIFAR10. The graph on the left plots the top-1 validation accuracy with respect to time, and the graph on the right plots a rolling average of the loss with respect to the epoch. We see that Adam initially outperforms SGD with momentum, but falls short around epoch 100 (~2 hours in the graph on the left), and does not reach a validation accuracy of 93%.

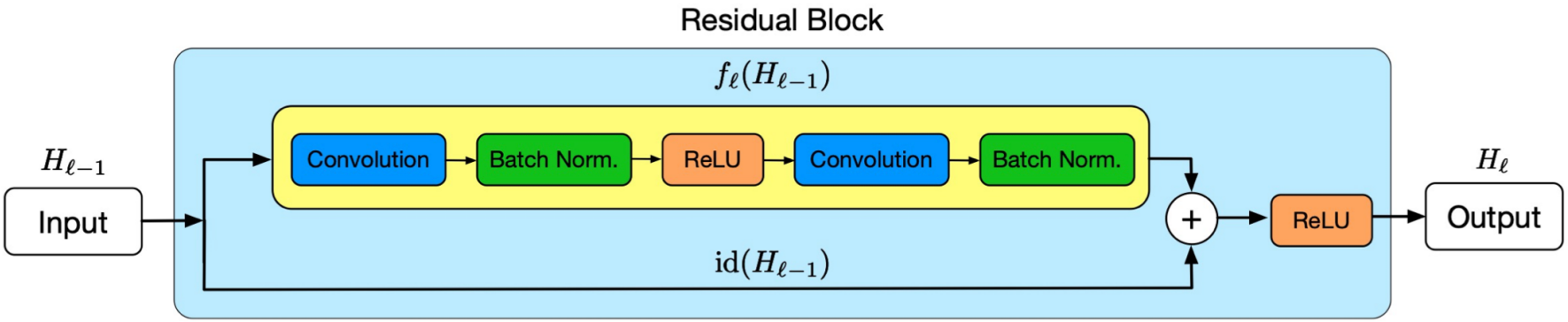
Coleman et al. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. NIPS 2017

Interaction Between Optimizations

- Do benefits from different optimizations compose?
  - Will the benefits add up (e.g., 2x with O1 and 3x with O2 will translate to 6x with O1 and O2)
- Example scenario considering 3 candidate optimizations:
  - ADAM optimizer
  - Single-node multi-GPU training
  - Stochastic Depth*
    - Regularization similar to dropout
    - Entire layers are randomly dropped during training,
    - Full network is used to perform inference
  - Larger batch size

*Huang et al. Deep Networks with Stochastic Depth

Recall Residual block



$$H_\ell = \mathtt{ReLU}(f_\ell(H_{\ell-1}) + \mathrm{id}(H_{\ell-1}))$$

Stochastic Depth

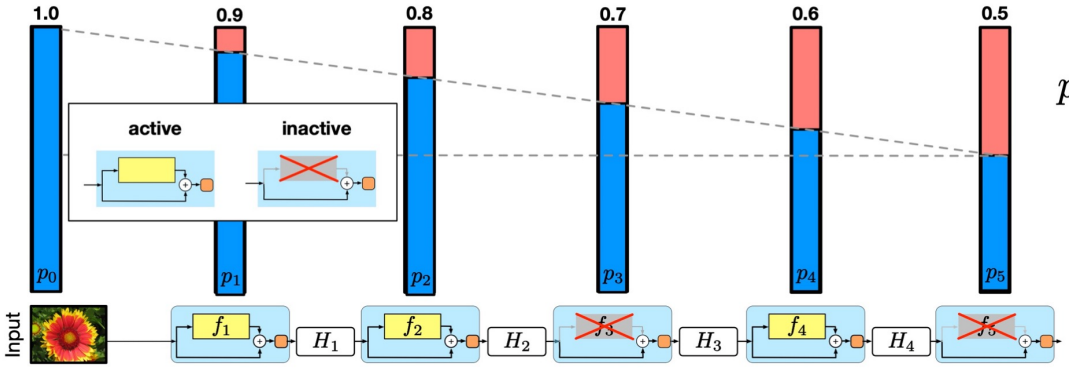
- Reduce the effective length of a neural network during training by randomly skipping layers entirely
- For each mini-batch randomly select sets of layers and remove their corresponding transformation functions, only keeping the identity skip connections
- Stochastic depth aims to shrink the depth of a network during training, while keeping it unchanged during testing.
- Random drop probability for each layer

