COMP 4901Q: High Performance Computing (HPC)

Lecture 12: A Case Study on Large-Scale Machine Learning in HPC

Instructor: Shaohuai SHI (shaohuais@cse.ust.hk)
Teaching assistants: Mingkai TANG (mtangag@connect.ust.hk)
Yazhou XING (yxingag@connect.ust.hk)

Course website: https://course.cse.ust.hk/comp4901q/

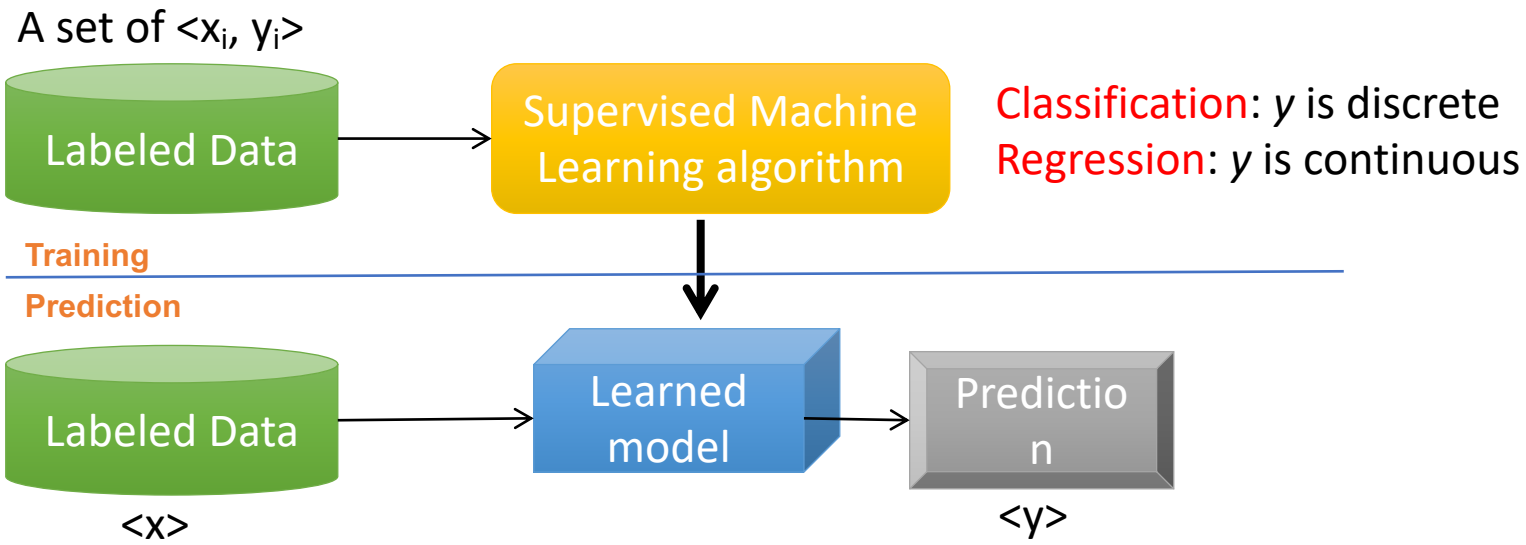
Outline

- Machine Learning Basics
  - Supervise Machine Learning
  - Review of Calculus
  - Gradient Decent
- (Deep) Neural Networks
- Model Training: SGD and Backpropagation
- How GPU Clusters Accelerate Deep Neural Networks Training

Machine Learning Basics

- Machine learning (ML) is a field of computer science that gives computers the ability to **learn without being explicitly programmed**
- We focus on **supervised machine learning**

Supervised learning problem: given a new data sample ***x***, what is the corresponding label *y*?



Methods that can learn from and make predictions on data

How does Supervised ML work?

- A machine learning (ML) model is a complicated function with many unknown parameters. Given an input, it can generate an output.
  - Remark: Output can be close to the truth or not.
  - A good model will have a high probability to give GOOD predictions (i.e., close to the ground truth) even for unseen data (i.e., the data not in the existing dataset).

- A general framework to find a good ML model:
  - 1. Divide the dataset into three sets: training/validation/testing
  - 2. Design the architecture of the model (i.e., the format of the function)
  - 3. Train on the training dataset: to determine the unknown model parameters. This step usually takes a long time. It tries to solve an optimization problem:
    - how to set the unknown parameters to minimize the overall differences between the ground truth and the predicted output?
  - 4. Validate the performance of the model on the validation dataset. If not good enough, go back to Step 2 (e.g., refine the model, or change to a new method)
  - 5. Once the model has been fixed, test its performance on a testing dataset.

Differentiation

- Consider a single-variable function y = f(x), where x is a real-value input, and y is a real-value output.
- Derivative of function y = f(x) measures the sensitivity to the change of output y with respect to a change in the input x.
- We say, the derivative of function f(x) at point x is:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- Leibniz's notation: for function y = f(x), its first derivative can be denoted by:

$$\frac{dy}{dx} \text{ or } \frac{df}{dx} \text{ or } \frac{d}{dx} f$$

- **Differentiation** is the process of finding the derivative of a function.

Derivative of Common Functions

c = constant

$$\frac{d}{dx}c = 0$$

$$\frac{d}{dx}c^x = c^x \ln c \qquad (c > 0)$$

$$\frac{d}{dx}x = 1$$

$$\frac{d}{dx}(cx) = c$$

$$\frac{d}{dx}e^x = e^x$$

$$\frac{d}{dx}|x| = \frac{|x|}{x} \qquad (x \neq 0)$$

$$\frac{d}{dx}\log_c x = \frac{1}{x \ln c} \qquad (c > 0,\ c \neq 1)$$

$$\frac{d}{dx}x^c = cx^{c-1} \quad \text{(where both } x^c \text{ and } cx^{c-1} \text{ are defined)}$$

$$\frac{d}{dx}\ln x = \frac{1}{x} \qquad (x > 0)$$

$$\frac{d}{dx}\left(\frac{1}{x}\right) = \frac{d}{dx}\left(x^{-1}\right) = -x^{-2} = -\frac{1}{x^2}$$

$$\frac{d}{dx}\ln|x| = \frac{1}{x}$$

$$\frac{d}{dx}\left(\frac{1}{x^c}\right) = \frac{d}{dx}\left(x^{-c}\right) = -cx^{-c-1} = -\frac{c}{x^{c+1}}$$

$$\frac{d}{dx}x^x = x^x(1+\ln x)$$

$$\frac{d}{dx}\sqrt{x} = \frac{d}{dx}x^{\frac{1}{2}} = \frac{1}{2}x^{-\frac{1}{2}} = \frac{1}{2\sqrt{x}} \qquad (x > 0)$$

Rules for Combined Functions

- Given function f(x) and g(x):
  - Sum rule: $$(\alpha f + \beta g)' = \alpha f' + \beta g'$$
  - Product rule: $$(fg)' = f'g + fg'$$
  - Quotient rule: $$\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$$
  - Chain rule: if $f(x) = h(g(x))$, then we have
$$f'(x) = h'(g(x)) \cdot g'(x).$$

Practice: Logistic Function

- In Artificial Neural Networks (ANNs), sigmoid functions are commonly used as the activation function. The following logistic function is one among many sigmoid functions.

$$f(x) = \frac{1}{1 + e^{-x}}$$

Please prove that: $f'(x) = f(x) \cdot (1\text{-}f(x))$.

This is a very nice feature because:

- ❑ $f(x)$ is calculated in the feed-forward step.
- ❑ $f'(x)$ can then be quickly calculated in the backpropagation step.

Partial Derivatives

- Now we consider functions with multiple input variables.
  - The output of the function depends on all variables.
  - But we still can measure the sensitivity to the change of the output with respect to the change of a single variable.
- The **partial derivative** of a multi-variable function $f(x_1, \ldots, x_n)$ with respect to variable x_i at the point $(a_1, \ldots, a_n)$ is defined as:

$$\frac{\partial f}{\partial x_i}(a_1, \ldots, a_n) = \lim_{h \to 0} \frac{f(a_1, \ldots, a_i + h, \ldots, a_n) - f(a_1, \ldots, a_i, \ldots, a_n)}{h}.$$

- A simple example: $z = f(x, y) = x^2 + xy + y^2$.
  - By fixing variable y (or x) as a constant, we can have:

$$\frac{\partial z}{\partial x} = 2x + y \qquad \frac{\partial z}{\partial y} = x + 2y$$

Rules of Partial Differentiation

Product Rule: $$\frac{\partial}{\partial \boldsymbol{x}}\big(f(\boldsymbol{x})g(\boldsymbol{x})\big) = \frac{\partial f}{\partial \boldsymbol{x}}g(\boldsymbol{x}) + f(\boldsymbol{x})\frac{\partial g}{\partial \boldsymbol{x}}$$

Sum Rule: $$\frac{\partial}{\partial \boldsymbol{x}}\big(f(\boldsymbol{x}) + g(\boldsymbol{x})\big) = \frac{\partial f}{\partial \boldsymbol{x}} + \frac{\partial g}{\partial \boldsymbol{x}}$$

Chain Rule: $$\frac{\partial}{\partial \boldsymbol{x}}(g \circ f)(\boldsymbol{x}) = \frac{\partial}{\partial \boldsymbol{x}}\big(g(f(\boldsymbol{x}))\big) = \frac{\partial g}{\partial f}\frac{\partial f}{\partial \boldsymbol{x}}$$

Gradient

- A function with n variables has n partial derivatives.

