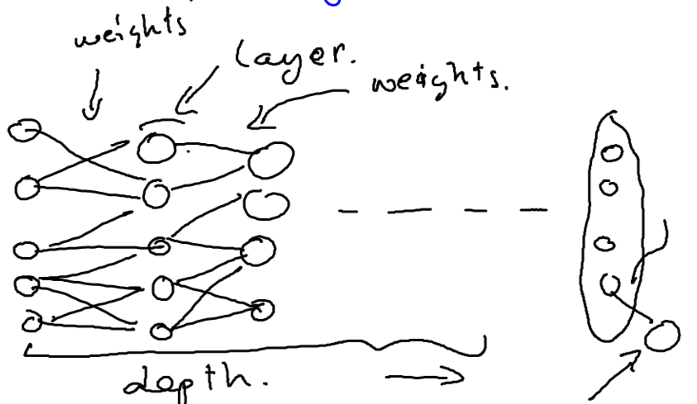
Optimization for Machine Learning (Lecture 6).

- A Neural Network (NN)



$L_1 \Rightarrow L_2 \Rightarrow \cdots$ Loss function.

How to compute ∇F
"the gradient function"?

Backpropagation.

A rule from calculus, the chain rule: z depend on y and y depends on x.

$$\frac{dz}{dx} = \underbrace{\frac{dz}{dy}}_{(1)} \; \underbrace{\frac{dy}{dx}}_{(2)}$$

$$z = (y)^2 \;(1), \quad y = 2x \quad \Rightarrow z = (2x)^2 = 4x^2$$

$$\frac{dz}{dy} = 2y, \quad \frac{dy}{dx} = 2 \quad \Rightarrow \frac{dz}{dx} = 2y \times 2 = 4y$$
$$= 8x$$

$$z' = 4 \times (2x) = 8x.$$

How does it translate to multivariate function or having vector as parameters.

$y = (y_1, \cdots, y_m)$.

$(x_1, x_2, \cdots, x_m) = x \in \mathbb{R}^m, \; y \in \mathbb{R}^n, \; z \in \mathbb{R}$: variables

$g: \mathbb{R}^m \to \mathbb{R}^n, \; f: \mathbb{R}^n \to \mathbb{R}$: functions.

$$\to z = f(y), \quad y = g(x)$$

Loss.

$$\to x \xrightarrow{g} y \xrightarrow{f} z$$

$$\underbrace{\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i}}_{\nabla \text{ components}}$$ (chain rule). (generalized to vectors).

Can be written compactly as: transpose

$$\nabla_x(z) = \underbrace{\left(\frac{\partial y}{\partial x}\right)^T}_{\text{Correspond to (2).}} \underbrace{\nabla_y(z)}_{\text{correspond to (1)}}$$

$\frac{\partial y}{\partial x}$ is $n \times m$ matrix: Jacobian of g or y.

$$J = \frac{\partial y}{\partial x} = \begin{array}{cc} & \\ n & \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & & \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{pmatrix} \\ & m \end{array}$$

matrix of all possible partial derivative I can take.

$$\left(\frac{\partial y}{\partial x}\right)^T = \begin{array}{cc} \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{pmatrix} & m \\ n & \end{array}$$

$m \times n$ matrix.

$$\left(\frac{\partial y}{\partial x}\right)^T \nabla_y(z) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{pmatrix} \begin{pmatrix} \frac{\partial z}{\partial y_1} \\ \vdots \\ \frac{\partial z}{\partial y_n} \end{pmatrix}$$

$(m \times n) \times (n \times 1) = m \times 1$

$$= \left(\sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_1} \cdots\cdots \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_m} \right)^T$$

$$= \nabla_x z$$ ($m \times 1$) matrix aka vector.

$$\boxed{\nabla_x(z) = \left(\frac{\partial z}{\partial y}\right)^T \nabla_y(z)} \quad (3)$$

(can be recursively called).

A derivative can be computed recursively in the reverse order. (Let's say for each function we implement a Jacobian, then we multiply as in (3) to obtain the gradient of the loss function).

