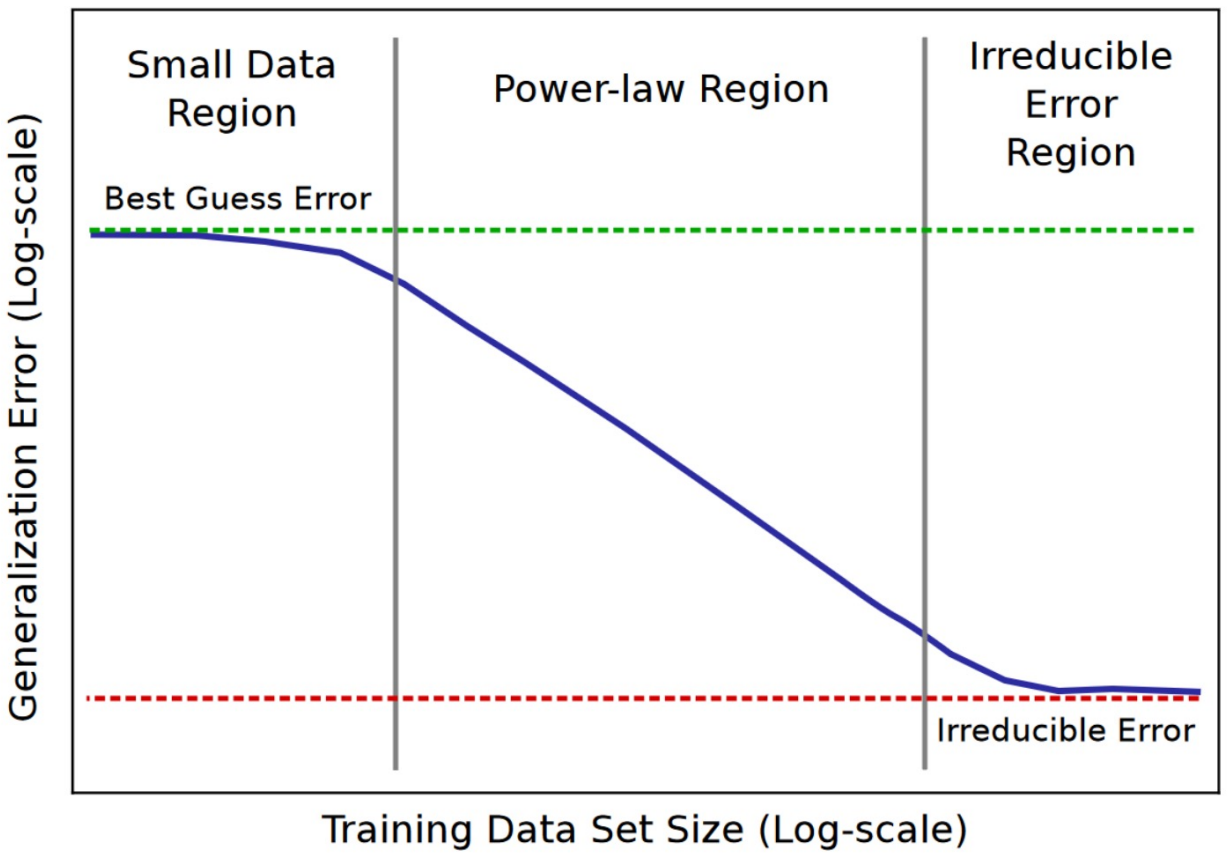
FUNDAMENTALS OF DEEP LEARNING FOR MULTIPLE GPUS

CONTEXT: WHY USE MULTIPLE GPUS?

IMPROVING ACCURACY

- Over the last few years, the largest trained model has increased in size by over 1000x, from a few hundred million parameters to half a trillion (and beyond) parameters
- These increases on model size were driving by increasing datasets
- They also led to in model quality, suggesting that this trend will continue
- Indeed researchers have identified three main eras of AI computing, requiring different FLOPs

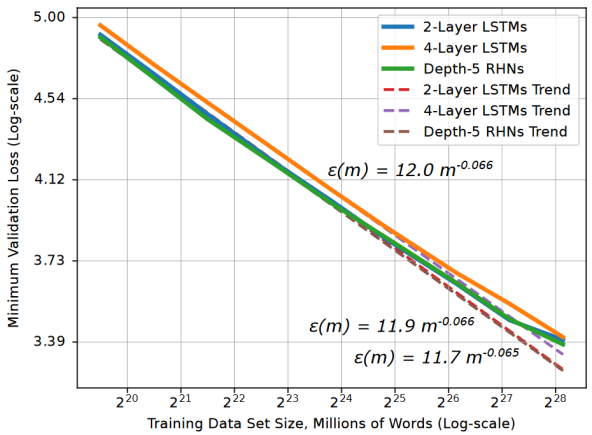
DATASET SIZE VS QUALITY

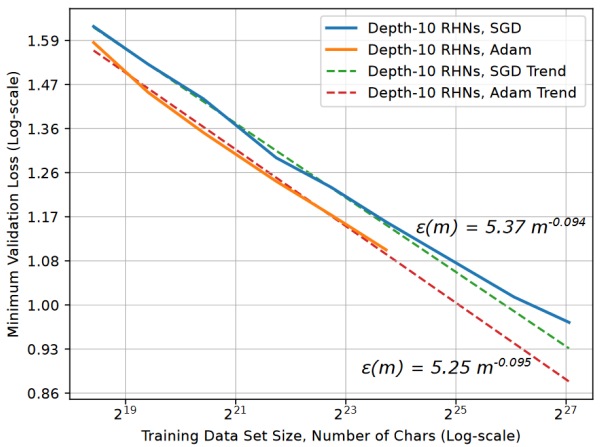


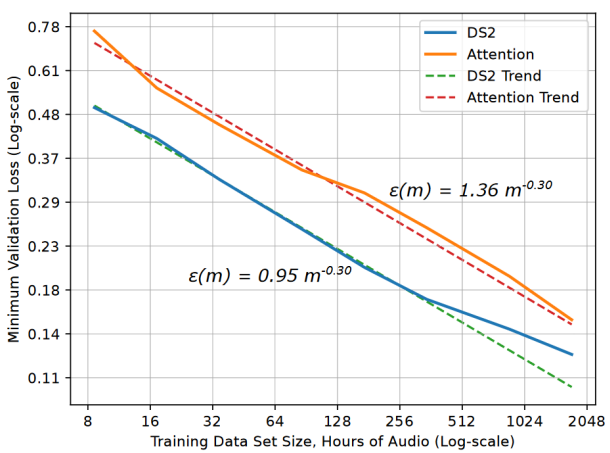
Hestness, J., et al. (2017). Deep Learning Scaling is Predictable, Empirically. arXiv: 1712.00409

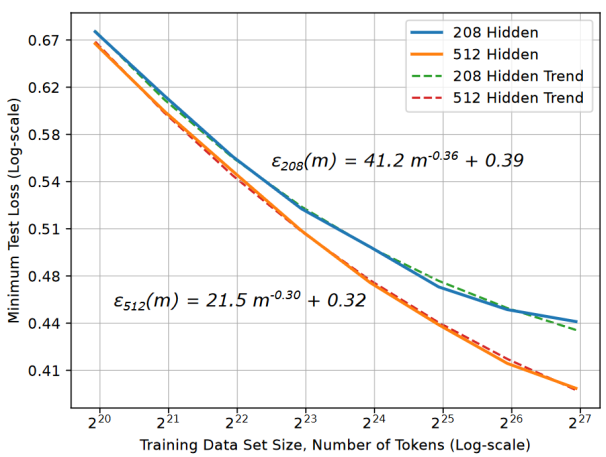
DATASET SIZE VS QUALITY

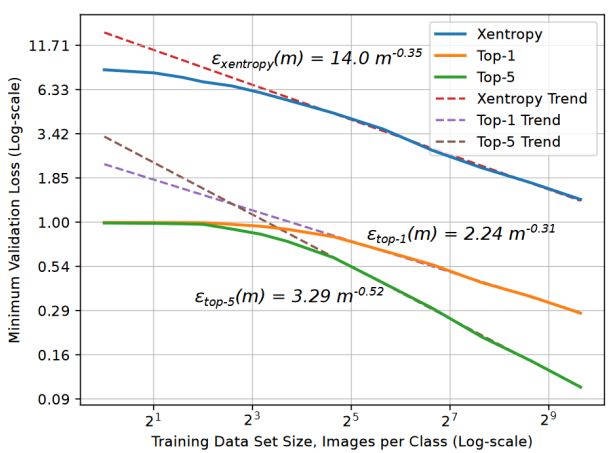
- Trend across different domains
  - Translation
  - Language Models
  - Character Language Models
  - Image Classification
  - Attention Speech Models





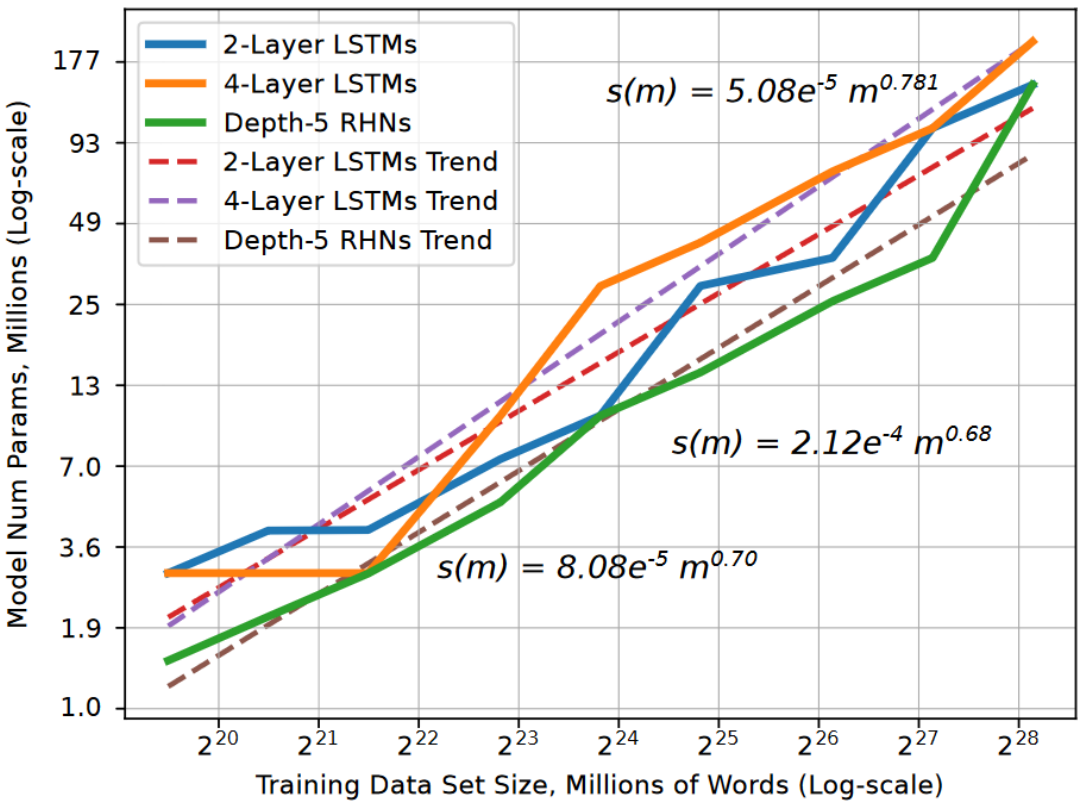






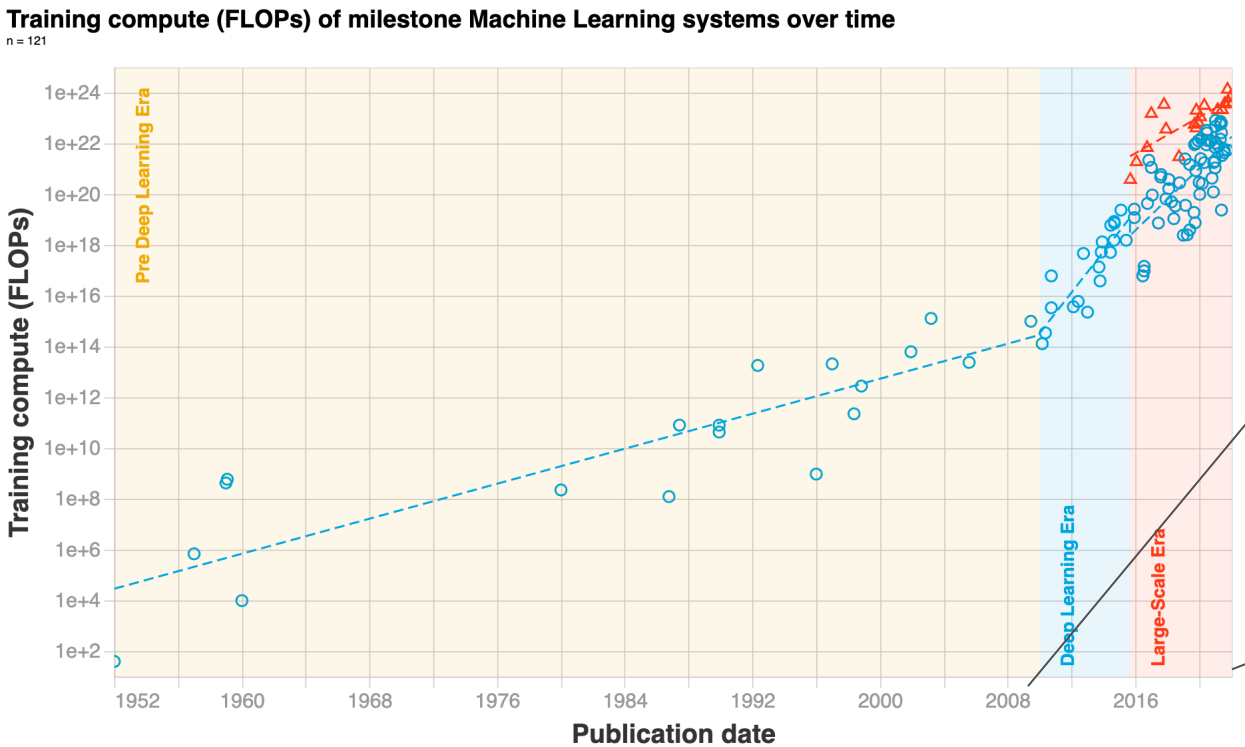
Hestness, J., et al. (2017). Deep Learning Scaling is Predictable, Empirically. arXiv: 1712.00409

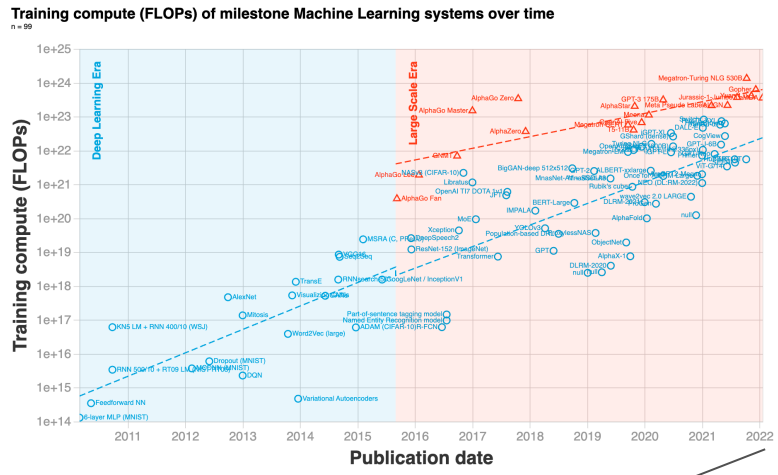
SUBLINEAR GROWTH OF MODELS TOO



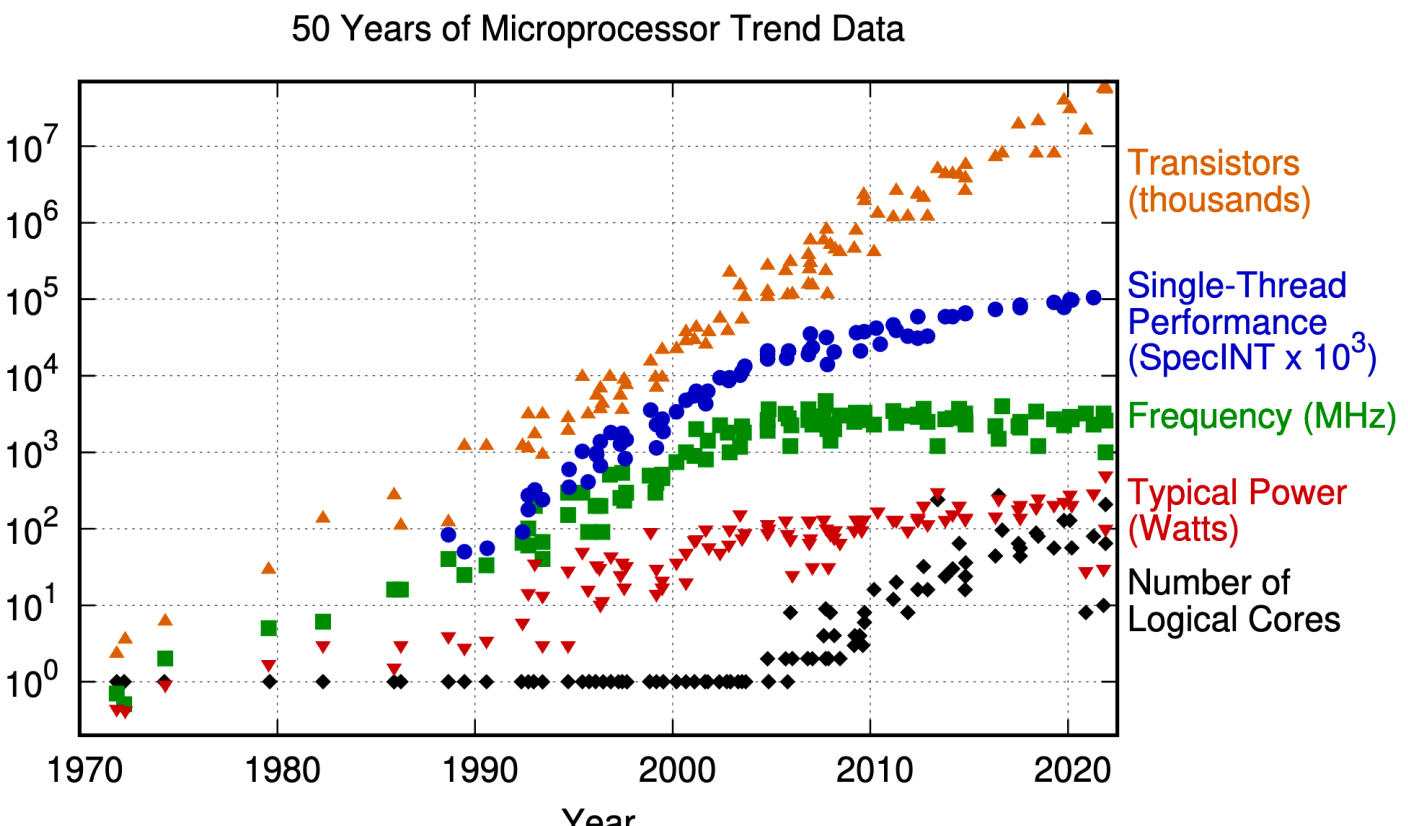
Hestness, J., et al. (2017). Deep Learning Scaling is Predictable, Empirically. arXiv: 1712.00409