- The **gradient** of a function $f(x_1, \ldots, x_n)$ is the **vector** of partial derivatives.

$$\nabla f = \frac{df}{dx} = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

Gradient with Chain Rule

- If $f(x_1, x_2)$ is a function of x_1 and x_2, where $x_1(s, t)$ and $x_2(s, t)$ are themselves functions of two variables s and t.

- The chain rule yields the following partial derivatives:

$$\frac{\partial f}{\partial s} = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial s} + \frac{\partial f}{\partial x_2}\frac{\partial x_2}{\partial s}$$

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial t} + \frac{\partial f}{\partial x_2}\frac{\partial x_2}{\partial t}$$

This may help you understand the proof of backpropagation algorithm
http://neuralnetworksanddeeplearning.com .

Training Machine Learning Models

- A machine learning model can be regarded as a mathematical function

$$y = f(x, w),$$

where x is a give input sample, w is the model parameter, and y is the output. In the general case, x, y, w are all vectors.

- Training is the process of finding the model parameters w such that for a given dataset D = $\{(x_i, y_i) | 1 \le i \le n\}$, the average loss $L(D, w)$ is minimized. Using MSE as an example,

$$L(D, w) = \frac{1}{n}\sum_{i=1}^{n} \|y_i - \tilde{y}_i\|^2 = \frac{1}{n}\sum_{i=1}^{n} \|y_i - f(x_i, w)\|^2$$

Minimizing the non-negative loss is an optimization problem.

If the model can make perfect prediction on all samples, then L(D, w) = 0; otherwise, $L(D, w) > 0$. **Smaller loss means better model**.

Gradient Decent

- Gradient decent is a popular optimization solution to iteratively reduce the loss by adjusting *w*.
  - To simplify our discussion, assume *w* is a vector of two real values w_1 and w_2.
- Assume data set D is fixed, we can write $L(D, w)$ as $L(w_1, w_2)$.
- Now we introduce tiny changes on *w*: $\Delta w = (\Delta w_1, \Delta w_2)$.
- The change on the loss $\Delta L \approx \frac{\partial L}{\partial w_1} \Delta w_1 + \frac{\partial L}{\partial w_2} \Delta w_2$.
- What we what to do is to make ΔL negative, i.e., to reduce the loss by adjusting w_1 and w_2.
- If we choose $\Delta \boldsymbol{w} = -\boldsymbol{r}\left(\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial w_1}}, \frac{\boldsymbol{\partial L}}{\boldsymbol{\partial w_2}}\right)$, then $\Delta L \approx -r\left(\left(\frac{\partial L}{\partial w_1}\right)^2 + \left(\frac{\partial L}{\partial w_2}\right)^2\right) \leq 0$ if $r > 0$.
- The **gradient** $\left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}\right)$ is denoted by ∇L, then $\Delta L \approx -r\|\nabla L\|^2$.

Gradient Decent

- We can generalize the previous discussion to *k*-dimensional *w*.
- The update rule is simply:

$$w \leftarrow w - r\,\nabla L,$$

where:

- r is the (positive) learning rate, ∇L is the gradient vector.
- If we decompose the vector:

$$w = (w_1, w_2, \ldots, w_k),\ \nabla L = \left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_k}\right),$$

then the update rule is:

$$w_i \leftarrow w_i - r\,\frac{\partial L}{\partial w_i},\ 1 \le i \le k.$$

- The remaining challenge is how to calculate $\frac{\partial L}{\partial w_i}$ for each parameter w_i.

Pseudocode of Gradient Decent

- Problem: finding the model parameters to minimize the loss $L(D, w)$ on a given dataset $D = \{(x_i, y_i) | 1 \le i \le n\}$.

/* **Gradient decent algorithm** */
Initialize w
while not converge {
    calculate $L(D, w)$
    calculate $\nabla L = (\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_k})$
    $w \leftarrow w - r \nabla L$
}

Drawback of gradient decent:

Given a large dataset D,
it may take a very long time to
calculate $L(D, w)$:

for(i = 1; i ≤ n; i++) {
    calculate $f(x_i, w)$
    …
}

We adjust the parameters only once, after going through the whole dataset.

Mini-batch Stochastic Gradient Decent (SGD)

- A more practical solution: Randomly divide dataset D into multiple mini-batches, each with *B* samples.
  - Now we have $S = D/B$ mini-batches, denoted as D_1, D_2, …, D_S. According to statistics, we hope $L(D_i, w)$ can approximate $L(D, w)$.

```
/* Mini-batch SGD algorithm */
Initialize w
while not converge {
    Randomly shuffle D and divide it into mini-batches D_1, ..., D_S
    for(i = 1; i ≤ S; i++)
        calculate L(D_i, w)
        calculate ∇L = (∂L/∂w_1, ∂L/∂w_2, ..., ∂L/∂w_k)
        w ← w − r ∇L
    } // this is called one epoch
}
```

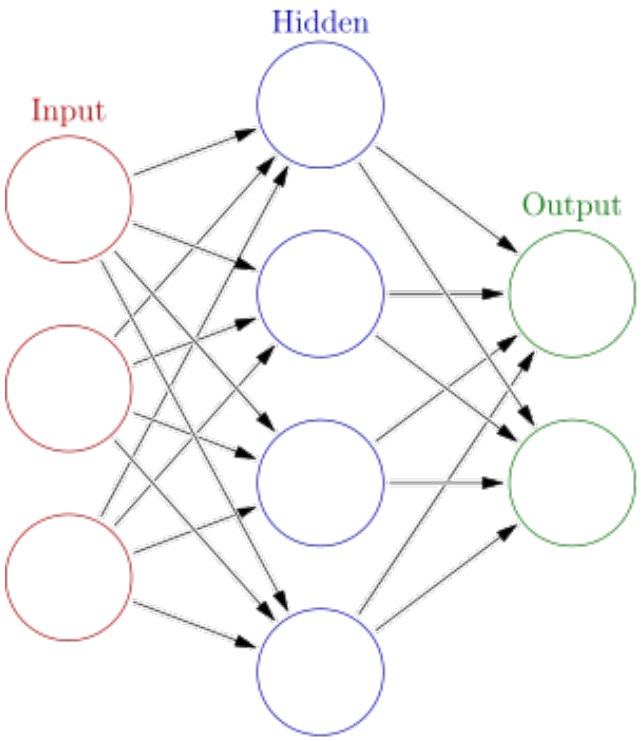
Details later

We adjust the parameters *S* times in one epoch!

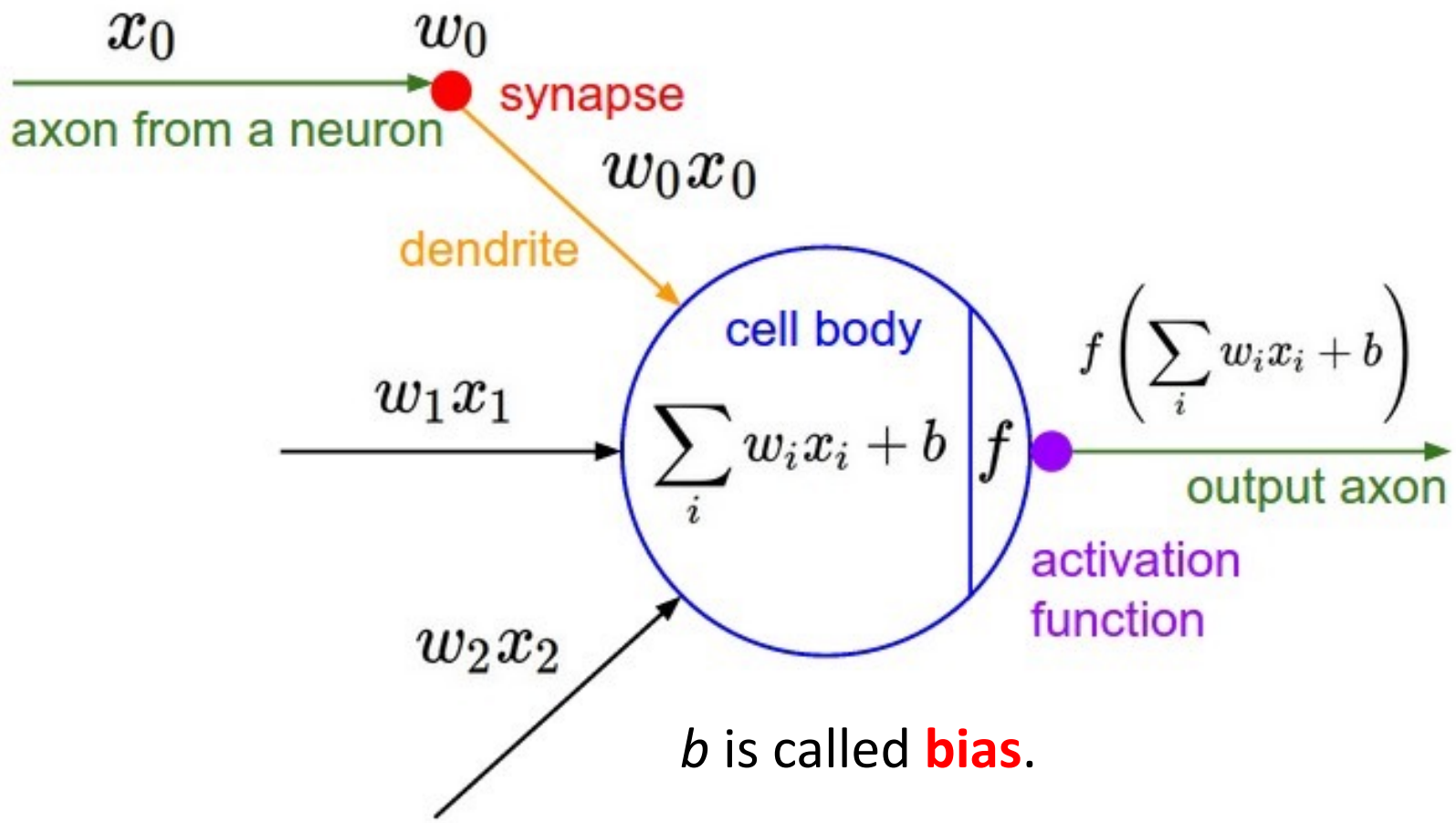
What is Artificial Neural Network (ANN)

- ANN is a category of machine learning models inspired by biological neurons.