Stochastic depth and layer survival probability

$b_\ell \in \{0, 1\}$ denote a Bernoulli random variable

$$H_\ell = \texttt{ReLU}(b_\ell f_\ell(H_{\ell-1}) + \mathrm{id}(H_{\ell-1}))$$

"survival" probability of ResBlock ℓ as $p_\ell = \Pr(b_\ell = 1)$



$$p_\ell = 1 - \frac{\ell}{L}(1 - p_L)$$

Expected network depth

- $\tilde{L}$ be random variable for the depth of network at any mini-batch
- $E[\tilde{L}] = \sum_{l=1}^{L} p_l$
- With linear decay rule and $p_L = 0.5$

The picture can't be displayed.

- When a residual block is bypassed for a specific iteration, there is no need to perform forward-backward computation or gradient updates for that block
- Stochastic depth speeds-up the training; 25%-40% training time speed-up

Composing Optimizations

- Do different optimizations compose well ?

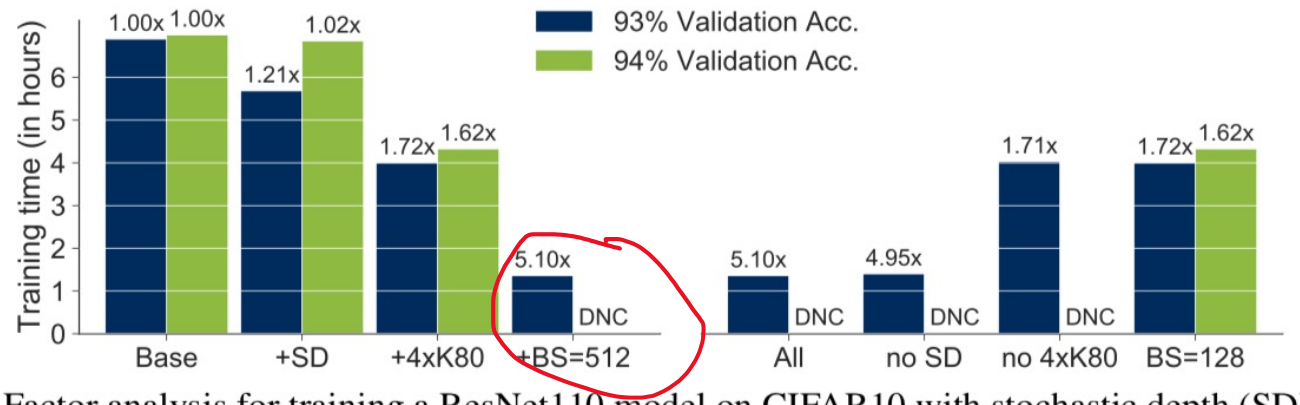


Figure 3: Factor analysis for training a ResNet110 model on CIFAR10 with stochastic depth (SD), 4 Nvidia K80 GPUs on a single node (4xK80), and a minibatch size (BS) of 512. Cumulatively enabling each optimization reduces the time to 93% top-1 accuracy, but combined, the model does not converge (DNC) to the 94% accuracy threshold. By removing the larger minibatch size, the model reaches the higher accuracy target.

Optimizations can interact in non-trivial ways when used in conjunction, producing lower speed-ups and less accurate models