Another example (More simple). (why we need a forward pass for backprop).

$$\text{input} \to x \xrightarrow{g} \overset{g(x)}{y} \xrightarrow{f} \overset{f(y)=f(g(x))}{z} \xrightarrow{e} u$$

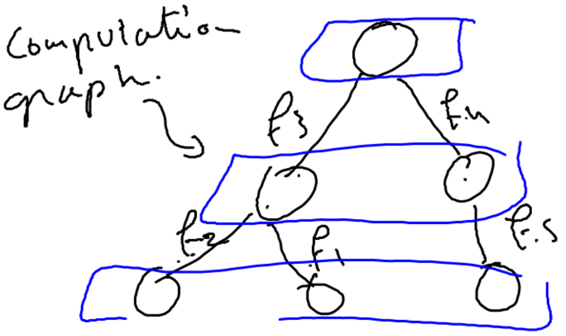
$$\frac{\partial u}{\partial x} = \underbrace{\frac{\partial u}{\partial z}}_{,} \times \underbrace{\frac{\partial z}{\partial y}} \times \underbrace{\frac{\partial y}{\partial x}}, \qquad u = e(z) = e'(z)$$

$$\frac{\partial u}{\partial x} = \underline{e'}(\underline{f(g(x))}) \times \underline{f'}(g(x)) \times \underline{g'}(x)$$

How to obtain: $\underline{f(g(x))}$, $\underline{g(x)}$

I use a forward pass.

Computation complexity:



forward pass. $O(n)$
n: # of nodes.

• Assuming that function evaluations have cost ?, $f(\cdot)$ // // $\frac{\partial f(\cdot)}{\partial x}$ has cost ?.

m : # of edges.

Back propagations: $O(n) + O(m) = O(n^2)$

In a fully connected graph. $O(n^2)$

Because NNs have chain structure

practically backprop. ~ $O(n)$ complexity.

Two types of derivates used in practice.

symbol-to-number: Torch, caffe.

symbol-to-symbol: Theano, Tensorflow.

↳ adds extra nodes to the computational graph, and the derivatives are computed by traversing the graph, where sa. symbol to-number are computed within nodes.

Techniques used in NN training.

Surrogate loss functions and Early stopping.

what is a surrogate loss function is a gradient friendly function (friendly = I can take the derivate), for example, (0-1) loss. $\mathbb{1}\{y_i \neq \hat{y}_i\}$ instead to work with this directly, I can work (NLL) negative Log-Likelihood.

- As the optimization is executed. I use a criterion on the validation set based on the true loss (0-1 loss) (to avoid overfitting).

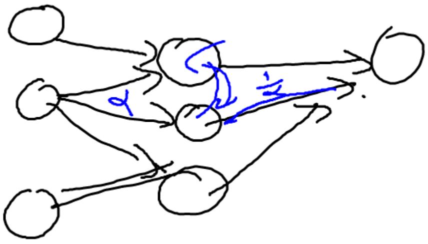
I stop when it's satisfied. ⇒

example (loss is not changing or low enough).

the gradient may still be big.

≠ Classical optimization $\nabla F \simeq 0$.

NN Landscapes & Local minimas



I can swap nodes and I obtain an equivalent model.

(Model Identifiability).

- Weight space symmetry.
- scaling of the weights } equivalent models.
  $\frac{1}{\alpha}$, α.

⇒ a NN can have uncountably infinitly many local minimas.

- In practice, one suspects local minima values aren't much differend than global minima.

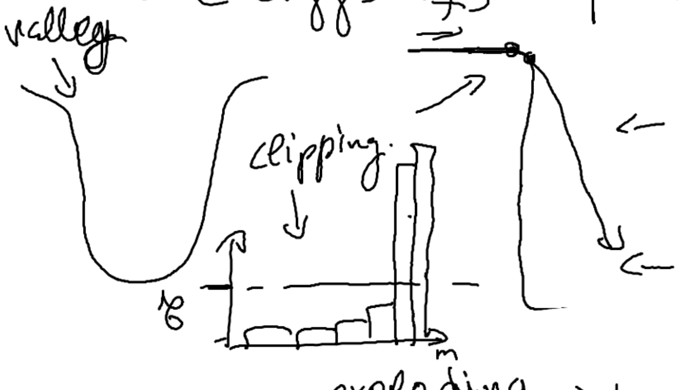
Other critical points (Maximas not much of a problem)

because it's unstable, usually SGD manges. because of randomness.

- saddle points Empirically gradient met
  (problem for (second order methods) ← they are attracted by saddle points.
  , 2nd order information, makes you attracted).

- Cliffs causes ⇒ exploding (big.) gradients.

method that is not proven too have optimality prop.