- Like many other machine learning algorithms, an ANN model takes some input (such as text, image, audio) and generates some output.



Neurons



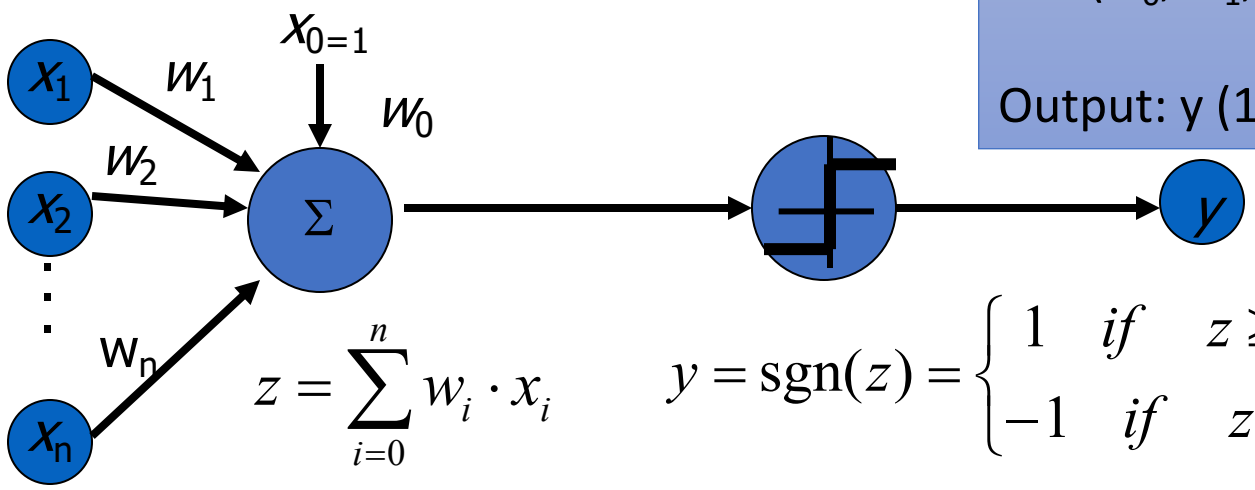
b is called **bias**.

Perceptron Learning based on Binary Neuron

The bias item can be denoted by $\boldsymbol{w_0}$, corresponding to an implicit input $x_0 = 1$.

Input:
$x = (1, x_1, x_2, \ldots, x_n)$
Weight parameters:
$\boldsymbol{w} = (w_0, w_1, \ldots, w_n)$

Output: y (1 or -1)



$$z = \sum_{i=0}^{n} w_i \cdot x_i$$

$$y = \mathrm{sgn}(z) = \begin{cases} 1 & if \quad z \geq 0 \\ -1 & if \quad z < 0 \end{cases}$$

Vector form:

$$z = \boldsymbol{w}^T \boldsymbol{x}$$

Activation function: a step function which is discontinuous

Perceptron Learning (1957, by Frank Rosenblatt)

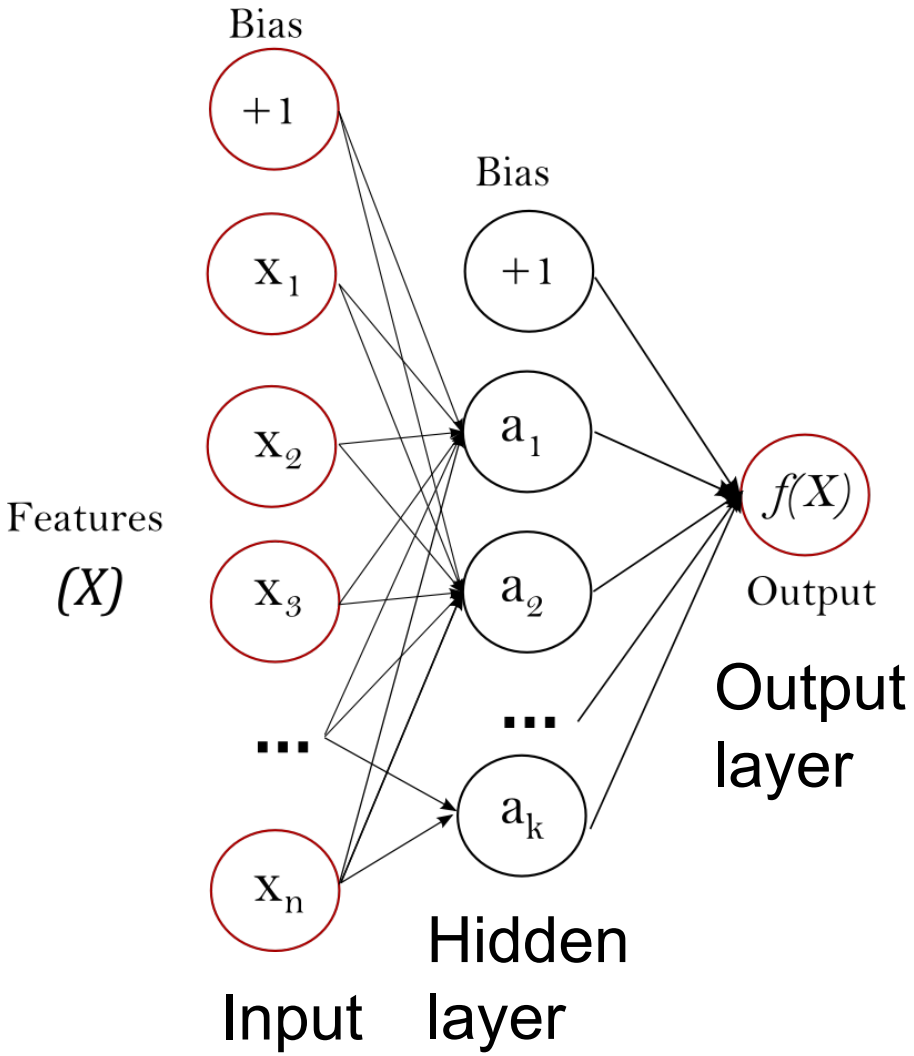
- Perceptron can work as a simple **binary classifier**, i.e., the output of the model is either +1 or -1, corresponding to two different classes.
- Question: how to find the parameters $\boldsymbol{w}$ from a set of training data?

An iterative solution:

