Lecture 9:

Parallel Deep Network Training

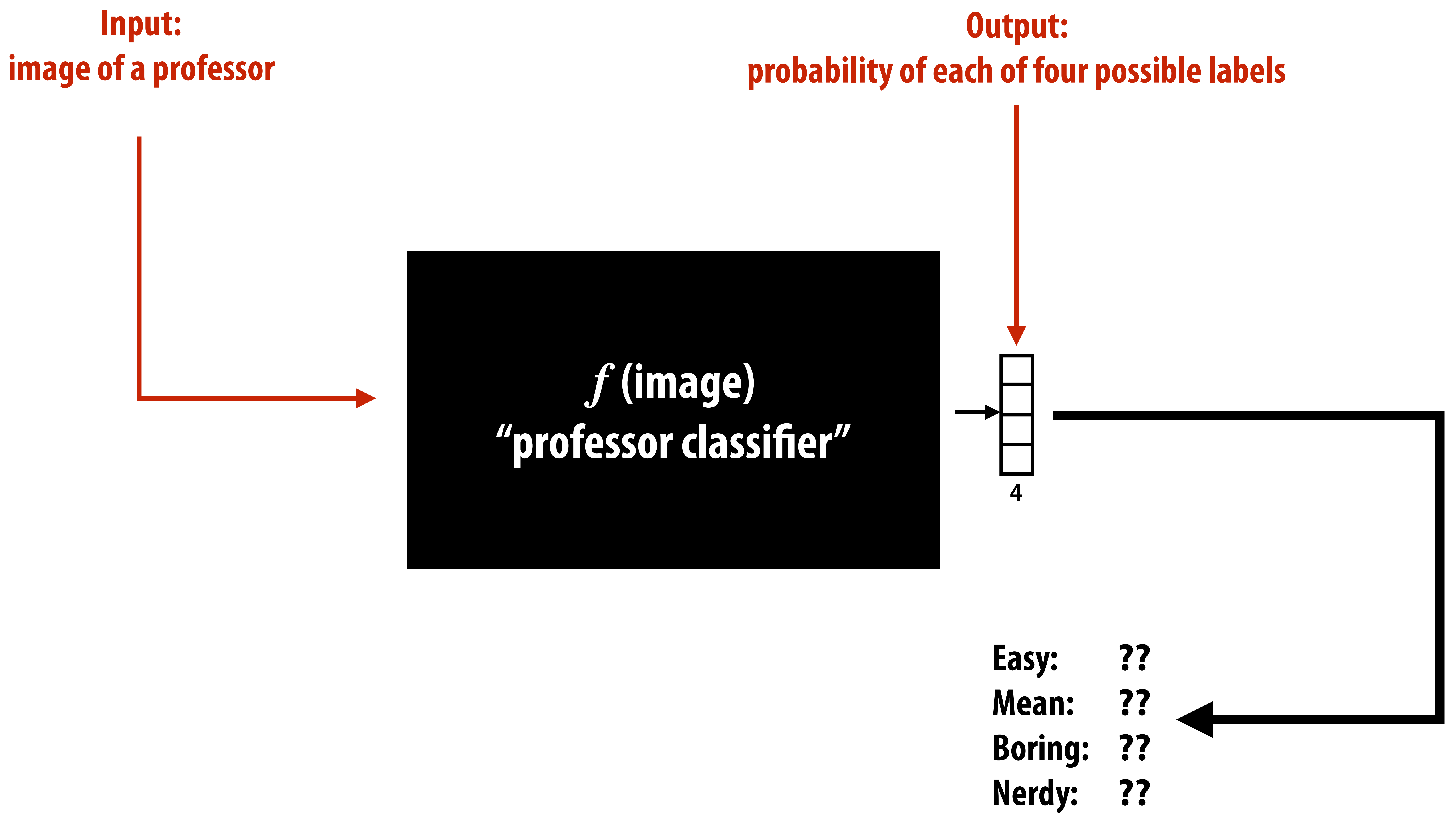
Visual Computing Systems
Stanford CS348V, Winter 2018

How would you describe this professor?

Easy?
Mean?
Boring?
Nerdy?

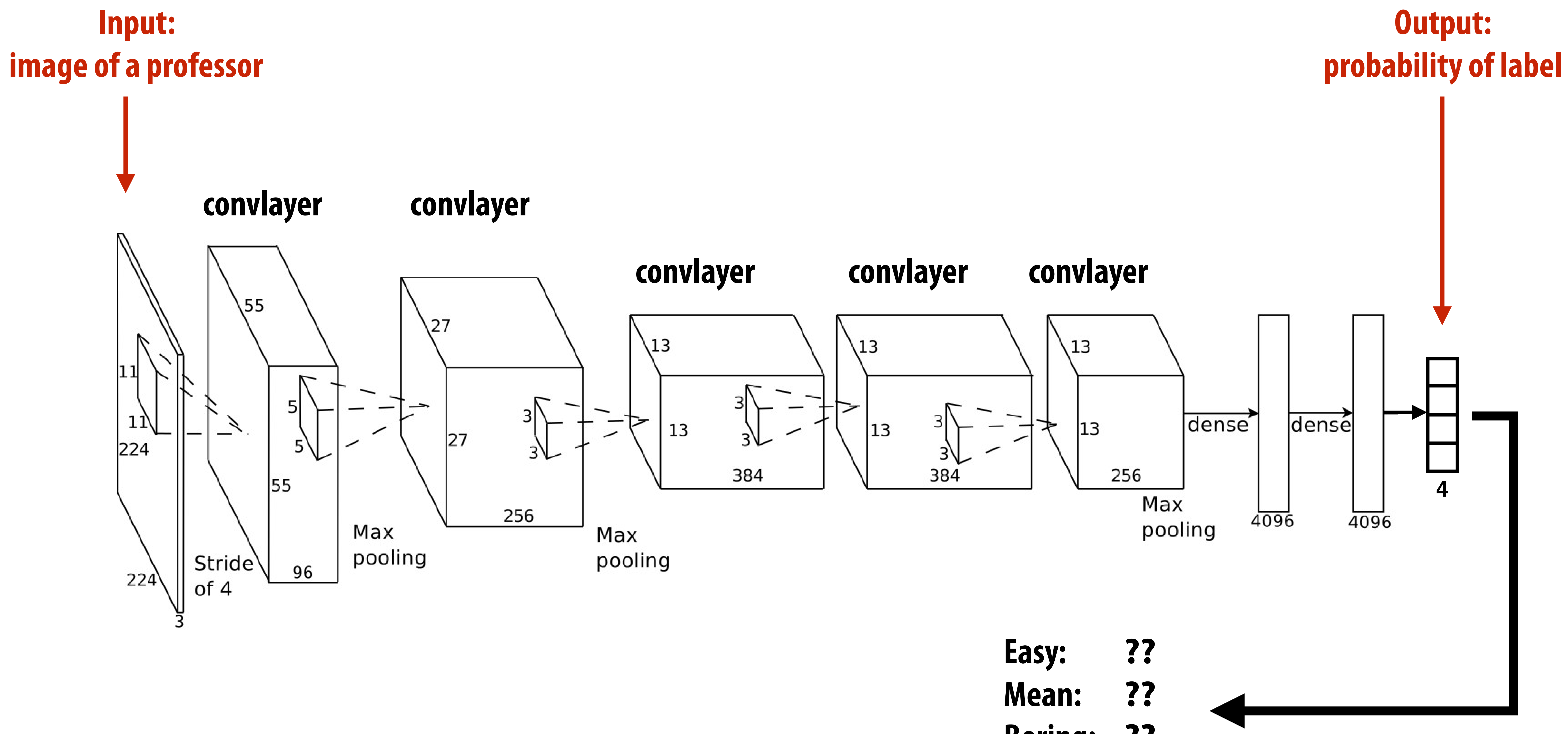
Professor classification task

Classifies professors as easy, mean, boring, or nerdy based on their appearance.



Professor classification network

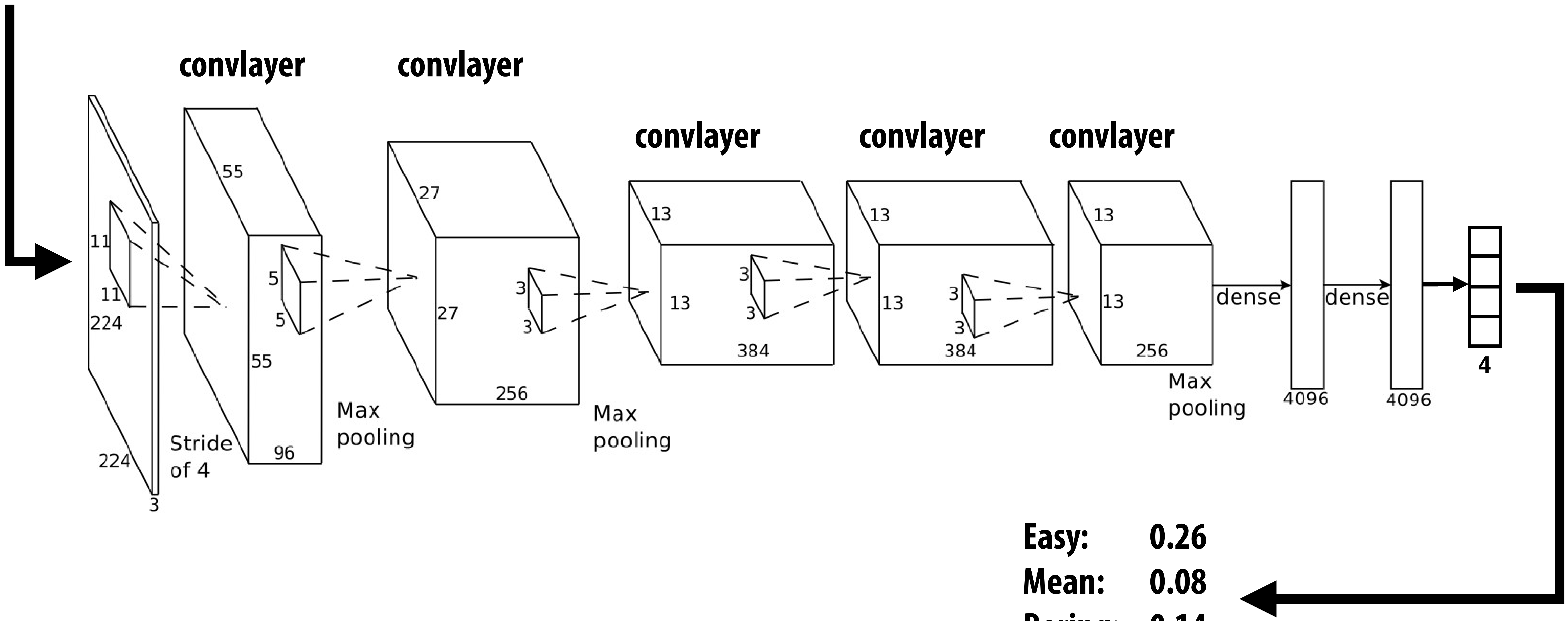
Classifies professors as easy, mean, boring, or nerdy based on their appearance.



Easy:	**??**
Mean:	**??**
Boring:	**??**
Nerdy:	**??**

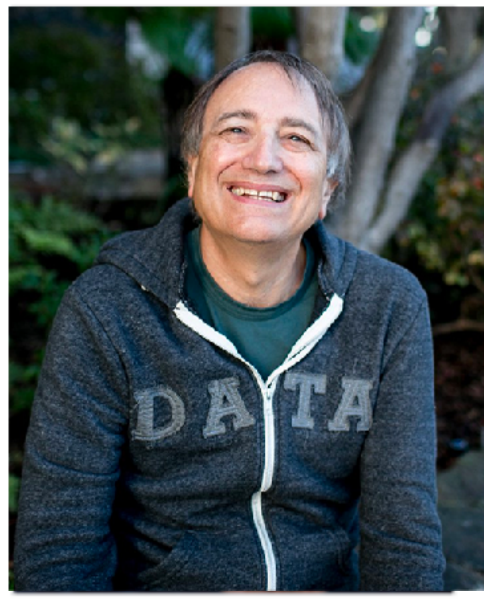
Recall: large networks may have 10's-100's of millions of parameters

Professor classification network



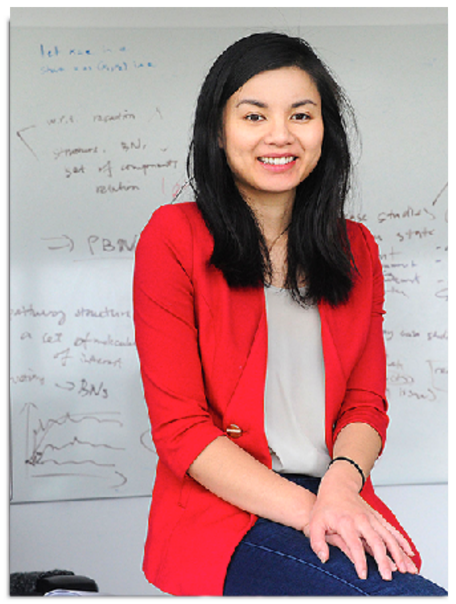
Easy:	**0.26**
Mean:	**0.08**
Boring:	**0.14**
Nerdy:	**0.52**

Training data (ground truth answers)



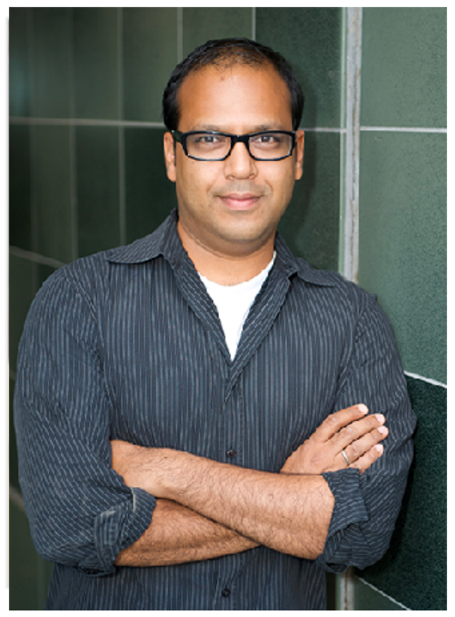
[label omitted]

[label omitted]



[label omitted]

Nerdy



[label omitted]

[label omitted]

[label omitted]

[label omitted]

[label omitted]

Nerdy

[label omitted]

[label omitted]

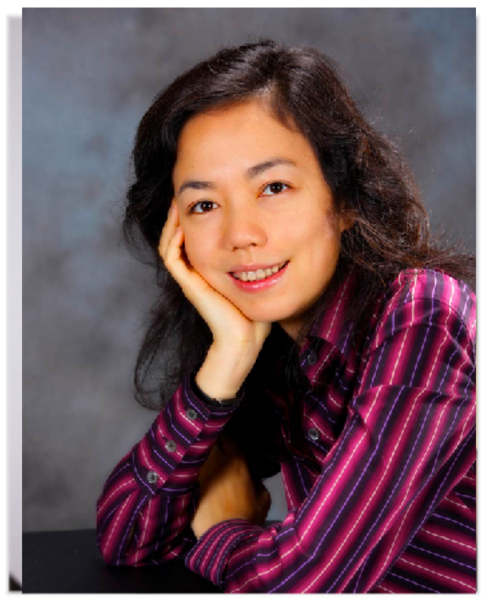
Nerdy

[label omitted]

[label omitted]

Nerdy

Nerdy



[label omitted]

[label omitted]

[label omitted]

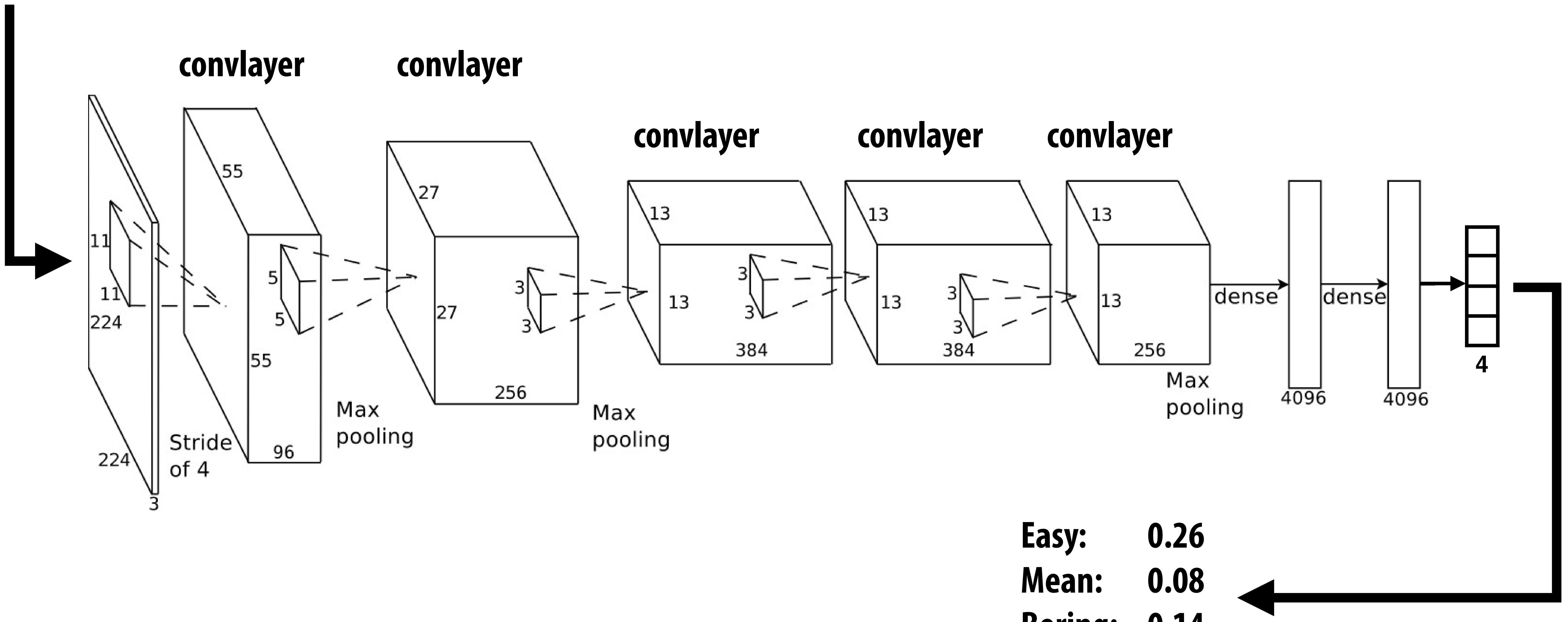
Nerdy

Professor classification network

New image of Kayvon
(not in training set)

Ground truth
(what the answer should be)

Easy:	0.0
Mean:	0.0
Boring:	0.0
Nerdy:	1.0



Easy:	0.26
Mean:	0.08
Boring:	0.14
Nerdy:	0.52

Network output

Error (loss)

Ground truth:
(what the answer should be)

Easy:	0.0
Mean:	0.0
Boring:	0.0
Nerdy:	1.0

Network output: *

Easy:	0.26
Mean:	0.08
Boring:	0.14
Nerdy:	0.52

Common example: softmax loss:

$$L = -log\left(\frac{\overset{\text{Output of network for correct category}}{e^{f_c}}}{\underset{\text{Output of network for all categories}}{\sum_j e^{f_j}}}\right)$$

* In practice a network using a softmax classifier outputs unnormalized, log probabilities (f_j), but I'm showing a probability distribution above for clarity

Training

Goal of training: learning good values of network parameters so that the network outputs the correct classification result for any input image

Idea: minimize loss for all the training examples (for which the correct answer is known)

$$L = \sum_i L_i$$

(total loss for entire training set is sum of losses *L_i* for each training example *x_i*)

Intuition: if the network gets the answer correct for a wide range of training examples, then hopefully it has learned parameter values that yield the correct answer for future images as well.