Coleman et al. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. NIPS 2017

Variability Due to Hardware and Software

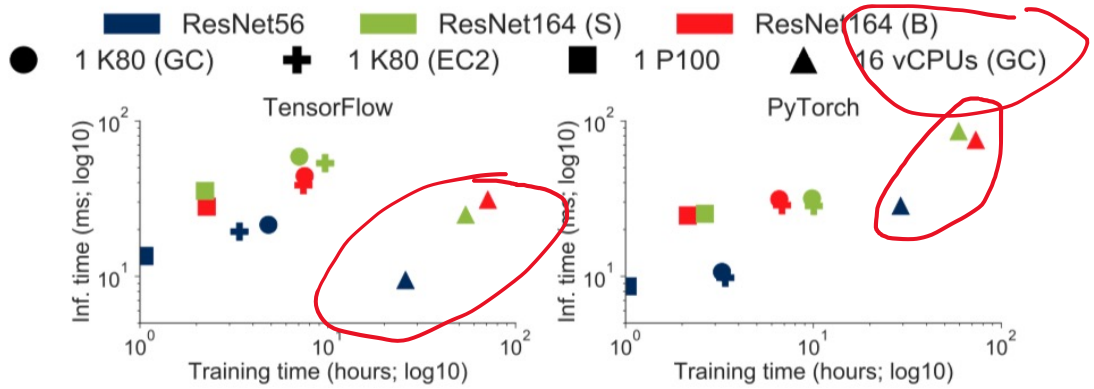
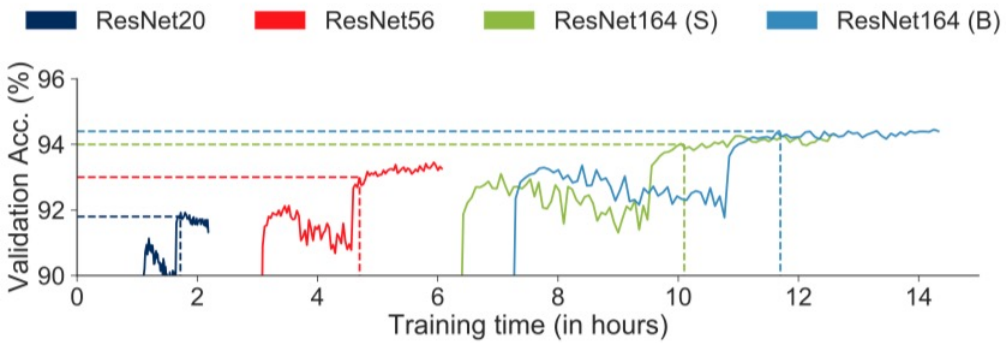


Figure 5: Inference time vs. training time to 93% validation accuracy, for different hardware, frameworks, and model architectures in DAWNBench's seed entries. ResNet164 (S) uses a simple building block, while (B) uses a bottleneck building block. Amazon EC2 instances use a `p2.xlarge` instance type (4vCPUs, 61 GB memory).

- TensorFlow is faster than PyTorch on CPUs but slightly slower on GPUs for inference
- Training and inference time are proportional to the depth of the model

Coleman et al. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. NIPS 2017

Identifying Optimal DL Architecture



Accuracy threshold	Time to threshold	Fastest model
91.8%	1h 43m	ResNet20
93%	4h 42m	ResNet56
94%	10h 6m	ResNet164 (simple)
94.4%	11h 42m	ResNet164 (bottleneck)

Figure 6: Validation accuracy vs. training time for different ResNet architectures on CIFAR10. Horizontal lines indicate accuracy thresholds of 91.8%, 93%, 94%, and 94.4%. ResNet20, ResNet56, ResNet164 (with simple building blocks), and ResNet164 (with bottleneck building blocks) are fastest to the corresponding accuracy thresholds.

For lower accuracy thresholds, shallower architectures reach the threshold faster.

Coleman et al. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. NIPS 2017

Training Cost vs Training Time

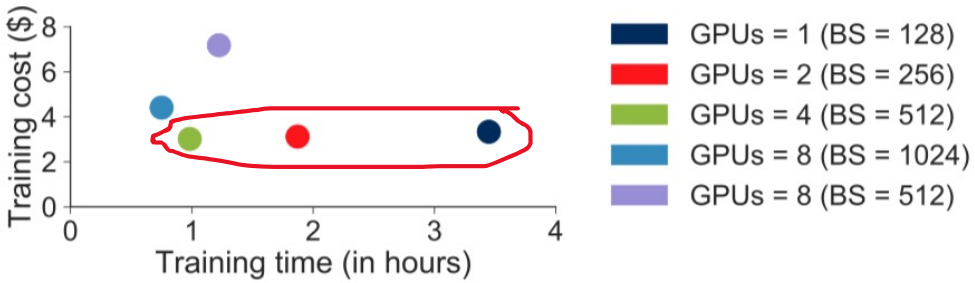


Figure 7: Training cost vs. training time for ResNet56 on the CIFAR10 dataset, using different numbers of GPUs, with an accuracy threshold of 92.5%. The cost of training stays roughly the same, regardless of the number of GPUs used, until 8 GPUs. Training time scales almost linearly with the inverse of the number of GPUs.