← huge gradient. } heuristic: to clip the gradient & only take direction information.

exploding → { - numerical instability.

L • driven away from minimas.

Adaptive Learning rates modifications for SGD

ADAGRAD: Adaptive gradient.

$$G_k = \sum_{k'=1}^{k} g(w_{k'}, \xi_{k'})\, g(w_{k'}, \xi_{k'})^T$$

$$(G_k)_{jj} = \Big(\sum_{k'=1}^{k} g\, \tilde{g}^T \Big)_{jj} \qquad \begin{pmatrix} a_1 & \\ & a_n \end{pmatrix}^{-\frac{1}{2}} = \begin{pmatrix} \frac{1}{\sqrt{a_1}} & \\ & \frac{1}{\sqrt{a_n}} \end{pmatrix}$$

Hadamard product.

$$w_{k+1} = w_k - \underbrace{\eta}_{\text{fixed}} \operatorname{diag}(G_k)^{-\frac{1}{2}} \odot g$$

$$(w_{k+1})_j = (w_k)_j - \frac{\eta}{\sqrt{G_{jj}}} \big(g(w_k, \xi_k)\big)_j$$

$\sqrt{G_{jj}} \simeq$ estimate of the Lipchitz constant

inspired by online learning techniques.

(ADAM) (Adaptive moment estimation).

Walking averge,

$$\begin{cases} m_{k+1} = \beta_1 \underbrace{m_k} + (1-\beta_1)\, g(w_k, \xi_k) \\ v_{k+1} = \beta_2 \underbrace{v_k}_{\text{past}} + \underbrace{(1-\beta_2)\big(g(w_k, \xi_k)\big)^2}_{\text{new value}} \end{cases}$$

$\underbrace{\beta_1, \beta_2}$: forgetting factors.

→ I forget faster.

$$\tilde{m} = \frac{m_{k+1}}{1-(\beta_1)^{k+1}} \qquad \tilde{v} = \frac{v_{k+1}}{1-(\beta_2)^{k+1}}$$

as k becomes larger I give more importance to recent values of v and m.

$$w_{t+1} = w_t - \eta \times \frac{\tilde{m}}{\sqrt{\tilde{v}} + \varepsilon}$$

gradient

to avoid division over zero.

RMS PROP (Root mean square propagation).

$$v(w_t) = \gamma\, v(w_{t-1}) + (1-\gamma)(\nabla g)^2.$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v(w,t)}}\, g \ldots$$

Parameter Initialization

- Break symmetry. somthing that we want.
- Randomly initialize the weights.
- If I make the weight large $\Rightarrow$

Break better the symmetry.

- " " large $\Rightarrow$ exploding gradients.

give $m \times n$ stage

- 2 general rule of thumbs. $w_{ij} \sim U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$
- $w_{ij} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$.

$\Rightarrow$ compromize between constant activation variance and constant gradient variance.

Batch normalization

Reparametrize the nn. Let H be a minibatch of activations at a hidden layer (we take the a subset).

$$\underset{\sim}{H'} = \frac{H-\mu}{\sigma} \quad \Bigg| \quad \mu = \frac{1}{M}\sum_{i=1}^{M} H_i$$

$$\sigma = \sqrt{\delta + \frac{1}{M}\sum_i (H-\mu)_i^2}$$

$\delta > 0$

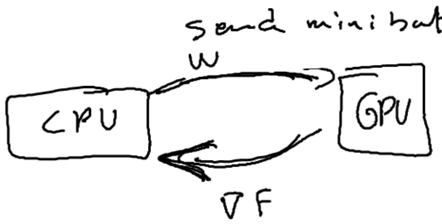
to have $\delta > 0$.
to avoid division over 0.

Distributed optimization

Usually it's benificial to split the work load if I have more hardware.

- I delegate tasks to Processing Units (PUs): CPUs, GPUs. TPUs.
  Tensor PUs.

Assume I am delegating tasks by sending minibatches then I receive gradient. n_{mb}: size of minibatche



t_c: compute time in GPU per data

compute time $= n_{mb} \times t_c$.

$$d_1 + \underbrace{\underline{n_{mb}\, t_T}}_{\text{time to transmit}} + \underbrace{\underline{n_{mb}\, t_c}}_{\text{time to comput}} + d_2 \leftarrow \text{time to prepare info}$$

transmission. $d_1 = d_2 = d$

time
to give data to GPU
iteration time.