Intuition: gradient descent

Say you had a function *f* that contained hidden parameters *p_1 and p_2*: $f(x_i)$

And for some input *x_i*, your training data says the function should output 0.

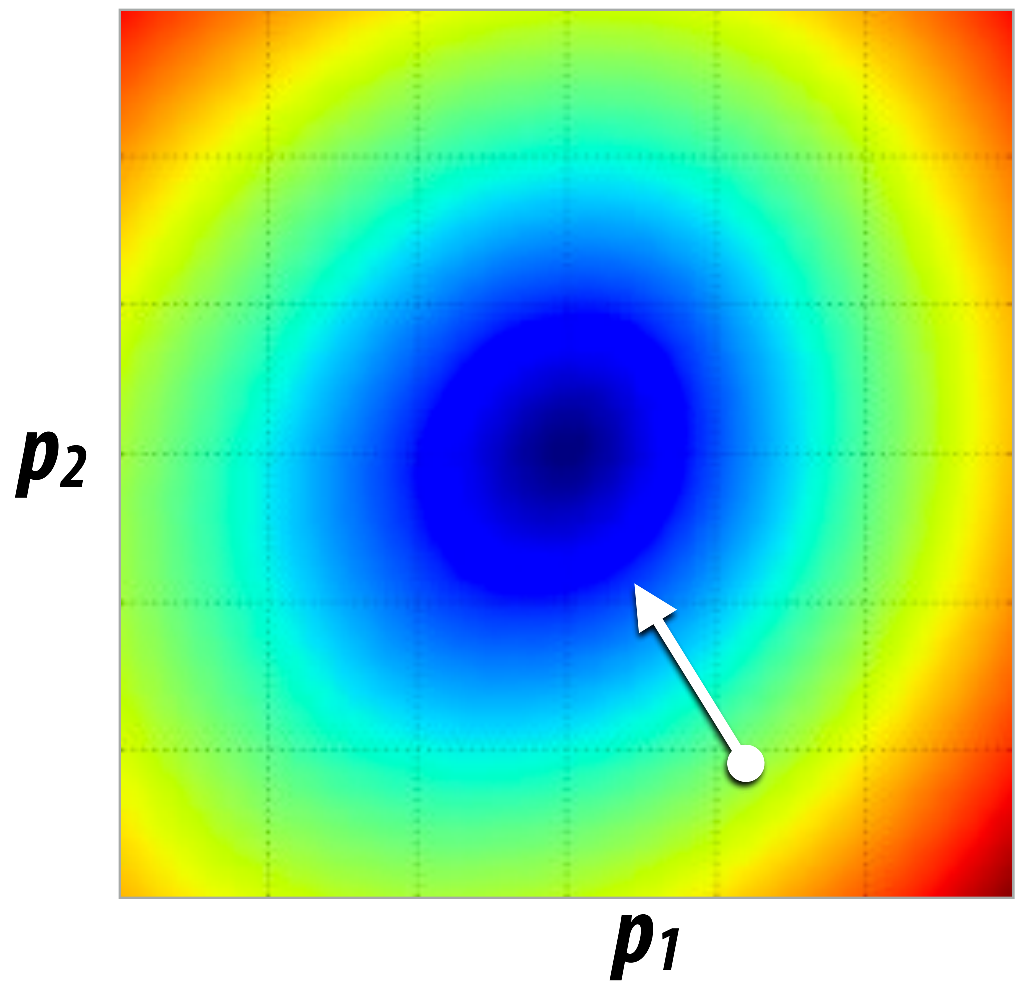
But for the current values of *p_1 and p_2*, it currently outputs 10.

$$f(x_i, p_1, p_2) = 10$$

And say I also gave you expressions for the derivative of *f* with respect to *p_1 and p_2* so you could compute their value at *x_i*.

$$\frac{df}{dp_1} = 2 \quad \frac{df}{dp_2} = -5 \qquad \nabla f = [2, -5]$$

red = high values of f
blue = low values



How might you adjust the values *p_1 and p_2* to reduce the error for this training example?

Basic gradient descent

```
while (loss too high):
  for each epoch: // a pass through the training dataset
     for each item x_i in training set:
        grad = evaluate_loss_gradient(f, params, loss_func, x_i)
        params += -grad * learning_rate;
```

Mini-batch stochastic gradient descent (mini-batch SGD): choose a random (small) subset of the training examples to use to compute the gradient in each iteration of the while loop

```
while (loss too high):
  for each epoch: // a pass through the training dataset
     for all mini batches in training set:
       grad = 0;
       for each item x_i in minibatch:
          grad += evaluate_loss_gradient(f, params, loss_func, x_i)
       params += -grad * learning_rate;
```

How do we compute dLoss/dp for a deep neural network with millions of parameters?

Quick review of back-propagation

Derivatives using the chain rule

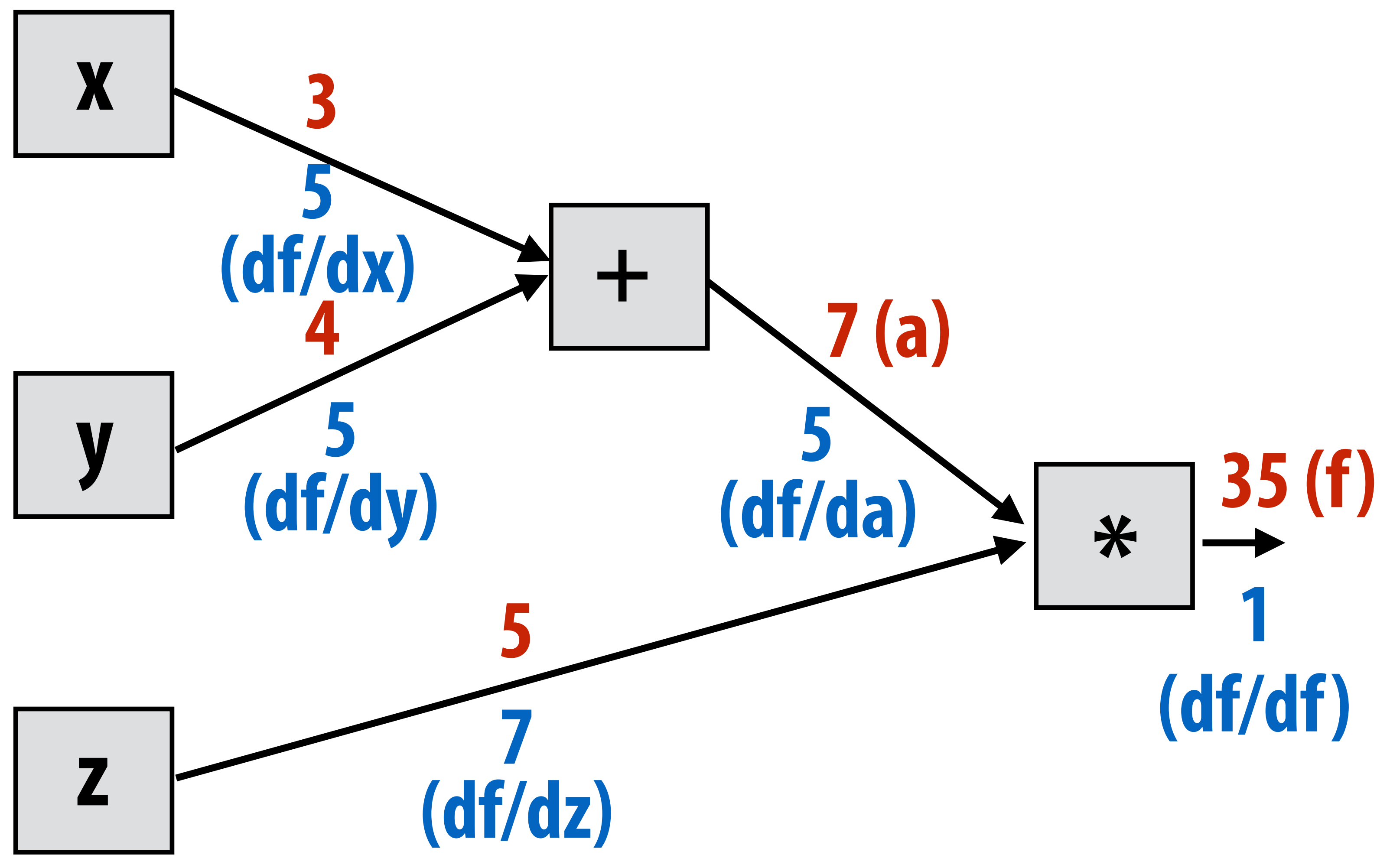
$$f(x, y, z) = (x + y)z = az$$

Where: $a = x + y$

$$\frac{df}{da} = z \qquad \frac{da}{dx} = 1 \qquad \frac{da}{dy} = 1$$

So, by the derivative chain rule:

$$\frac{df}{dx} = \frac{df}{da}\frac{da}{dx} = z$$



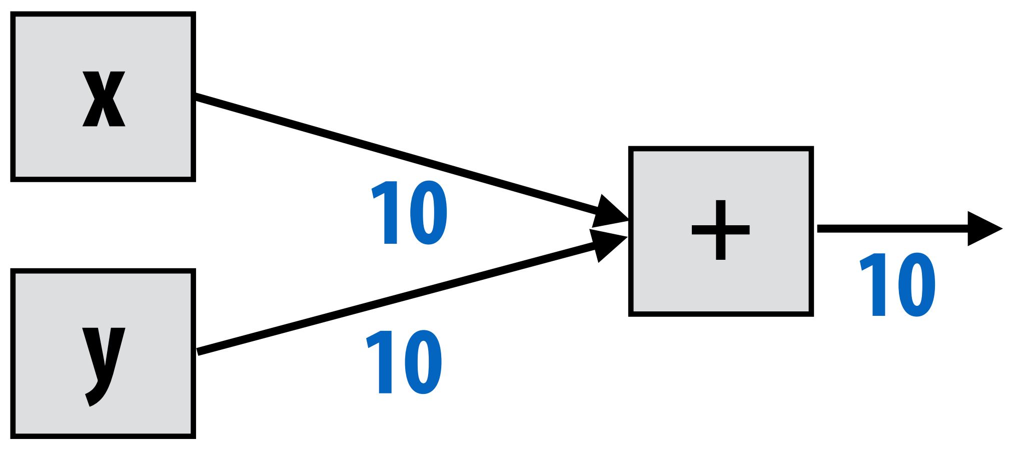
Red = output of node
Blue = df/dnode

Backpropagation

Red = output of node

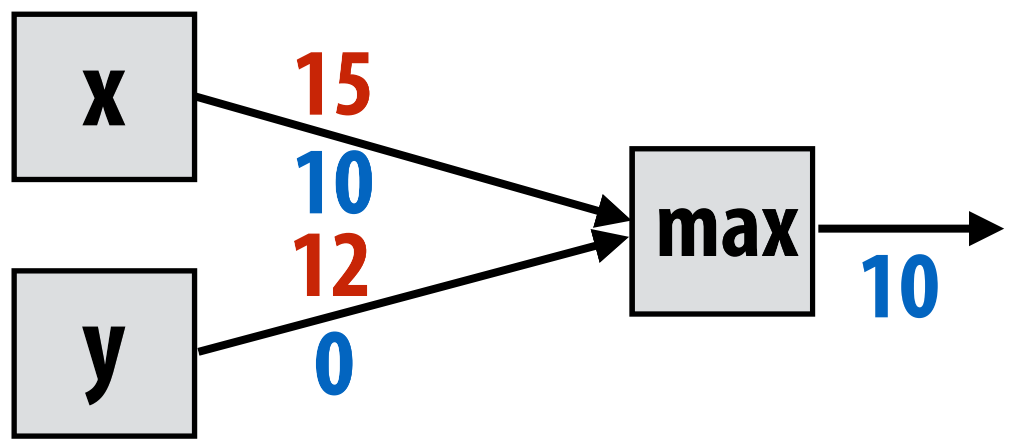
Blue = df/dnode

Recall: $$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$$



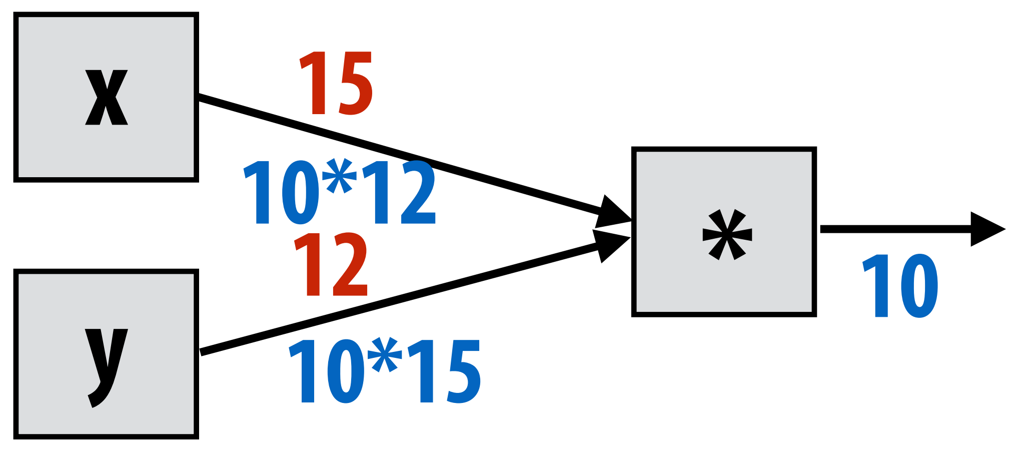
$$g(x, y) = x + y$$

$$\frac{dg}{dx} = 1\,, \quad \frac{dg}{dy} = 1$$



$$g(x, y) = \max(x, y)$$

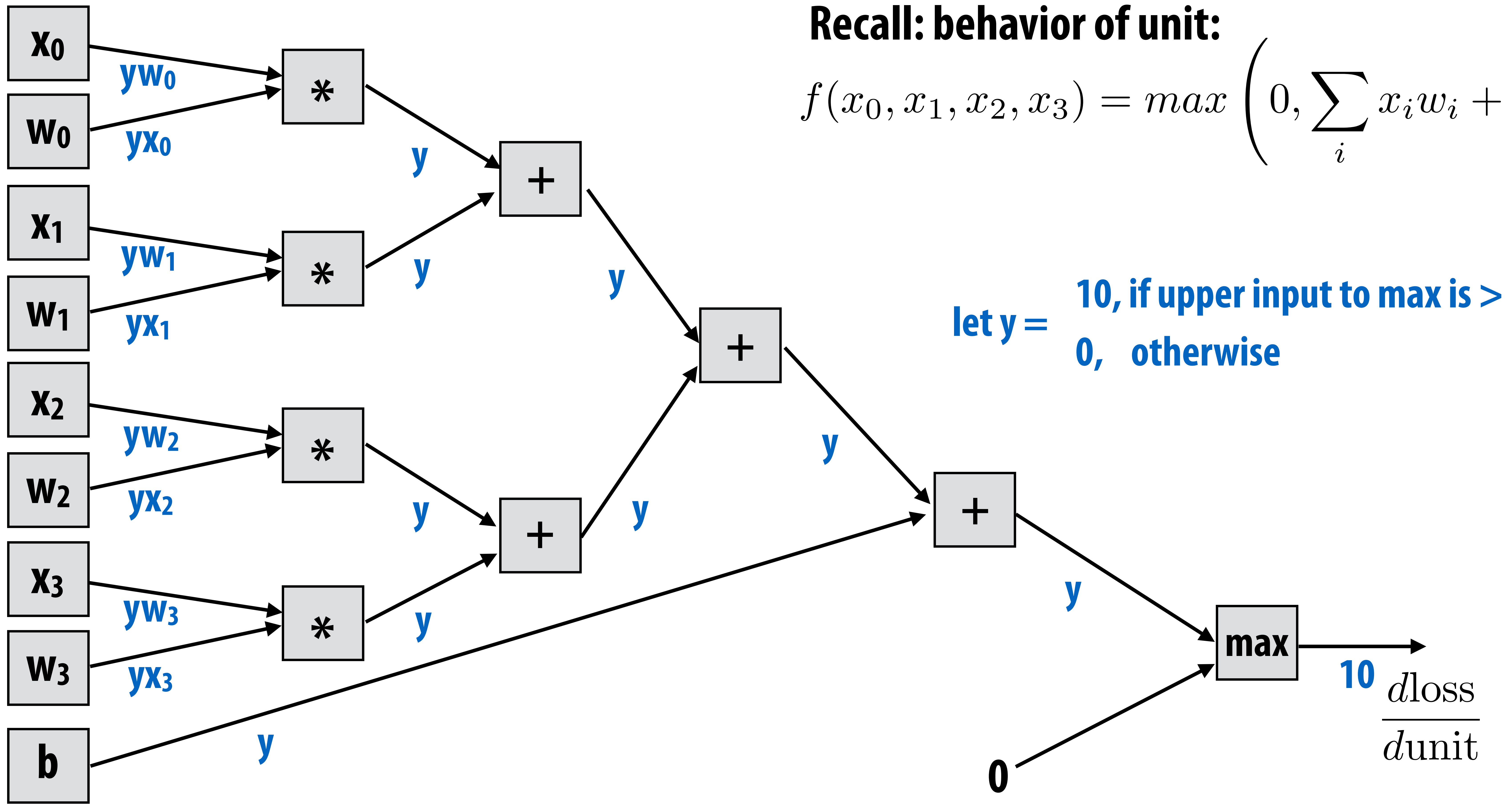
$\frac{dg}{dx} =$ **1, if x > y**; **0, otherwise**