ML COMPUTATIONAL DEMANDS OVER TIME



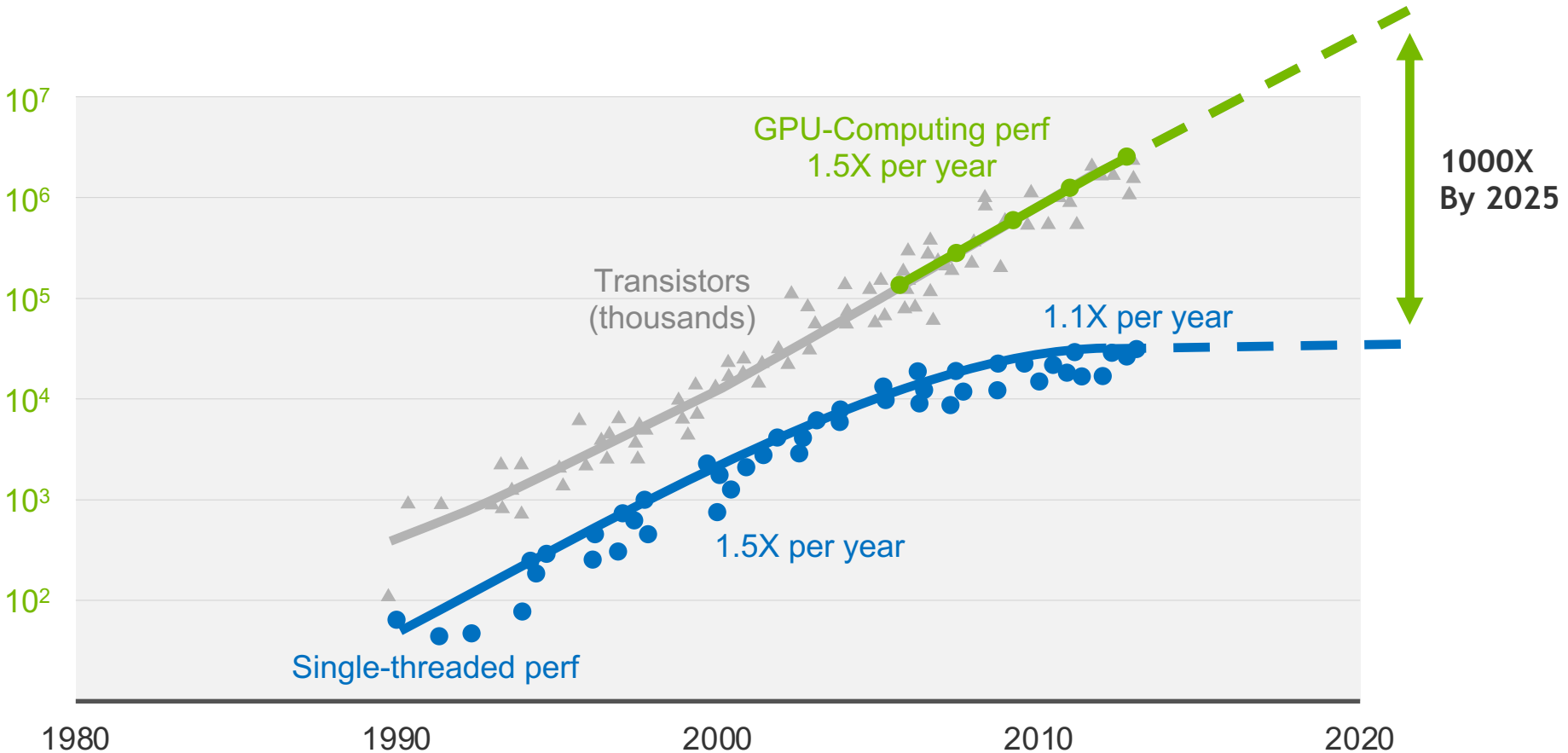


TRENDS IN COMPUTATIONAL POWER



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

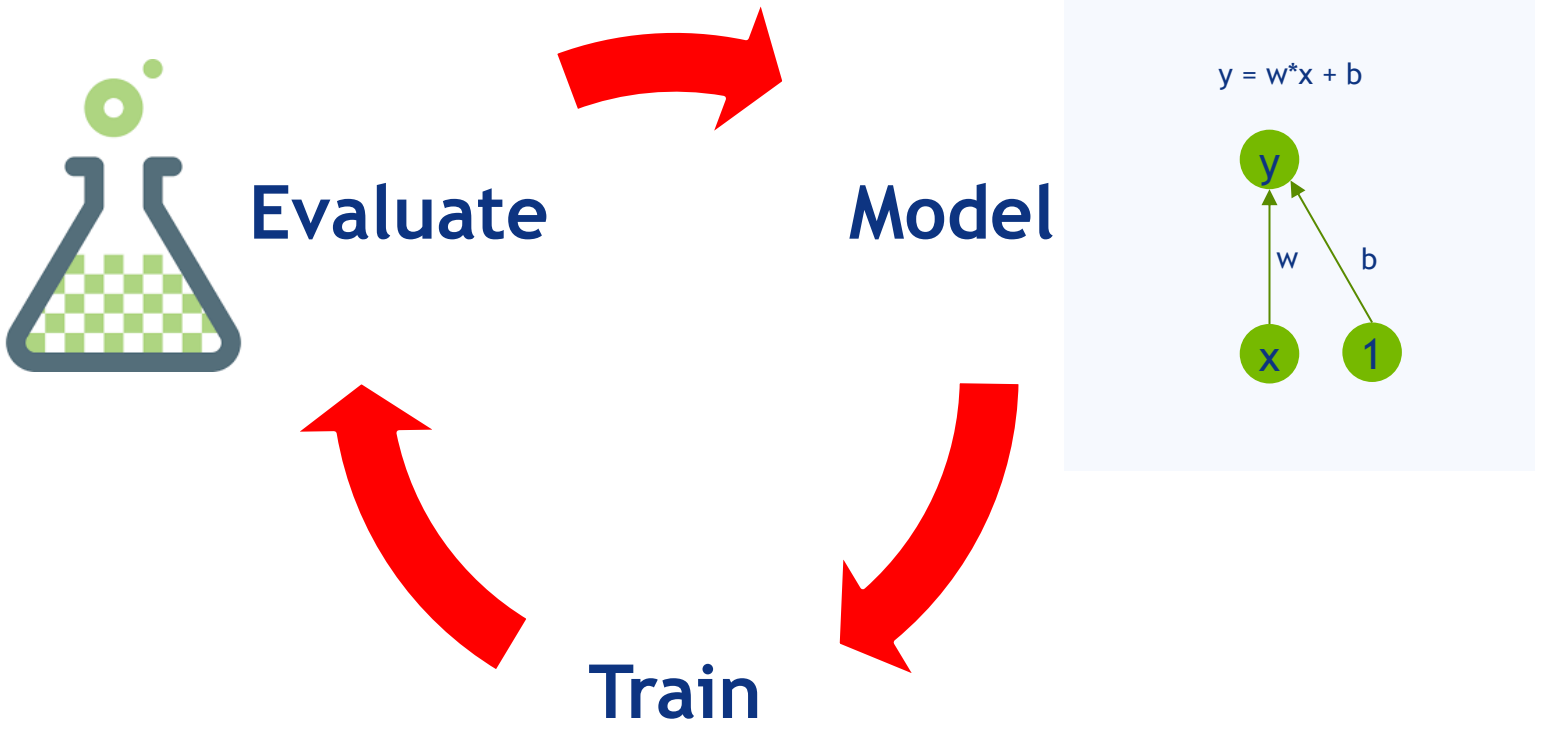
TRENDS IN COMPUTATIONAL POWER



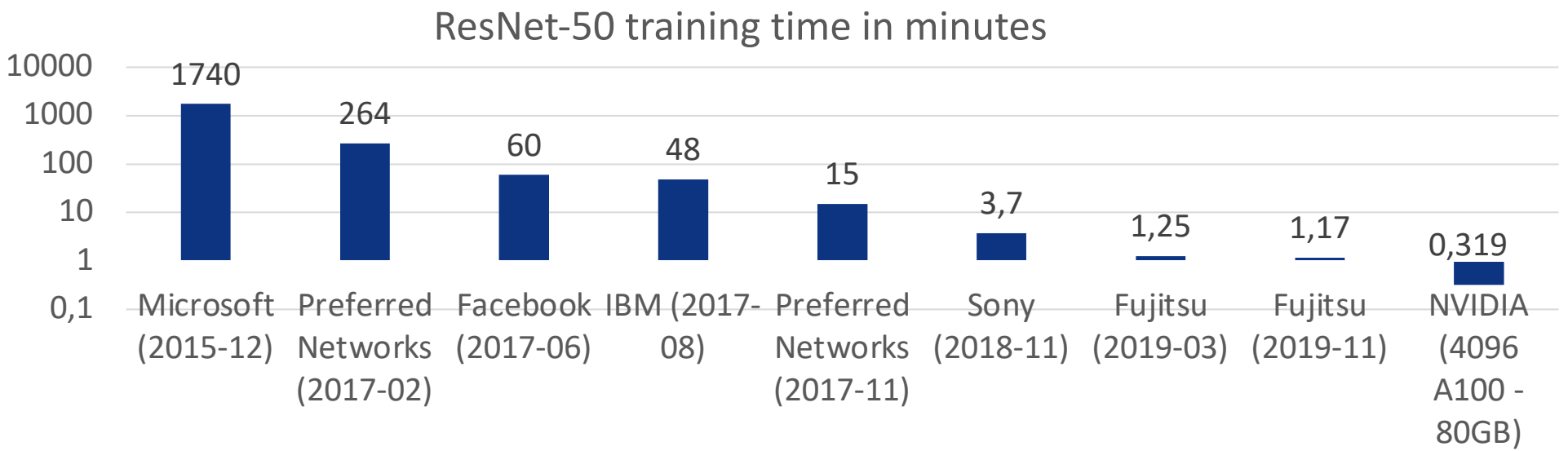
For some workloads: O(100 years) on a dual CPU server O(30 Days) DGX H100

SCIENTIFIC METHOD

What does it take for leveraging ML?



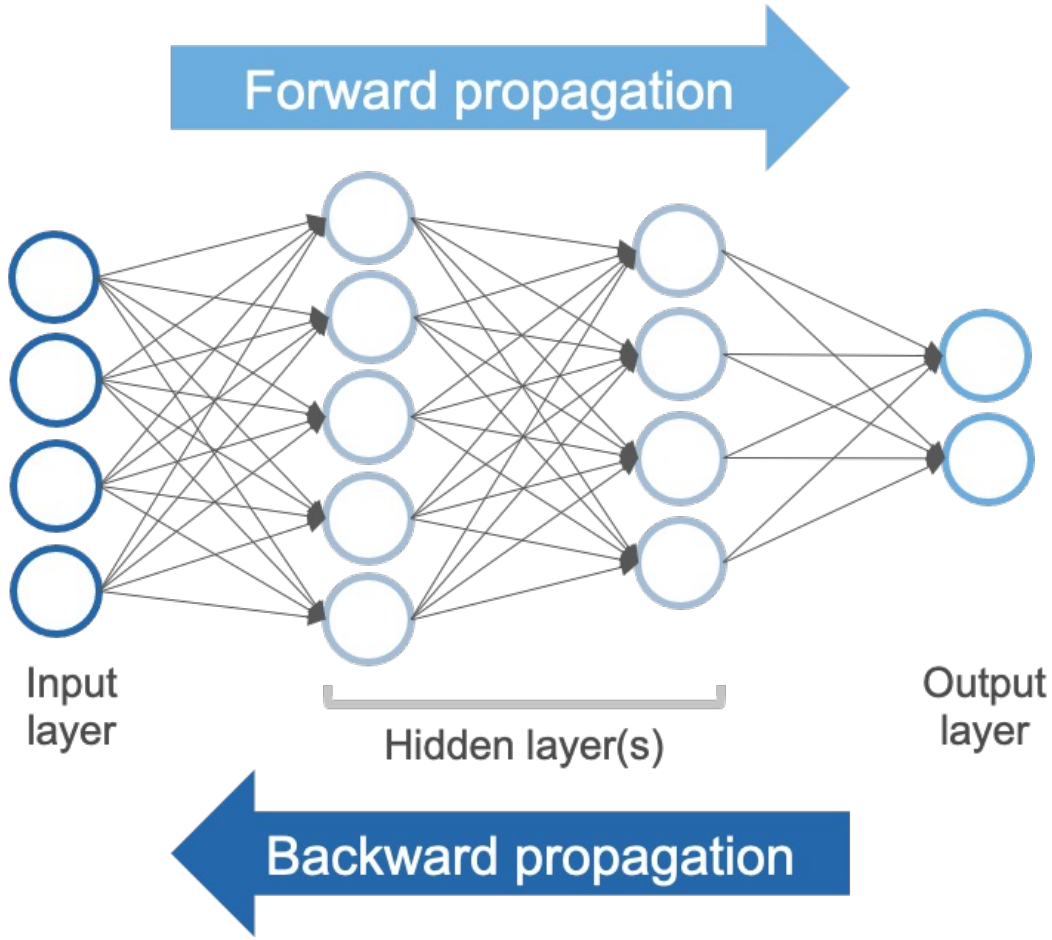
SHORT ITERATION TIME IS FUNDAMENTAL FOR SUCCESS



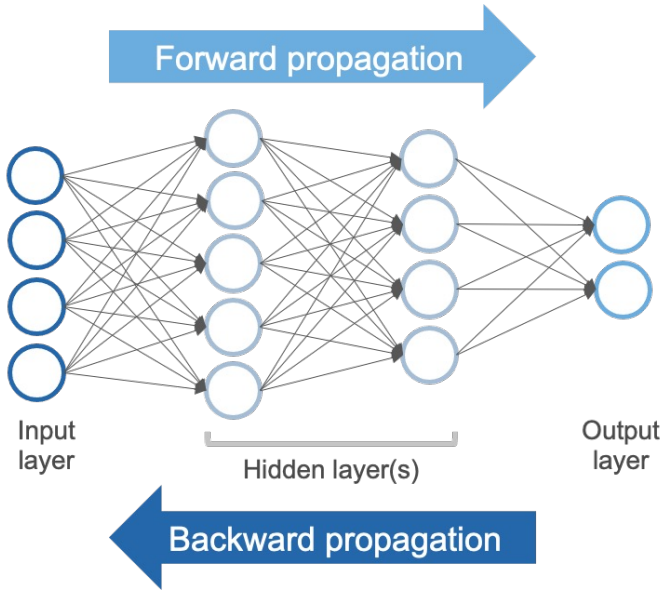
IT ALL STARTS WITH TRAINING …

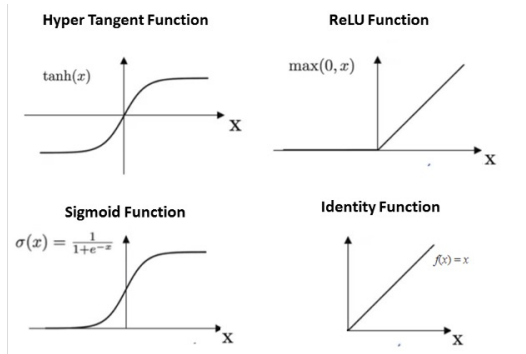
BRIEF REVIEW OF NEURAL NETWORKS TRAINING

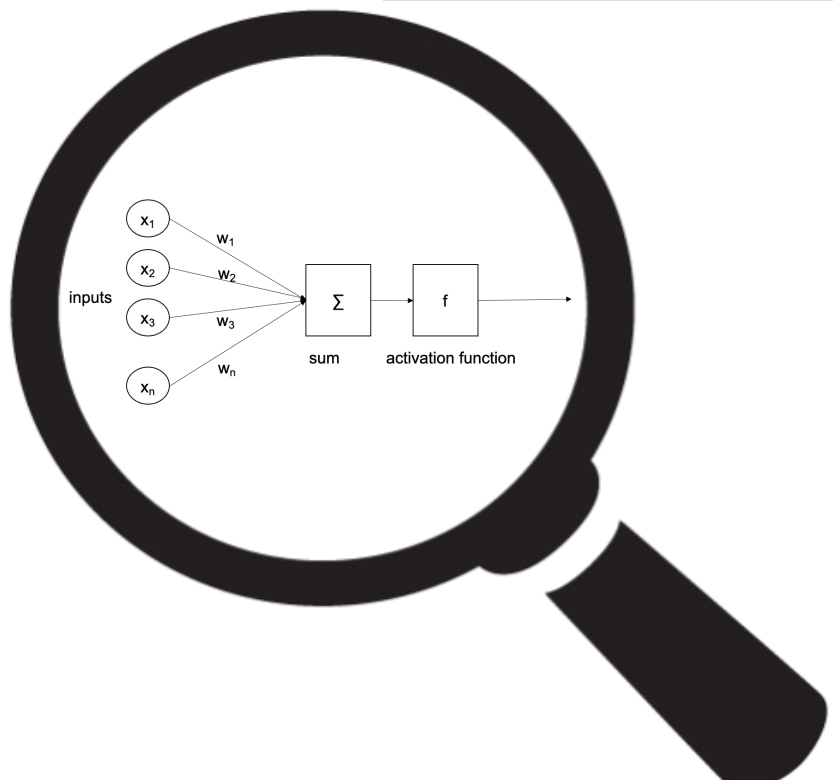
MACHINE LEARNING WITH (DEEP) NEURAL NETWORKS



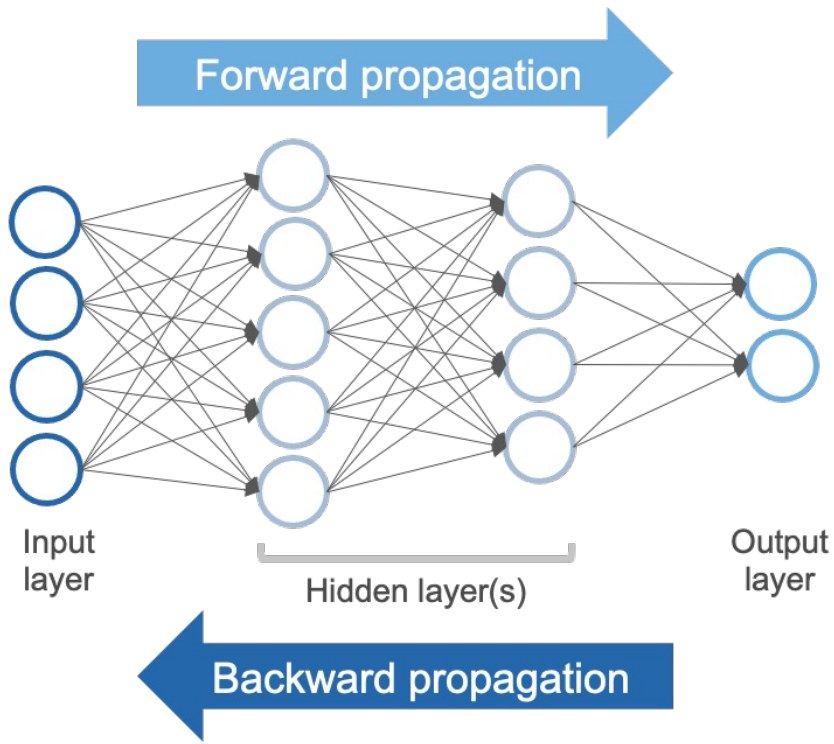
MACHINE LEARNING WITH (DEEP) NEURAL NETWORKS







MACHINE LEARNING WITH (DEEP) NEURAL NETWORKS



Loss function

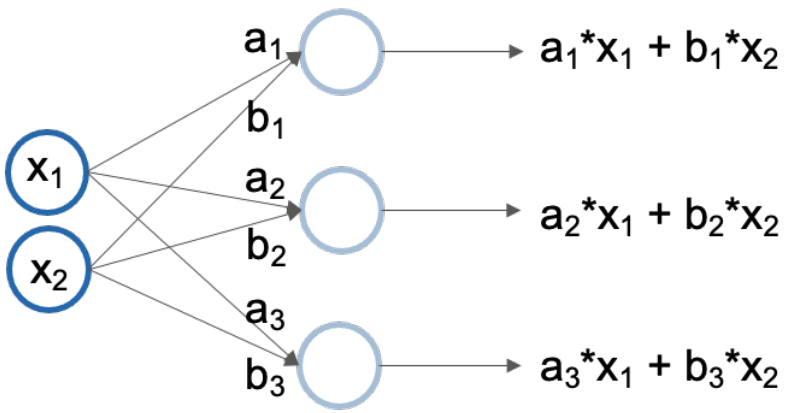
$$l = MSE = \frac{\sum_{i=1}^{n}(y_i - y_i^p)^2}{n}$$