- Scaling from 1 to 4 GPUs
  - Training time scales perfectly linearly with the inverse of the number of GPUs used
  - Cost remains constant despite training time going down
- Scaling to 8 GPUs
  - Increase in the cost of training
  - Training time does not decrease enough to counter the doubling in instance cost per unit time (related to scaling efficiency)

Coleman et al. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. NIPS 2017

MLPerf

- https://mlperf.org
- Training: https://mlperf.org/training-overview/#overview
- Inference: https://mlperf.org/inference-overview/#overview

Time to Accuracy (TTA) Metric

- TTA measures time for a system to train to a target, near-state-of-the-art accuracy level on a held-out dataset
- TTA combines both generalization and speed
- Dawnbench was the first multi-entrant benchmark competition to use the TTA metric
- MLPerf benchmark also uses TTA as its primary metric
- Entries compete to achieve target accuracy in the fastest time
  - Imagenet training from 30 mins to less than 2 mins
  - Very large-scale distributed training, with large batch sizes, GPUs, CPUs, TPUs
  - Major companies Google, Intel, NVIDIA compete with optimized solutions with the goal to reduce TTA
  - Entries provide an opportunity to study ML systems optimized heavily for *training performance*

Questions about TTA and DL systems performance

- Is TTA metric stable or do the entries only represent the best result out of many trials?
- Do models optimized for TTA still generalize well ?
  - Are they implicitly adapting to the held-out dataset used in the benchmark through extensive hyperparameter tuning?
- How close are these entries from fully utilizing hardware platforms and what are the computational bottlenecks?

Variability of TTA

Coefficient of Variation % = (standard deviation/mean)x100

Coefficient of Variation of TTA for DAWNBench

Entry name	Coeff. of variation	Frac. of runs
ResNet-50, p3.16xlarge	5.3%	80%
ResNet-50, 4xp3.16xlarge	11.2%	60%
ResNet-50, 8xp3.16xlarge	9.2%	100%
ResNet-50, 16xp3.16xlarge	12.2%	100%
ResNet-50, 1xTPU	4.5%	100%
AmoebaNet-D, 1xTPU	2.3%	100%
ResNet-50, 1/2 TPU Pod	2.5%	100%

Coefficient of Variation of TTA for MLPerf

Entry	Coeff. of variation
ResNet, NVIDIA, 1xDGX-1	6.7%
SSD, NVIDIA, 1xDGX-1	0.5%
SSD, NVIDIA, 8xDGX-1	6.7%
Mask, NVIDIA, 1xDGX-1	3.9%
Mask, NVIDIA, 8xDGX-1	0.8%
GNMT, NVIDIA, 1xDGX-1	0.2%
Transformer, NVIDIA, 1xDGX-1	13.8%

Low coefficient of variation => low variability in TTA across runs
TTA is quite stable across different runs

Coleman et al. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. 2019

Generalization of Optimized Models

- Evaluation on new data for image classification
  - Labeled set of 2,864 images from Flickr
  - Only images posted after January 1st, 2014 were used (no overlap with Imagenet images)
  - Images spanned 886 (out of 1000) classes.

Model	Accuracy (top-5, unseen data)
ResNet-18 (pretrained)	89.5%
ResNet-50 (pretrained)	92.2%
ResNet-152 (pretrained)	93.2%
ResNet-50, 1xTPU	92.6%
ResNet-50, p3.16xlarge	91.9%
ResNet-50, 4xp3.16xlarge	91.3%
ResNet-50, 8xp3.16xlarge	91.5%
ResNet-50, 16xp3.16xlarge	91.3%
AmoebaNet-D, 1xTPU	91.3%

(a) DAWNBENCH submissions, top-5 accuracy. ResNet-50 on p3.16xlarge instances used non-standard optimizations such as progressive resizing.