Time to converge to ε precision.

$$T_\varepsilon \propto \left(\frac{1}{\varepsilon} \times \frac{1}{nmb}\right) \times \left(2d + nmb \times (t_c + t_T)\right)$$

Real time.

if $d = 0$. $nmb \in$ doesn't matter.

if $d > 0$ $nmb \gg 1$.

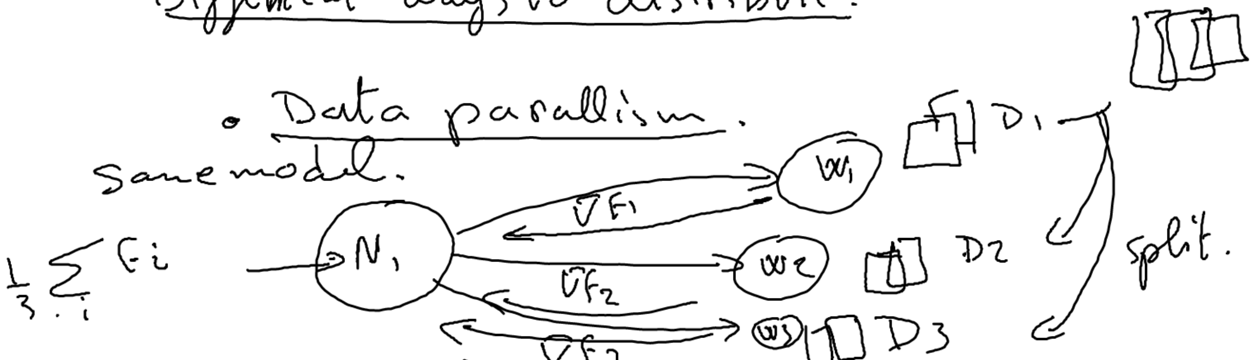
when $nmb \gg 1$

$$\boxed{T_\varepsilon = \frac{1}{\varepsilon}\left(t_c + t_T\right)}$$

Different ways to distribute.

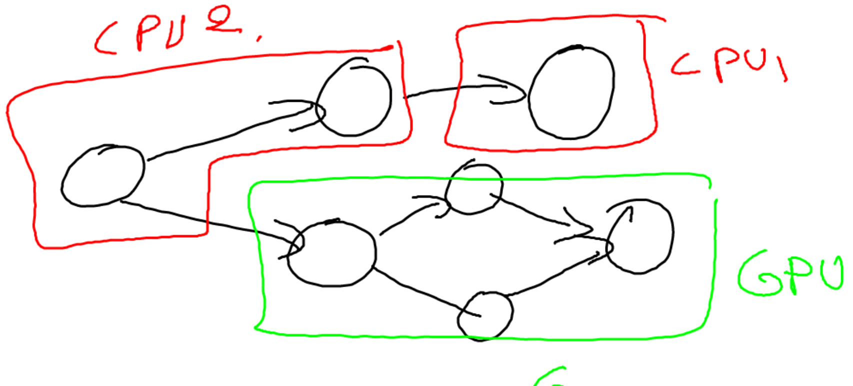
- Data parallism.

same model.



- Model parallism.

Instead to lean $(\underbrace{w_1, w_2}_{W_1}\ \underbrace{\_\ \_}_{W_2}\ \_\ \underbrace{\_\ w_d}_{W_{10}})$ in one machine, I can split the Learning to diffent machines.