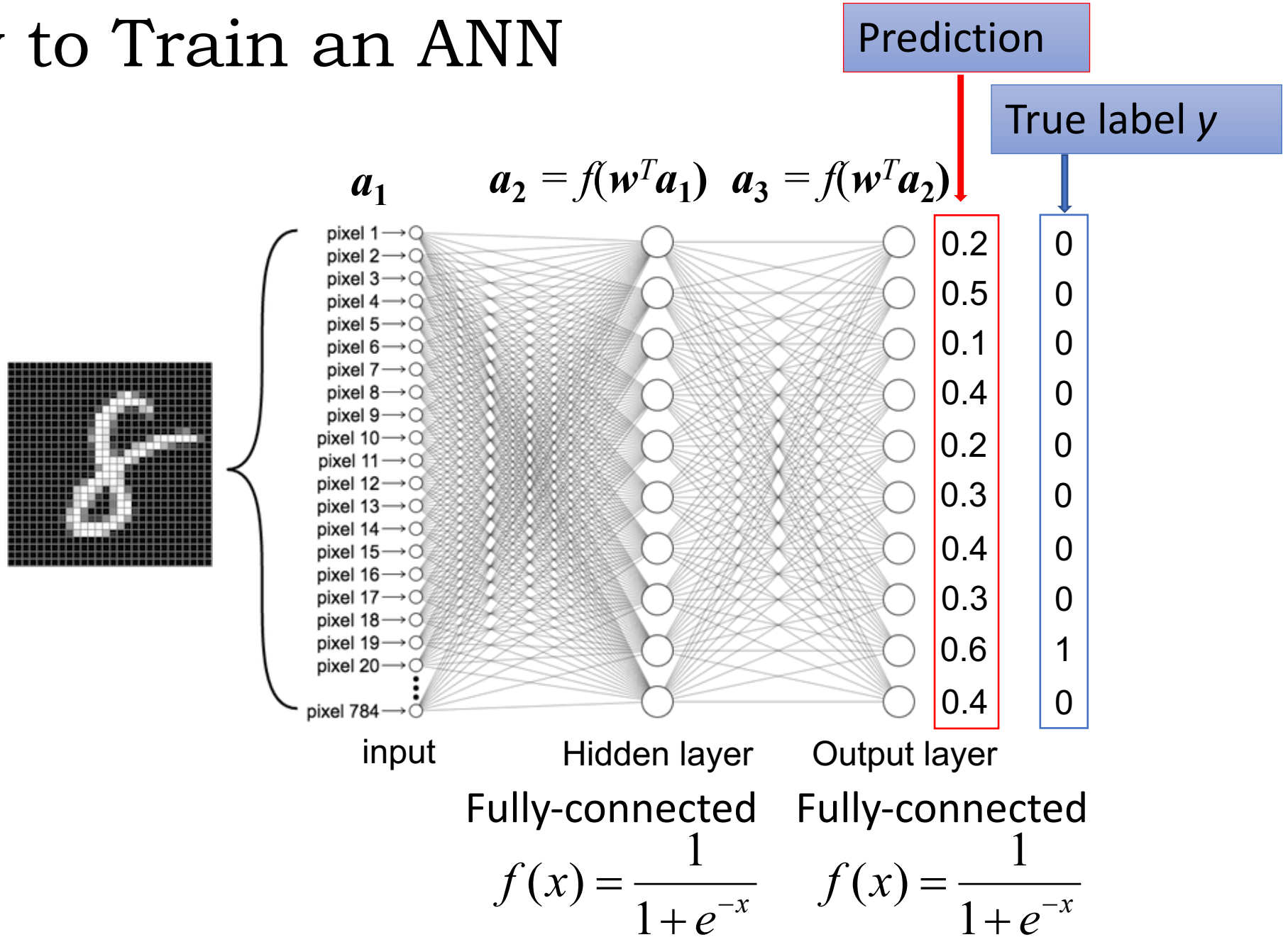
1. Initialize the weights to 0, or small random numbers
2. [An **epoch**]: for each training sample $(\boldsymbol{x}^i, \boldsymbol{y}^i)$ in the training data set, perform the following steps:
   - Compute the prediction value $\hat{\boldsymbol{y}} = \boldsymbol{sgn}(\boldsymbol{w}^T\boldsymbol{x}^i)$
   - If $\hat{\boldsymbol{y}}$ is different from the true $\boldsymbol{y}^i$, update the weights
     - For j = 0 to n: $\boldsymbol{w}_j = \boldsymbol{w}_j + \boldsymbol{r}(\boldsymbol{y} - \hat{\boldsymbol{y}})\boldsymbol{x}_j^i$, where $0 < \boldsymbol{r} < 1$ is called **learning rate**, $\boldsymbol{w}_j$ is the j-th component of vector $\boldsymbol{w}$, $\boldsymbol{x}_j^i$ is the j-th component of vector $\boldsymbol{x}^i$
3. Go back to Step 2, until there is no weight update or we reach the maximum number of epochs.

Multi-layer Perceptron as Neural Network

- Perceptron is a single-layer neural network. It has limited capacity in approximating functions (linearly separable).
- But it turns out that multi-layer perceptron (MLP) can approximate very complex functions.
- Between the input and output, there is one or more hidden layers.
- A neuron in a layer is connected to neurons in its previous layer. Each link has a weight value.

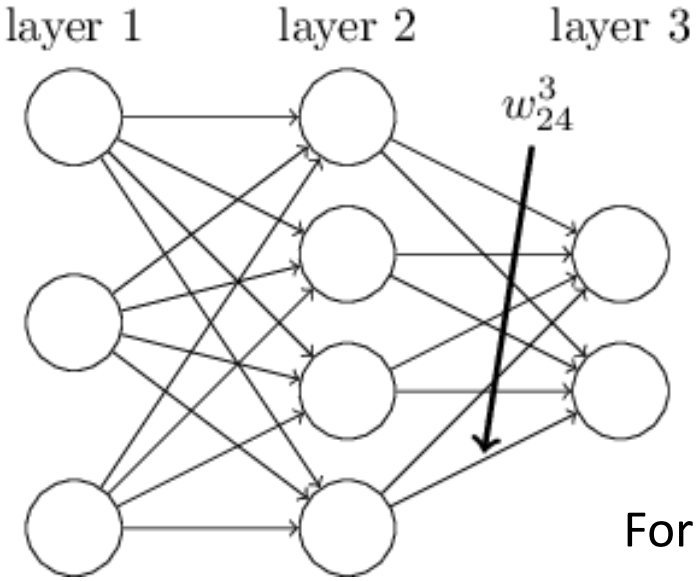


How to Train an ANN



Mini-batch SGD + Backpropagation

- Mini-batch SGD provides the architecture of training an ANN.
- However, we need to find a way to calculate $\nabla L = (\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_k})$, which is not an easy task for ANN.
- Error backpropagation is a method of finding gradients for ANNs.



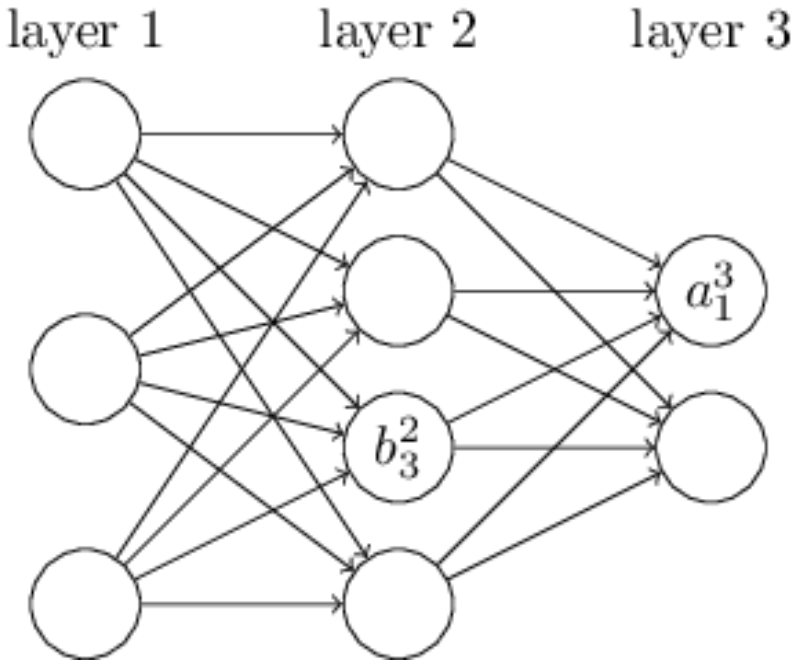
w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

For example, weights of layer 3 is: $\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix}$

Ack: Some diagrams are from http://neuralnetworksanddeeplearning.com/chap2.html

Feed-forward

- We use b^l_j for the bias of the j-th neuron in the l-th layer.
- We use a^l_j for the activation of the j-th neuron in the l-th layer.



- Elementwise:

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

- Vectorized form:

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

where a^l is the activation vector, w^l is the weight matrix, and b^l is the bias vector.

Notations

- The input to the *j*-th neuron at layer *l*: $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$
- The activation of the *j*-th neuron: $a_j^l = \sigma(z_j^l)$

- Define the **error** ${\delta_j}^l$ of the *j*-th neuron: ${\delta_j}^l = \frac{\partial C}{\partial z_j^l}$

- Error backpropagation is a general method of finding all "errors", from the last layer back to the first layer.

- The gradients

$$\frac{\partial C}{\partial w_{jk}^l} \text{ and } \frac{\partial C}{\partial b_j^l}$$

can then be calculated based on these errors.

Vector and Matrix Forms

- Weight matrix for layer *l*: w^l
- Bias vector for layer *l*: b^l
- Input vector for layer *l*: z^l
- Activation vector for layer *l*: a^l
- Vectorizing the activation function: $z^l = w^l a^{l-1} + b^l$
- Hadamard product $s \odot t$: element-wise product of two vectors *s* and *t* of the same dimension. E.g., $a^l = \sigma(z^l)$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

Finding Errors of Last Layer

- Error depends on the loss function. Usually, the average loss depends on individual loss of a single input *x*, so let's focus on a single loss. Consider the following loss function:

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2$$

- From calculus, we can have:

$$\partial C / \partial a_j^L = (a_j^L - y_j)$$

- Use $\nabla_a C$ to denote the vector of partial derivatives $\partial C / \partial a_j^L$.

- The errors in the last layer (layer L) can be calculated by:

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

Error Backpropagation

- The errors of layer *l* are calculated from the errors of layer *l*+1 and the weight matrix of layer *l*+1.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

- Gradients are then calculated from errors:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

Backpropagation Algorithm

the "details" part of the SGD algorithm in Page 17

1. **Input a set of training examples**

2. **For each training example** x**:** Set the corresponding input activation $a^{x,1}$, and perform the following steps:

   - **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$.

   - **Output error** $\delta^{x,L}$**:** Compute the vector $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.