Model	Accuracy (top-1, unseen data)
ResNet-18 (pretrained)	71.7%
ResNet-50 (pretrained)	77.4%
ResNet-152 (pretrained)	79.4%
ResNet-50, DGX-1	77.6%

(b) MLPERF submission, top-1 accuracy.

- Models optimized for TTA achieve nearly the same accuracy or higher than the pre- trained ResNet-50
- Optimizing for TTA does not sacrifice generalization performance

Coleman et al. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. 2019

System Scale vs TTA vs System Throughput

Model	System scale	BSes	Epochs	Thpt. speedup	TTA speedup
Trans.	1, 24	10k, 492k	2, 6	10.9×	3.6×
GNMT	1, 32	1k, 8.2k	3, 5	10.9×	6.5×
ResNet	1, 80	4k, 16k	63, 82	28.2×	21.6×
SSD	1, 8	1.2k, 2k	49, 55	4.6×	4.1×
Mask R-CNN	1, 8	32, 128	13, 14	4.2×	3.9×

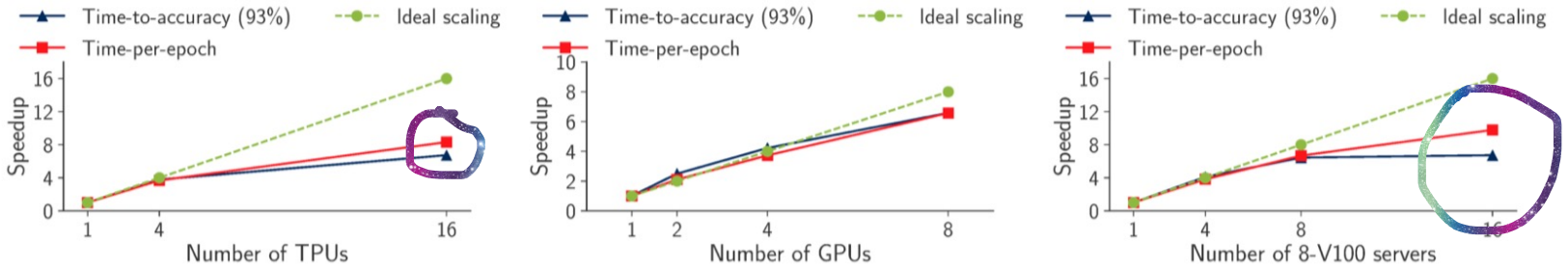
Table 7: Model, system scale (in number of DGX-1s), batch size (BS), number of epochs for convergence, throughput speedup, and TTA speedup. Numbers are given for two system scales per model using official MLPERF entries. As shown, throughput does not directly correlate with TTA and speedups can differ by up to 3× (10.9× vs 3.6× for transformer).

Increasing the batch size to increase throughput may increase the number of epochs required for convergence

System scale does not correlate with Thpt speedup and TTA speedup

Coleman et al. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. 2019

Scaling of TTA with Distributed Training



(a) AmoebaNet across TPUs, TPU pod. **(b)** ResNet-50 within `p3.16xlarge` server. **(c)** ResNet-50 across `p3.16xlarge` servers.

Figure 3: Speedup with respect to a single worker vs. number of workers for three ImageNet models, one on a TPU pod, another on a single `p3.16xlarge` instance with 8 NVIDIA V100 GPUs, and a third on multiple `p3.16xlarge` instances for selected official DAWNBENCH entries. As the number of workers increases, the scaling performance drops off (over 2× gap from ideal scaling).