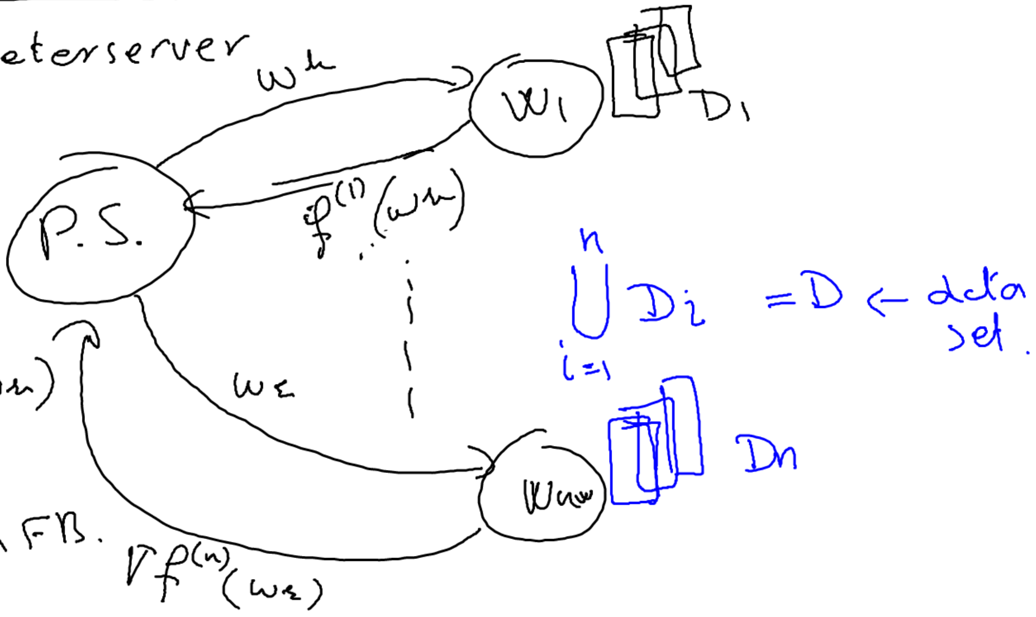


Synchroneous Parameter Server Architecture

P.S. : parameterserver

$$w_{k+1} = w_k + \eta \sum_{i=1}^{n_w} \nabla f^{(i)}(w_k)$$

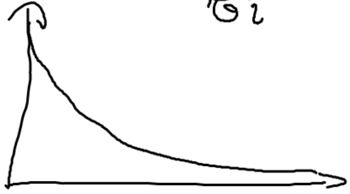
coved even FB.



Why it is called synchroneous, because I need to wait all the results from each worker to go to the next iteration.

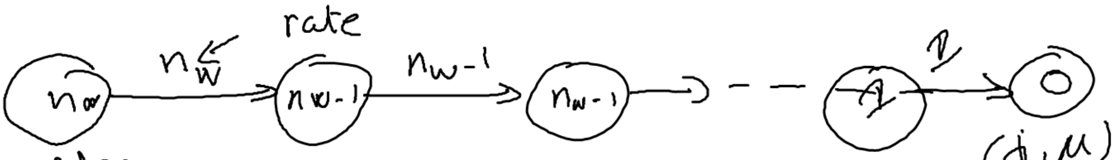
"Bad IDea in practice because of straggler effect". (straggler : someone who takes forever to get the job done compared to others).

I am waiting for n_w workers, Each worker gets the job done in random time $\sim Exp(1)$ τ_i



τ : tau.