Optimizer SGD

$$\theta_t \leftarrow \theta_{t-1} - n_t * g(\theta_t; B_t)$$

$$g(\theta_t; B_t) = \frac{1}{|B_t|}\sum_{z \in B_t} \nabla l(\theta_t; z)$$

FORWARD OPERATIONS



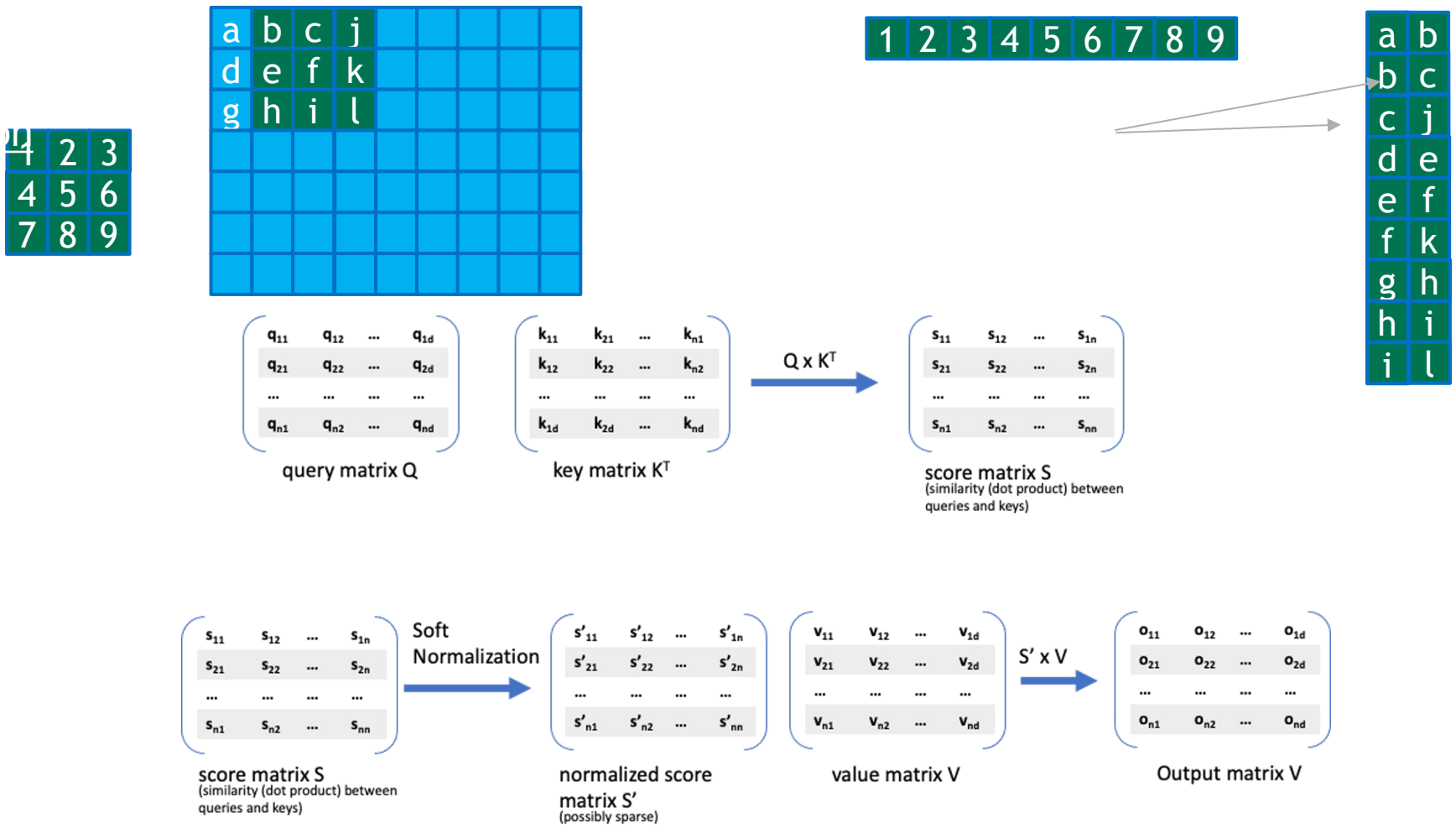
Matrix Multiplication Operation

$$\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \\ a_3 & b_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} a_1*x_1 + b_1*x_2 \\ a_2*x_1 + b_2*x_2 \\ a_3*x_1 + b_3*x_2 \end{pmatrix}$$

$$\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \\ a_3 & b_3 \end{pmatrix} \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \end{pmatrix} = \begin{pmatrix} a_1*x_1 + b_1*x_2 & a_1*y_1 + b_1*y_2 \\ a_2*x_1 + b_2*x_2 & a_2*y_1 + b_2*y_2 \\ a_3*x_1 + b_3*x_2 & a_3*y_1 + b_3*y_2 \end{pmatrix}$$

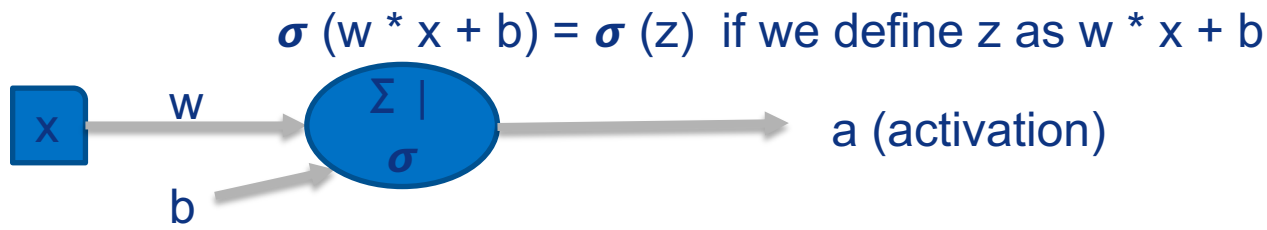
batch of two inputs

FORWARD OPERATIONS



BACK PROPAGATION AND GRADIENT DESCENT

Linear regression y = w * x + b (I.e., a NN of a single neuron, and identity, f(x) = x, as activation function)



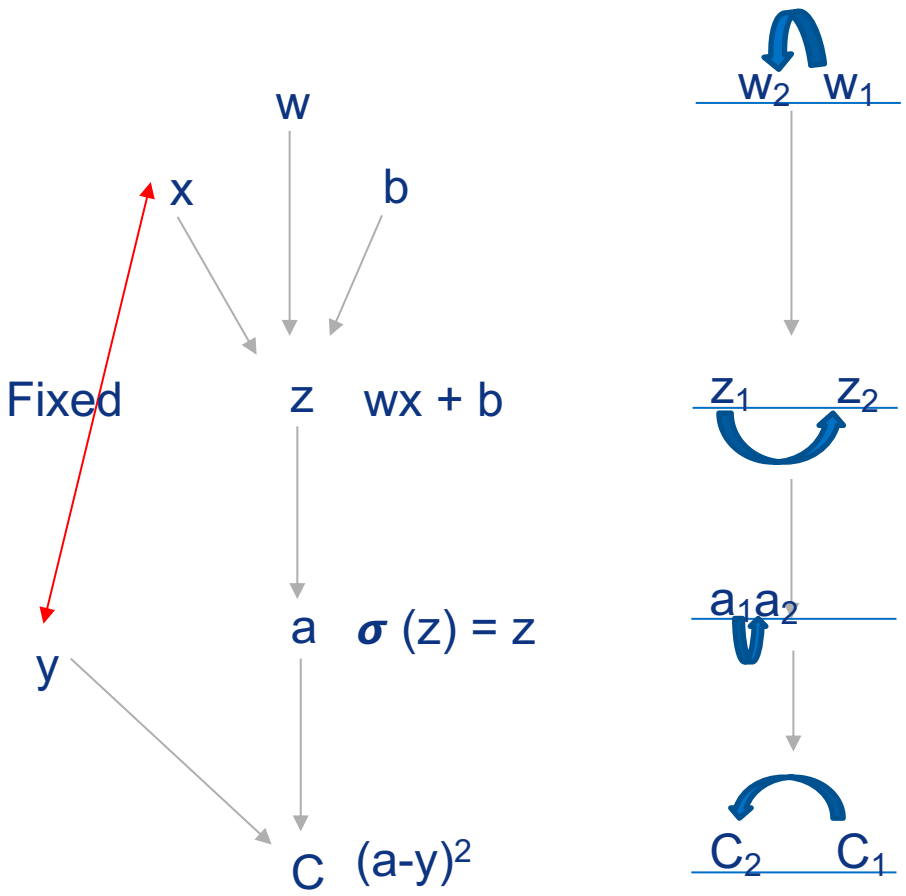
Loss function defined as C = (a – y)2

How does C change with w and b variations?

- compute the ratio at with C changes with changes in w and b
- use this ratio to modify then w and b in order to move C towards a minimum

COMPUTING THE GRADIENT

Gradient with a single input, that generates prediction a



$$\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w}\frac{\partial a}{\partial z}\frac{\partial C}{\partial a} = 2x(a-y)$$

$$\frac{\partial z}{\partial w} = x$$

$$\frac{\partial a}{\partial z} = 1$$

$$\frac{\partial C}{\partial a} = 2(a-y)$$

$$\frac{\partial C}{\partial b} = \frac{\partial z}{\partial b}\frac{\partial a}{\partial z}\frac{\partial C}{\partial a} = 2(a-y)$$

$$\frac{\partial z}{\partial b} = 1$$

$$\frac{\partial a}{\partial z} = 1$$

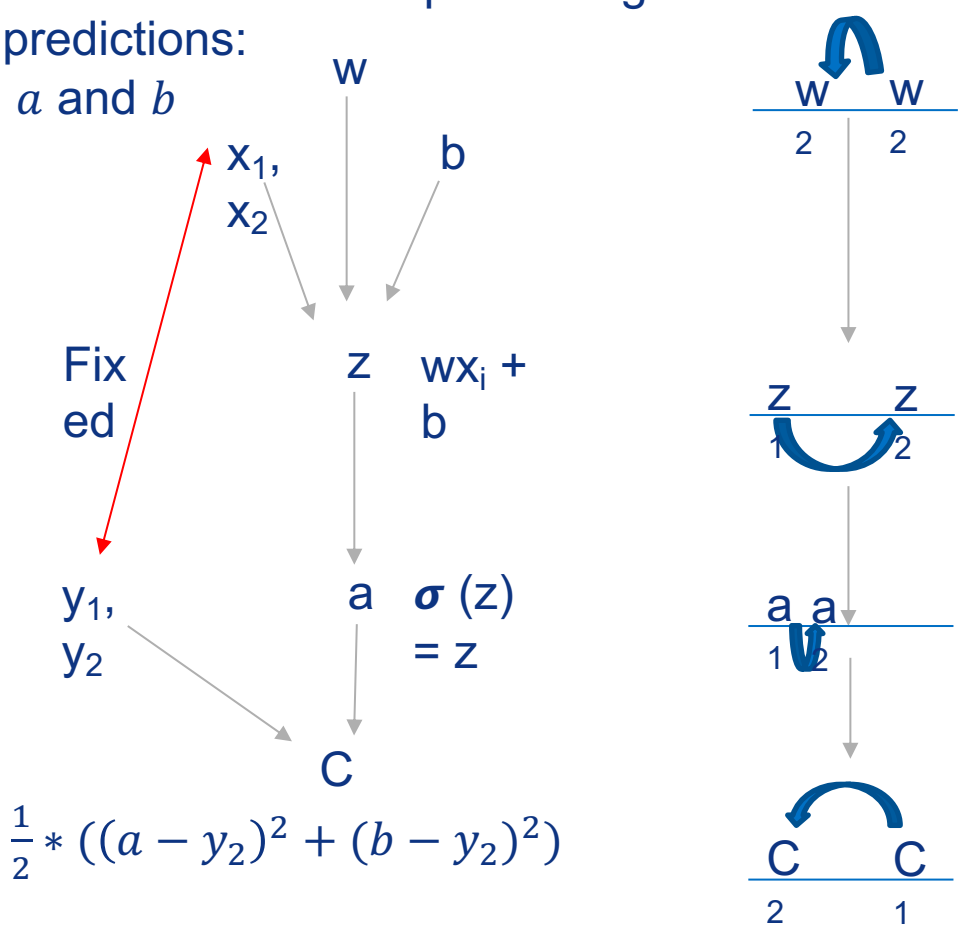
$$\frac{\partial C}{\partial a} = 2(a-y)$$

Gradient Vector

$$\begin{pmatrix} \frac{\partial C}{\partial w} \\ \frac{\partial C}{\partial b} \end{pmatrix} = \begin{pmatrix} 2x(a-y) \\ 2(a-y) \end{pmatrix}$$

COMPUTING THE GRADIENT

Gradient with two inputs that generates predictions:
a and b



$$\frac{1}{2} * ((a - y_2)^2 + (b - y_2)^2)$$

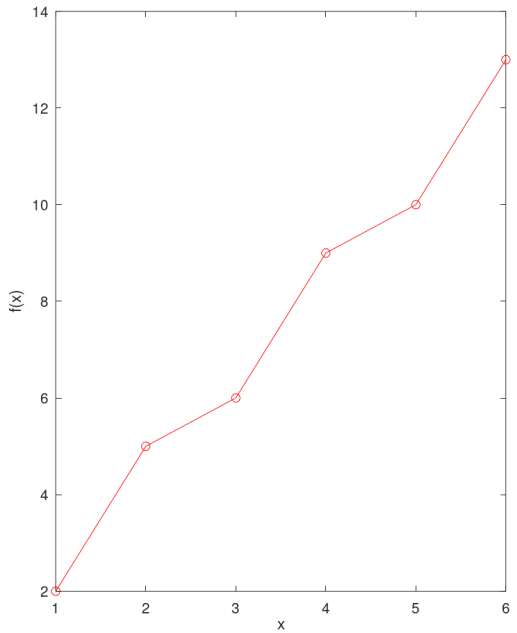
$$\frac{\partial C}{\partial w} = \frac{1}{2} * (2x(a - y_1) + 2x(b - y_2))$$

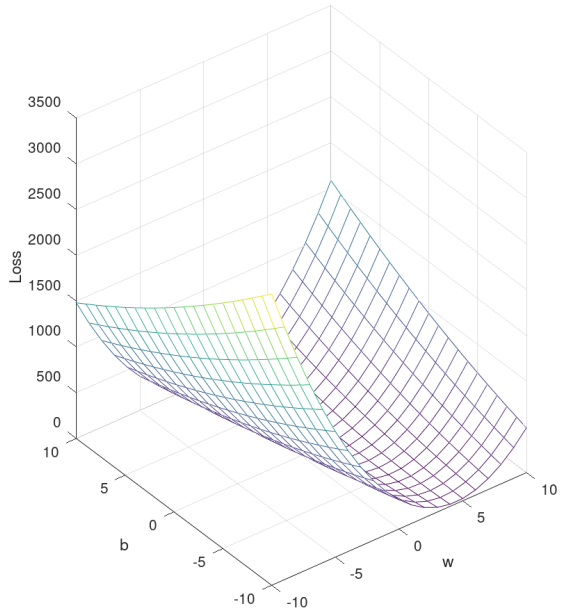
$$\frac{\partial C}{\partial b} = \frac{1}{2} * (2(a - y_1) + 2(b - y_2))$$

Gradient Vector

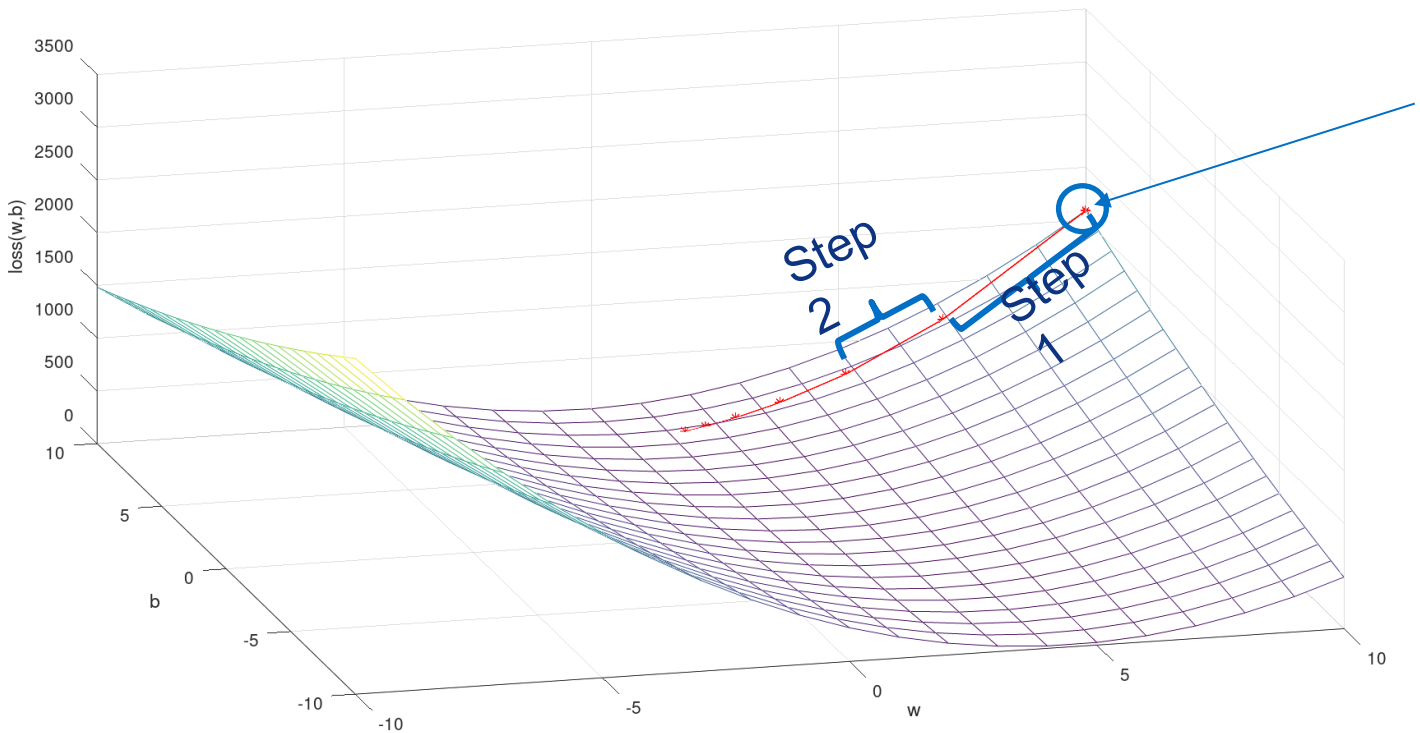
$$\begin{pmatrix} \frac{\partial C}{\partial w} \\ \frac{\partial C}{\partial b} \end{pmatrix} = \begin{pmatrix} \frac{1}{2} * (2x(a - y_1) + 2x(b - y_2)) \\ \frac{1}{2} * (2(a - y_1) + 2(b - y_2)) \end{pmatrix}$$