Within a TPU pod, TPU devices (TPUs) are connected connected to each other over dedicated high-speed networks

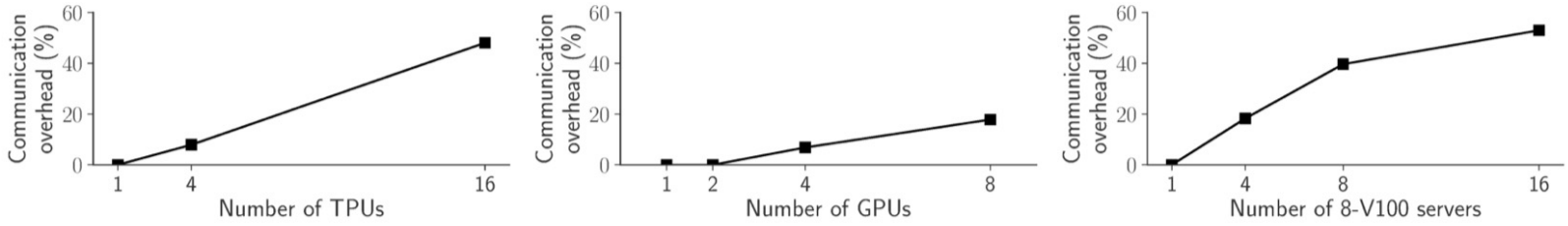
- Both time-per-epoch and TTA scale almost linearly with the number of workers *within* a server
- Both time-per-epoch and TTA do not scale as well for training that spans multiple servers
- TTA scales worse than time-per-epoch
  - Greater number of epochs are needed to converge to the same accuracy target for the larger minibatch size.

Coleman et al. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. 2019

Poor scaling

- Question: Why is the scaling poorer than Goyal et al (Facebook) ?
  - Use of faster V100 GPUs (compared to P100 GPUs)
  - Slower network interfaces (up to 25 Gigabits/second on AWS compared to 50 Gigabits/second in a private Facebook cluster)

Communication Overhead



(a) AmoebaNet across TPUs, TPU pod. **(b)** ResNet-50 within `p3.16xlarge` server. **(c)** ResNet-50 across `p3.16xlarge` servers.

Figure 5: Percentage of time in an epoch spent communicating vs. number of workers for three ImageNet models, one on a TPU pod, another on a single `p3.16xlarge` instance, and a third on multiple `p3.16xlarge` instances. Within a 8-V100 server, communication overhead is low (17.82%), but cross-machine communication is more expensive (53%).

- Communication is a bottleneck in DL at scale
- Communication optimized libraries like Horovod are helpful
- Gradient compression/quantization techniques are helpful; Widespread usage missing; Needs integration with standard frameworks
- Techniques other than data parallelization

Coleman et al. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. 2019

Single Worker Utilization: Roofline Analysis

No network overhead

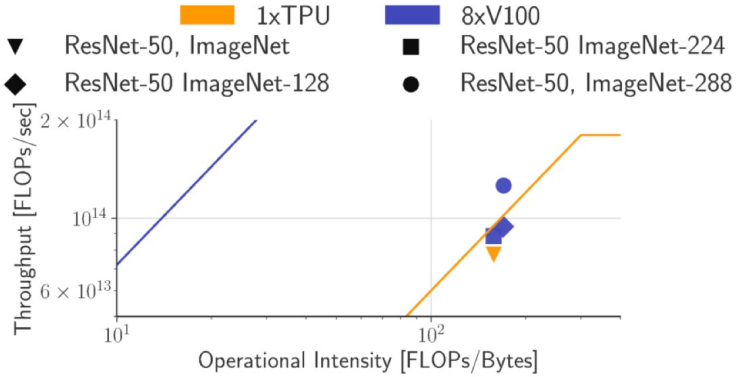


Figure 7: Roofline models for the various DAWNBENCH entries. All of the entries under-utilize the hardware resources, by up to 10×.