$$g(x, y) = xy$$

$$\frac{dg}{dx} = y\,, \quad \frac{dg}{dy} = x$$

Back-propagating through single unit



$\frac{\text{dloss}}{\text{dunit}}$

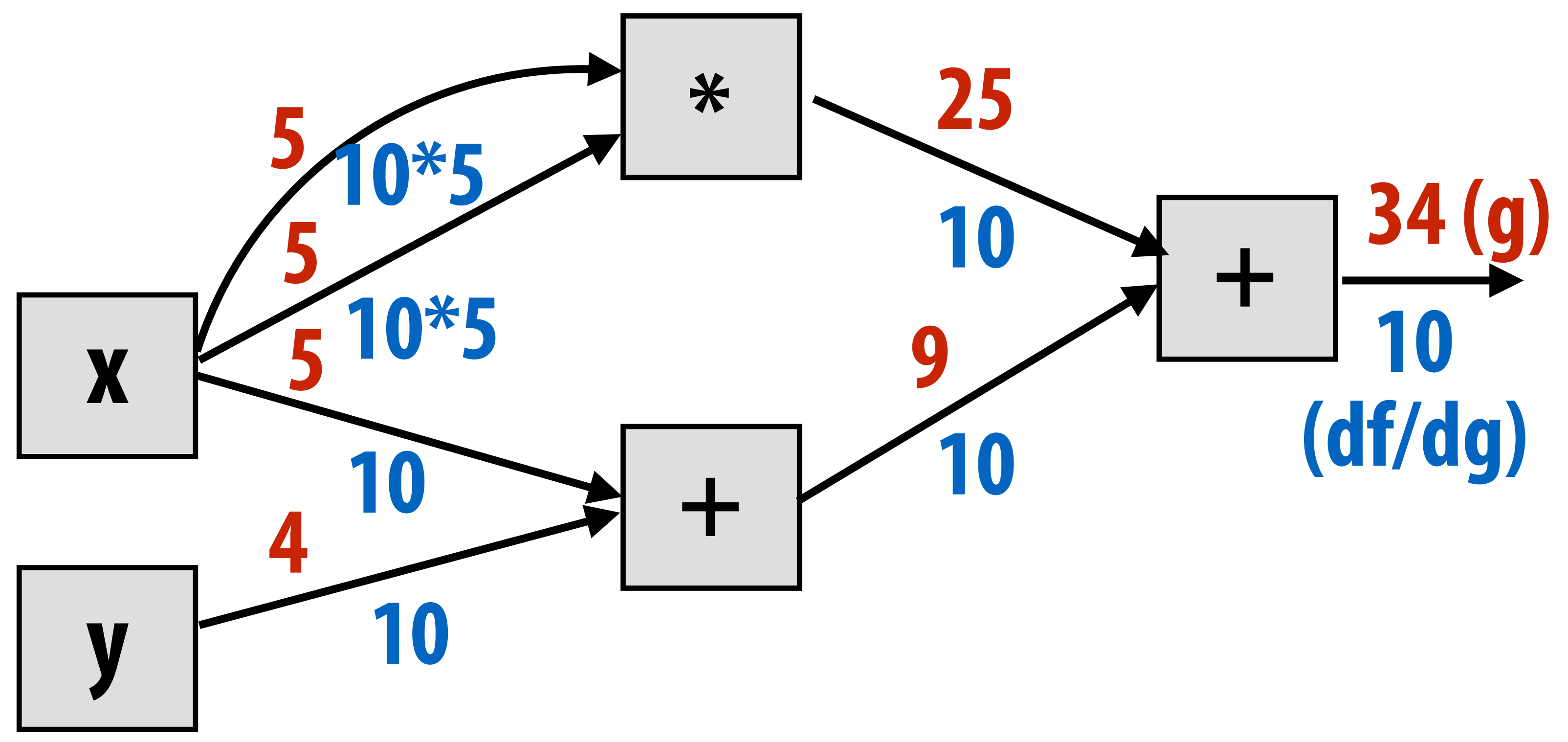
Recall: behavior of unit:

$$f(x_0, x_1, x_2, x_3) = max\left(0, \sum_i x_i w_i + b\right)$$

let y = 10, if upper input to max is > 0
0, otherwise

Observe: output of prior layer must be retained in order to compute weight gradients for this unit during backprop.

Multiple uses of an input variable



$$g(x,y) = (x+y) + x*x = a + b$$

$$\frac{da}{dx} = 1\,,\ \frac{db}{dx} = 2x$$

$$\frac{dg}{dx} = \frac{dg}{da}\frac{da}{dx} + \frac{dg}{db}\frac{db}{dx} = 2x + 1$$

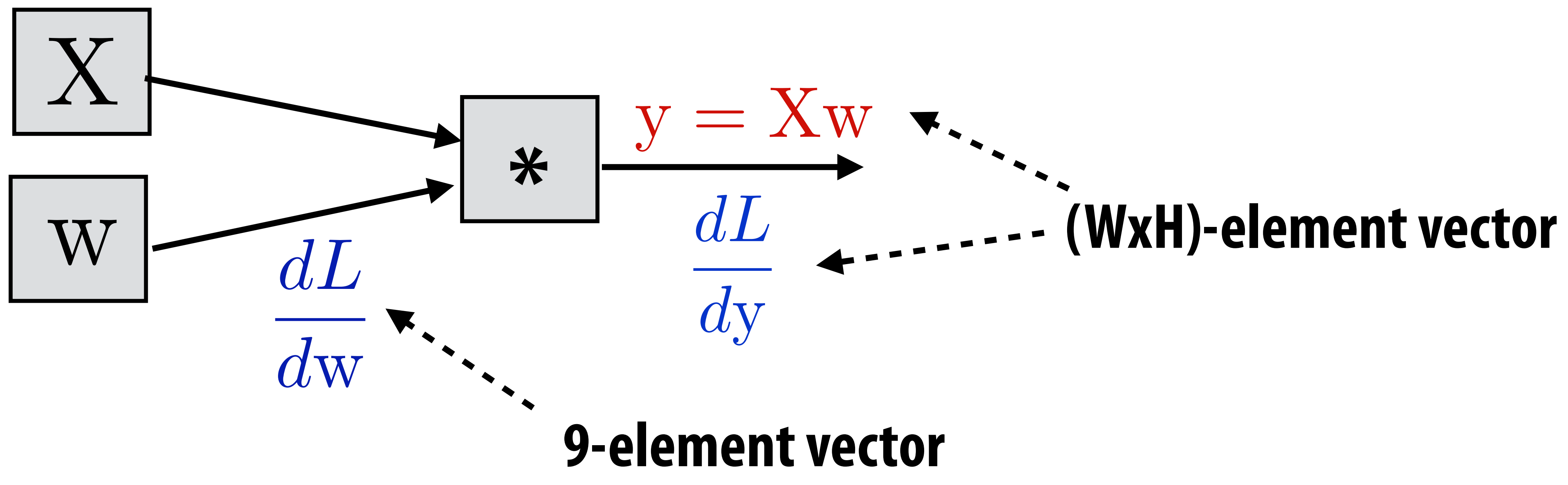
Sum gradients from each use of variable:

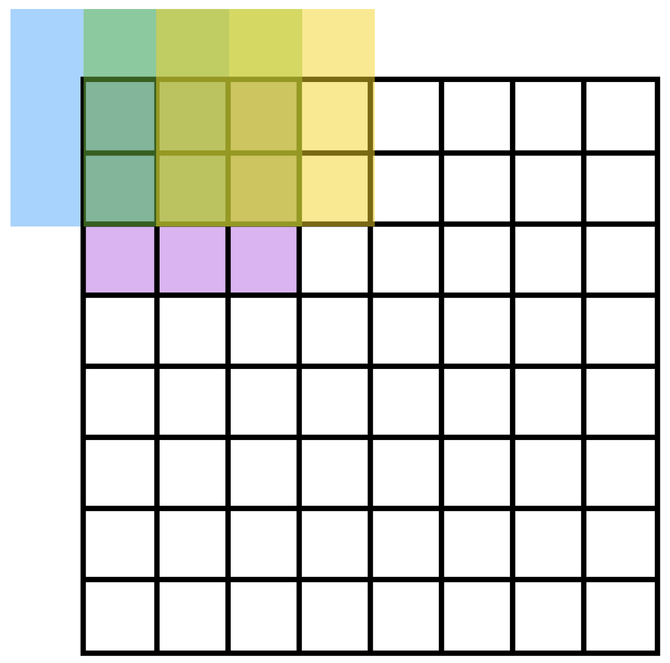
Here:

$$\begin{aligned}\frac{df}{dx} &= \frac{df}{dg}\frac{dg}{dx} \\ &= 10\frac{dg}{dx} \\ &= 10(2x+1) \\ &= 10(10+1) = 110\end{aligned}$$

Implication: backpropagation through all units in a convolutional layer adds gradients computed from each unit to the overall gradient for the shared weights

Back-propagation: matrix form





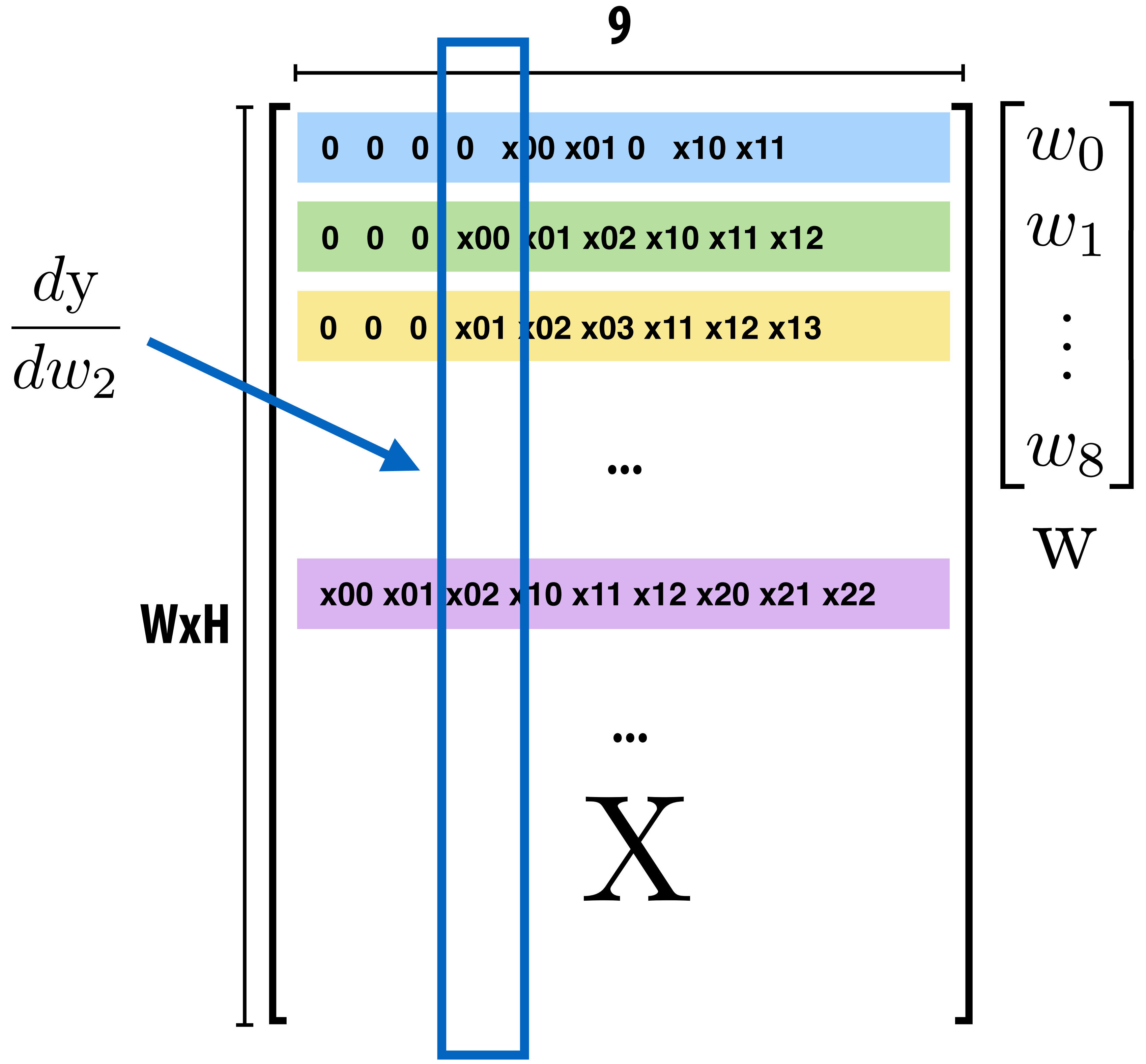
$$\frac{dy_j}{dw_i} = X_{ji}$$

$$\frac{dL}{dw_i} = \sum_j \frac{dL}{dy_j}\frac{dy_j}{dw_i}$$

$$= \sum_j \frac{dL}{dy_j} X_{ji}$$

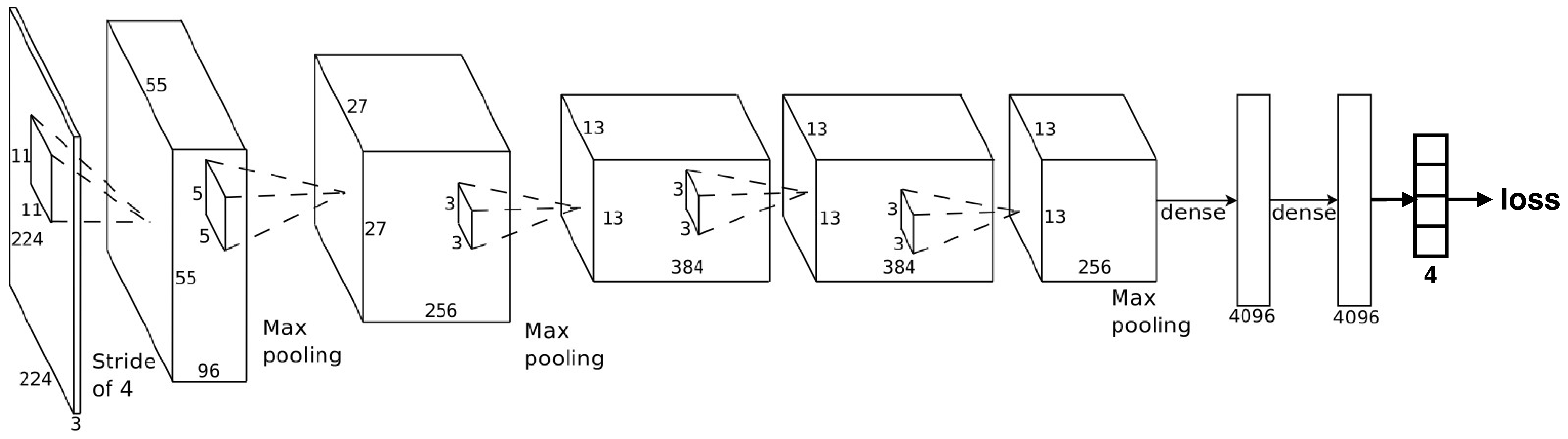
Therefore:

$$\frac{dL}{dw} = X^T \frac{dL}{dy}$$



9
$\frac{dy}{dw_2}$
WxH
0 0 0 0 x00 x01 0 x10 x11
0 0 0 x00 x01 x02 x10 x11 x12
0 0 0 x01 x02 x03 x11 x12 x13
...
x00 x01 x02 x10 x11 x12 x20 x21 x22
...
X
w_0
w_1
$\vdots$
w_8
w

Backpropagation through the entire professor classification network



```
For each training example $x_i$ in mini-batch:
    Perform forward evaluation to compute loss for $x_i$
    Compute gradient of loss w.r.t. final layer's outputs
    Backpropagate gradient to compute gradient of loss w.r.t. all network parameters
    Accumulate gradients (over all images in minibatch)
Update all parameter values: w_new = w_old – learning_rate * grad
```

Recall from last class: VGG memory footprint

Calculations assume 32-bit values (image batch size = 1)