BACK PROPAGATION AND GRADIENT DESCENT

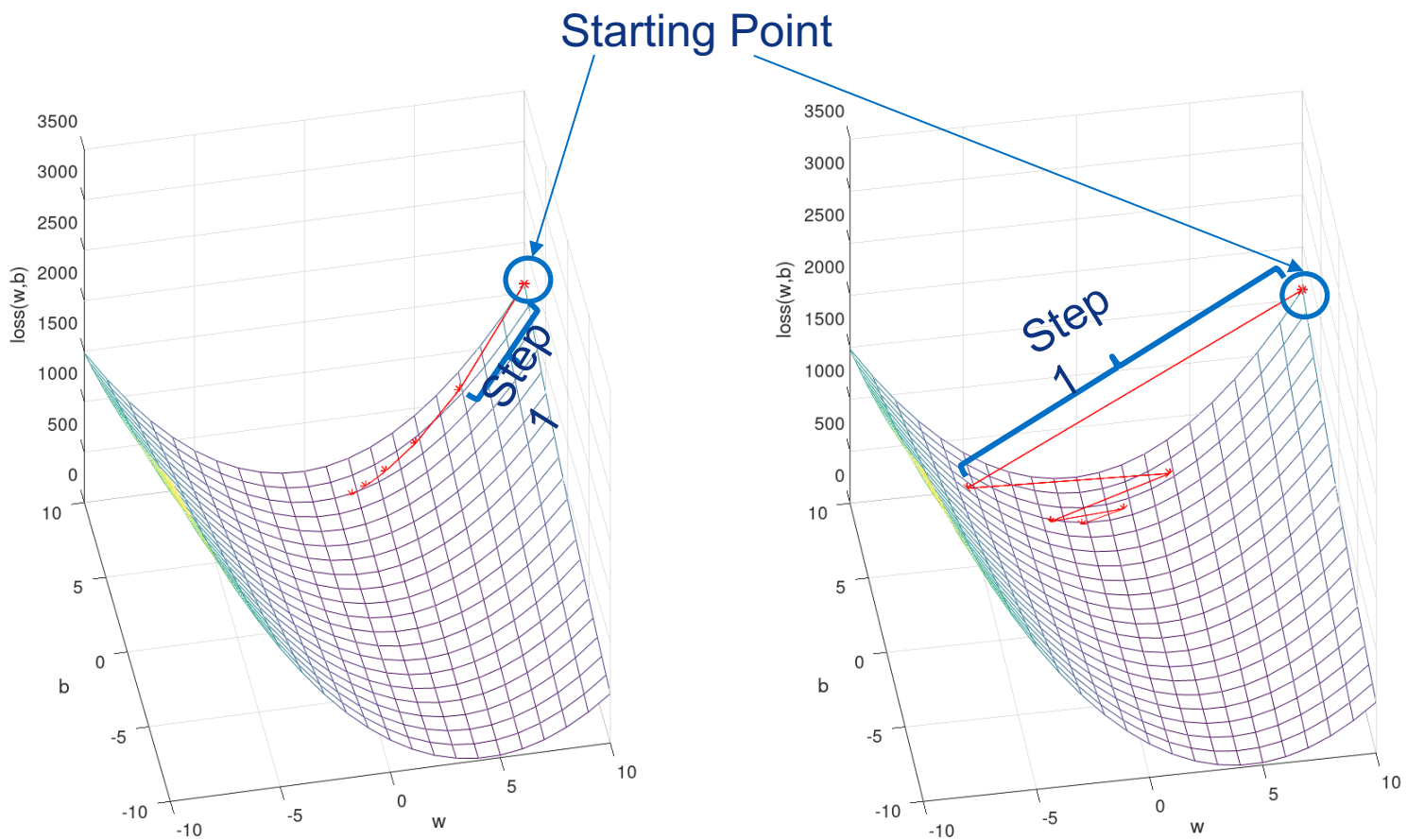




BACK PROPAGATION AND GRADIENT DESCENT



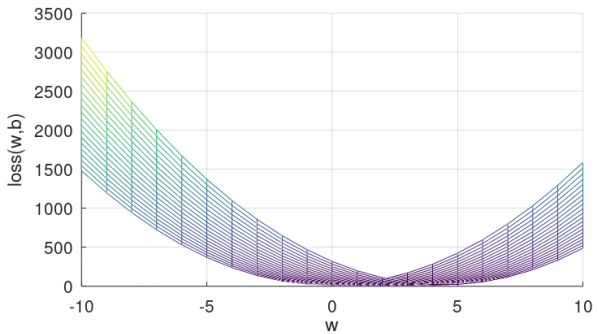
BACK PROPAGATION AND GRADIENT DESCENT

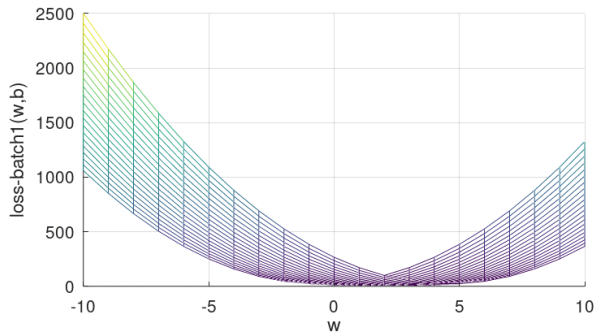


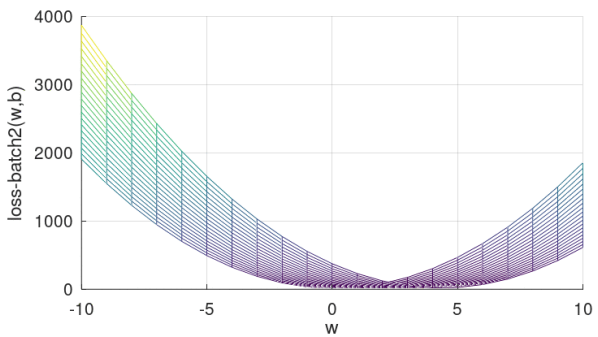
smaller learning rate

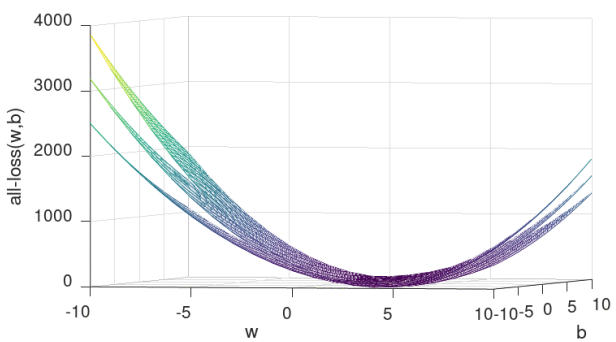
larger learning rate

BACK PROPAGATION AND GRADIENT DESCENT

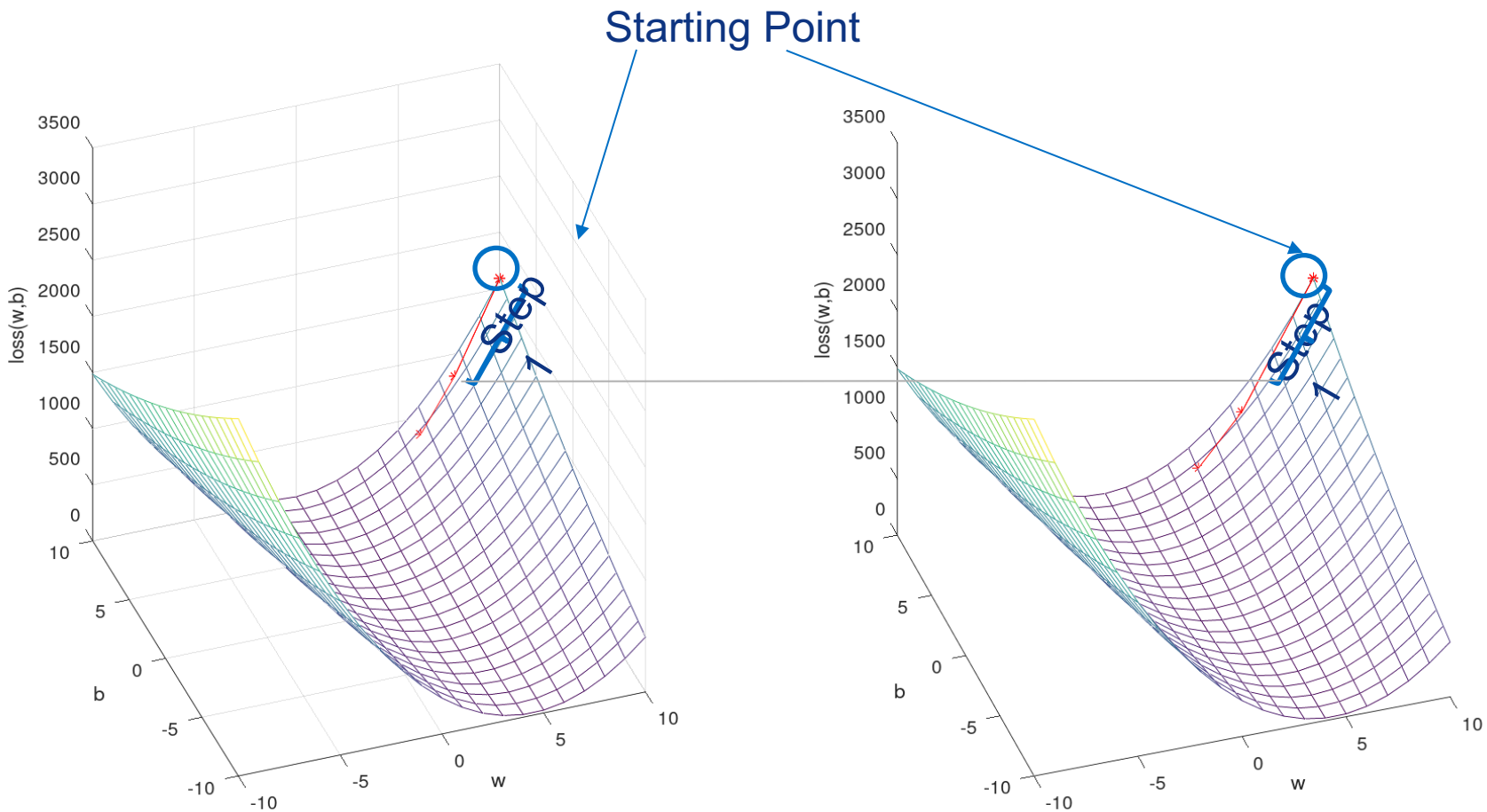








BACK PROPAGATION AND GRADIENT DESCENT



gradient with batch 1

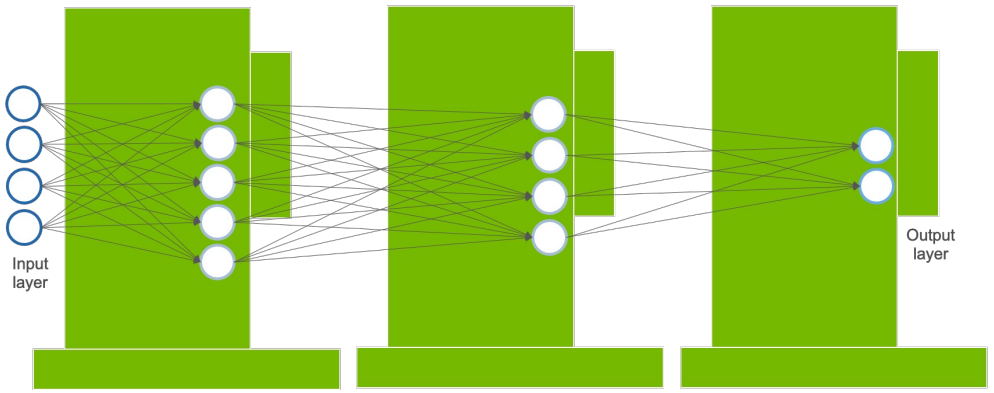
gradient with batch 2

BACK PROPAGATION AND GRADIENT DESCENT

Batch size implications

- Smaller batches imply more steps per epoch:
  - More updates to weights --> More updates to the net
- Smaller batches do not imply larger/smaller gradients

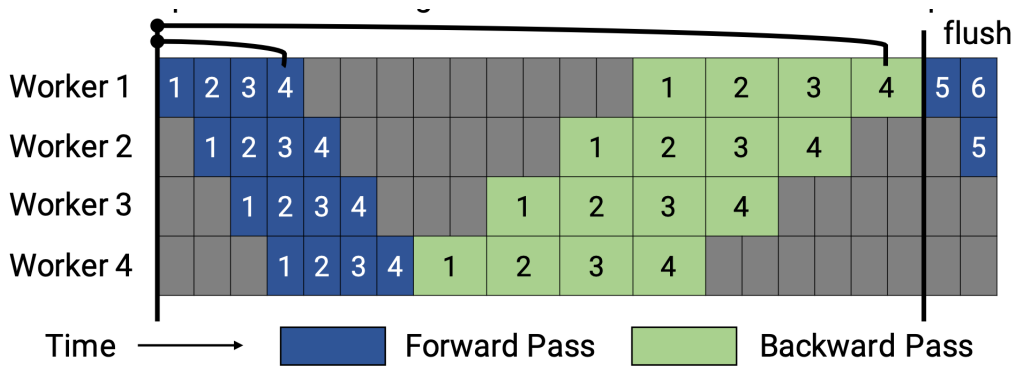
PARALLEL/DISTRIBUTED ML TRAINING



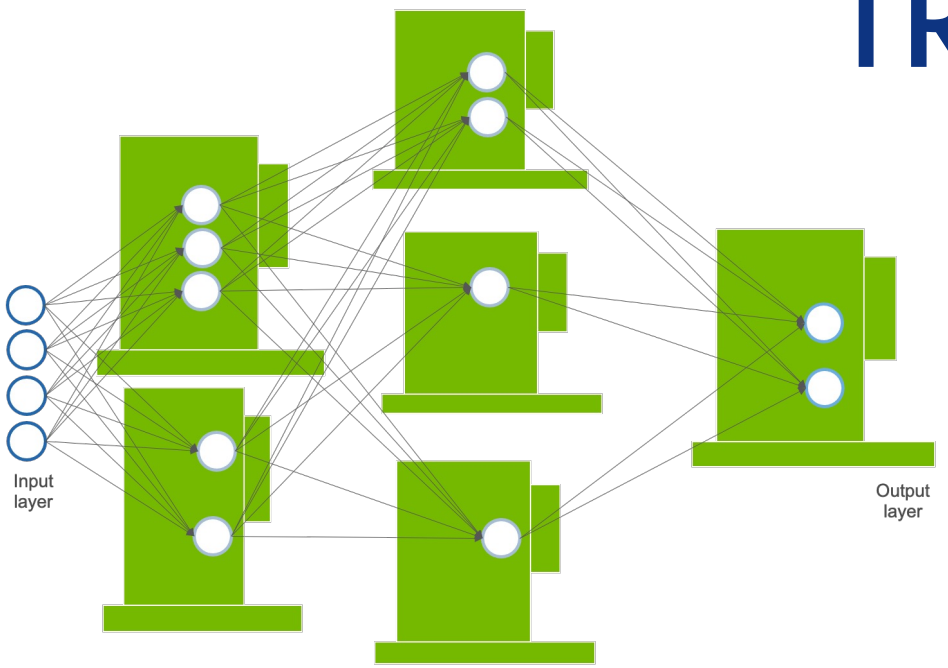
1. Model Parallelism: Memory usage and computation of a model distributed across devices
Two main variants:
   a) Pipeline parallelism
   b) Tensor parallelism

Pipeline Model

- Complete layer per device
  - Weights stay within device
- Activations are communicated between GPUs
- Non efficient implementations may lead to inefficient usage of resources
  - Research area



PARALLEL/DISTRIBUTED ML TRAINING

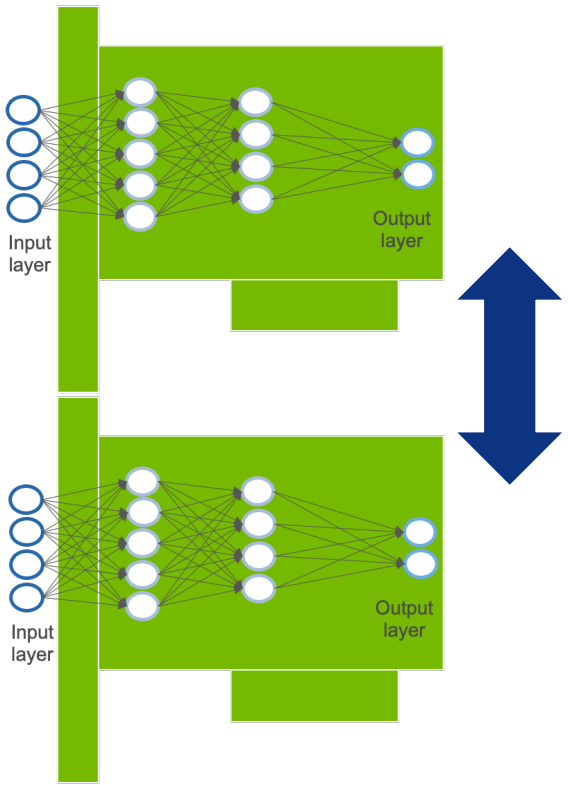


1. Model Parallelism: Memory usage and computation of a model distributed across devices
Two main variants:
   a) Pipeline parallelism
   b) Tensor parallelism

Tensor Parallelism

- Tensor operations (e.g., computing a layer output) distributed across device
  - Allows larger, more computationally expensive models
- Activations are communicated between GPUs
- Further points for inefficiencies
  - A device might depend on the activations computed by more than one device

PARALLEL/DISTRIBUTED ML TRAINING



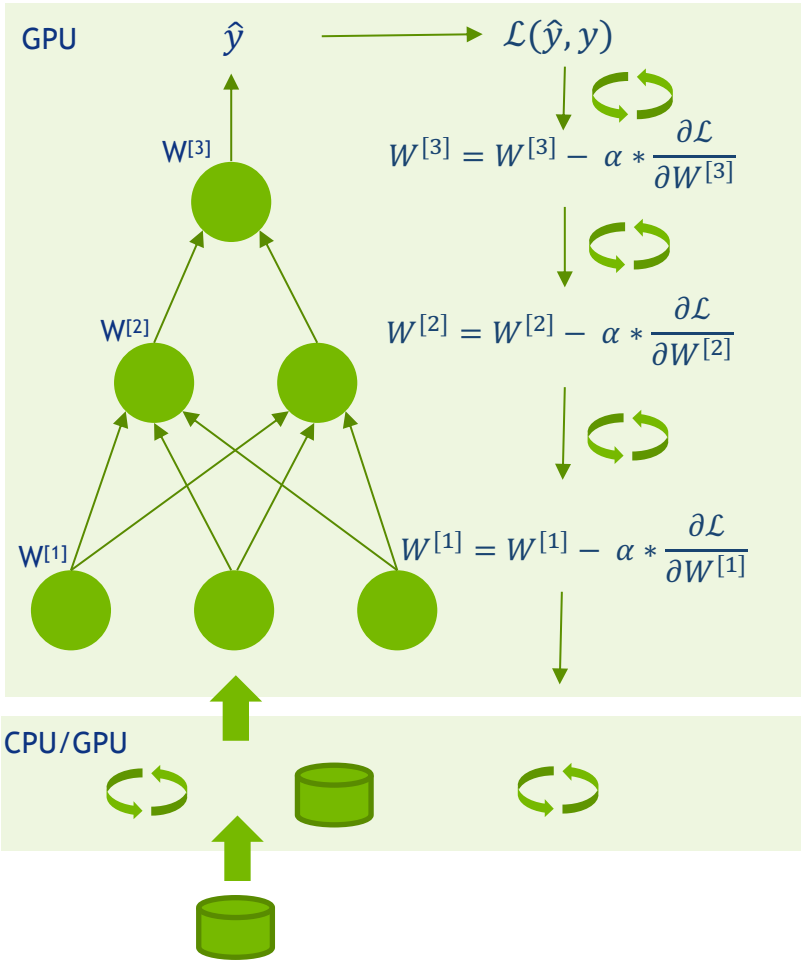
2. Data Parallelism: Training mini-batch is split across devices