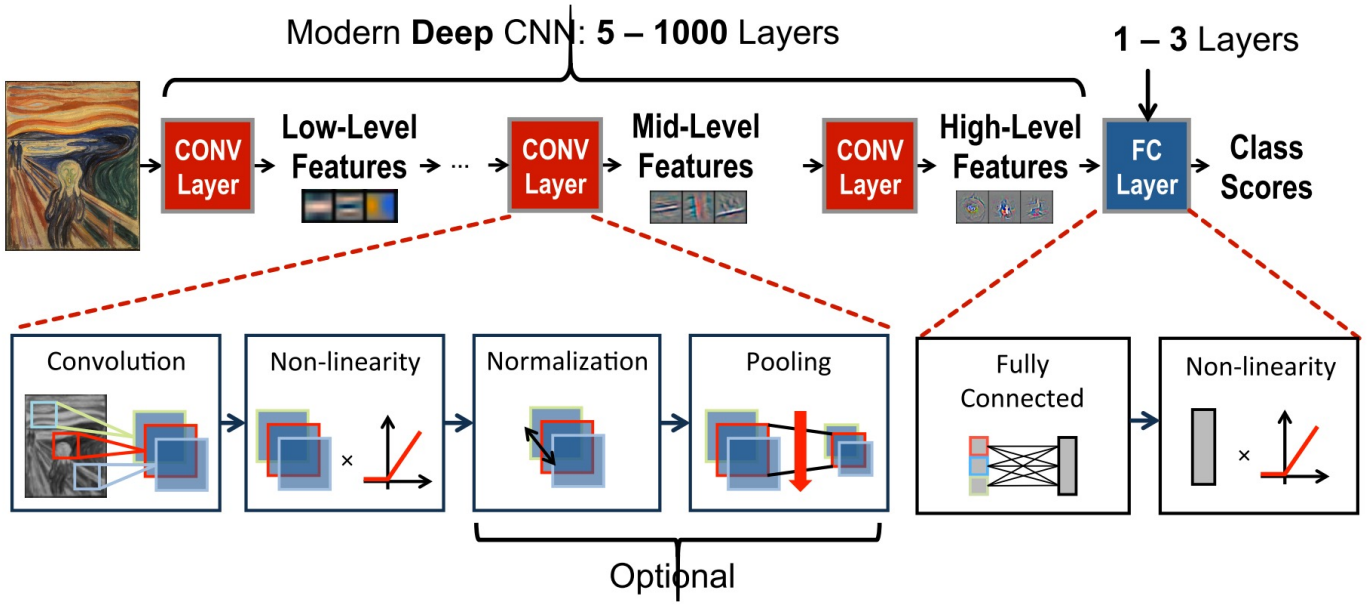
   - **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.

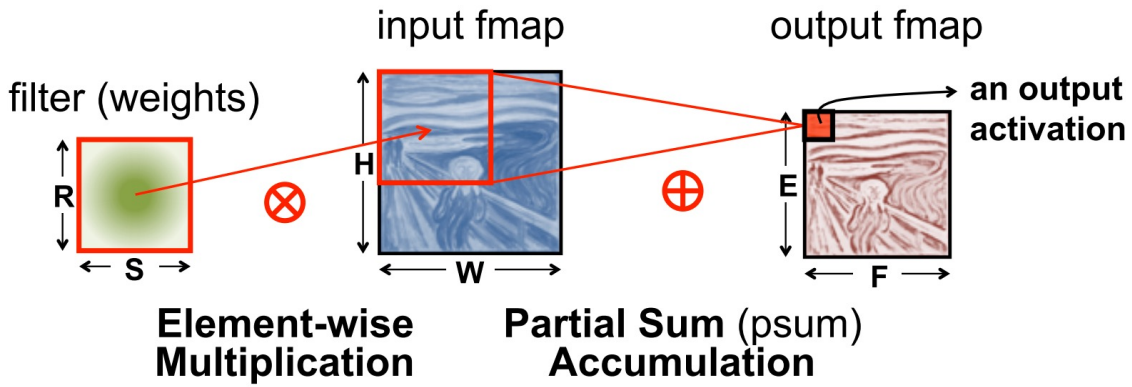
3. **Gradient descent:** For each $l = L, L-1, \ldots, 2$ update the weights according to the rule $w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and the biases according to the rule $b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

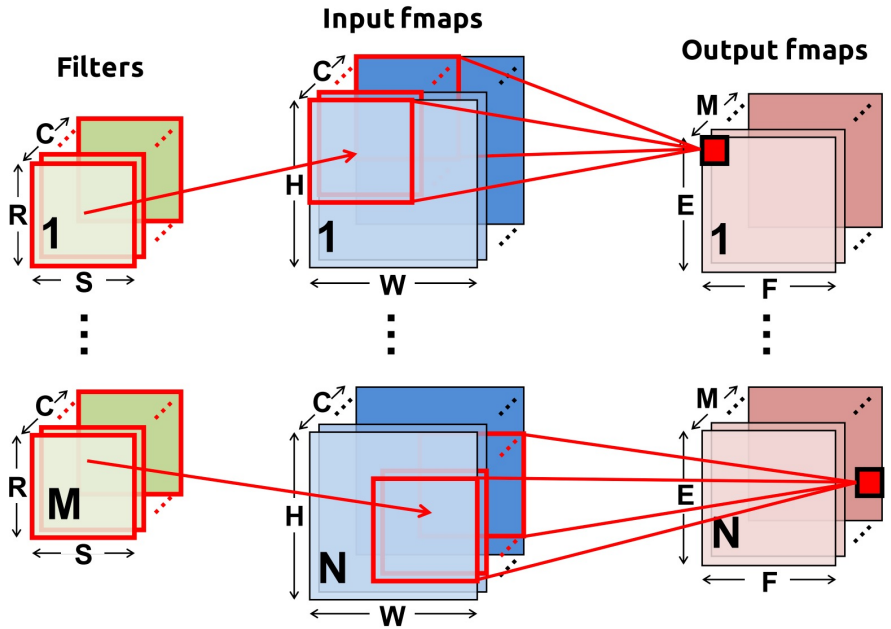
Notice that this outer product generates a matrix.

DNN Training is Computing-intensive

- Convolutional neural networks (CNNs) for Computer Vision (CV)







Source: Sze, Vivienne, et al. "Efficient processing of deep neural networks: A tutorial and survey." *Proceedings of the IEEE* 105.12 (2017): 2295-2329.

CNN Workloads (One Iteration)

Layer Type	Eval.	Work (W)
Activation	y	$O(NCHW)$
	∇w	$O(NCHW)$
	∇x	$O(NCHW)$
Fully Connected	y	$O(C_{out} \cdot C_{in} \cdot N)$
	∇w	$O(C_{in} \cdot N \cdot C_{out})$
	∇x	$O(C_{in} \cdot C_{out} \cdot N)$
Convolution (Direct)	y	$O(N \cdot C_{out} \cdot C_{in} \cdot H' \cdot W' \cdot K_x \cdot K_y)$
	∇w	$O(N \cdot C_{out} \cdot C_{in} \cdot H' \cdot W' \cdot K_x \cdot K_y)$
	∇x	$O(N \cdot C_{out} \cdot C_{in} \cdot H \cdot W \cdot K_x \cdot K_y)$
Pooling	y	$O(NCHW)$
	∇w	–
	∇x	$O(NCHW)$
Batch Normalization	y	$O(NCHW)$
	∇w	$O(NCHW)$
	∇x	$O(NCHW)$

Workloads of each different types of layers

N is the mini-batch size

Metrics	LeNet 5	AlexNet	Overfeat fast	VGG 16	GoogLeNet v1	ResNet 50
Top-5 error†	n/a	16.4	14.2	7.4	6.7	5.3
Top-5 error (single crop)†	n/a	19.8	17.0	8.8	10.7	7.0
Input Size	28×28	227×227	231×231	224×224	224×224	224×224
# of CONV Layers	2	5	5	13	57	53
Depth in # of CONV Layers	2	5	5	13	21	49
Filter Sizes	5	3,5,11	3,5,11	3	1,3,5,7	1,3,7
# of Channels	1, 20	3-256	3-1024	3-512	3-832	3-2048
# of Filters	20, 50	96-384	96-1024	64-512	16-384	64-2048
Stride	1	1,4	1,4	1	1,2	1,2
Weights	2.6k	2.3M	16M	14.7M	6.0M	23.5M
MACs	283k	666M	2.67G	15.3G	1.43G	3.86G
# of FC Layers	2	3	3	3	1	1
Filter Sizes	1,4	1,6	1,6,12	1,7	1	1
# of Channels	50, 500	256-4096	1024-4096	512-4096	1024	2048
# of Filters	10, 500	1000-4096	1000-4096	1000-4096	1000	1000
Weights	58k	58.6M	130M	124M	1M	2M
MACs	58k	58.6M	124M	130M	1M	2M
Total Weights	60k	61M	146M	138M	7M	25.5M
Total MACs	341k	724M	2.8G	15.5G	1.43G	3.9G

Summary of popular CNNs (MACs at feed-forward for one data sample)

*Backpropagation is typically around 2 times larger workloads than feed-forward

Source:

Ben-Nun, Tal, and Torsten Hoefler. "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis." *ACM Computing Surveys (CSUR)* 52.4 (2019): 1-43.