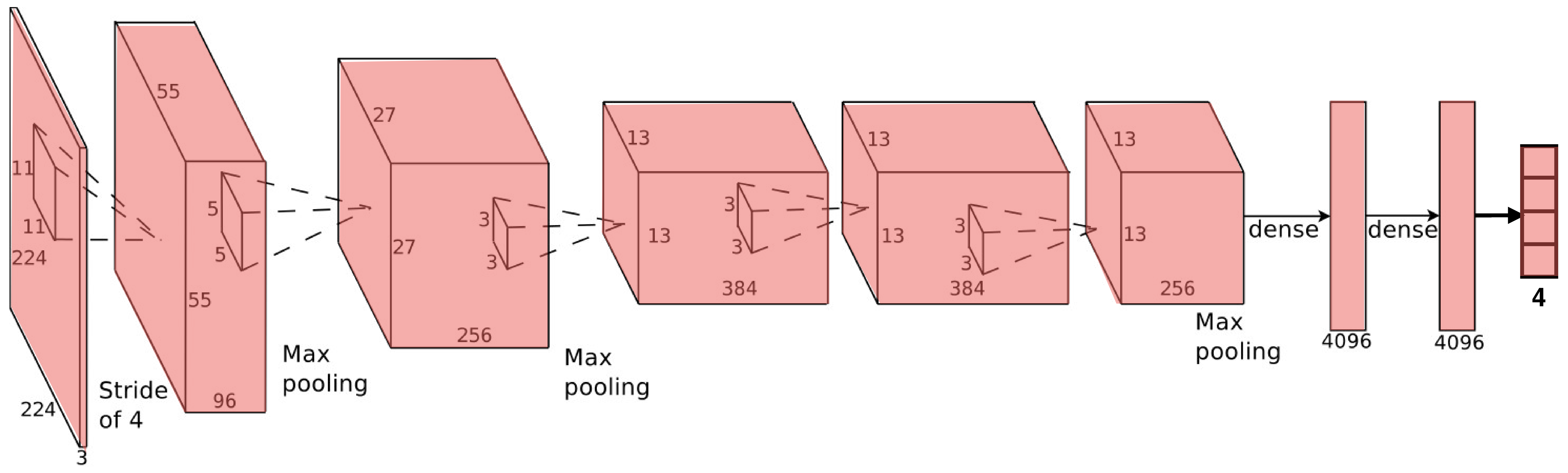
	weights mem:	output size (per image)	(mem)
input: 224 x 224 RGB image	—	224x224x3	150K
conv: (3x3x3) x 64	6.5 KB	224x224x64	12.3 MB
conv: (3x3x64) x 64	144 KB	224x224x64	12.3 MB
maxpool	—	112x112x64	3.1 MB
conv: (3x3x64) x 128	228 KB	112x112x128	6.2 MB
conv: (3x3x128) x 128	576 KB	112x112x128	6.2 MB
maxpool	—	56x56x128	1.5 MB
conv: (3x3x128) x 256	1.1 MB	56x56x256	3.1 MB
conv: (3x3x256) x 256	2.3 MB	56x56x256	3.1 MB
conv: (3x3x256) x 256	2.3 MB	56x56x256	3.1 MB
maxpool	—	28x28x256	766 KB
conv: (3x3x256) x 512	4.5 MB	28x28x512	1.5 MB
conv: (3x3x512) x 512	9 MB	28x28x512	1.5 MB
conv: (3x3x512) x 512	9 MB	28x28x512	1.5 MB
maxpool	—	14x14x512	383 KB
conv: (3x3x512) x 512	9 MB	14x14x512	383 KB
conv: (3x3x512) x 512	9 MB	14x14x512	383 KB
conv: (3x3x512) x 512	9 MB	14x14x512	383 KB
maxpool	—	7x7x512	98 KB
fully-connected 4096	392 MB	4096	16 KB
fully-connected 4096	64 MB	4096	16 KB
fully-connected 1000	15.6 MB	1000	4 KB
soft-max		1000	4 KB

Many weights in fully-connected players

Storing convolution layer outputs (unit "activations") can get big in early layers with large input size and many filters

Note: multiply these numbers by N for batch size of N images

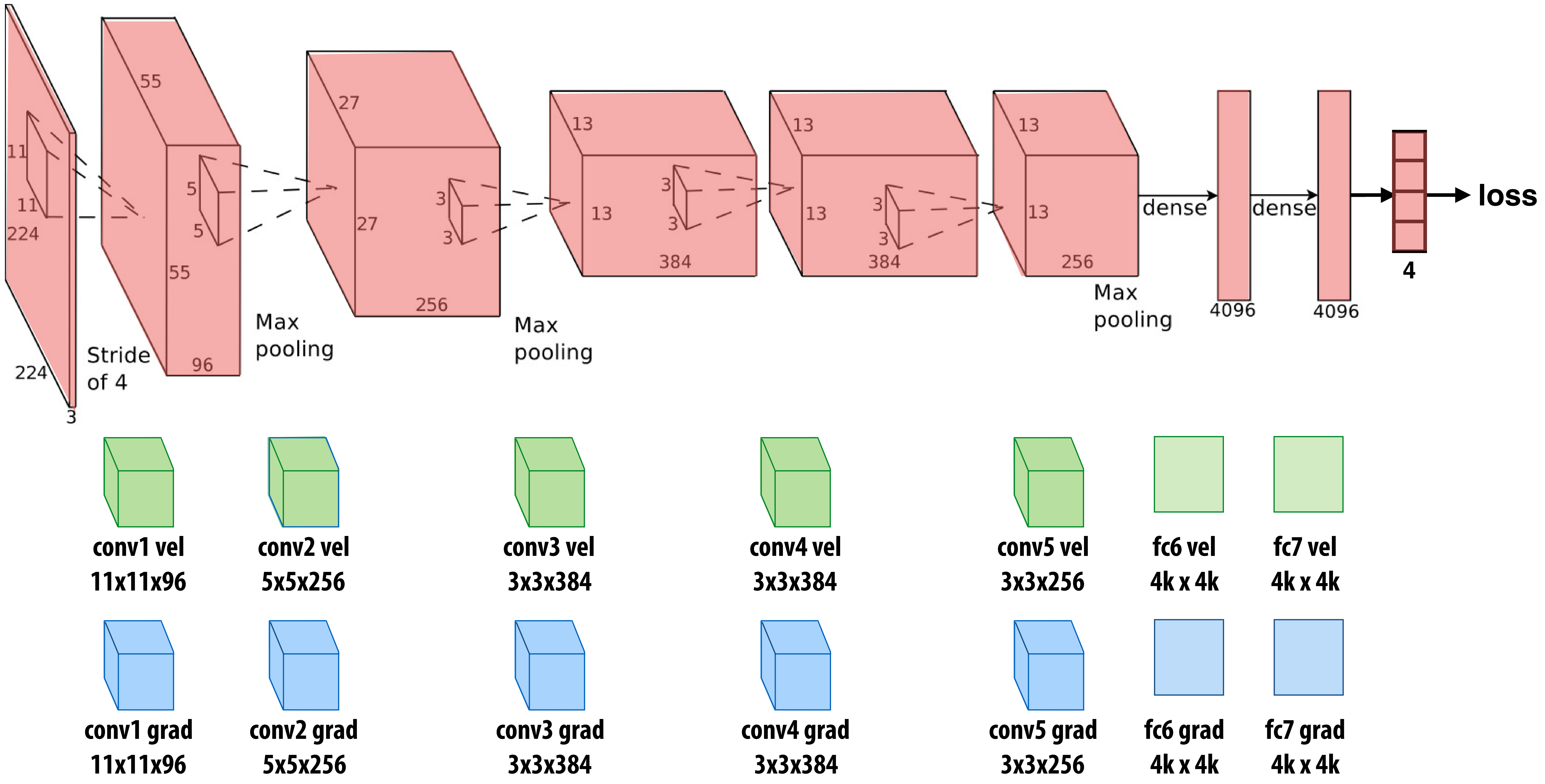
Data lifetimes during network evaluation



Weights (read-only) reside in memory

After evaluating layer i, can free outputs from layer i-1

Data lifetimes during training



- Must retain outputs for all layers because they are needed to compute gradients during back-prop
- Parallel back-prop will require storage for per-weight gradients (more about this in a second)
- In practice: may also store per-weight gradient velocity (if using SGD with "momentum") or step size cache in adaptive step size schemes like Adagrad

```
vel_new = mu * vel_old - step_size * grad
w_new = w_old + vel_new
```

VGG memory footprint

Calculations assume 32-bit values (image batch size = 1)

inputs/outputs get multiplied by mini-batch size

Unlike forward evaluation: cannot immediately free outputs once consumed by next level of network

Must also store per-weight gradients

Many implementations also store gradient "momentum" as well (multiply by 3)

	weights mem:	output size (per image)	(mem)
input: 224 x 224 RGB image	—	224x224x3	150K
conv: (3x3x3) x 64	6.5 KB	224x224x64	12.3 MB
conv: (3x3x64) x 64	144 KB	224x224x64	12.3 MB
maxpool	—	112x112x64	3.1 MB
conv: (3x3x64) x 128	228 KB	112x112x128	6.2 MB
conv: (3x3x128) x 128	576 KB	112x112x128	6.2 MB
maxpool	—	56x56x128	1.5 MB
conv: (3x3x128) x 256	1.1 MB	56x56x256	3.1 MB
conv: (3x3x256) x 256	2.3 MB	56x56x256	3.1 MB
conv: (3x3x256) x 256	2.3 MB	56x56x256	3.1 MB
maxpool	—	28x28x256	766 KB
conv: (3x3x256) x 512	4.5 MB	28x28x512	1.5 MB
conv: (3x3x512) x 512	9 MB	28x28x512	1.5 MB
conv: (3x3x512) x 512	9 MB	28x28x512	1.5 MB
maxpool	—	14x14x512	383 KB
conv: (3x3x512) x 512	9 MB	14x14x512	383 KB
conv: (3x3x512) x 512	9 MB	14x14x512	383 KB
conv: (3x3x512) x 512	9 MB	14x14x512	383 KB
maxpool	—	7x7x512	98 KB
fully-connected 4096	392 MB	4096	16 KB
fully-connected 4096	64 MB	4096	16 KB
fully-connected 1000	15.6 MB	1000	4 KB
soft-max		1000	4 KB

SGD workload

```
while (loss too high):

    for each item x_i in mini-batch:
       grad += evaluate_loss_gradient(f, loss_func, params, x_i)

    params += -grad * step_size;
```

At first glance, this loop is sequential (each step of "walking downhill" depends on previous)

Parallel across images

large computation with its own parallelism (but working set may not fit on single machine)

sum reduction

trivial data-parallel over parameters

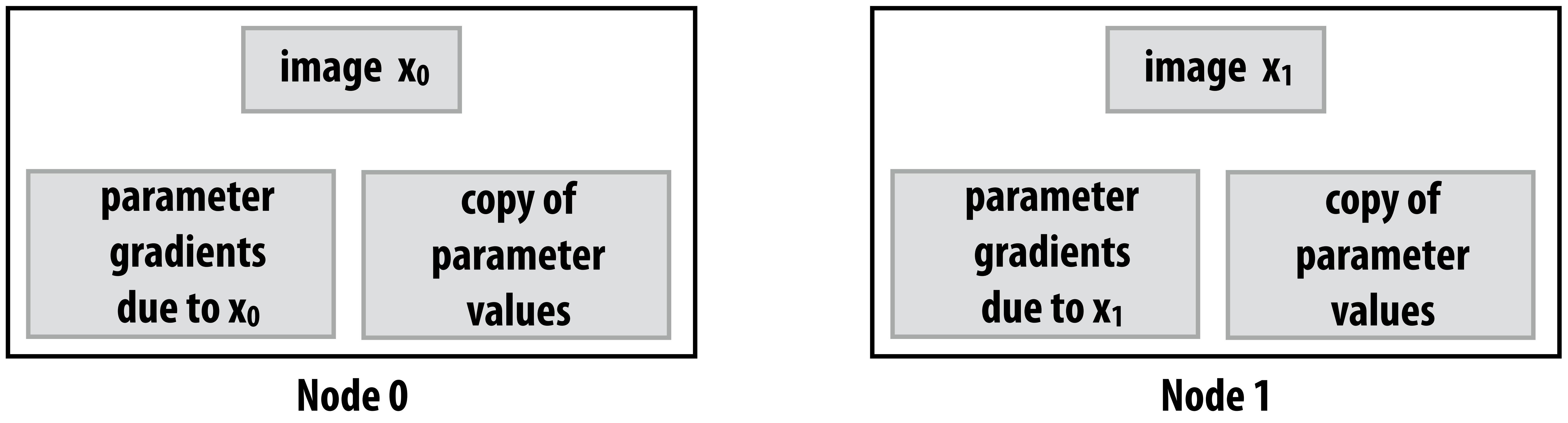
DNN training workload

- **Large computational expense**
  - Must evaluate the network (forward and backward) for millions of training images
  - Must iterate for many iterations of gradient descent (100's of thousands)
  - Training modern networks on big datasets takes days
- **Large memory footprint**
  - Must maintain network layer outputs from forward pass
  - Additional memory to store gradients/gradient velocity for each parameter
  - Recall parameters for popular VGG-16 network require ~500 MB of memory (training requires GBs of memory for academic networks)
  - Scaling to larger networks requires partitioning DNN across nodes to keep DNN + intermediates in memory
- **Dependencies /synchronization (not embarrassingly parallel)**
  - Each parameter update step depends on previous
  - Many units contribute to same parameter gradients (fine-scale reduction)
  - Different images in mini batch contribute to same parameter gradients

Synchronous data-parallel training (across images)

```
for each item x_i in mini-batch:
   grad += evaluate_loss_gradient(f, loss_func, params, x_i)
params += -grad * learning_rate;
```

Consider parallelization of the outer for loop across machines in a cluster



```
partition dataset across nodes
for each item x_i in mini-batch assigned to local node:
   // just like single node training
   grad += evaluate_loss_gradient(f, loss_func, params, x_i)
barrier();
sum reduce gradients, communicate results to all nodes
barrier();
update copy of parameter values
```

Synchronous training

- All nodes cooperate to compute gradients for a mini-batch *
- Gradients are summed (across the entire machine)
  - All-to-all communication (e.g., MPI_Allreduce)
  - Good implementations will sum gradients for layer *i* when computing backprop for *i*+1 (overlap communication and computation).
- Update model parameters
  - Typically done without wide parallelism (e.g. each machine computes its own update)
- All nodes proceed to work on next mini-batch given new model parameters

* If curious about batch norm in a parallel training setting. In practice each of *k* nodes works on a set of *n* images, with batch norm statistics computed independently for each set of n (mini-batch size is *kn*).

Challenges of scaling out (many nodes)

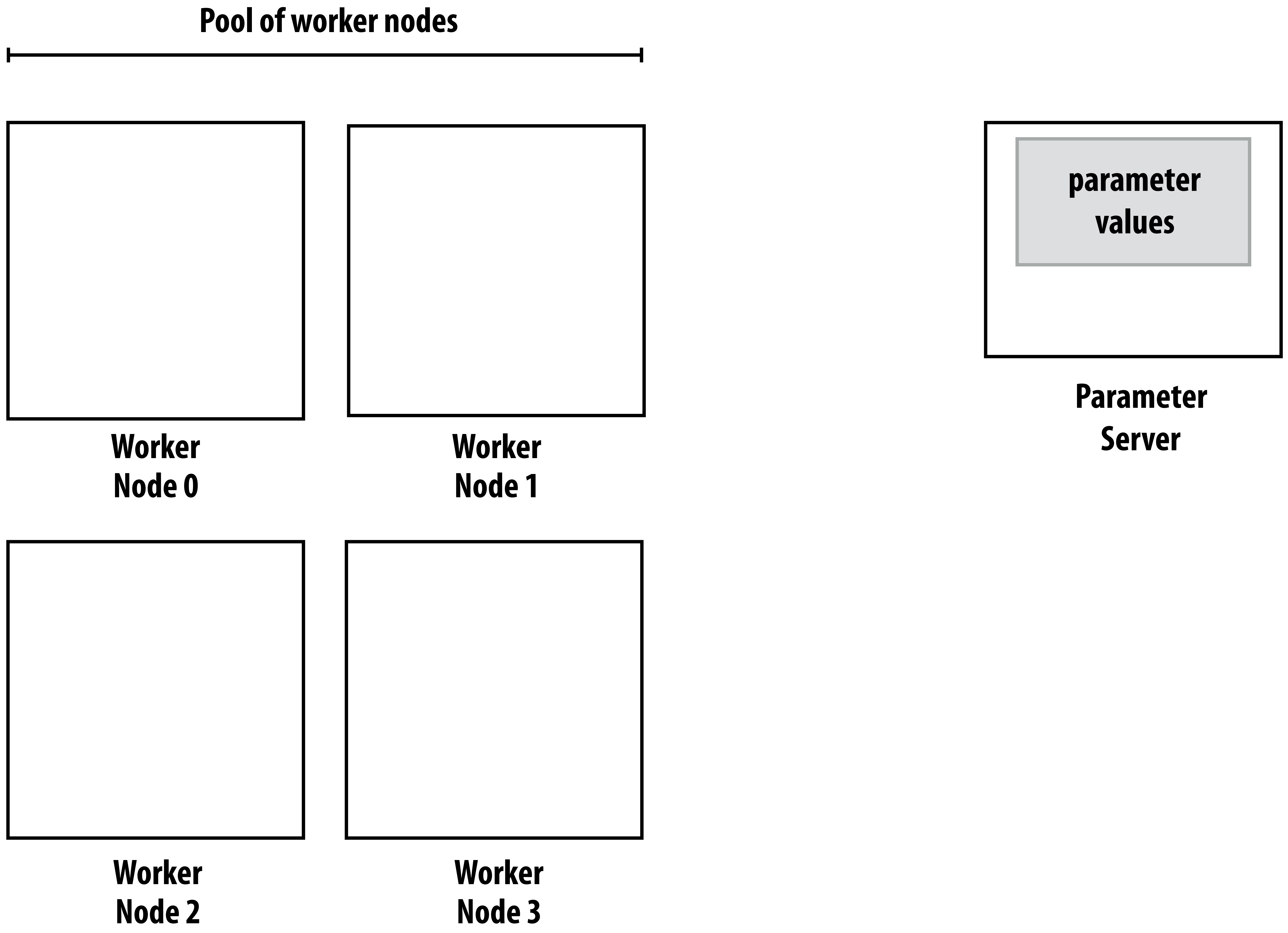
- Slow communication between nodes
  - Commodity clusters do not feature high-performance interconnects (e.g., infiniband) typical of supercomputers
  - Synchronous SGD involves all to all communication after each minibatch