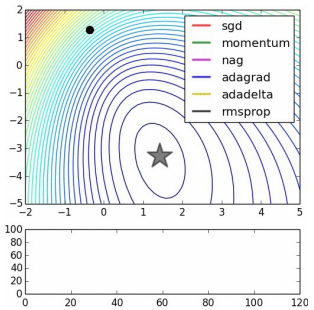
- Model must fit into the memory of a single device
- Weights are the same in each device
  - Gradients are communicated across all devices

  (all-to-all)

TRAINING A NEURAL NETWORK

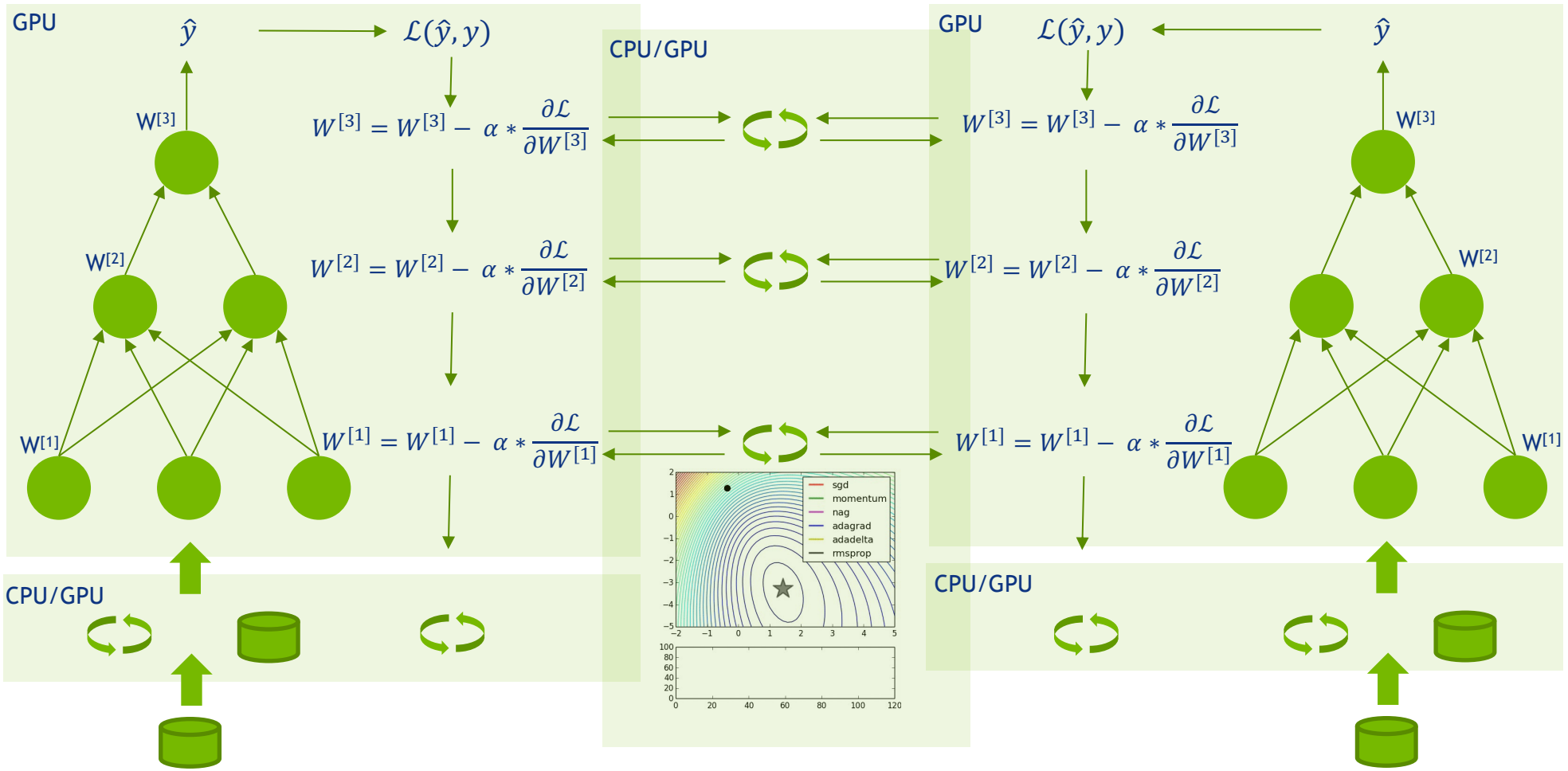
Single GPU





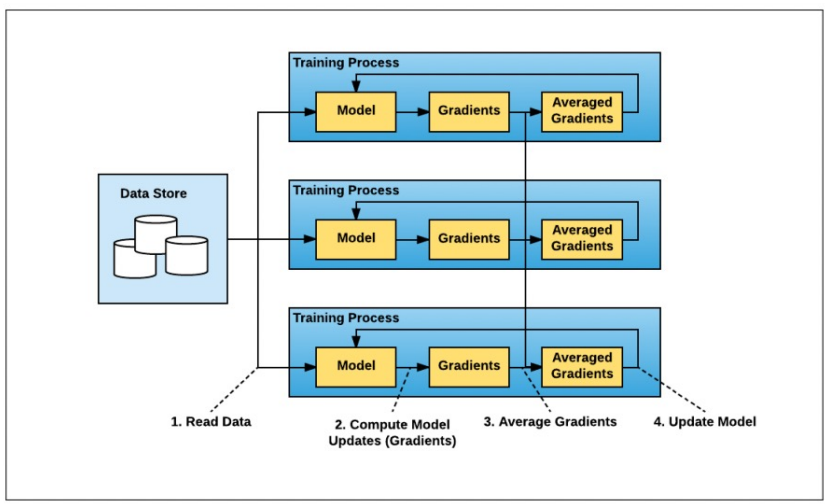
1. Read the data
2. Transport the data
3. Pre-process the data
4. Queue the data
5. Transport the data
6. Calculate activations for layer one
7. Calculate activations for layer two
8. Calculate the output
9. Calculate the loss
10. Backpropagate through layer three
11. Backpropagate through layer two
12. Backpropagate through layer one
13. Execute optimization step
14. Update the weights
15. Return control

TRAINING A NEURAL NETWORK



DATA PARALLELISM

- Traditionally, ML developers have scaled up models through **data parallelism**
  - which splits up your data and feeds it to horizontally-scaled model instances
- This **scales up training but** has an important limitation: it **requires that the model fits within a single hardware device**



1. Run several copies of the training. Each copy:
   1. reads a part of the data
   2. runs it through the model
   3. computes model updates (gradients)
2. Average the gradients from all the copies
3. Update the model
4. Repeat from Step 1a

DATA PARALLELISM
Implementations

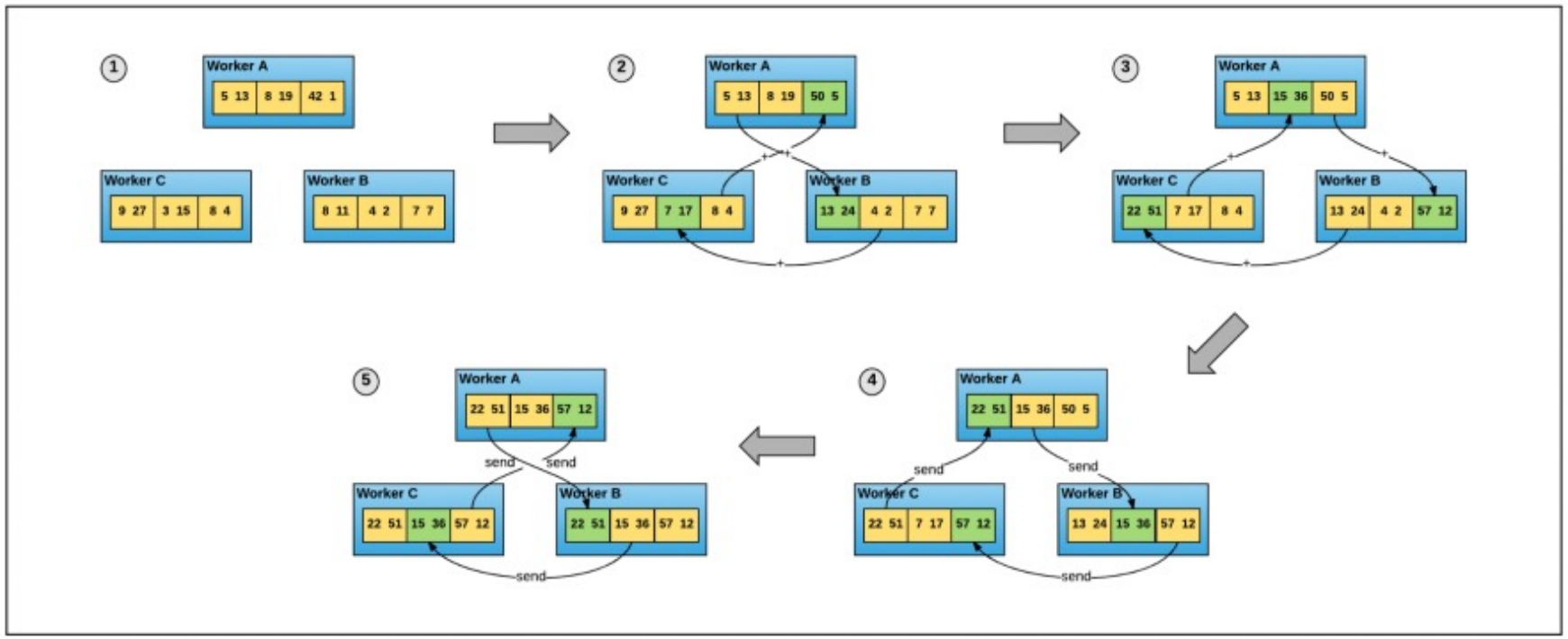
- Parameter server approach
  - Initial approach used by distributed Tensorflow
  - workers process training data, compute gradients, and send them to (a) parameter servers to be averaged
  - Challenges:
    - identifying the right ratio of worker to parameter servers to avoid networking or computational bottlenecks and network saturation in an "all-to-all" communication pattern.

DATA PARALLELISM
Implementations

- Ring-allreduce algorithm
  - Baidu (early 2017) proposed algorithm for averaging gradients and communicating those gradients to all nodes
    - The algorithm is called ring-allreduce
  - Bandwidth-optimal
  - Users utilize a Message Passing Interface (MPI) implementation such as Open MPI to launch all copies of the TensorFlow program and modify their program to average gradients using an allreduce() operation

DATA PARALLELISM

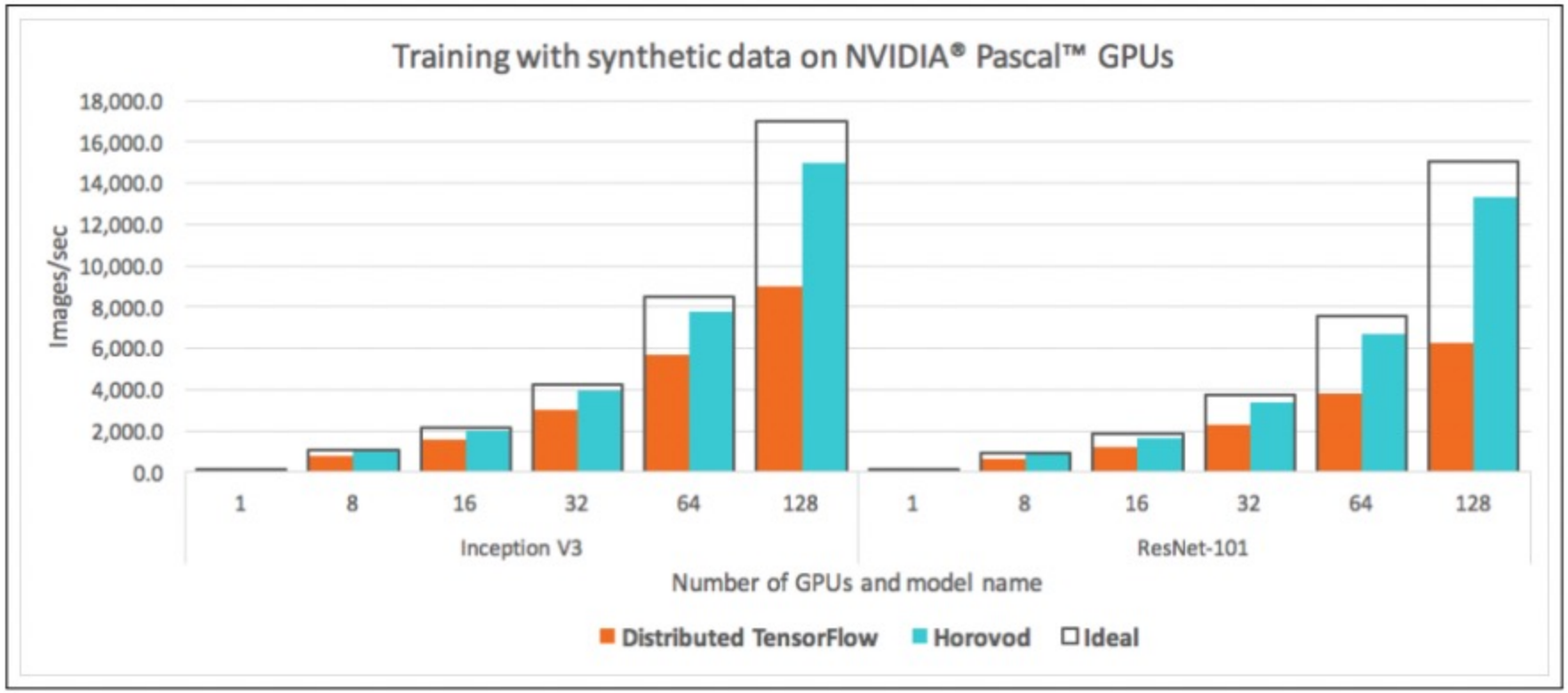
Implementations



HOROVOD

- Horovod is a Python package that implements distributed data parallel deep learning using ring-allreduce
- It replaces MPI with NCCL, NVIDIA's library for collective communication, which provides a highly optimized version of ring-allreduce
- NCCL 2 enables running ring-allreduce across **multiple machines** and supports models that fit inside a single server on multiple GPUs

HOROVOD



In the graph, Distributed TensorFlow refers to the implementation using parameter servers.

HOROVOD

- Works with stock TensorFlow, Keras, PyTorch, and MXNet
- Stand-alone package allows reducing the time required to install Horovod from about an hour to a few minutes, depending on the hardware

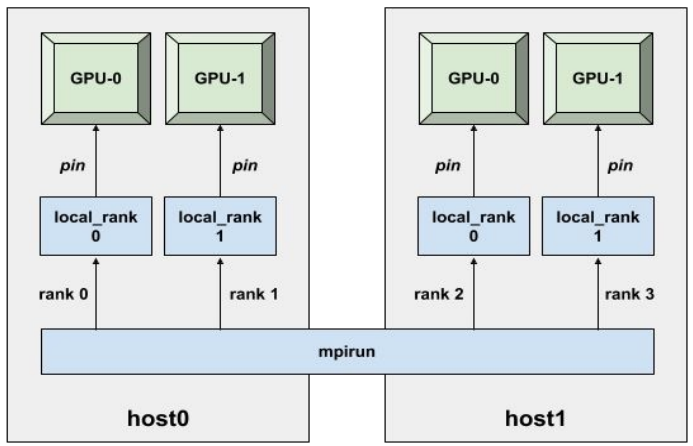
HOROVOD

```
import horovod.tensorflow.keras as hvd

hvd.init()
```

Import and initialize the package

```
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    tf.config.experimental.set_memory_growth(gpus[hvd.local_rank()], True)
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')
```



Pin processes to GPUs

- World size, Local Ranks, Global Ranks (ranks)

HOROVOD

```
opt = hvd.DistributedOptimizer(opt)
```

Wrap the optimizer into a distributed one

```
callbacks.append(hvd.BroadcastGlobalVariablesCallback(0))
```

Start from the model across all the GPUs

```
checkpoint = tf.keras.callbacks.ModelCheckpoint(...)

if hvd.rank() == 0:

    callbacks.append(checkpoint)

model.fit(…, callbacks,, verbose=1 if hvd.rank()== 0):
```

Only one process does checkpointing, and only one process is verbose

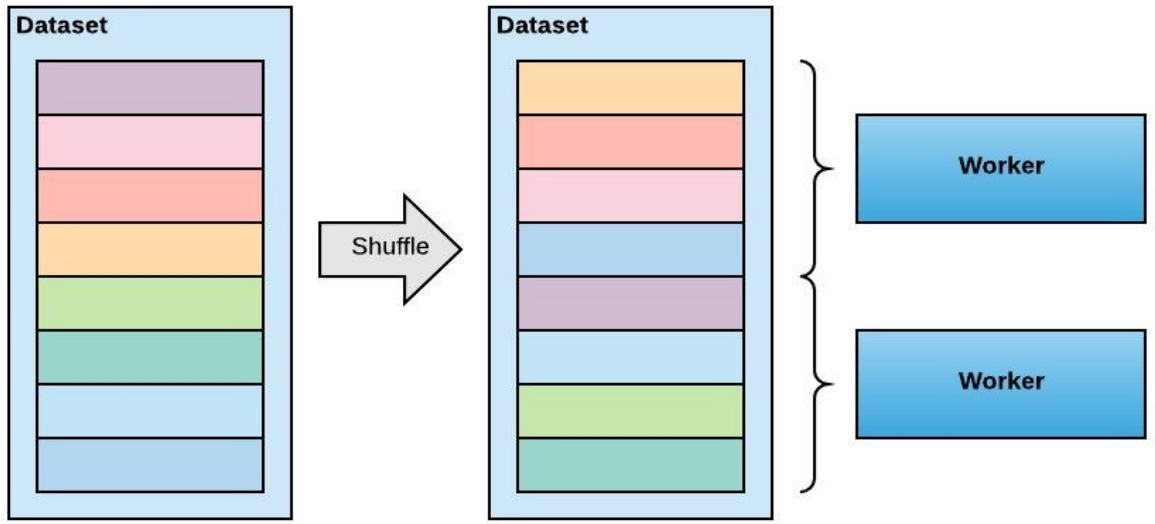
DATA PARTITIONING: OPTION 1

Shuffle the dataset

Partition records among workers

Train by sequentially reading the partition

After epoch is done, reshuffle and partition again



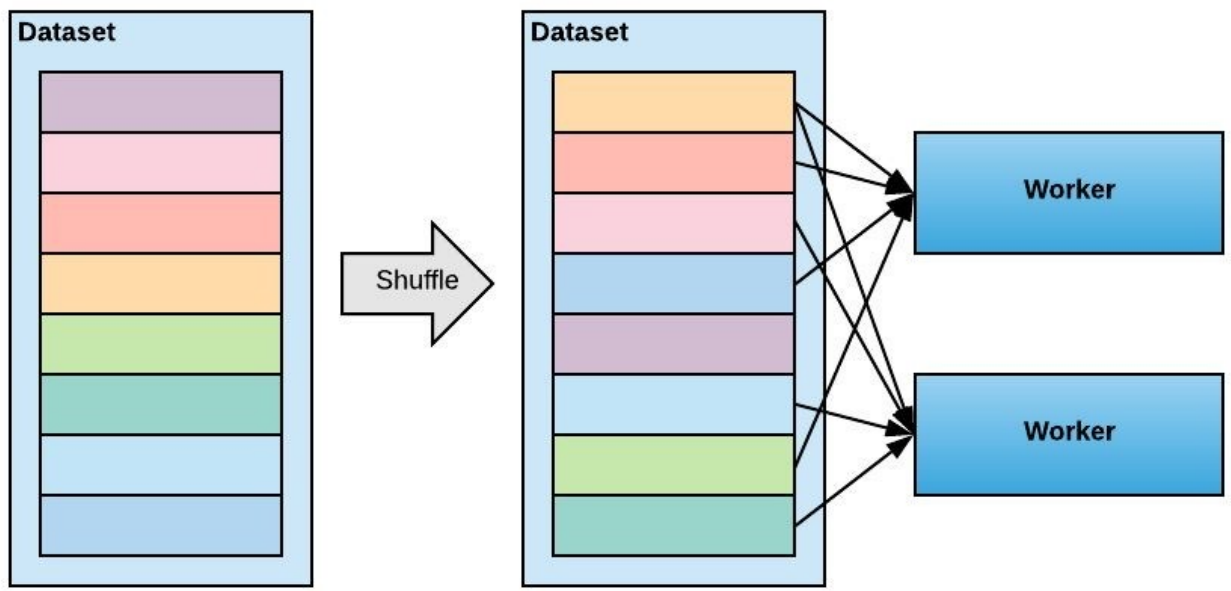
NOTE: make sure that all partitions contain the same number of batches, otherwise the training will deadlock