Sze, Vivienne, et al. "Efficient processing of deep neural networks: A tutorial and survey." *Proceedings of the IEEE* 105.12 (2017): 2295-2329.

CNN Workloads (Training)

- SGD is an iterative algorithm

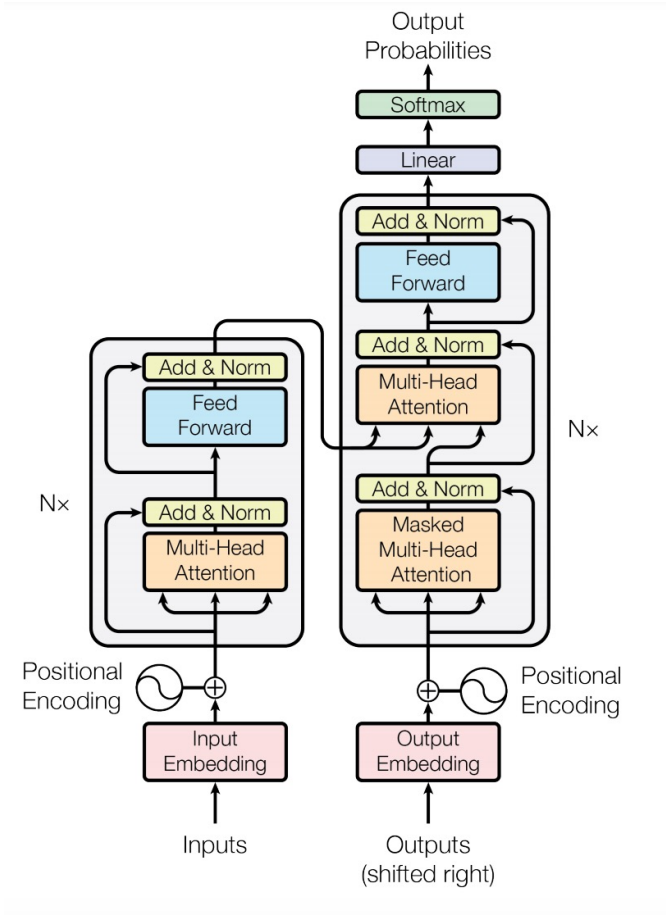
$$W_{i+1} = W_i - \eta \cdot \frac{1}{N} \sum_{j=1}^{N} \nabla L(W_i, D_i^j)$$

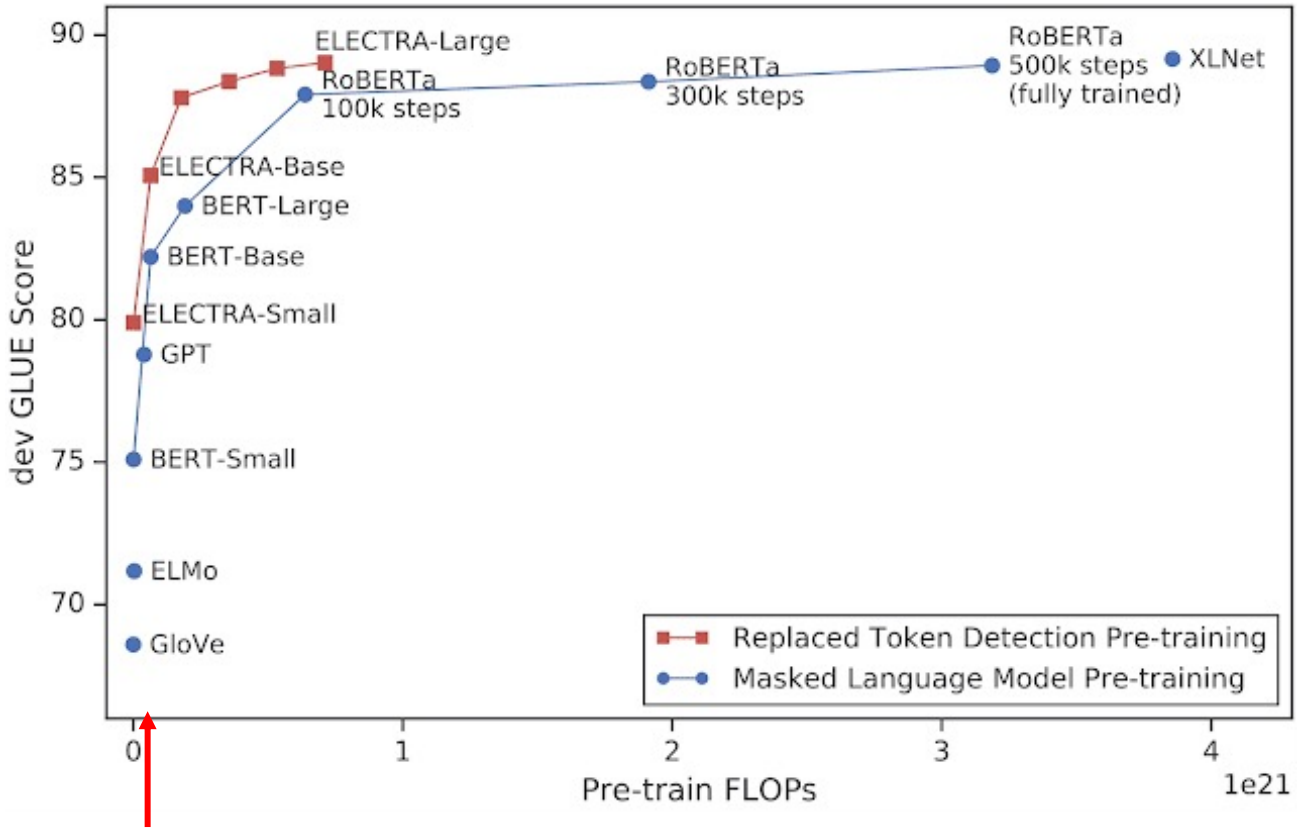
- Typically go through the whole dataset hundreds of times
  - **Epoch**: go through the data set one time
  - **Iteration**: each update of the above equation is one iteration
  - 100 epochs – 300 epochs to converge
  - An example
    - Dataset: ImageNet-2012 has ~1.28 million training images
    - CNN: ResNet-50 with ~11.7G MACs (Feed-forward+backward) and ~25M parameters
      - One epoch: $2*11.7G*1.28*10^6$=~30000TFLOP
      - Typically requires **90** epochs
    - Nvidia Tesla V100 GPU with 32GB memory: ~112 TFLOPS (with Tensor Cores)
    - Theoretical result
      - minimal time: 30000*90/112 = ~24000 seconds = **~6.7 hours**
    - Real-world best practice
      - MXNet with a mini-batch size of 256, runs at the speed of ~1,457 images/sec#
      - 90 epochs take 90 * $1.28*10^6$ / 1457 = ~79000 seconds = **~22 hours**

#. https://developer.nvidia.com/deep-learning-performance-training-inference

Transformer Workloads (Training)

- Transformer for natural language processing (NLP)
  - Many matrix multiplication operations





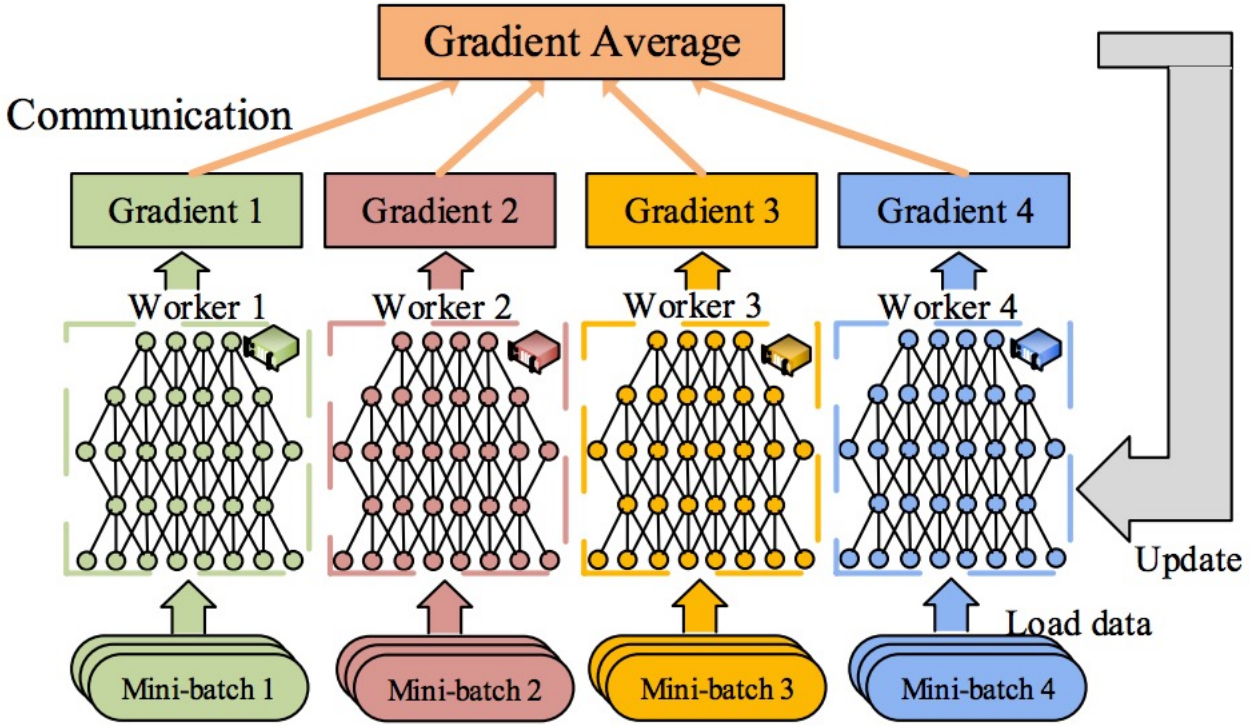
#1 supercomputer: 2.146176e18 FLOPS
(Half precision for AI)!