- Nodes with different performance (even if machines are the same)
  - Workload imbalance at barriers (sync points between nodes)

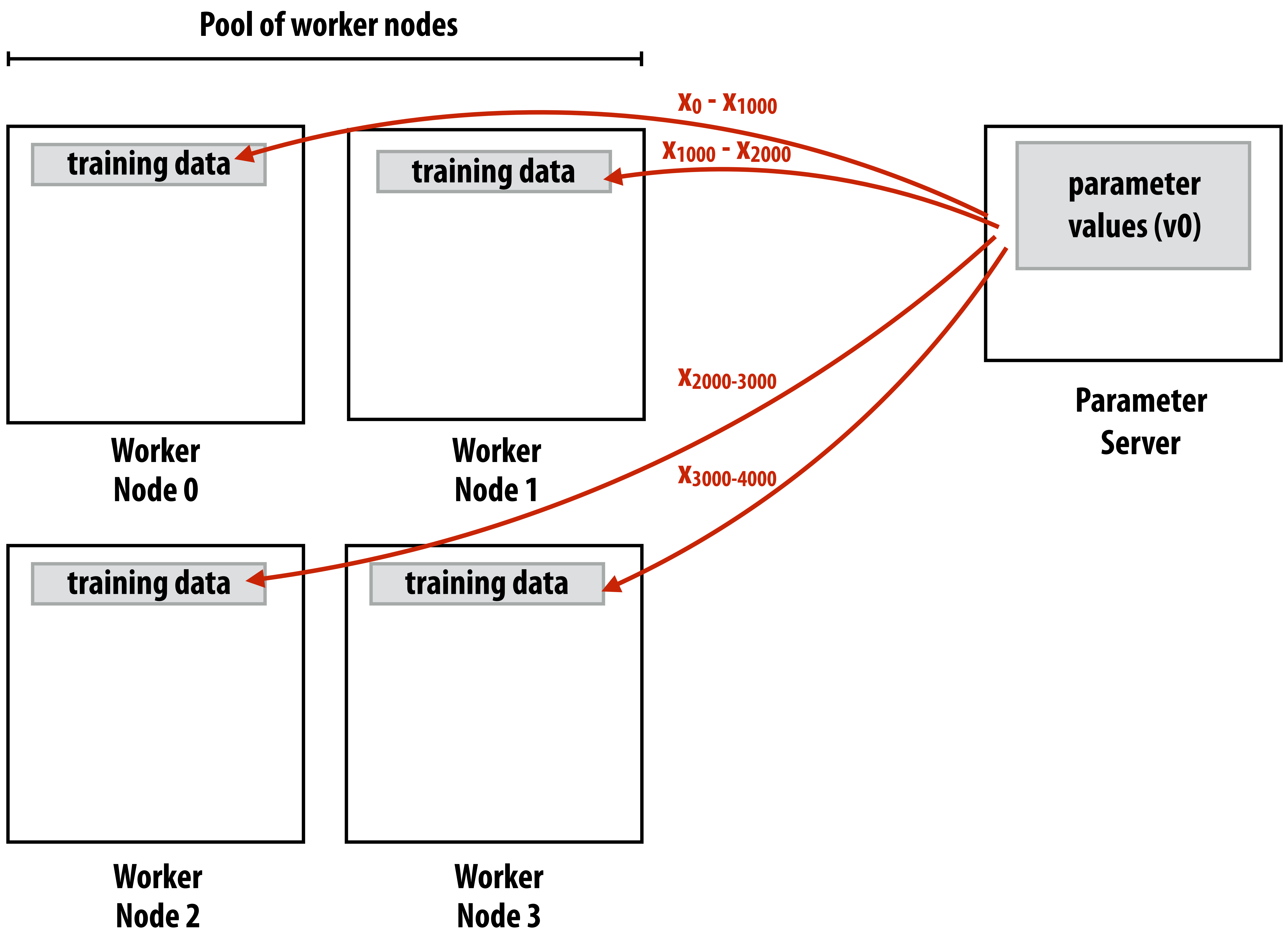
Alternative solution: exploit properties of SGD by using asynchronous execution

Parameter server design

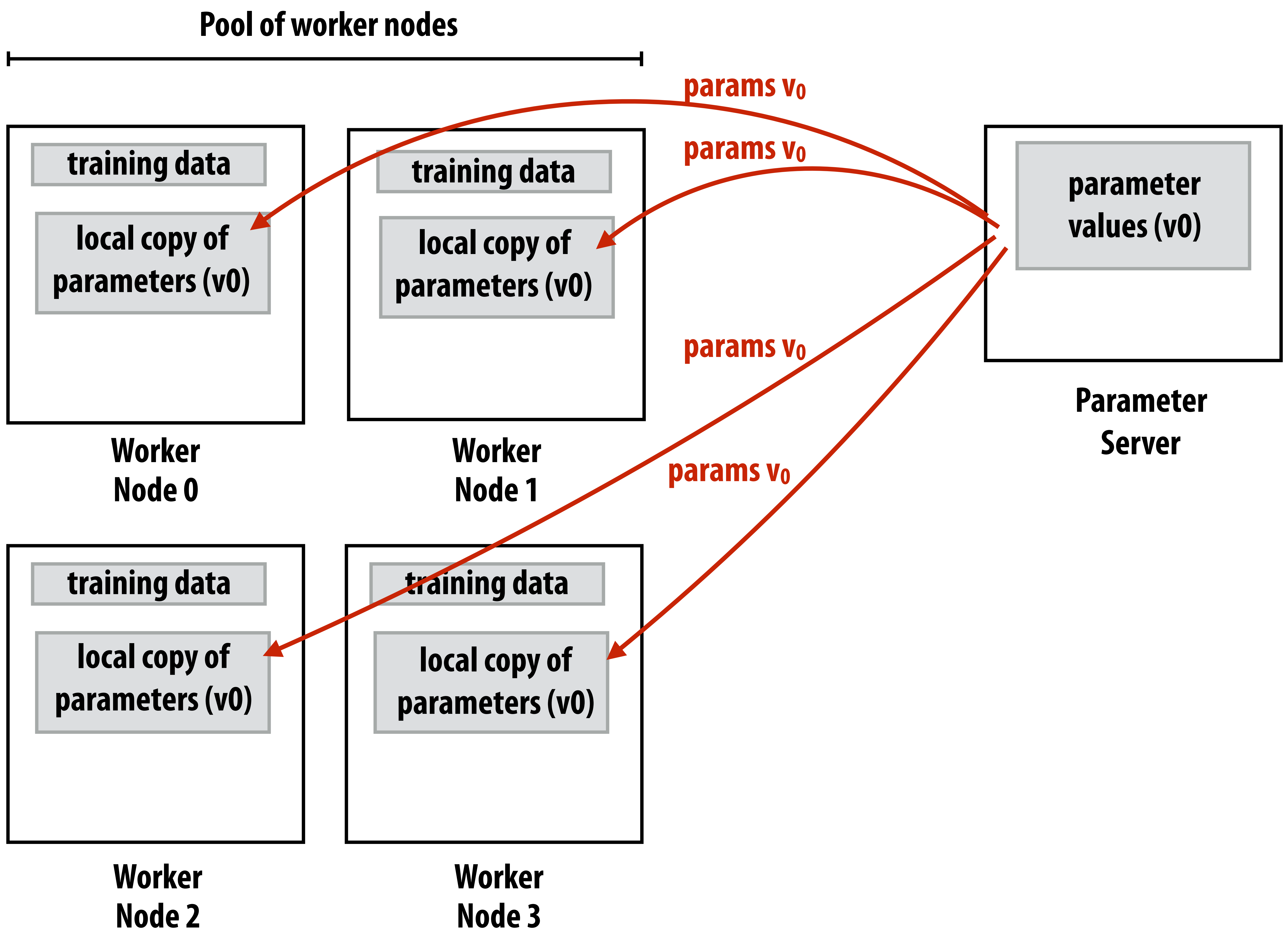
Parameter Server [Li OSDI14]
Google's DistBelief [Dean NIPS12]
Microsoft's Project Adam [Chilimbi OSDI14]



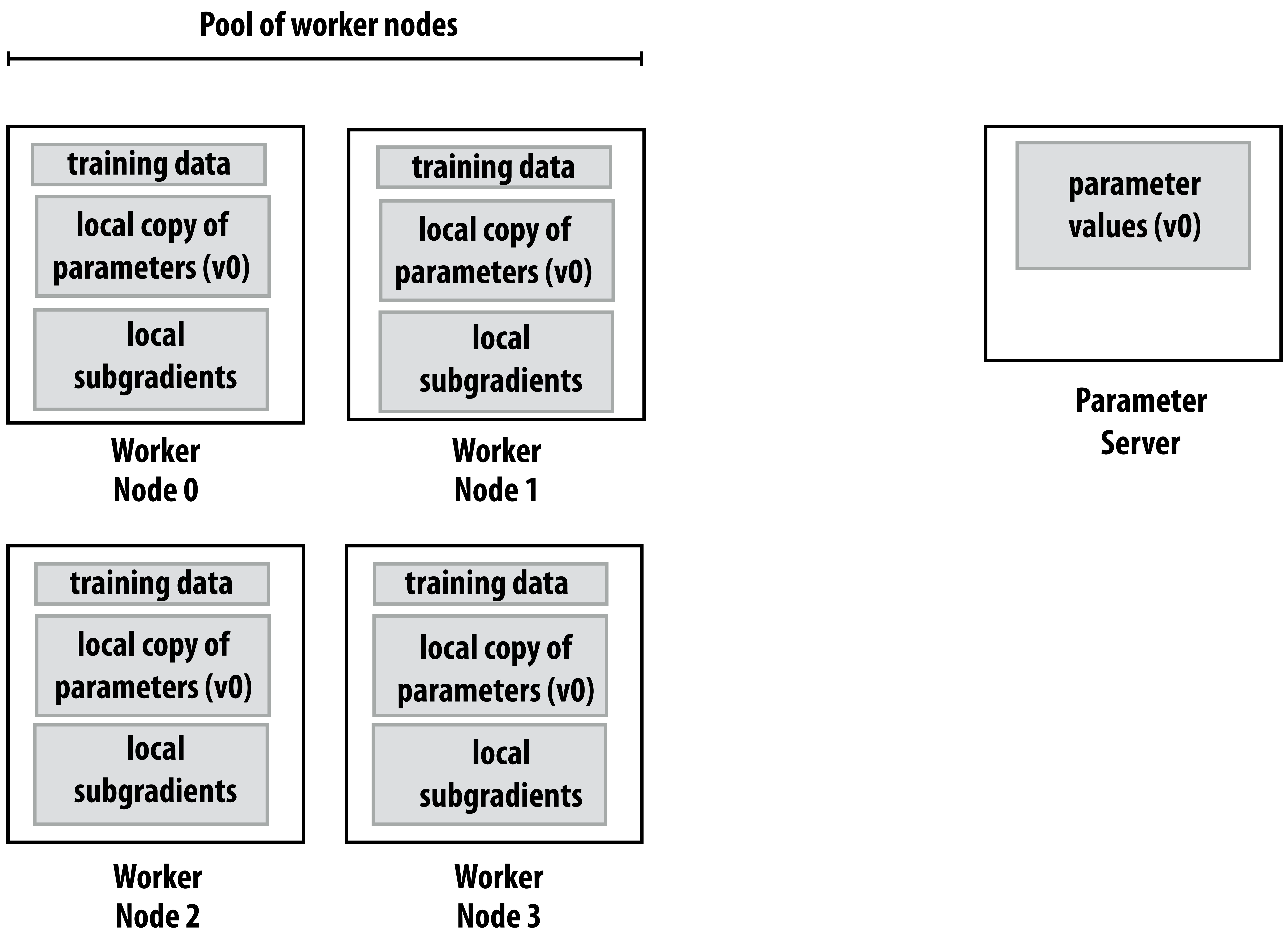
Training data partitioned among workers



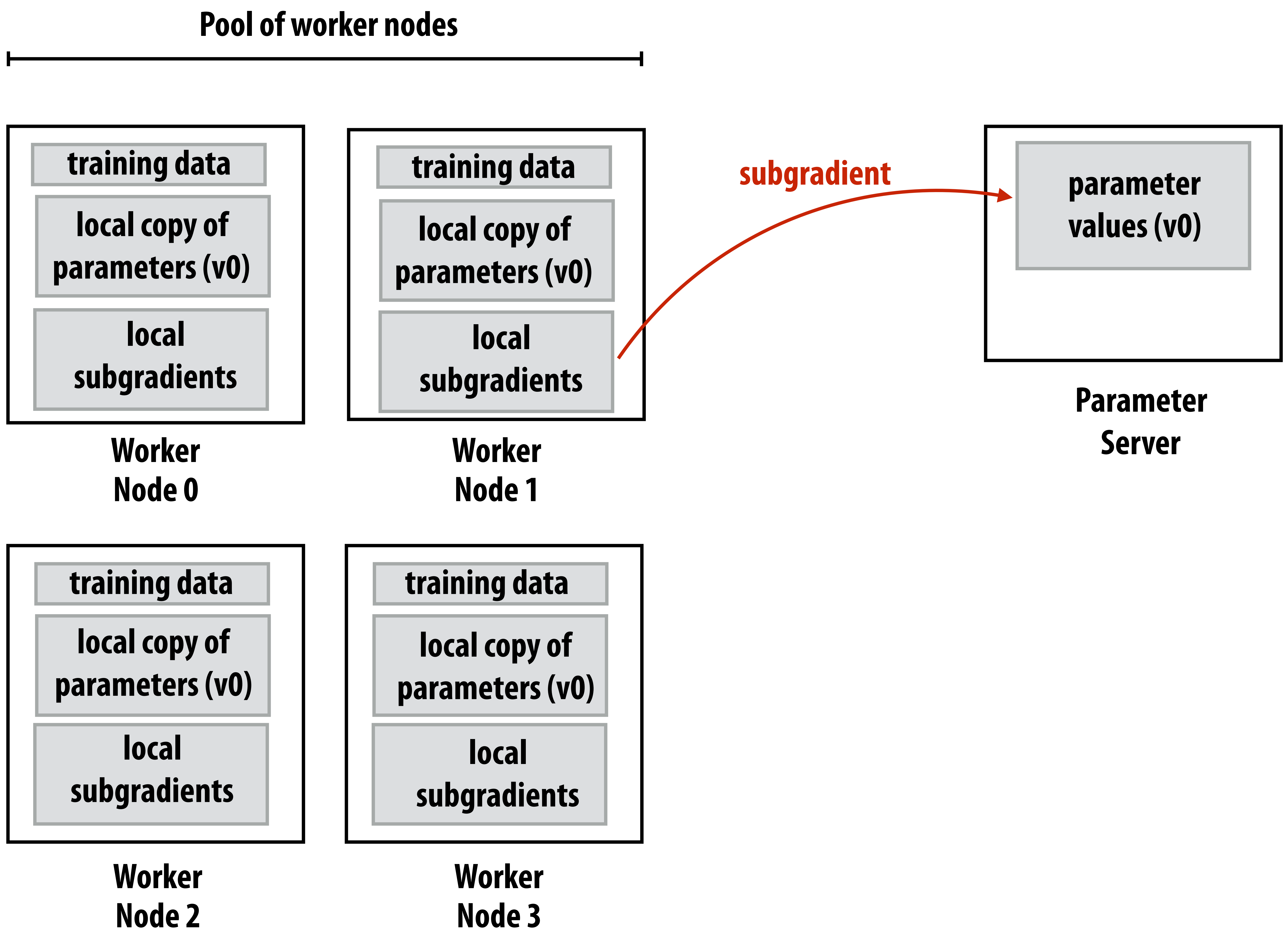
Copy of parameters sent to workers



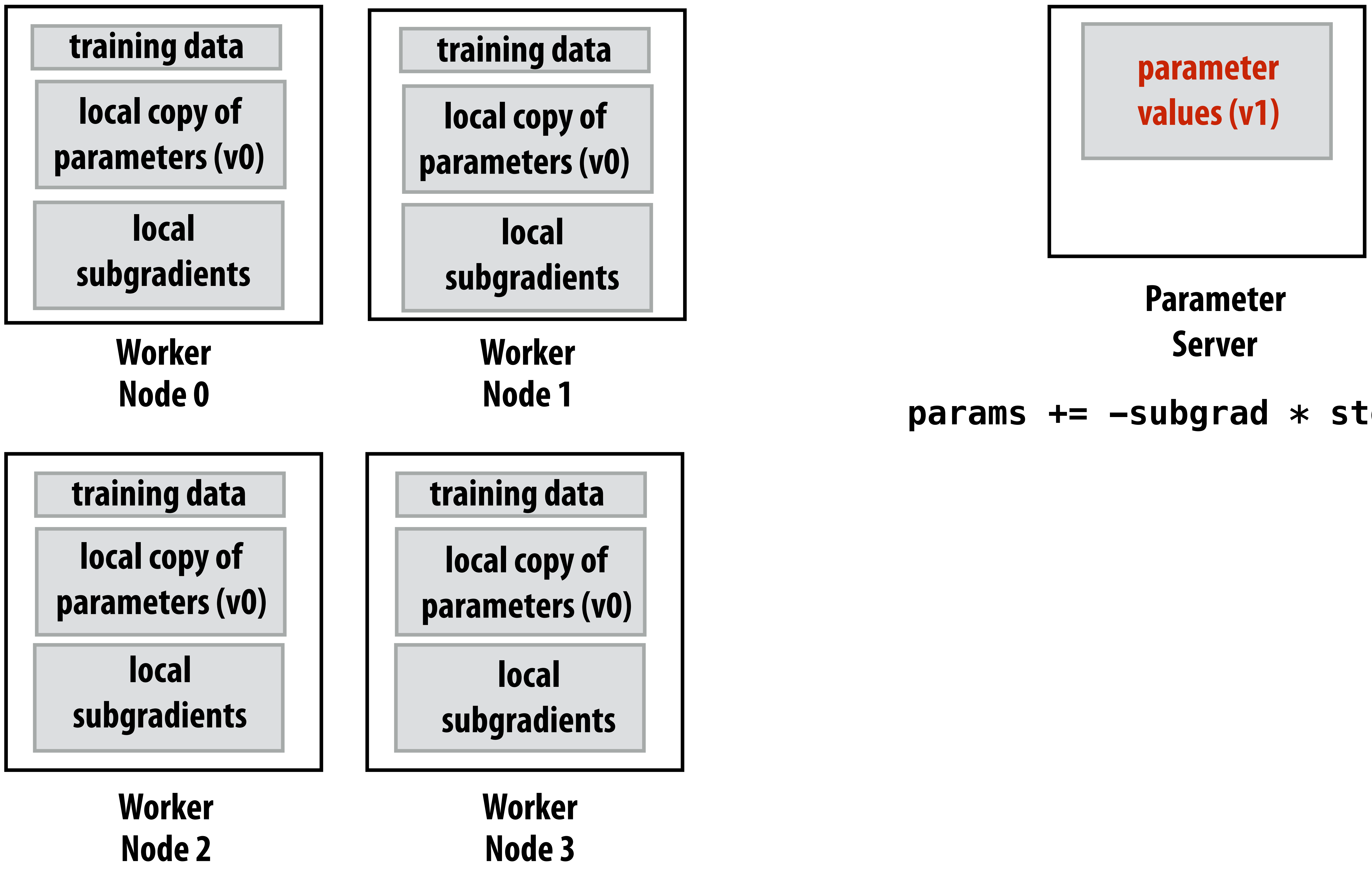
Workers independently compute local “subgradients"