DATA PARTITIONING: OPTION 2

Shuffle the dataset

Train by randomly reading data from whole dataset

After epoch is done, reshuffle



HOROVOD

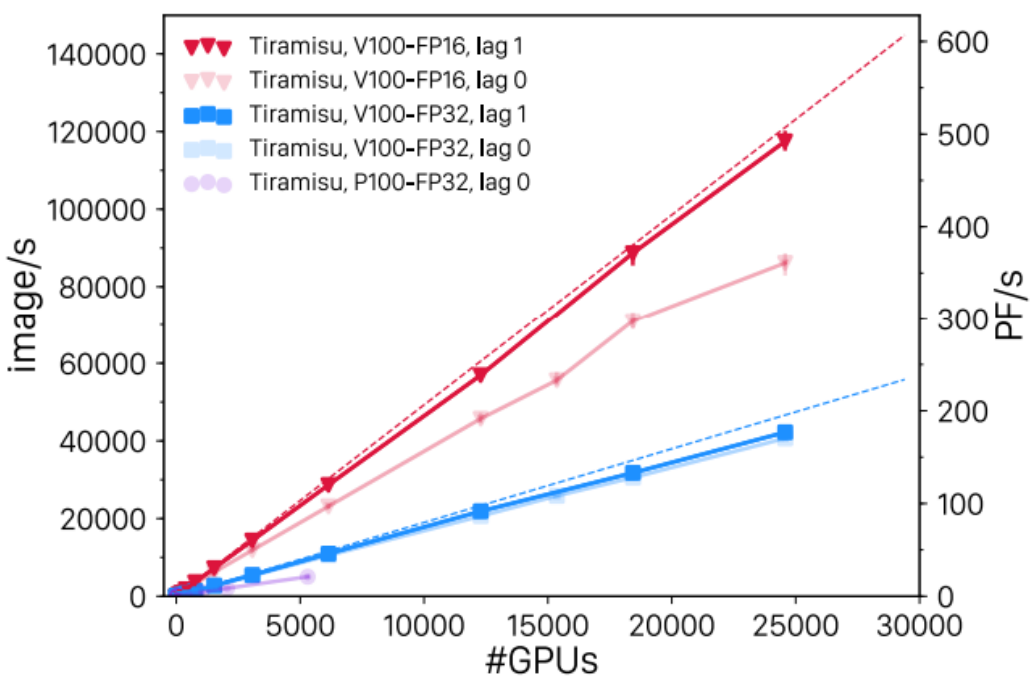
Single-node:

```
$ mpirun -np 4 python train.py
```

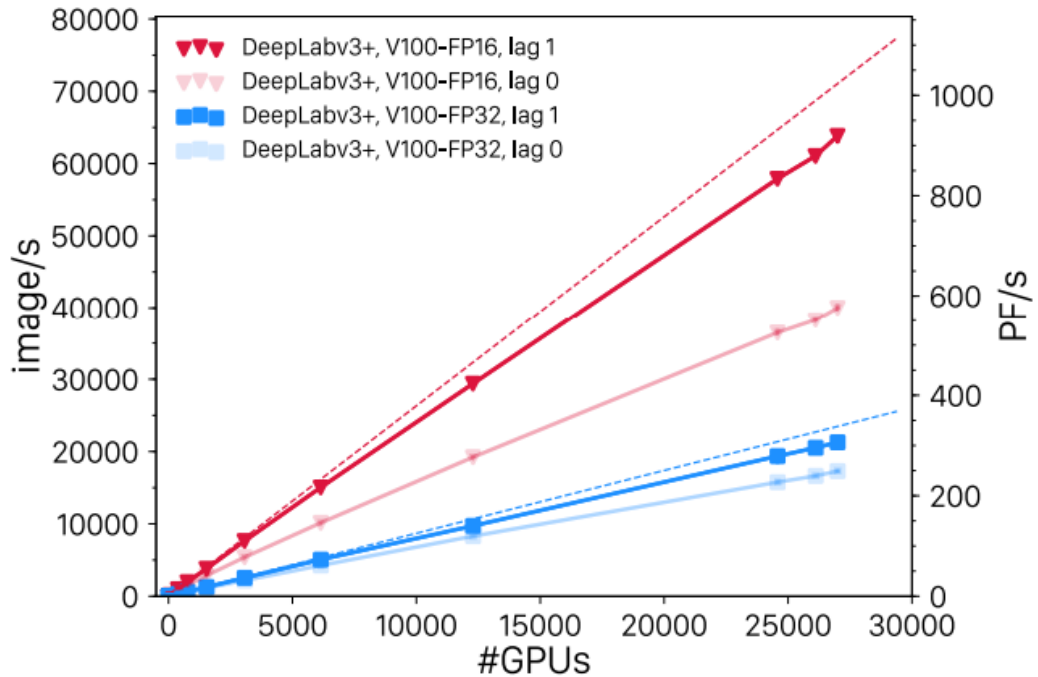
Multi-node:

```
$ mpirun -np 8 -H server1:4,server2:4 python train.py
```

THROUGHPUT WITH INCREASING BATCH SIZE



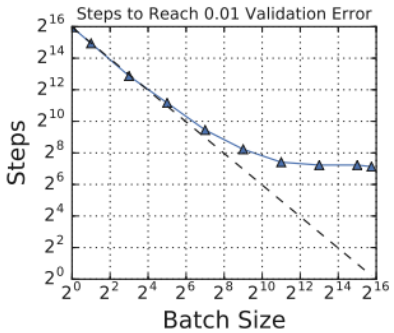
(a) Tiramisu



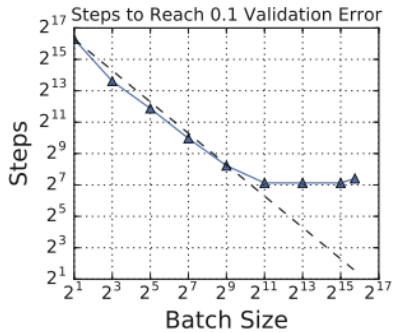
(b) DeepLabv3+

Kurth, T., Treichler, S., Romero, J., Mudigonda, M., Luehr, N., Phillips, E., ... & Houston, M. (2018, November). Exascale deep learning for climate analytics. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (p. 51). IEEE Press. arXiv:1810.01993

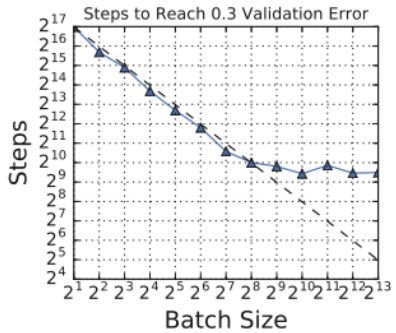
CRITICAL BATCH SIZE



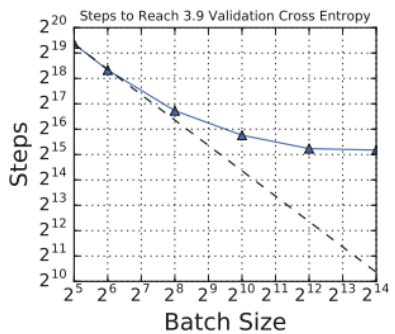
(a) Simple CNN on MNIST



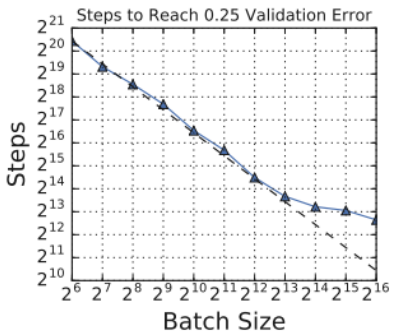
(b) Simple CNN on Fashion MNIST



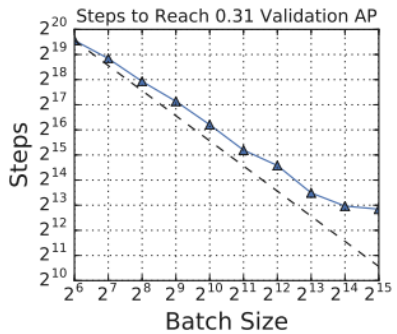
(c) ResNet-8 on CIFAR-10



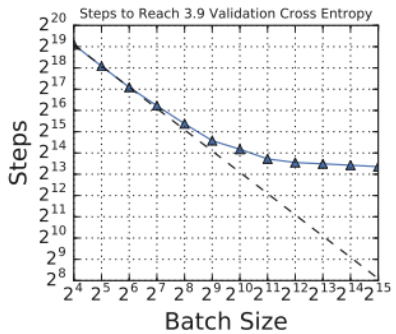
(g) Transformer on Common Crawl



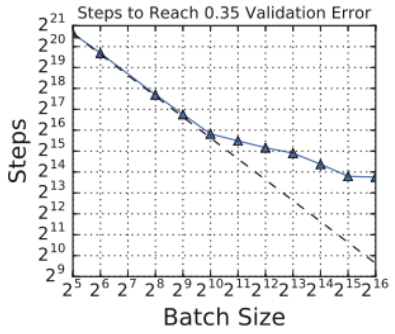
(d) ResNet-50 on ImageNet



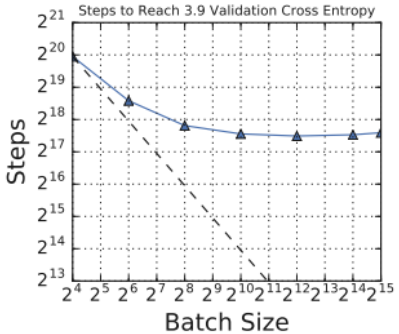
(e) ResNet-50 on Open Images



(f) Transformer on LM1B



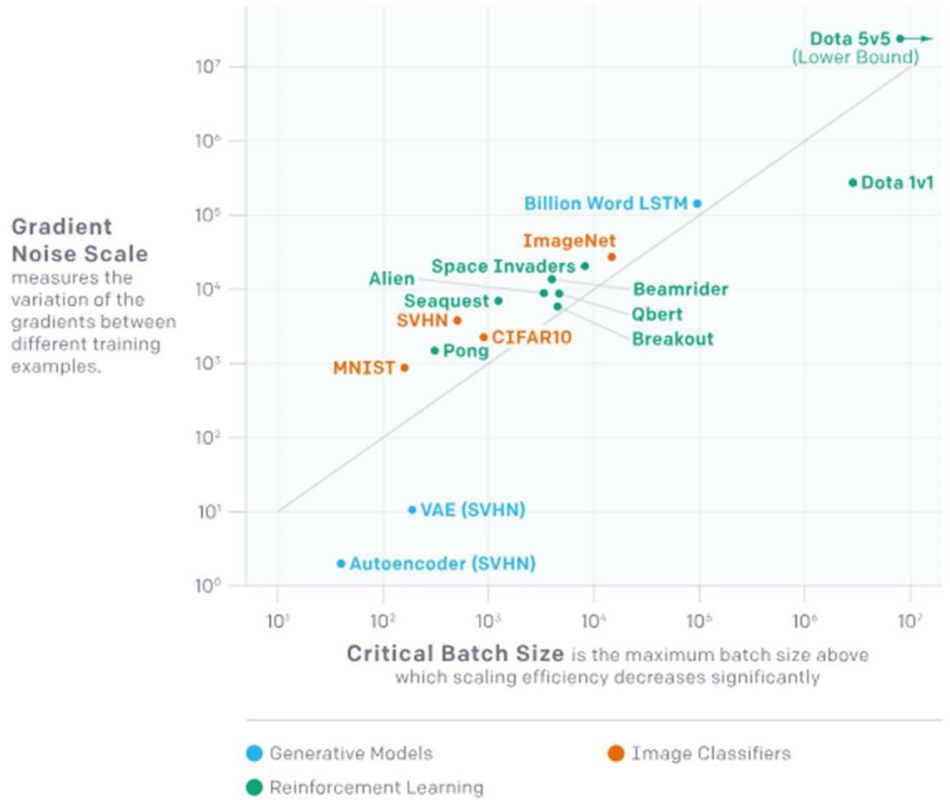
(h) VGG-11 on ImageNet



(i) LSTM on LM1B

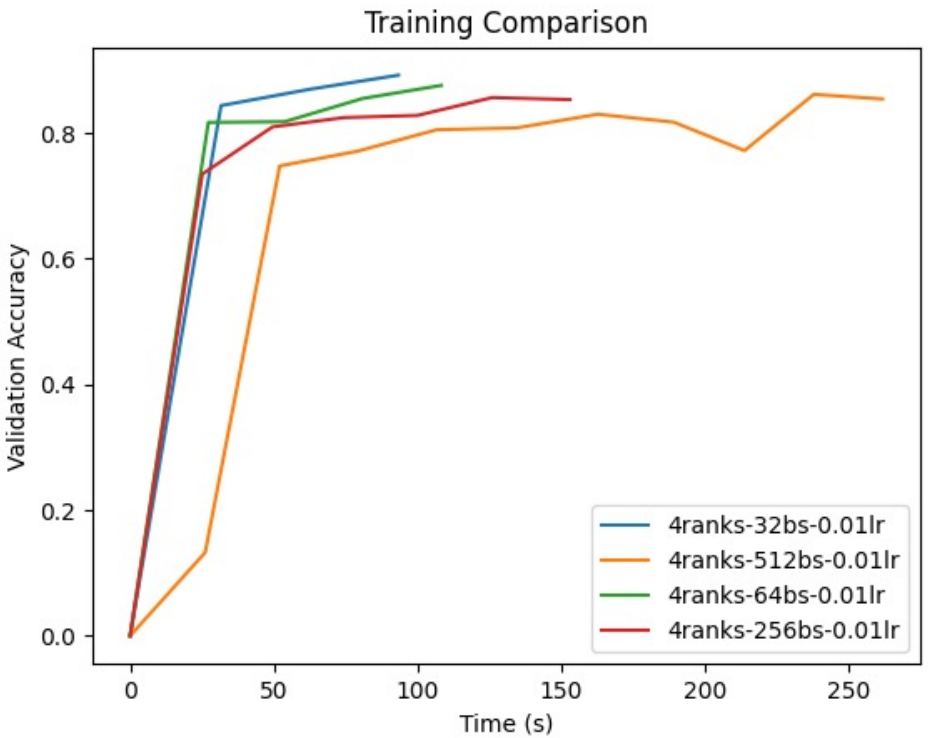
Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., & Dahl, G. E. (2018). Measuring the effects of data parallelism on neural network training. arXiv:1811.03600