Source:
Vaswani, Ashish, et al. "Attention is All you Need." NIPS. 2017.
https://ai.googleblog.com/2020/03/more-efficient-nlp-model-pre-training.html

How GPU Clusters Accelerate DNN Training

- Data parallelism

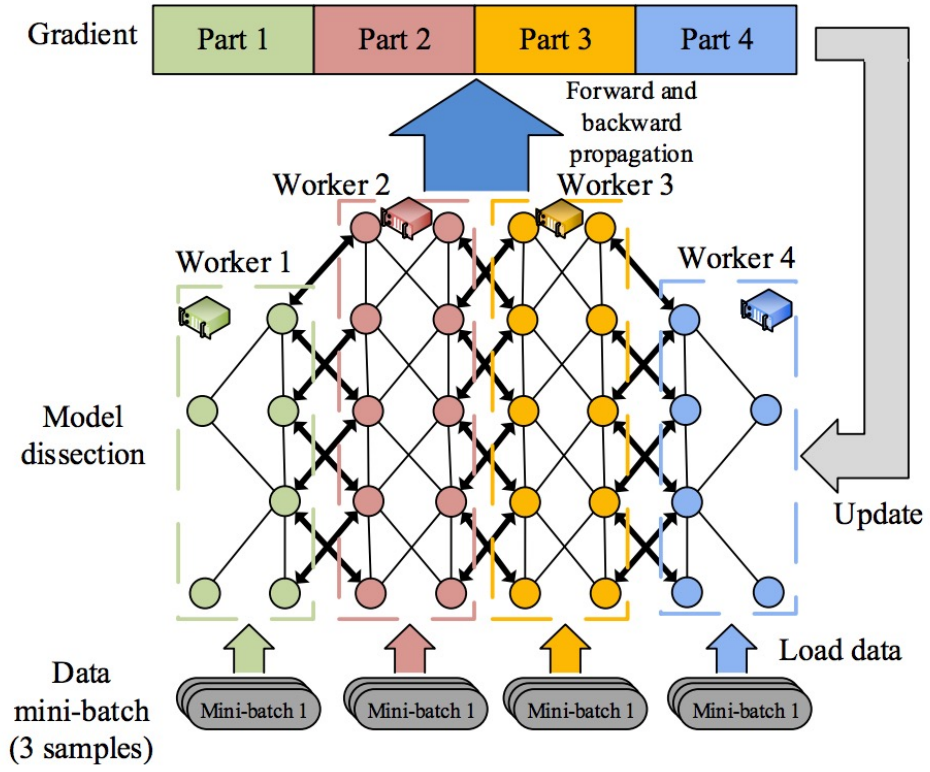
$$W_{i+1} = W_i - \eta \cdot \frac{1}{N} \sum_{j=1}^{N} \nabla L(W_i, D_i^j)$$

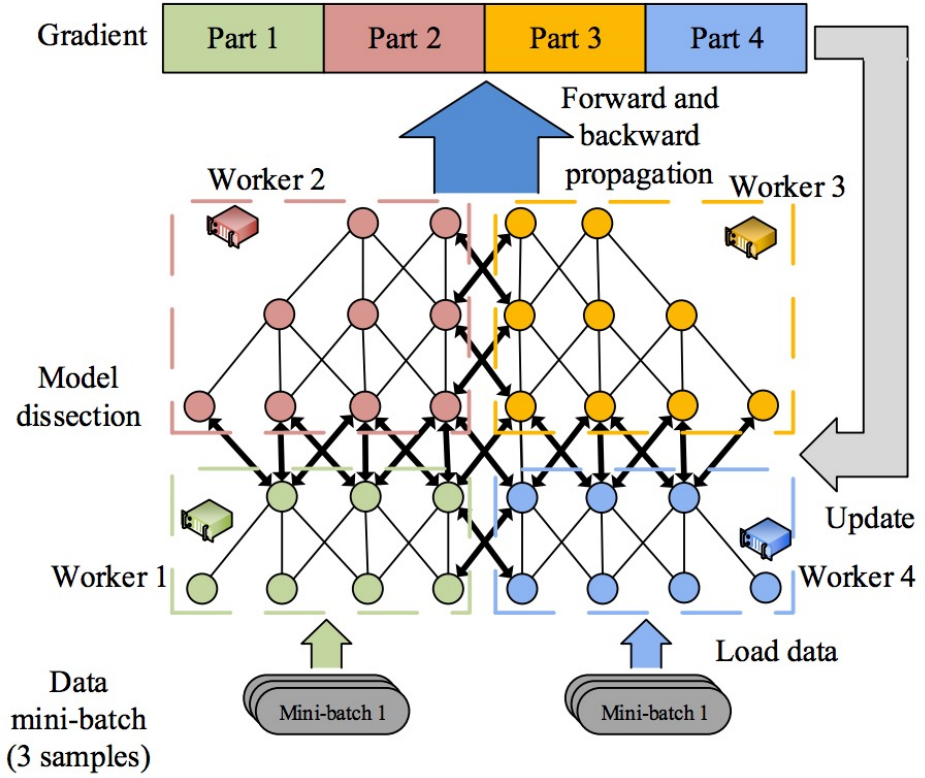


Source: Tang et al., Communication-Efficient Distributed Deep Learning: A Comprehensive Survey, 2020

How GPU Clusters Accelerate DNN Training

- Model parallelism
  - Two types of partition

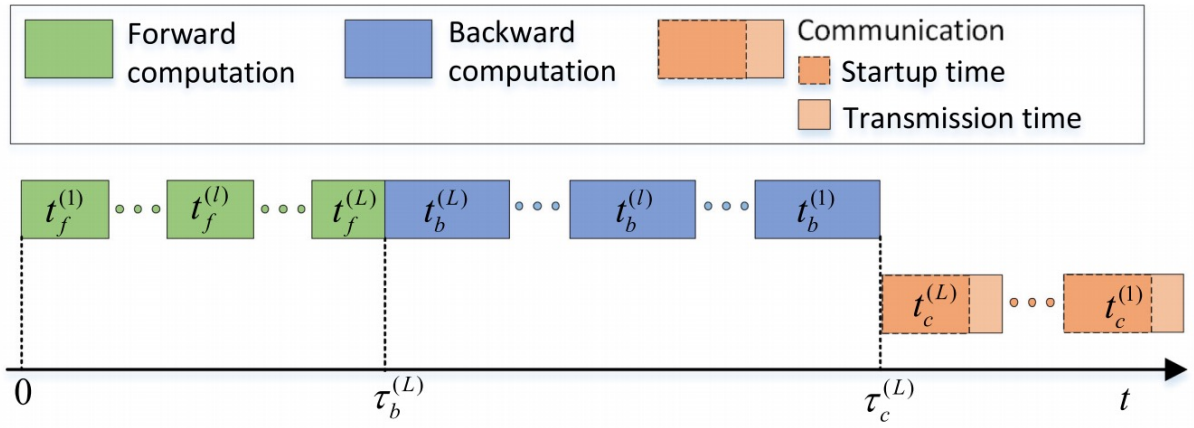




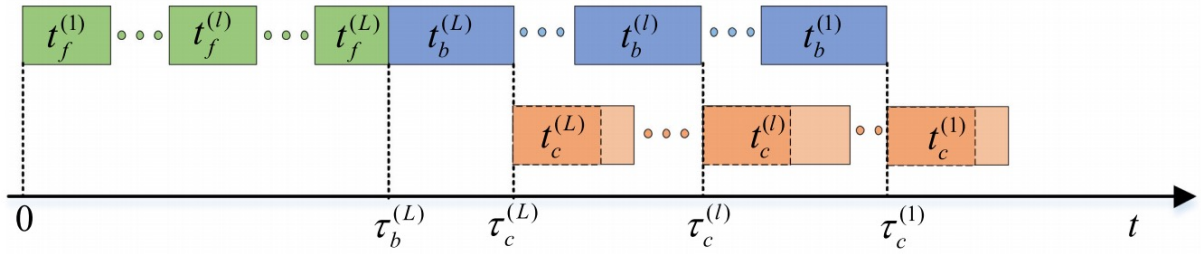
Source: Tang et al., Communication-Efficient Distributed Deep Learning: A Comprehensive Survey, 2020

How GPU Clusters Accelerate DNN Training

- Pipelining in data parallelism
  - Gradient communications can be overlapped with computations



(a) Naive S-SGD.