Worker sends subgradient to parameter server



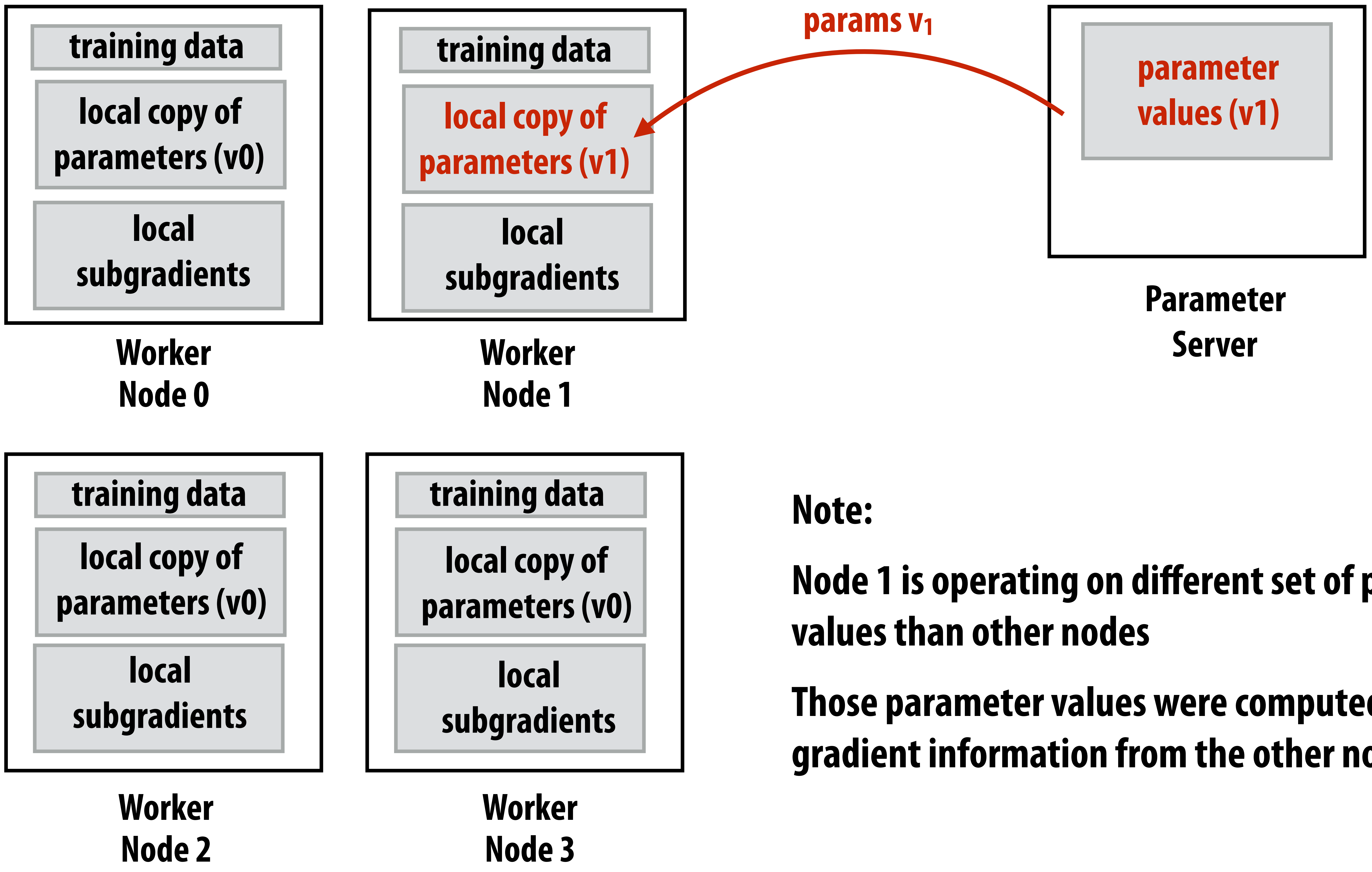
Server updates global parameter values based on subgradient



```
params += -subgrad * step_size;
```

Updated parameters sent to worker

Worker proceeds with another gradient computation step

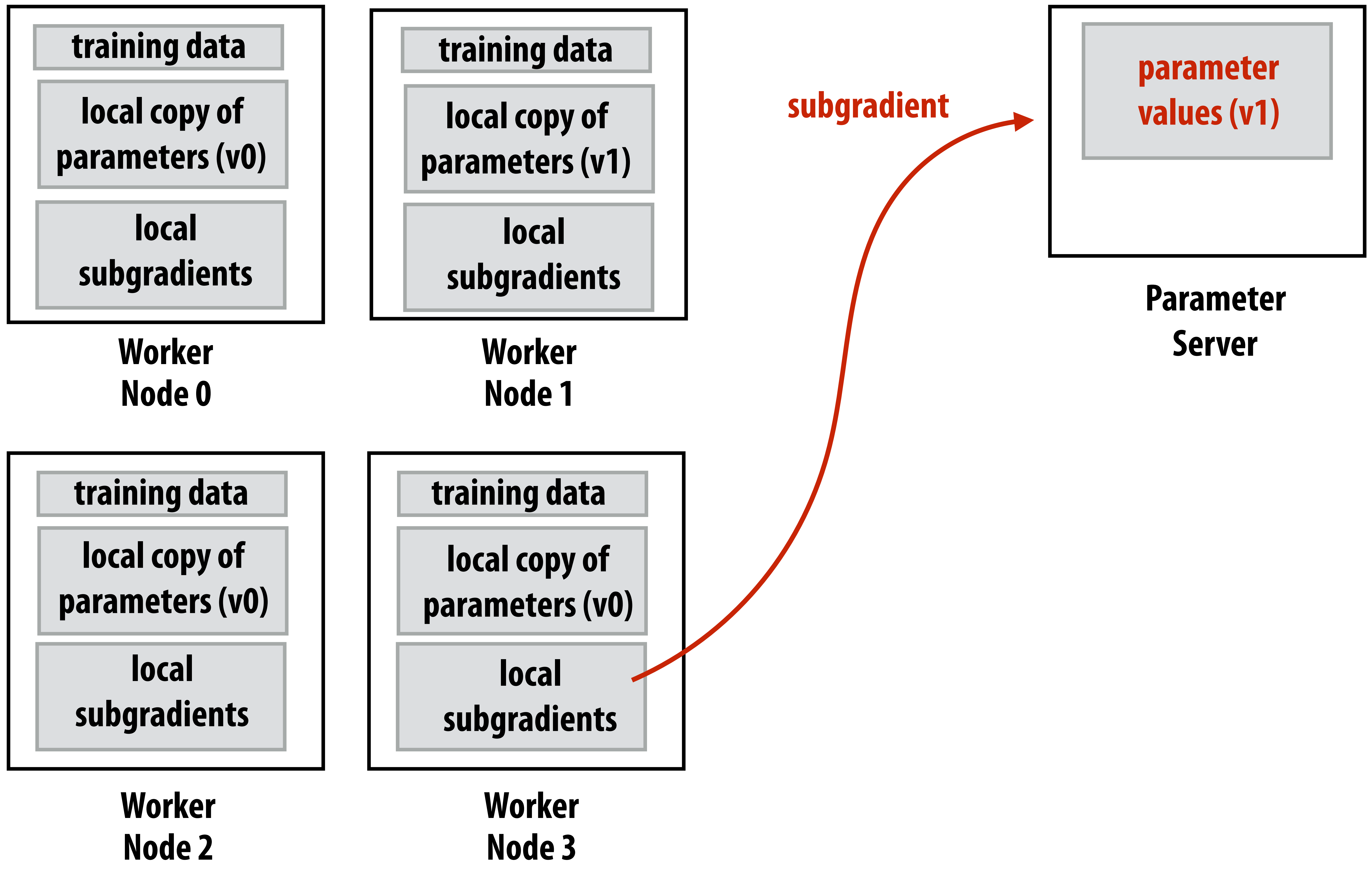


Note:

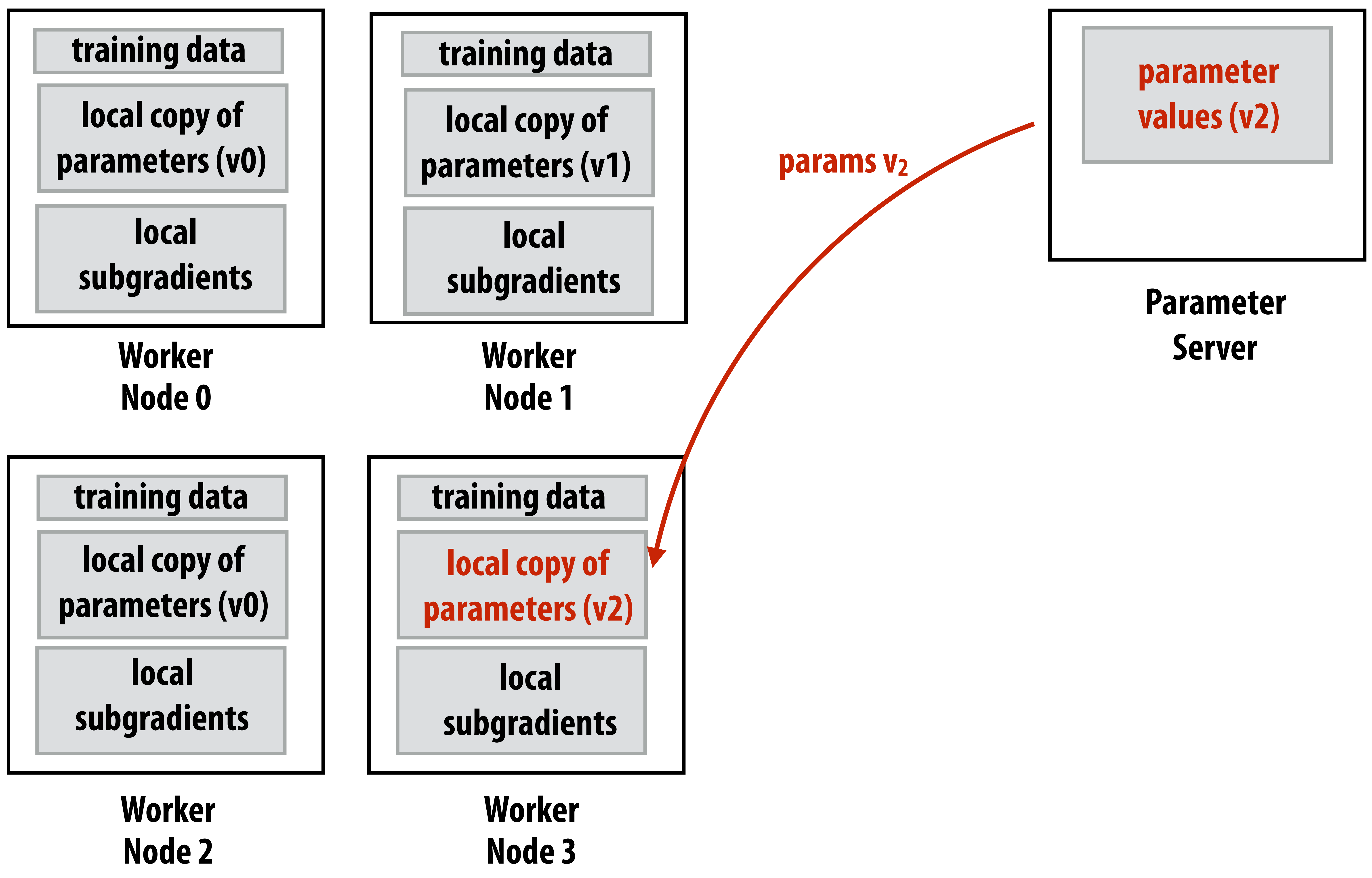
Node 1 is operating on different set of parameter values than other nodes

Those parameter values were computed without gradient information from the other nodes

Updated parameters sent to worker (again)



Worker continues with updated parameters

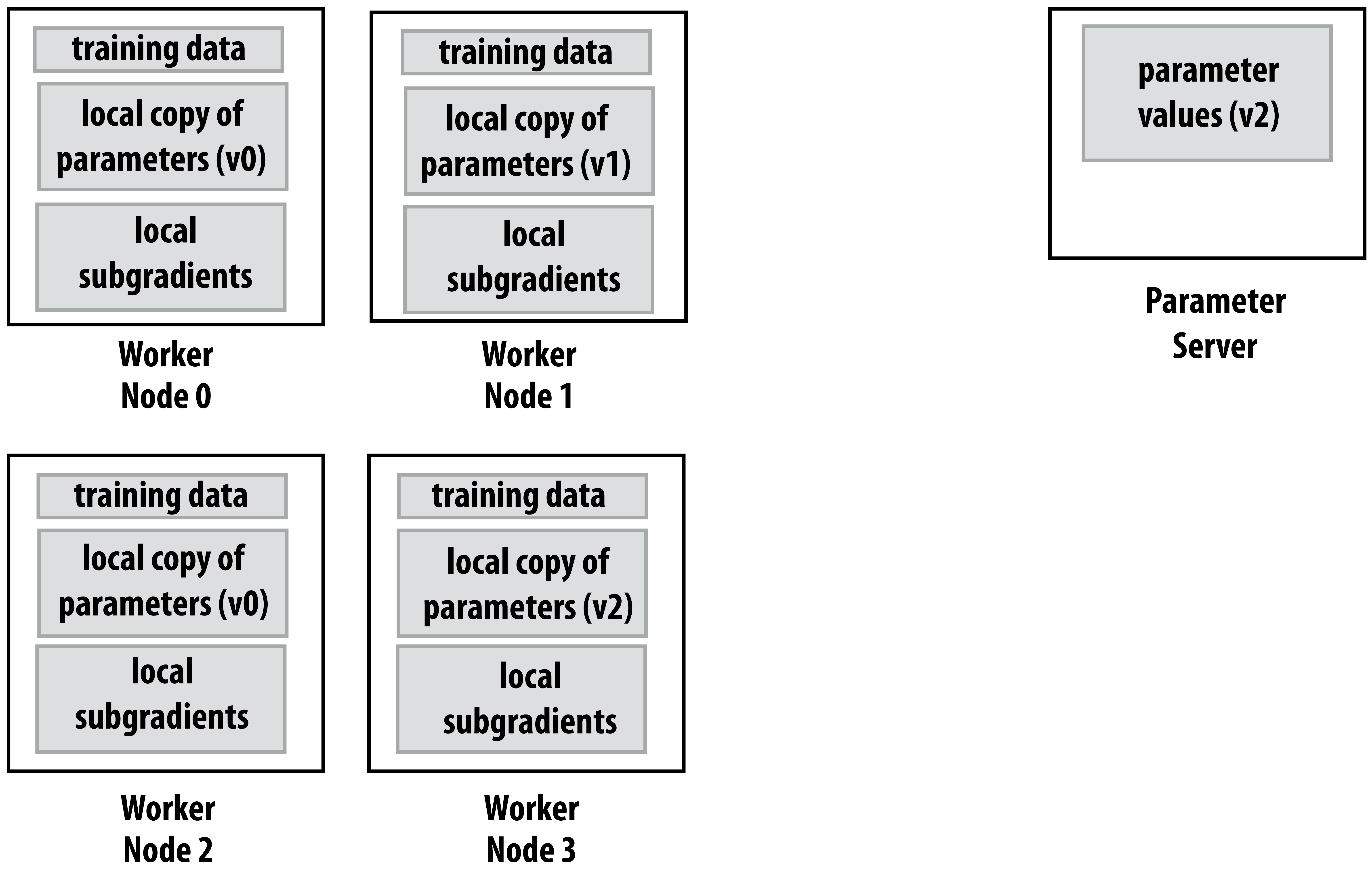


Summary: asynchronous parameter update

- **Idea: avoid global synchronization on all parameter updates between each SGD iteration**
  - Design reflects realities of cluster computing:
    - Slow interconnects
    - Unpredictable machine performance
- **Solution: asynchronous (and partial) subgradient updates**
- **Will impact convergence of SGD**
  - Node N working on iteration *i* may not have parameter values that result the results of the *i-1* prior SGD iterations

Bottleneck?