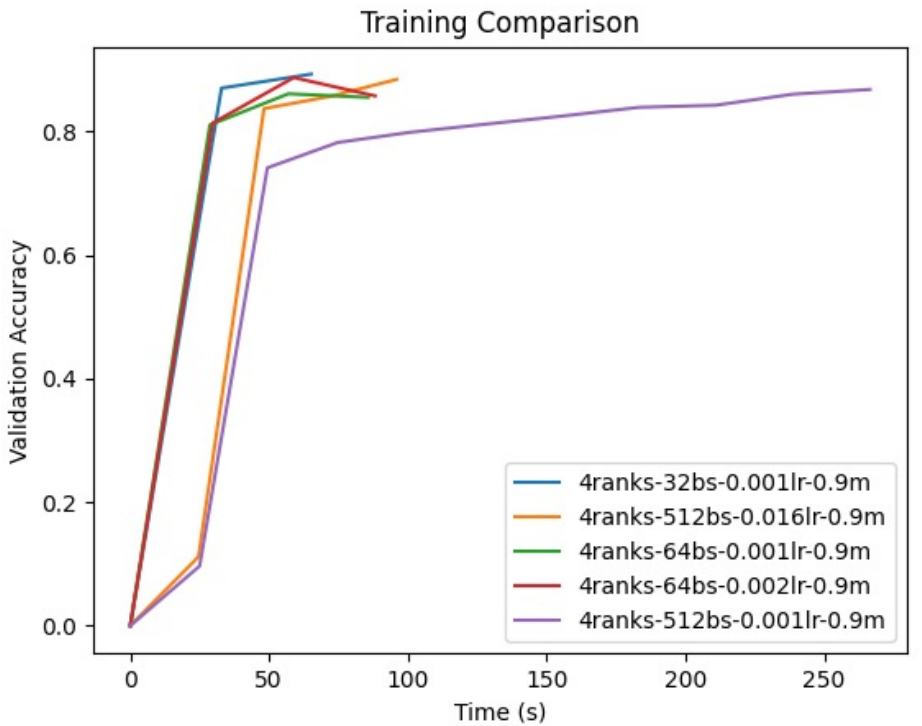
CRITICAL BATCH SIZE



https://blog.openai.com/science-of-ai/

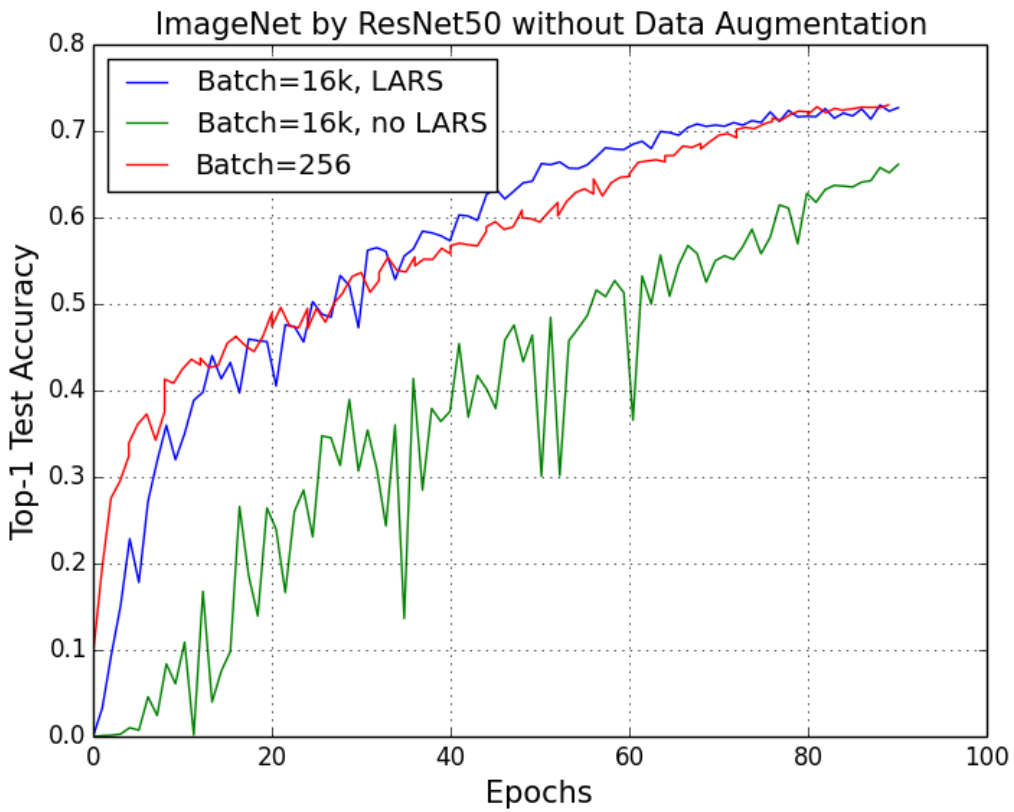
ACCURACY WITH INCREASING BATCH SIZES

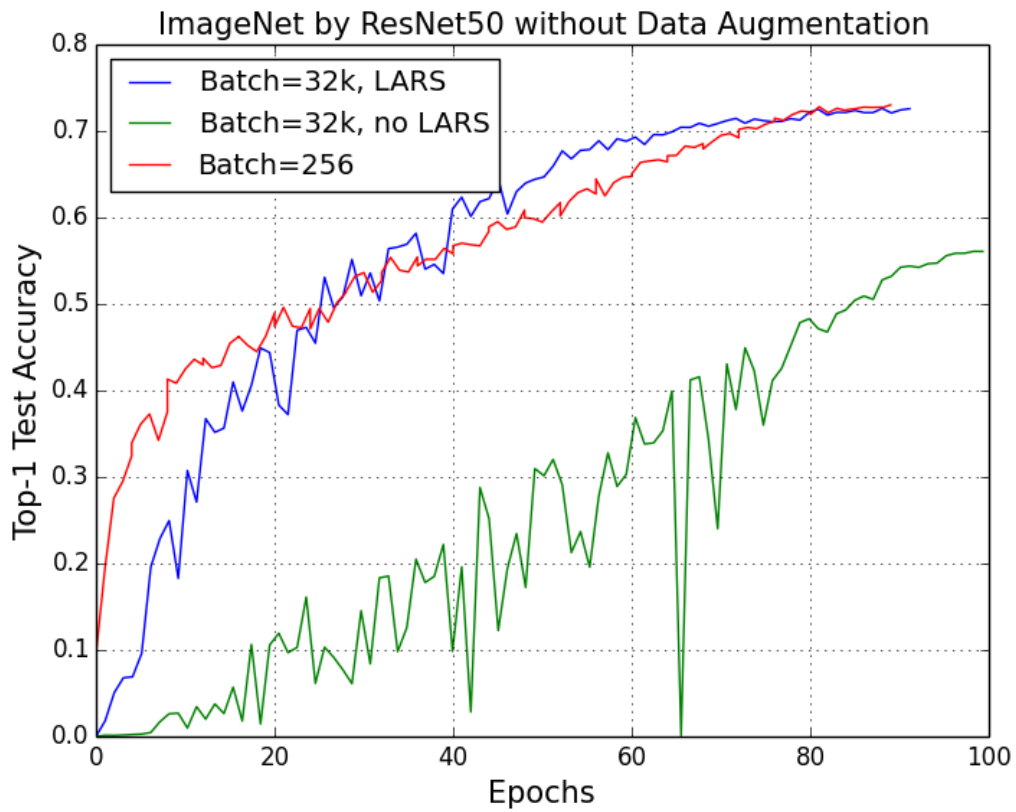




You, Y., Zhang, Z., Hsieh, C., Demmel, J., & Keutzer, K. (2017). ImageNet training in minutes. arXiv: 1709.05011

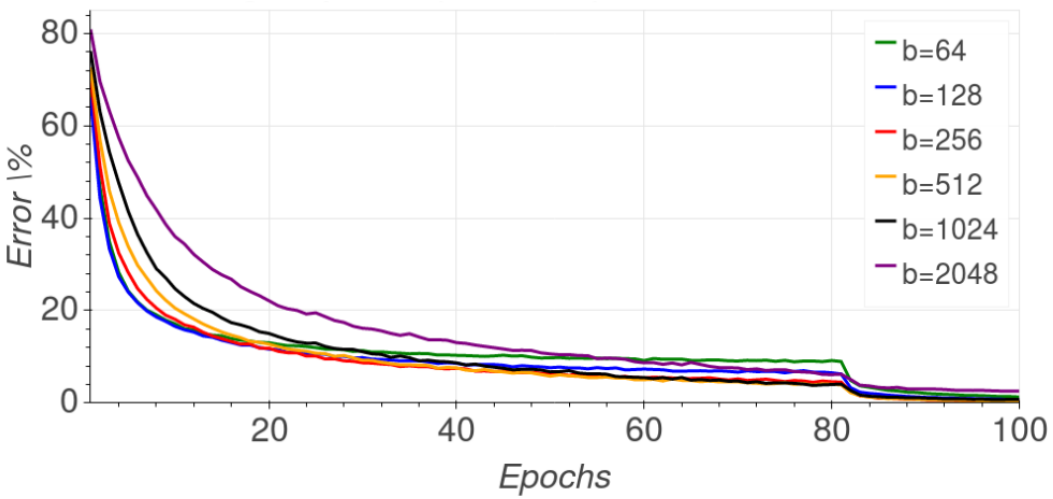
ACCURACY WITH INCREASING BATCH SIZES



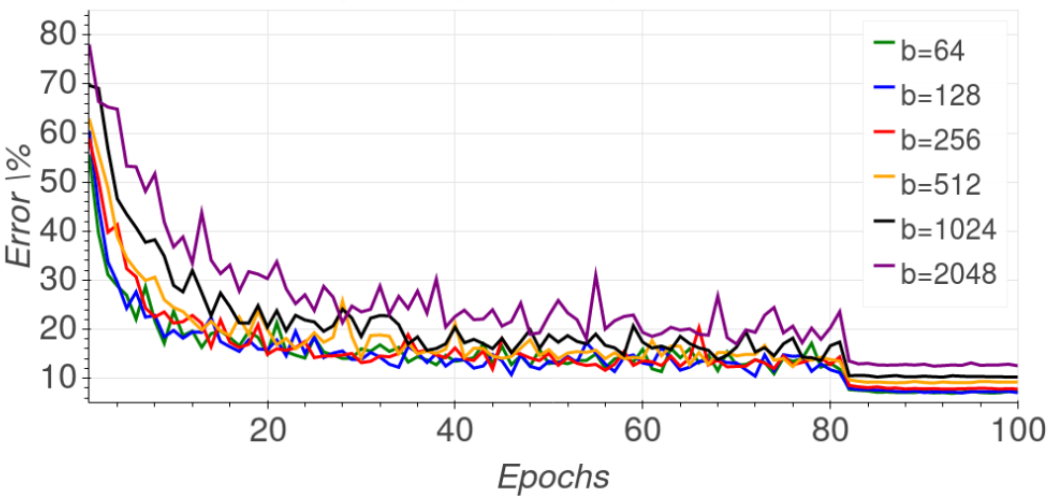


You, Y., Zhang, Z., Hsieh, C., Demmel, J., & Keutzer, K. (2017). ImageNet training in minutes. arXiv: 1709.05011

ACCURACY WITH INCREASING BATCH SIZES



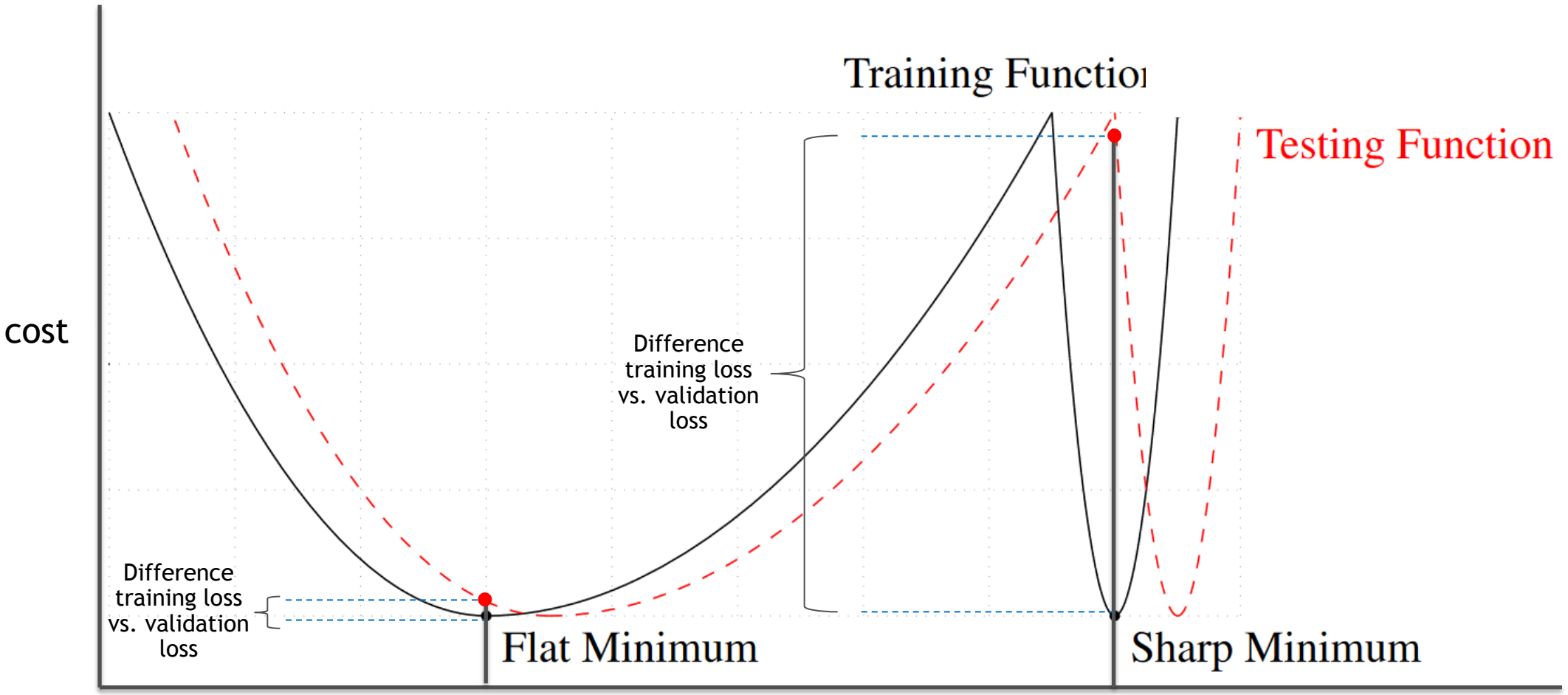
(a) Training error



(b) Validation error

Hoffer, E., Hubara, I., & Soudry, D. (2017). Train longer, generalize better: closing the generalization gap in large batch training of neural networks. arXiv:1705.08741

ACCURACY WITH INCREASING BATCH SIZES



Keskar, N. S., et al. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. arXiv:1609.04836

ACCURACY WITH INCREASING BATCH SIZES

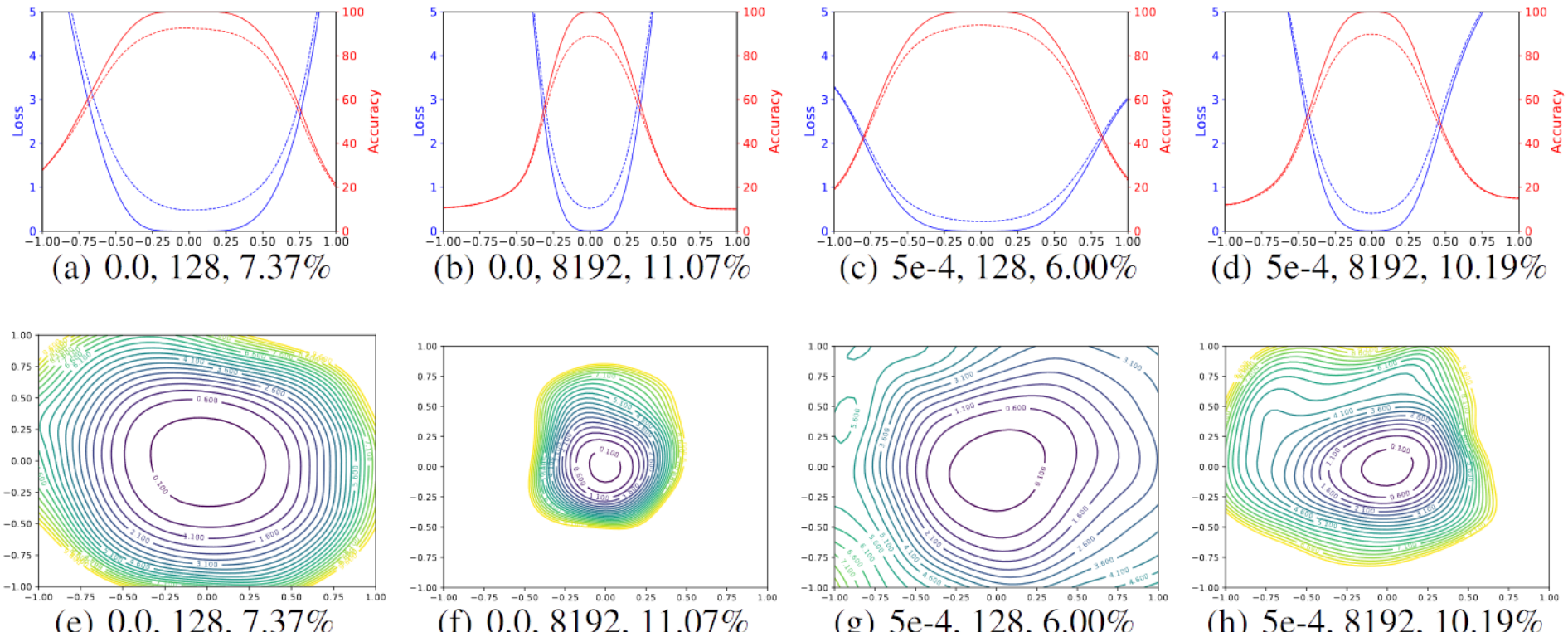


Figure 3: The 1D and 2D visualization of solutions obtained using SGD with different weight decay and batch size. The title of each subfigure contains the weight decay, batch size, and test error.

Li, H., Xu, Z., Taylor, G., & Goldstein, T. (2017). Visualizing the Loss Landscape of Neural Nets. arXiv:1712.09913

OPTIMIZATION WITH INCREASING BATCH SIZE

- Manipulate the learning rate?
- Add noise to the gradient?
- Manipulate the batch size?
- Change the learning algorithm?

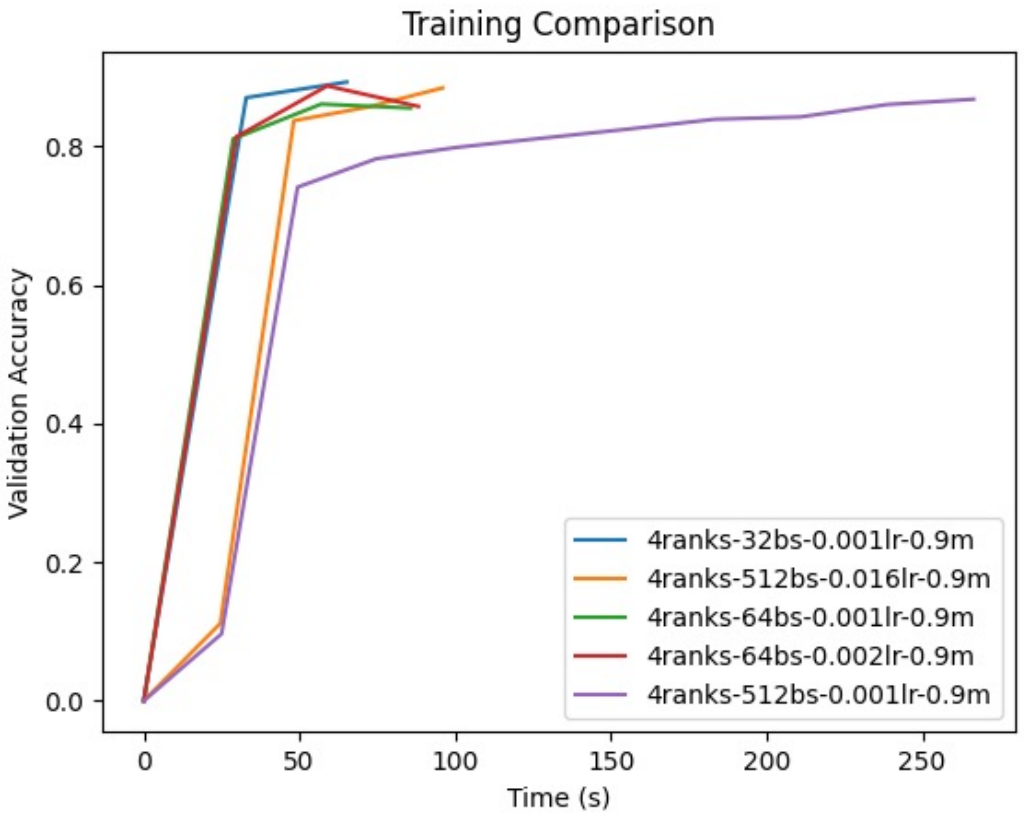
LEARNING RATE SCALING

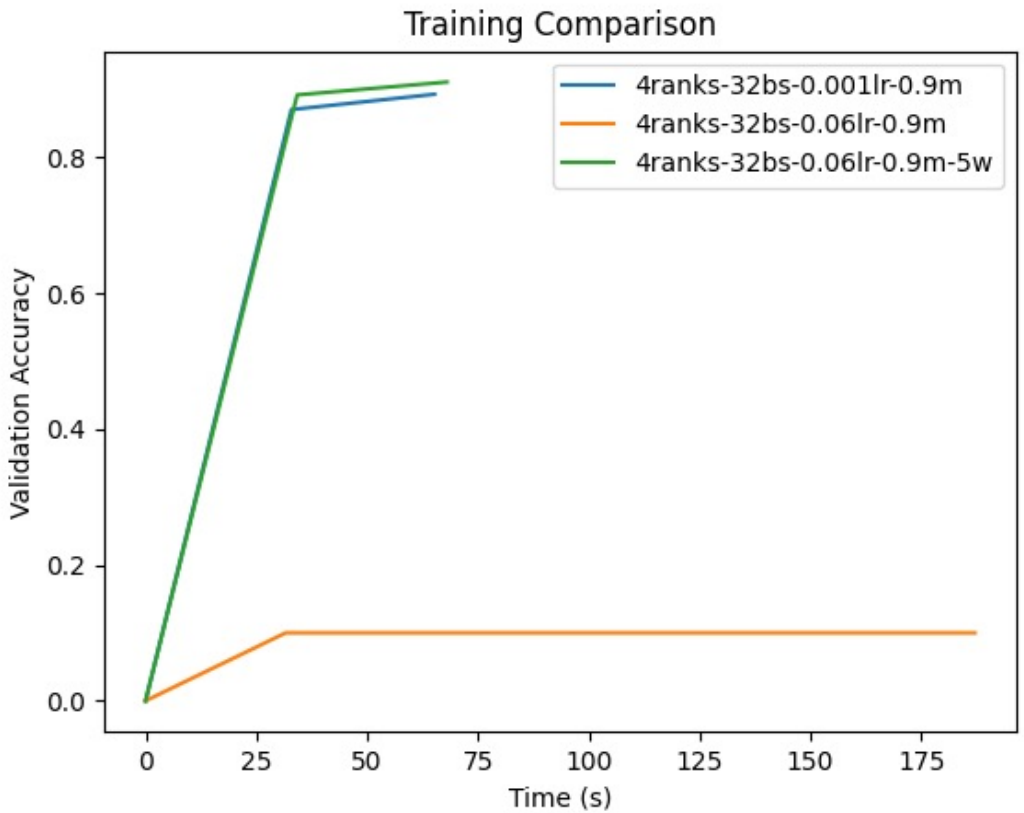
- Larger batch sizes imply less steps per epoch
- Intuitively
  - 2x batch size means 0.5x less updates of the gradients
  - To get from point X to point Y, it seems reasonable to do twice as larger steps (i.e., double the learning rate)
  - This suggest to increase the learning rate by the same factor the batch size is increased
- In practice the value that has been found to work well is to increase the learning rate on a factor $\sqrt{M}$ being M the factor by which you increase the batch size

LEARNING RATE SCALING

- A lot of networks will diverge early in the learning process
- A larger learning rate does make this divergence even larger
  - We can address this by starting with a smaller learning rate that increases over a few epochs (where the learning process should not diverge that much)
  - After these epochs, we can settle for the desired learning rate value / strategy
  - This technique is called warmup

LEARNING RATE SCALING





BATCH NORMALIZATION

- The idea is to normalize the inputs to all layers in every batch (this is more sophisticated than simply normalizing the input dataset)
- Minimizes drift in the distribution of inputs to a layer
- It allows higher learning rates and reduces the need to use dropout
- The original batch normalization paper suggests using the statistics for the entire batch
  - What to do in the data parallel training case?
  - Batch normalization is thus carried out in isolation on a per-GPU basis.
  - Additional noise by calculating smaller batch statistics (“ghost batches”).