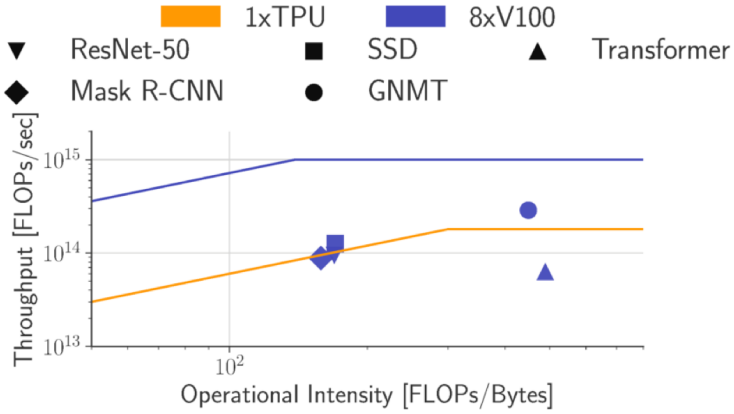
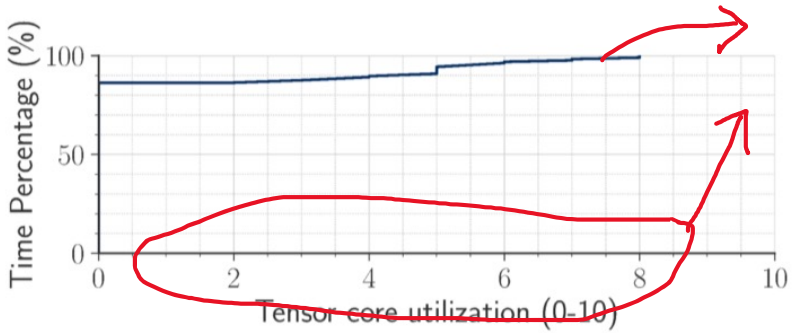


Figure 8: Roofline models for the various MLPERF entries. All of the entries under-utilize the hardware resources, by up to 10×.

All entries analyzed *severely* underutilize the available compute resources – each plotted point achieves a throughput significantly lower than peak device throughput.

Single Worker Utilization: Training Bottleneck

- Tensor core utilization of each GPU kernel in Pytorch



85% of the time is spent on kernels which have negligible tensor core utilization

Figure 9: CDF of tensor core utilization for the `fast.ai` ResNet50 model trained with `fp16` precision submitted to the DAWNBENCH competition. About 85% of time is spent on kernels that don't utilize the NVIDIA Tensor Cores *at all*, and no kernel achieves full utilization of the Tensor Core units.

Coleman et al. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. 2019

Single Worker Utilization: Kernel throughput

only achieves a throughput of 7.6 Teraflops, compared to peak device throughput of 15.7 Teraflops

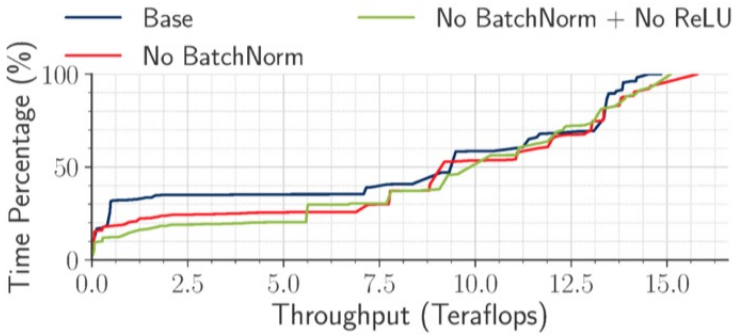


Figure 11: CDF of per-kernel throughput for ResNet50 models trained with `fp32` precision. The CDF is computed by percentage of time spent executing each GPU kernel. A standard ResNet-50 model spends about 40% time in low-throughput kernels (< 6 Teraflops). Removing the BatchNorm layer from the ResNet50 model decreases the percentage of time in low-throughput kernels to about 30%; removing the ReLU layers decreases this further.

- Memory-bound kernels like BatchNorm and ReLU take a significant percentage of total runtime.
- Optimizations like loop and kernel fusion can help reduce the impact of memory-bound kernels by reducing the number of DRAM reads and writes made

Coleman et al. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. 2019

Loop Fusion

```
1  // BatchNorm.
2  for (int i = 0; i < n; i++) {
3      y[i] = gamma * ((x[i] - mu) / sigma) + beta;
4  }
5  // ReLU.
6  for (int i = 0; i < n; i++) {
7      z[i] = max(y[i], 0);
8  }
```