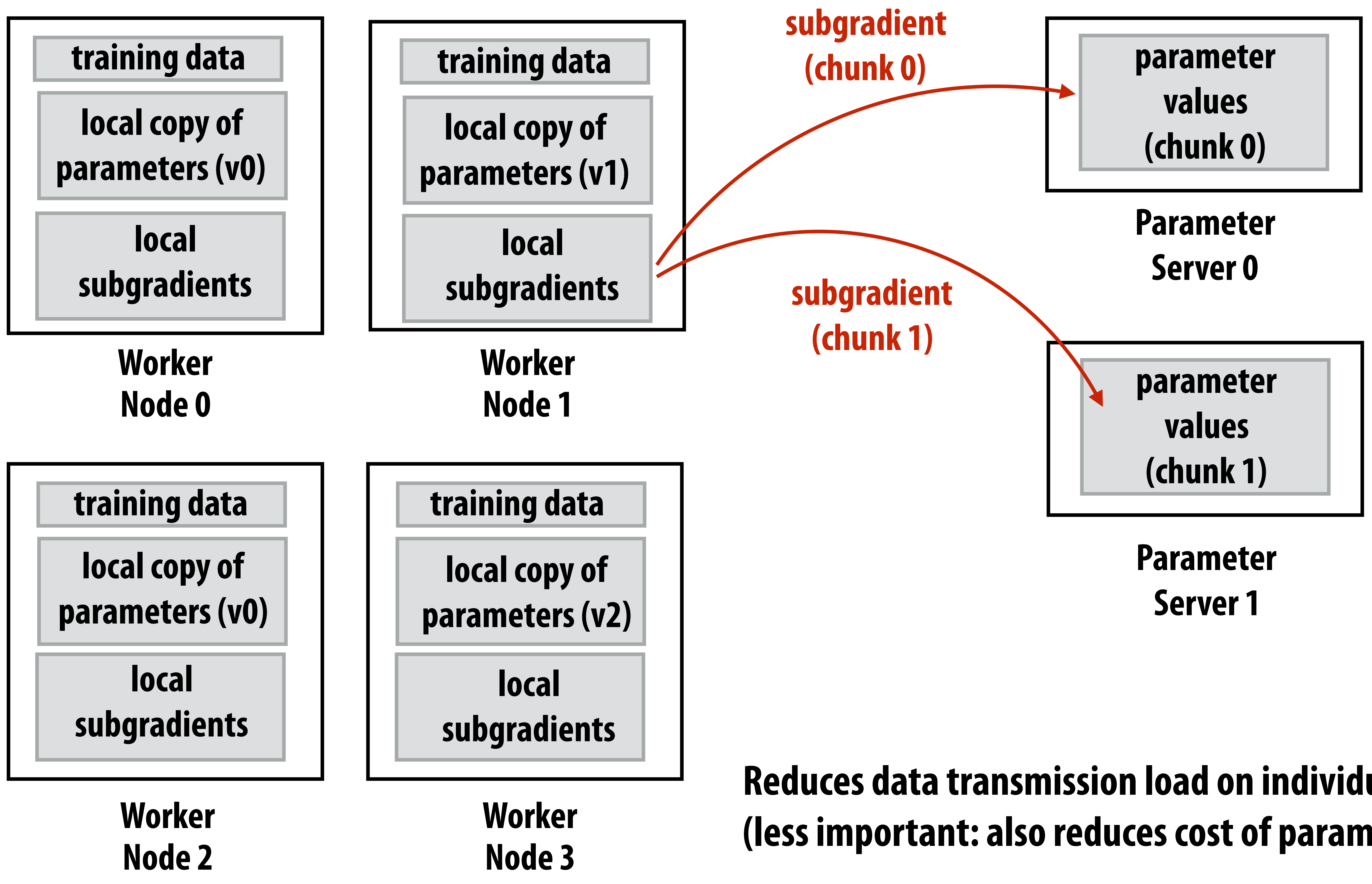
What if there is heavy contention for parameter server?



Shard the parameter server

Partition parameters across servers

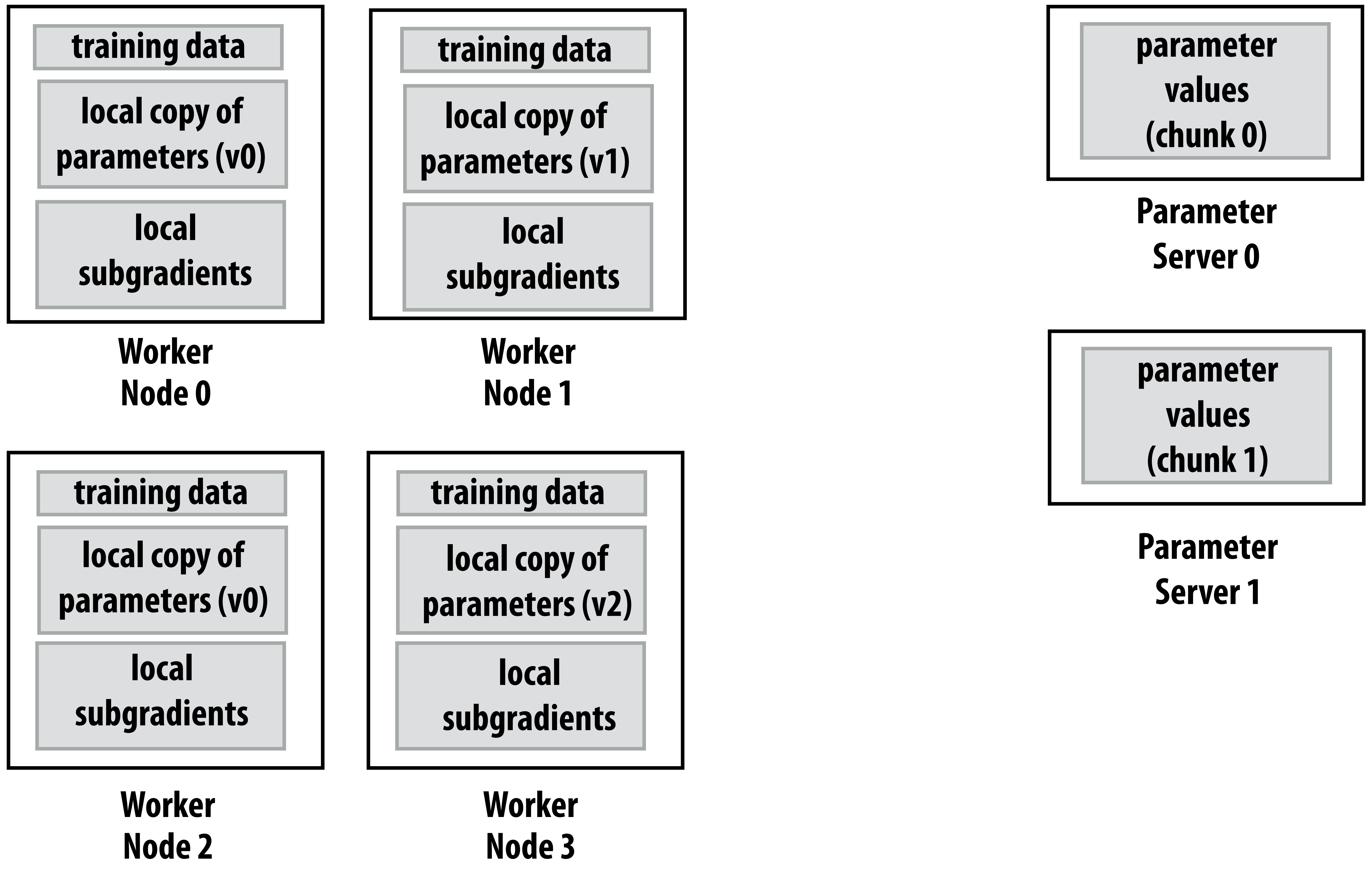
Worker sends chunk of subgradients to owning parameter server



Reduces data transmission load on individual servers
(less important: also reduces cost of parameter update)

What if model parameters do not fit on one worker?

Recall high footprint of training large networks (particularly with large mini-batch sizes)

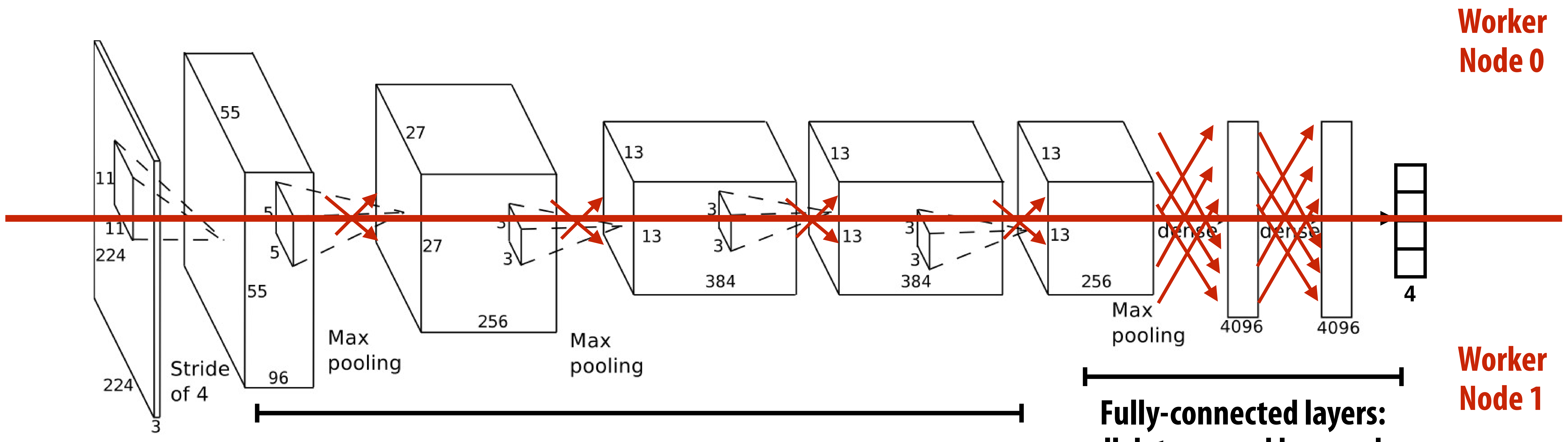


Model parallelism

Partition network parameters across nodes (spatial partitioning to reduce communication)

Reduce internode communication through network design:

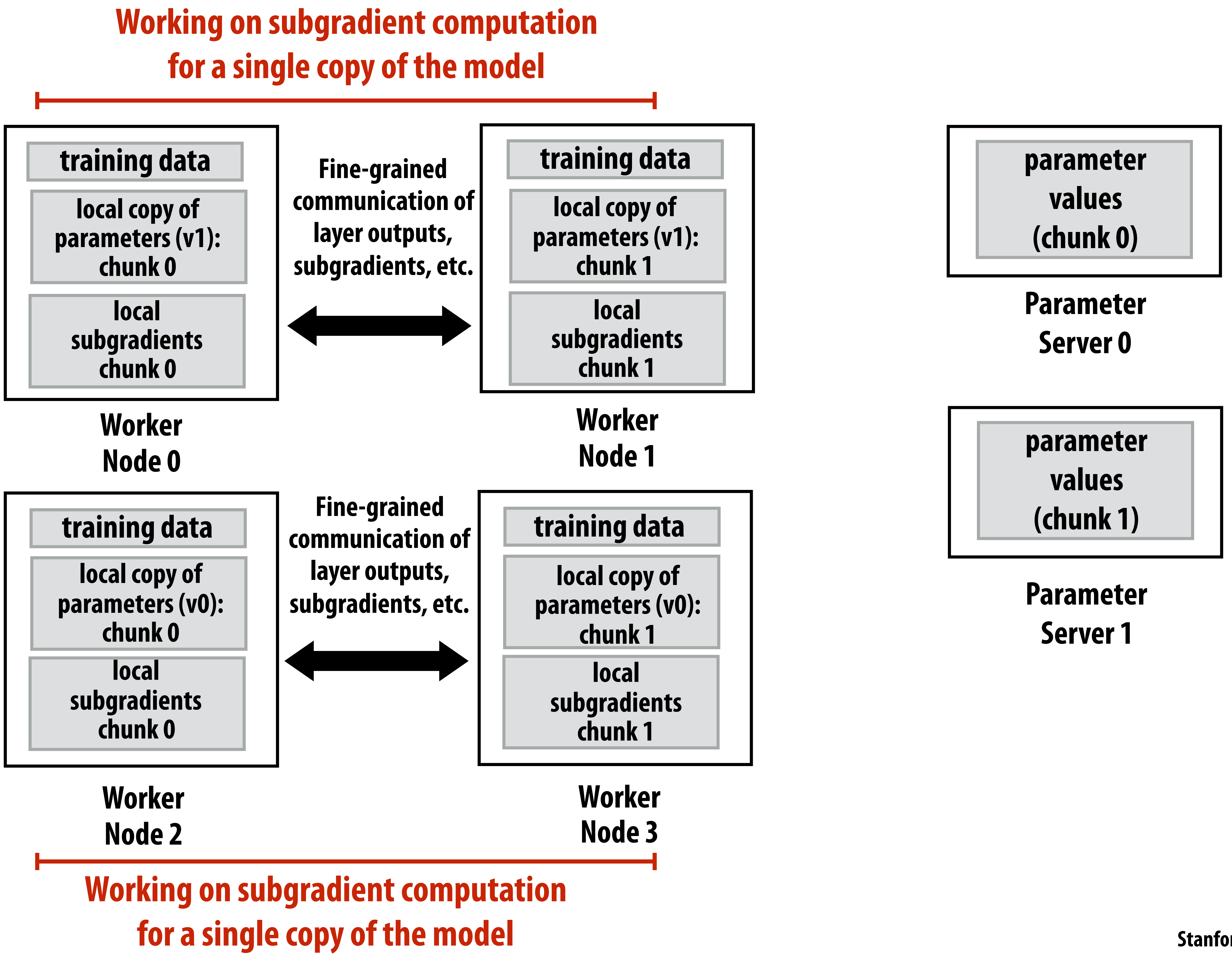
- Use small spatial convolutions (1x1 convolutions)
- Reduce/shrink fully-connected layers



Convolutional layers: only need to community outputs near spatial partition

Fully-connected layers: all data owned by a node must by communicated to other nodes

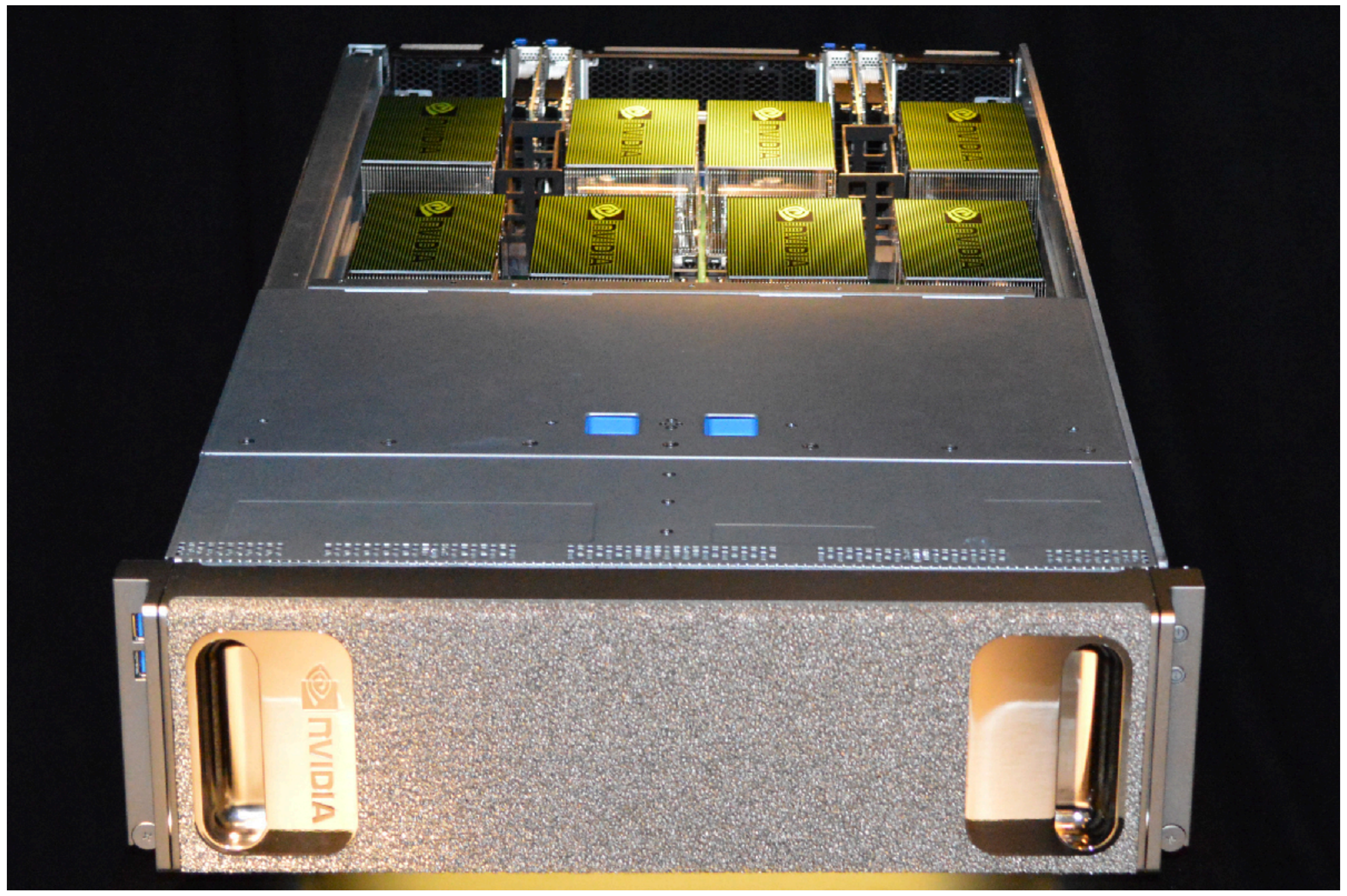
Training data-parallel and model-parallel execution


Working on subgradient computation for a single copy of the model
training data
local copy of parameters (v1): chunk 0
local subgradients chunk 0
Worker Node 0
Fine-grained communication of layer outputs, subgradients, etc.
training data
local copy of parameters (v1): chunk 1
local subgradients chunk 1
Worker Node 1
parameter values (chunk 0)
Parameter Server 0
parameter values (chunk 1)
Parameter Server 1
training data
local copy of parameters (v0): chunk 0
local subgradients chunk 0
Worker Node 2
Fine-grained communication of layer outputs, subgradients, etc.
training data
local copy of parameters (v0): chunk 1
local subgradients chunk 1
Worker Node 3
Working on subgradient computation for a single copy of the model

Better hardware? using supercomputers for training?