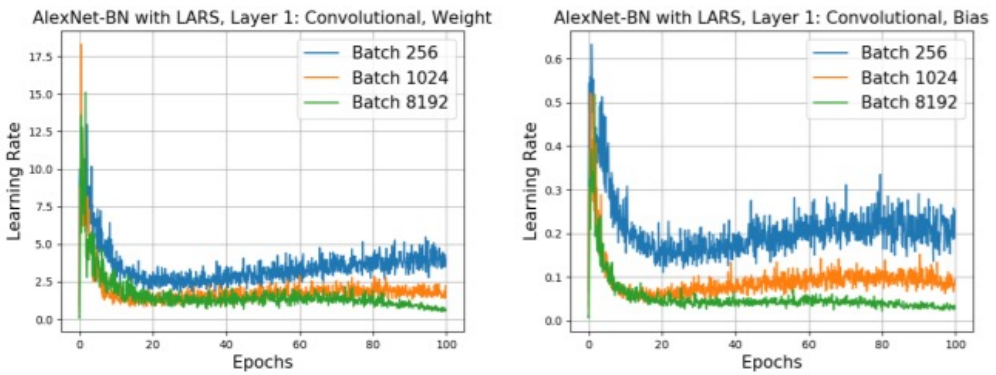
ALTERNATIVES TO SGD

- **LARS (Layer-wise Adaptive Rate Scaling)**
  - It aims to dynamically scale the learning rates of individual layers in a neural network based on their weight magnitudes
  - The motivation behind LARS is to address the challenge of training deep networks with very large weight updates, which can cause instability and hinder convergence
- **LARC (Layer-wise Adaptive Rate Clipping):**
  - LARC applies adaptive rate clipping to the gradients during optimization
  - It clips the gradients based on a threshold determined by the ratio of the norm of the layer's weights to the norm of the layer's gradients. If this ratio exceeds a predefined threshold, the gradients are rescaled or clipped to prevent them from being too large

ALTERNATIVES TO SGD

- **NOVOGRAD**
  - introduces gradient normalisation, which rescales the gradient to have a fixed norm or magnitude
  - This helps mitigate the impact of large gradients, making the optimization process more stable
  - Based on ADAM , which does not explicitly perform gradient normalisation, although it uses adaptive learning rates to handle varying gradient magnitudes

EXAMPLES WITH LARS



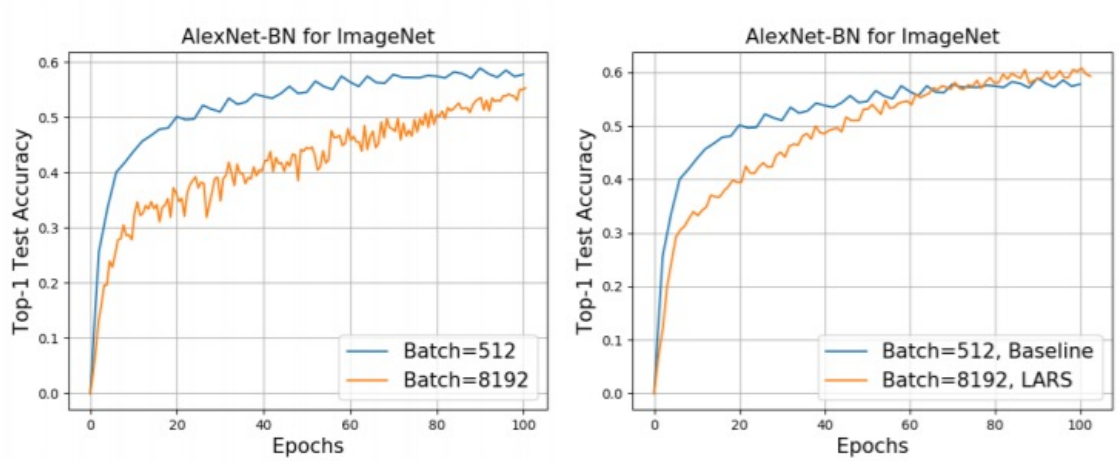
(a) Local LR, conv1-weights

(b) Local LR, conv1-bias

(c) Local LR , conv5-weights

(d) Local LR, conv5-bias

Figure 2: LARS: local LR for different layers and batch sizes



(a) Training without LARS

(b) Training with LARS

You, Y., Gitman, I., & Ginsburg, B. Large batch training of convolutional networks. arXiv:1708.03888

BEYOND HOROVOD AND DATA PARALLELISM

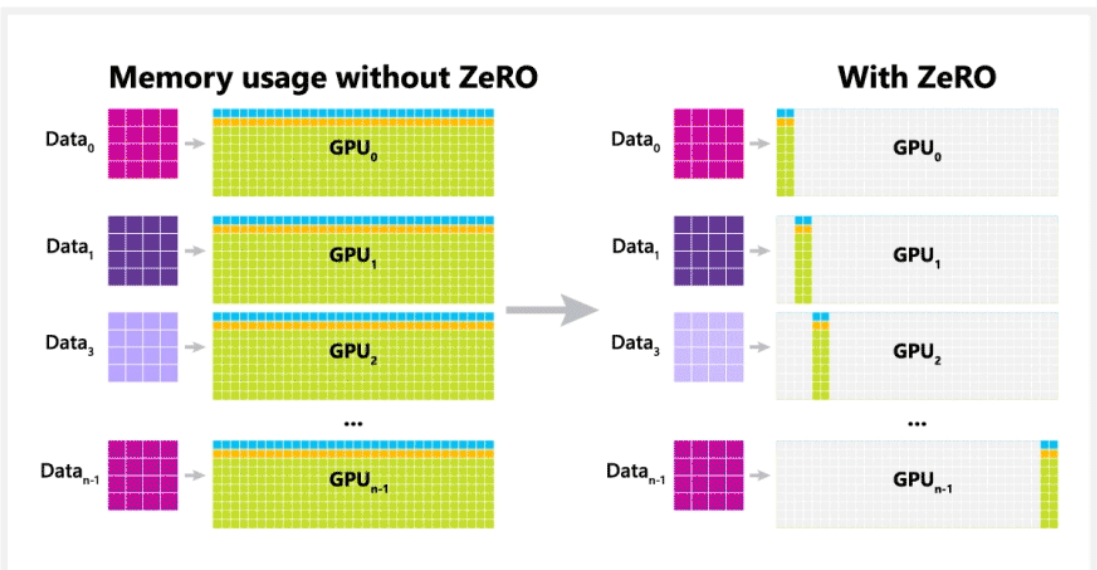
- DeepSpeed is a deep learning optimization library that integrates with popular deep learning frameworks like PyTorch and TensorFlow
- It provides a wide range of features for training large models efficiently. Some key features of DeepSpeed include:
  - Memory Optimization: memory footprint reduction using as activation checkpointing and offloading optimizer states to CPU memory. This allows for training larger models that would otherwise exceed GPU memory limits
  - Gradient compression techniques to reduce communication overhead in distributed training, enabling faster and more efficient communication between GPUs or machines

BEYOND HOROVOD AND DATA PARALLELISM

- It provides a wide range of features for training large models efficiently. Some key features of DeepSpeed include:
  - DeepSpeed integrates with ZeRO to enhance memory efficiency by partitioning model weights across multiple GPUs or machines, enabling training of larger models than the memory capacity of an individual device while still aiming at a data parallel training
  - DeepSpeed supports pipeline parallelism, which splits the model across multiple GPUs or machines to process different parts of the model simultaneously, leading to improved training performance.

BEYOND HOROVOD AND DATA PARALLELISM

DeepSpeed + ZeRO



https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/

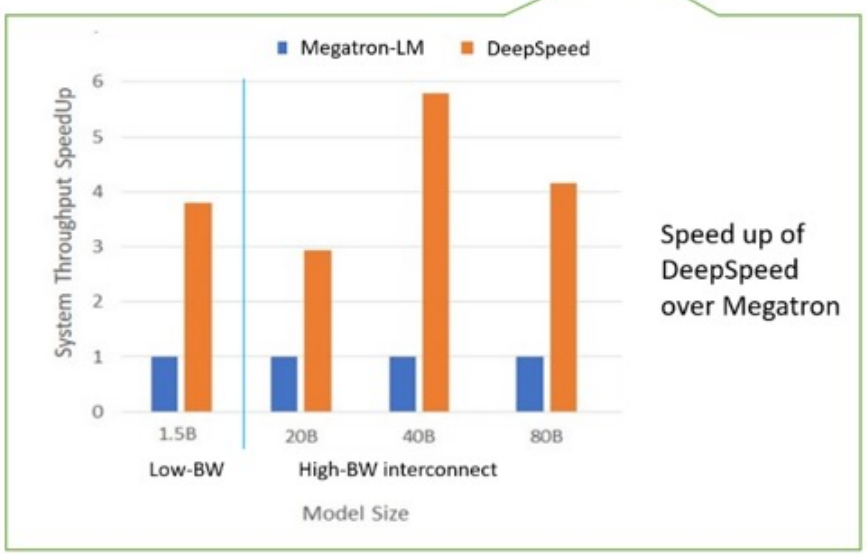
BEYOND HOROVOD AND DATA PARALLELISM

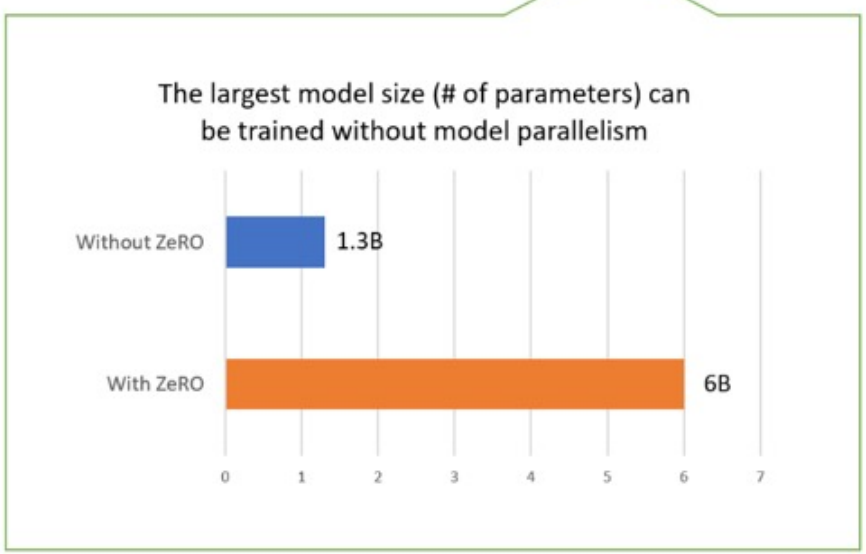
Scale
- 100B parameter
- 10X bigger

Speed
- Up to 5X faster

Cost
- Up to 5X cheaper

Usability
- Minimal code change





https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/

BEYOND HOROVOD AND DATA PARALLELISM

ZeRO 4-way data parallel training

Using:

- P_{os} (Optimizer state)
- P_g (Gradient)
- P_p (Parameters)

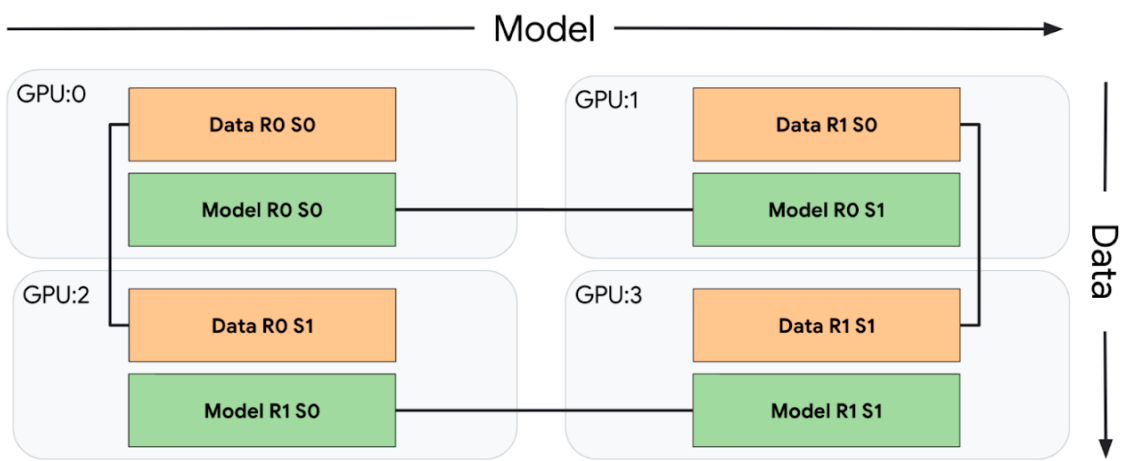
https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/

BEYOND HOROVOD AND DATA PARALLELISM

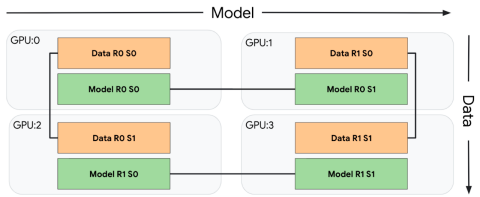
- DTensor enables larger and more performant model training by giving developers the flexibility to combine and fine-tune multiple parallelism techniques
- Allows leveraging data parallelism and model parallelism to improve the efficiency of training large models
- The ability to use multiple parallelism techniques and fine-tune them gives developers the tools they need to improve their model training and achieve better results
- While it sounds quite nice there is a main caveat: it is really low level in its current experimental phase

BEYOND HOROVOD AND DATA PARALLELISM

- Data and model parallelism are not only supported, but also can be directly combined to scale models even more efficiently
- Accelerator agnostic —TPUs, GPUs, or something else

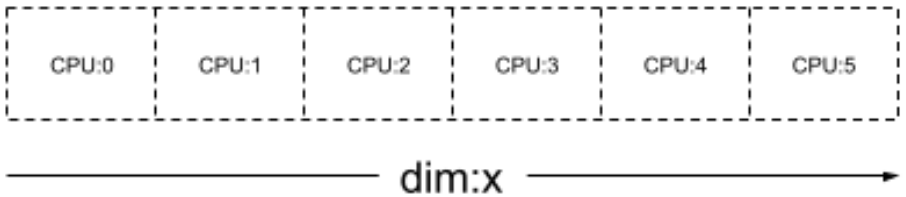


BEYOND HOROVOD AND DATA PARALLELISM



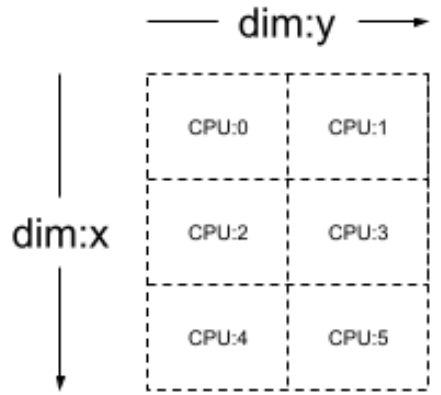
- DTensors works with the concepts of mesh and layout
- A mesh is a logical cartesian topology representation of the available resources
- Can be
  - 1D

```
dtensor.create_mesh([('x', 6)], devices=DEVICES)
```

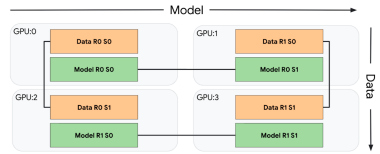


  - 2D

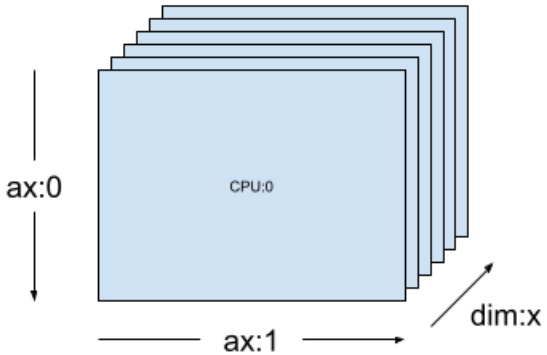
```
dtensor.create_mesh(
    [('x', 3), ('y', 2)],
    devices=DEVICES)
```



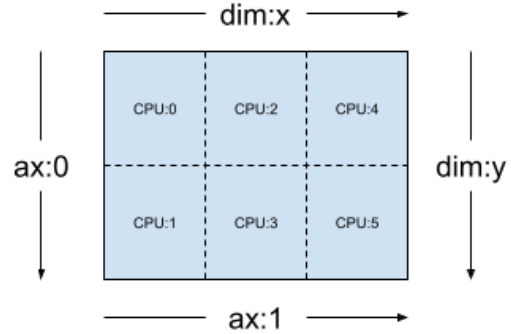
BEYOND HOROVOD AND DATA PARALLELISM



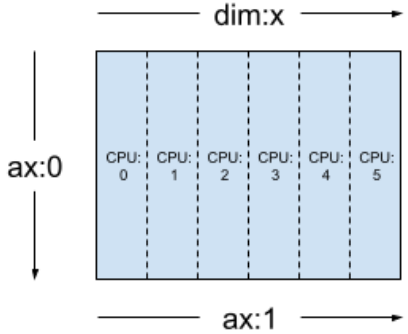
- A layout indicates how to shard (distribute) tensors over a mesh axis




```
dtensor.Layout(
[dtensor.UNSHARDED, dtensor.UNSHARDED],
mesh_1d)
```

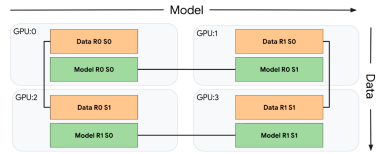



```
dtensor.Layout(
Layout(["x", dtensor.UNSHARDED],
mesh_2d)mesh_1d)
```




```
dtensor.Layout(
[dtensor.UNSHARDED, 'x’],
mesh_1d)
```

BEYOND HOROVOD AND DATA PARALLELISM




```
from typing import Tuple

class MLP(tf.Module):

  def __init__(self, dense_layouts: Tuple[dtensor.Layout, dtensor.Layout]):
    super().__init__()

    self.dense1 = Dense(
        1200, 48, (1, 2), dense_layouts[0], activation=tf.nn.relu)
    self.bn = BatchNorm()
    self.dense2 = Dense(48, 2, (3, 4), dense_layouts[1])

  def __call__(self, x):
    y = x
    y = self.dense1(y)
    y = self.bn(y)
    y = self.dense2(y)
    return y

mesh = dtensor.create_mesh([("batch", 8)], devices=DEVICES)

model = MLP([dtensor.Layout([dtensor.UNSHARDED, dtensor.UNSHARDED], mesh),
             dtensor.Layout([dtensor.UNSHARDED, dtensor.UNSHARDED], mesh),])
```