Expected finish time for a worker is $\mathbb{E}[\tau_i] = 1$

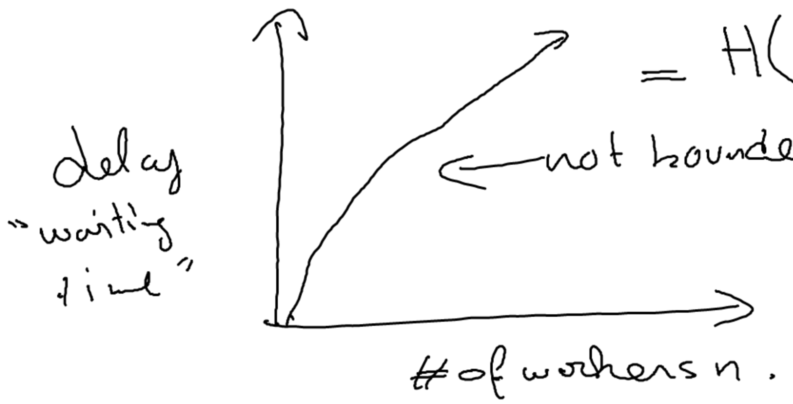


min of two exponential variables

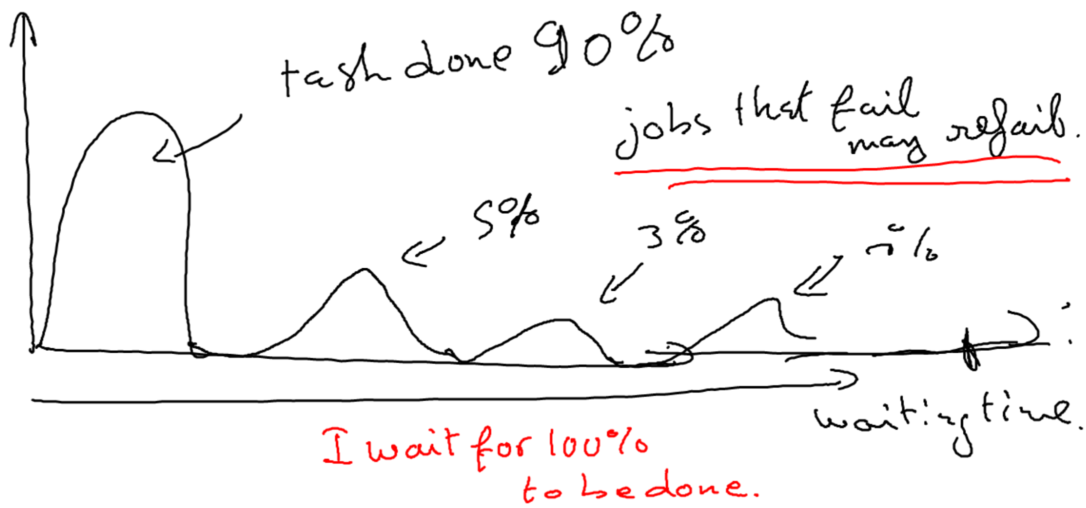
is cls exponential with rate $d + \mu$.

$$\mathbb{E}[\max \tau_i] = \frac{1}{nw} + \frac{1}{nw-1} + \cdots 1$$

$$= H(nw) \simeq \log(n)$$

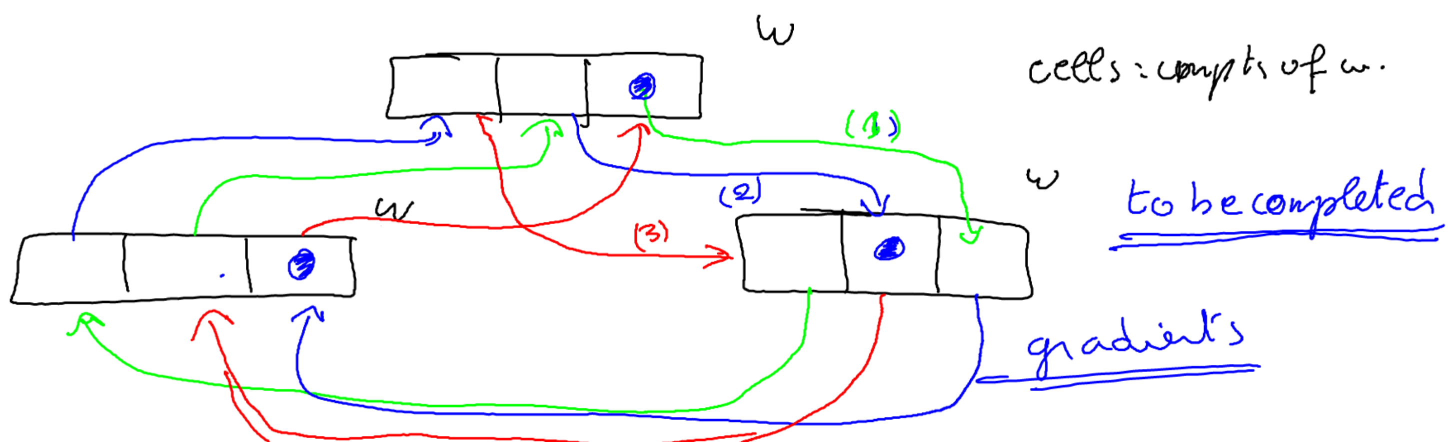


In practice it's even worse.

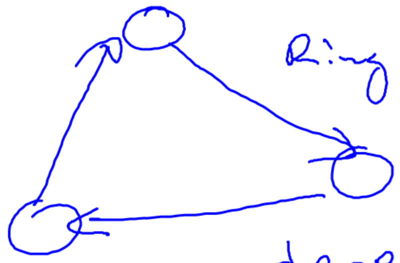


synchronous Parameter server is Bad! ← No one uses it.

Ring - All - Reduce Architecture



$e(nw-1)$ rounds of coms. split to two types $\begin{cases} \text{aggregation } (nw-1) \\ \text{exchange } (nw-1) \\ \text{of} \\ \text{completed} \\ \text{components} \end{cases}$



Ring topology.

dependency is on one worker rather than n.