- Fast interconnects critical for model-parallel training
  - Fine-grained communication of outputs and gradients
- Fast interconnects diminish need for async training algorithms
  - Avoid randomness in training due to schedule of computation (yes, there remains randomness due to SGD algorithm)

OakRidge Titan Supercomputer
(Cray low-latency interconnect)



NVIDIA DGX-1: 8 Pascal GPUs connected via high speed NV-Link interconnect

Better algorithmic techniques (again): improving scalability of synchronous training…

- **Larger mini-batches increase compute to communication ratio: communicate gradients summed over B training inputs**

```
for each item x in mini-batch on this node:
   grad += evaluate_loss_gradient(f, loss_func, params, x)
barrier();
sum reduce gradients across all nodes, communicate results to all nodes
barrier();
update copy of local parameter values
```

- **But large mini-batches (if used naively) reduce accuracy of model trained**

Linear scaling rule

size of mini batch = n
SGD learning rate = η

Recall: minibatch SGD parameter update

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in \mathcal{B}} \nabla l(x, w_t)$$

Consider processing of k minibatches (k steps of gradient descent)

$$w_{t+k} = w_t - \eta \frac{1}{n} \sum_{j<k} \sum_{x \in \mathcal{B}_j} \nabla l(x, w_{t+j})$$

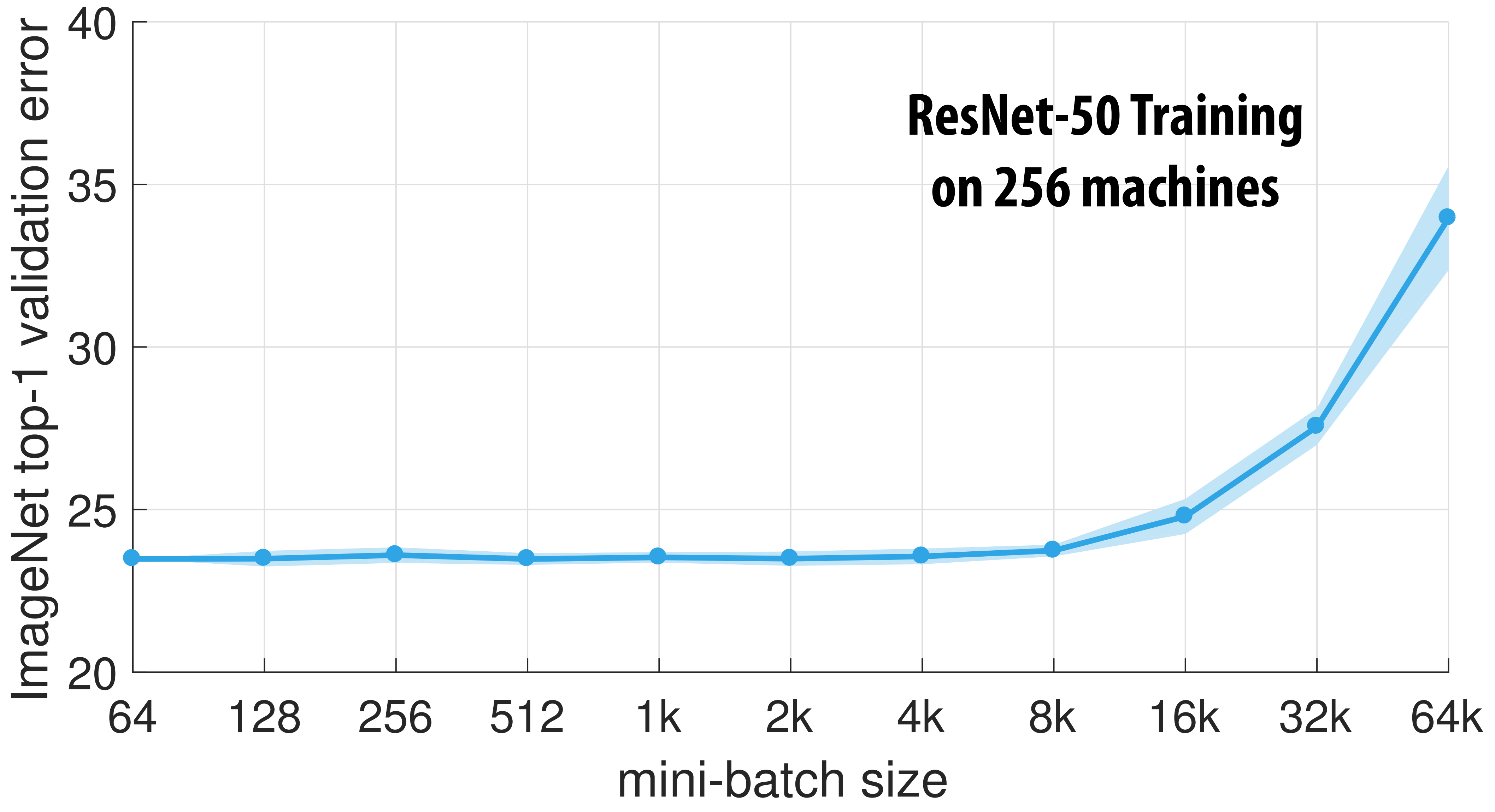
Consider processing one minibatch that is of size kn (one step of gradient descent)

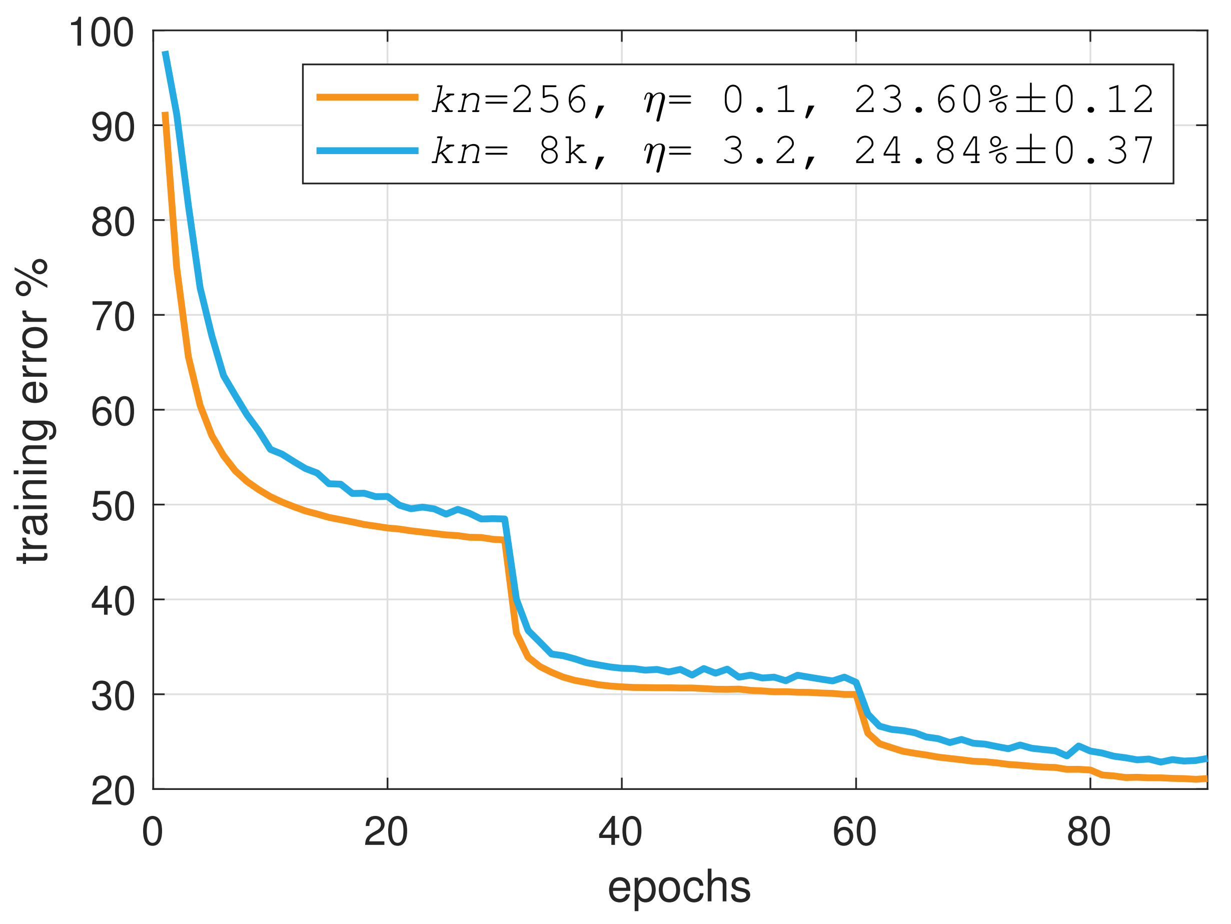
$$\hat{w}_{t+1} = w_t - \hat{\eta} \frac{1}{kn} \sum_{j<k} \sum_{x \in \mathcal{B}_j} \nabla l(x, w_t)$$

Suggests that if $\nabla l(x, w_t) \approx \nabla l(x, w_{t+j})$ for *j* < *k* then minibatch SGD with size *n* and learning rate η can be approximated by large mini batch SGD with size *kn* if the learning rate is also scaled to $k\eta$

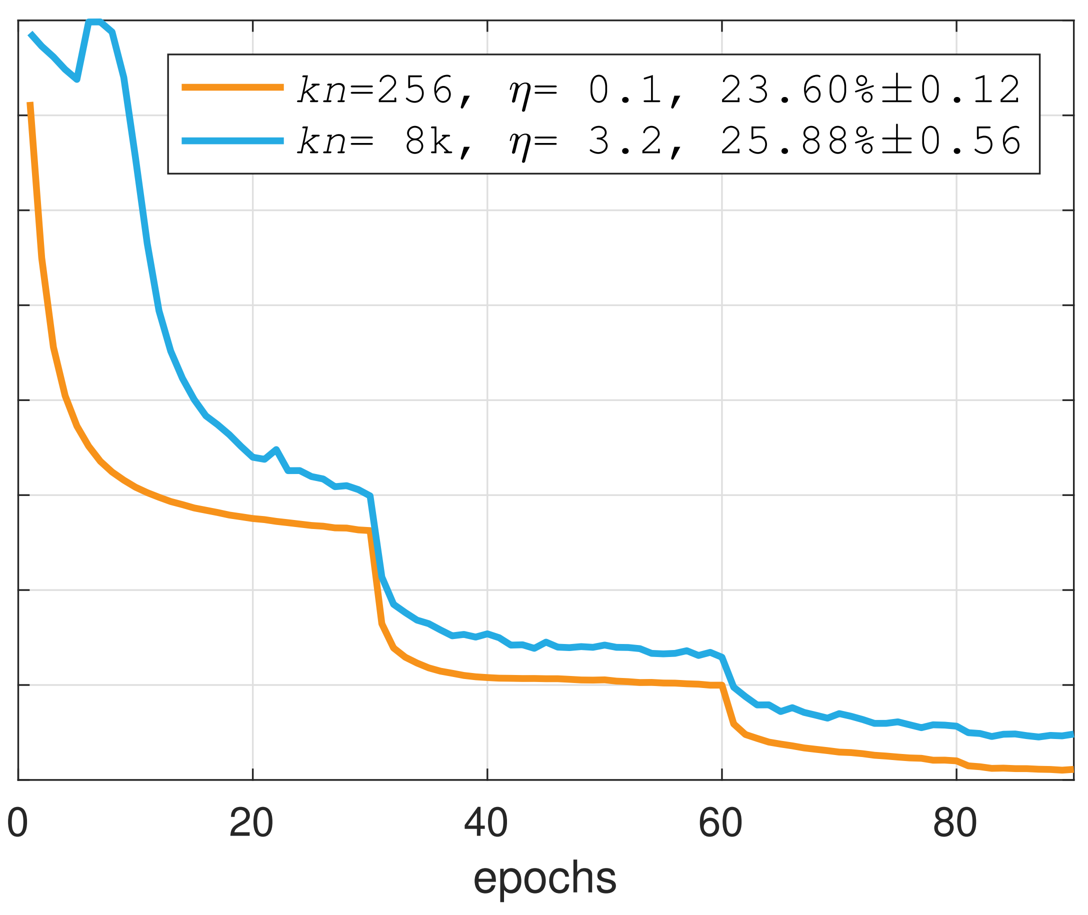
When does $\nabla l(x, w_t) \approx \nabla l(x, w_{t+j})$ not hold?

- **At beginning of training**
  - **Suggests starting training with smaller learning rate (learning rate "warmup")**
- **When minibatch size begins to get too large (there is a limit to scaling minibatch size)**

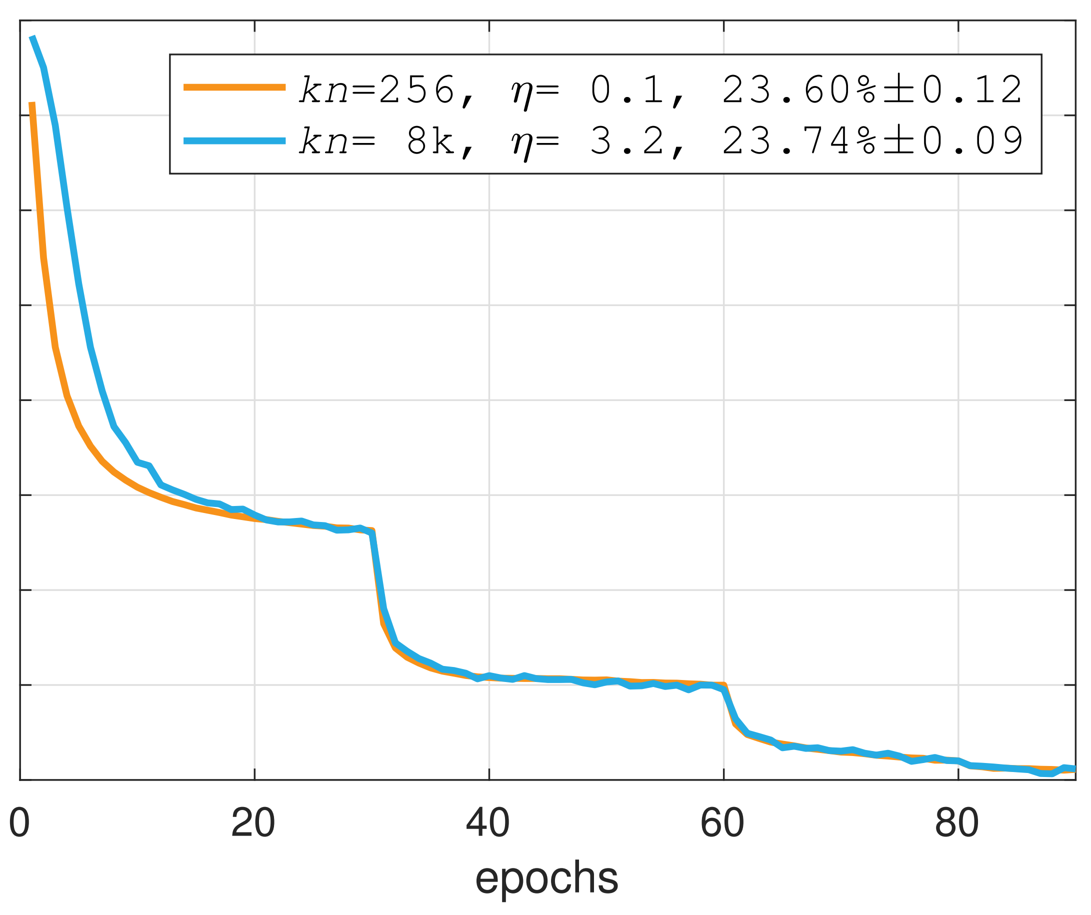




(a) no warmup



(b) constant warmup



(c) gradual warmup

Minibatch size 256 (orange) vs. 8192 (blue)

[Figure credit: Goyal et al. 2017]

Summary: training large networks in parallel

- Many cluster systems rely on data-parallel training with asynchronous update to efficiently use clusters of commodity machines
  - Modification of SGD algorithm to meet constraints of modern parallel systems
  - Effects on convergence are problem dependent and not particularly well understood
  - Efficient use of fast interconnects may provide alternative to these methods (facilitate tightly orchestrated solutions much like supercomputing applications)

- Modern DNN designs (with fewer weights), large minibatch sizes, careful learning rate schedules enable scalability without asynchronous execution on commodity clusters

- High-performance training of deep networks is an interesting example of constant iteration of algorithm design and parallelization strategy (a key theme of this course!)