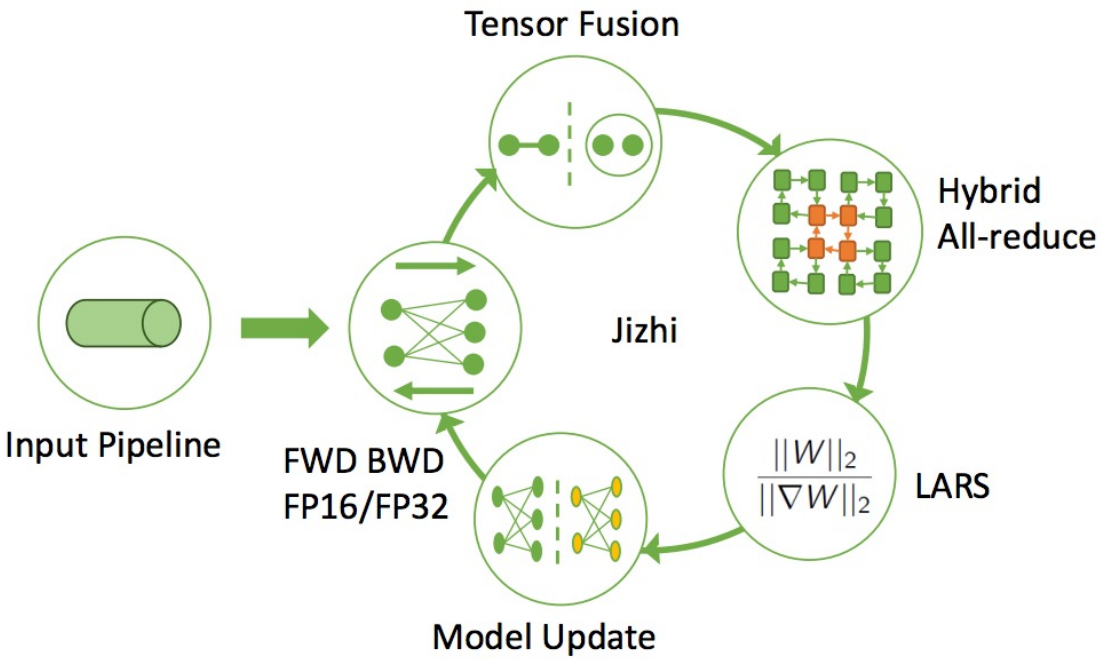


(b) WFBP S-SGD.

Source: Shi et al., MG-WFBP: Merging Gradients Wisely for Efficient Communication in Distributed Deep Learning, TPDS 2021

A Case: Scaling ResNet-50 Training to 1024 GPUs

- Input pipeline
  - **Parallelism** between data reading and computation: Reduce I/O time
  - Input data preprocessing
    - multi-processing/multi-threading/**OpenMP**
- FWD BWD
  - Feed-forward and backpropagation computation with FP16: Improve GPU throughput (**GPU Computing)**
- Tensor Fusion
  - Fuse small tensors to large tensors for communication: Improve communication throughput
- Hybrid All-Reduce
  - Intra-node communication vs. inter-node communication: Better utilize bandwidth resources to reduce communication time (**MPI Programming**)



- LARS
  - Convergence guarantees for large-batch training

Source: Jia et al., Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes, NeurIPS workshop 2018

Testbed

- Hardware
  - 256 nodes connected with 100Gbps Ethernet (i.e., Mellanox ConnectX-4 that supports RoCEv2)
    - Will have 128-node and 256-node experiments
  - Each node:
    - 8 Nvidia Tesla P40 GPUs (PCIe)
    - 2T NVMe SSDs
- Software
  - TensorFlow: Deep learning framework
    - low-level code: C/C++
    - high-level usage: Python
  - cuDNN: High-performance CUDA library for DNN operators on GPUs
  - Horovod and OpenMPI: Distributed training framework with data parallelism
  - NCCL: High-performance communication library for GPU clusters

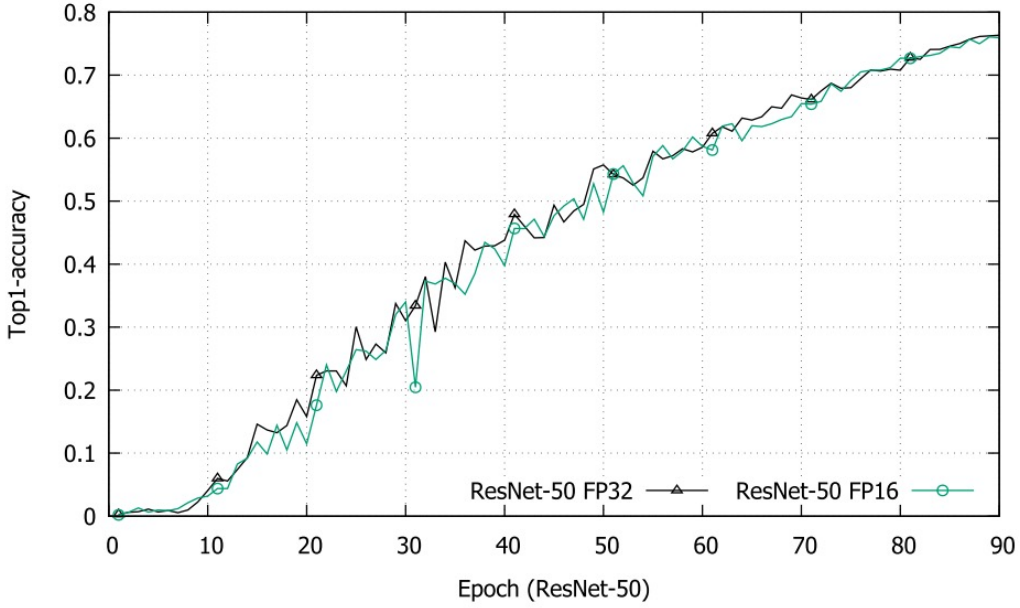
Source: Jia et al., Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes, NeurIPS workshop 2018

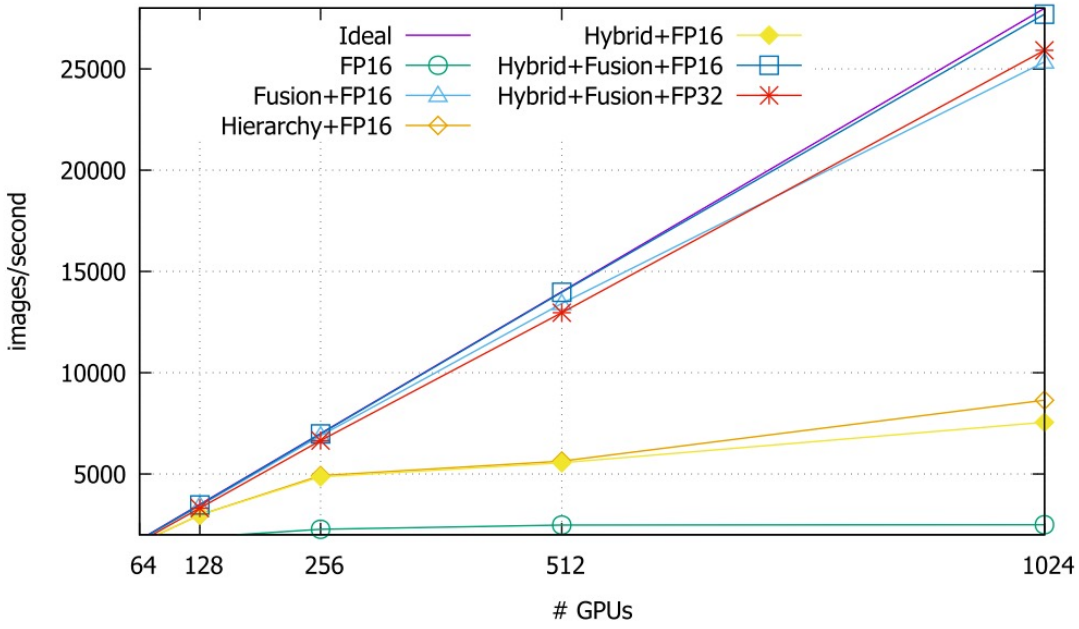
Results

- Convergence
  - Train the model in 90 epochs
  - Top-1 validation accuracy: 76.3%

- Weak-scaling performance
  - Single-node (8-GPU) performance: 218 images/s with a mini-batch size of 64 per GPU
  - 128-node (1024-GPU) performance: ~27500 images/s
  - Scaling efficiency: 25700/218/128 = ~99.2%
  - 90 epochs run at **7.8** minutes

- 256-node (2048 GPUs)
  - 90 epochs run at **6.6** minutes
  - Only **18%** improvement over 128-node with doubled GPUs!
    - Can be further optimized!





Source: Jia et al., Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes, NeurIPS workshop 2018

Production-ready Distributed Training Frameworks

- PyTorch-DDP
  - https://pytorch.org/docs/stable/notes/ddp.html
  - particularly for data parallelism
- DeepSpeed
  - https://github.com/microsoft/DeepSpeed
  - particularly for model parallelism
- Horovod
  - https://github.com/horovod/horovod
  - particularly for data parallelism
- Ray
  - https://www.ray.io/
  - For generic distributed computing

Summary

- Machine Learning Basics
  - Supervised Machine Learning
  - Review of Calculus
  - Gradient Decent
- (Deep) Neural Networks
  - From neurons to neural networks
- Model Training: SGD and Backpropagation
  - The fundamental training algorithm in deep learning
- How GPU Clusters Accelerate Deep Neural Networks Training
  - High-performance computers are necessary for large-scale deep learning