## Lecture 24:

# Parallel Deep Neural Networks

---

**Parallel Computer Architecture and Programming**
**CMU 15-418/15-618, Fall 2018**

# Training/evaluating deep neural networks

## Technique leading to many high-profile AI advances in recent years

**Speech recognition/natural language processing**

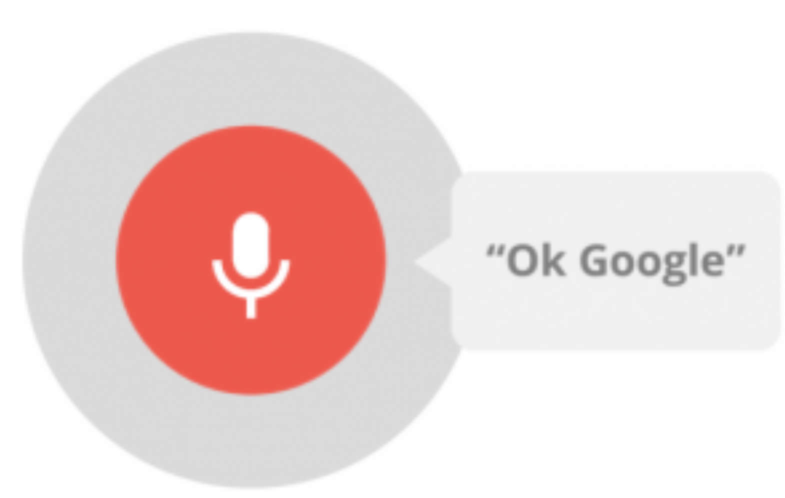
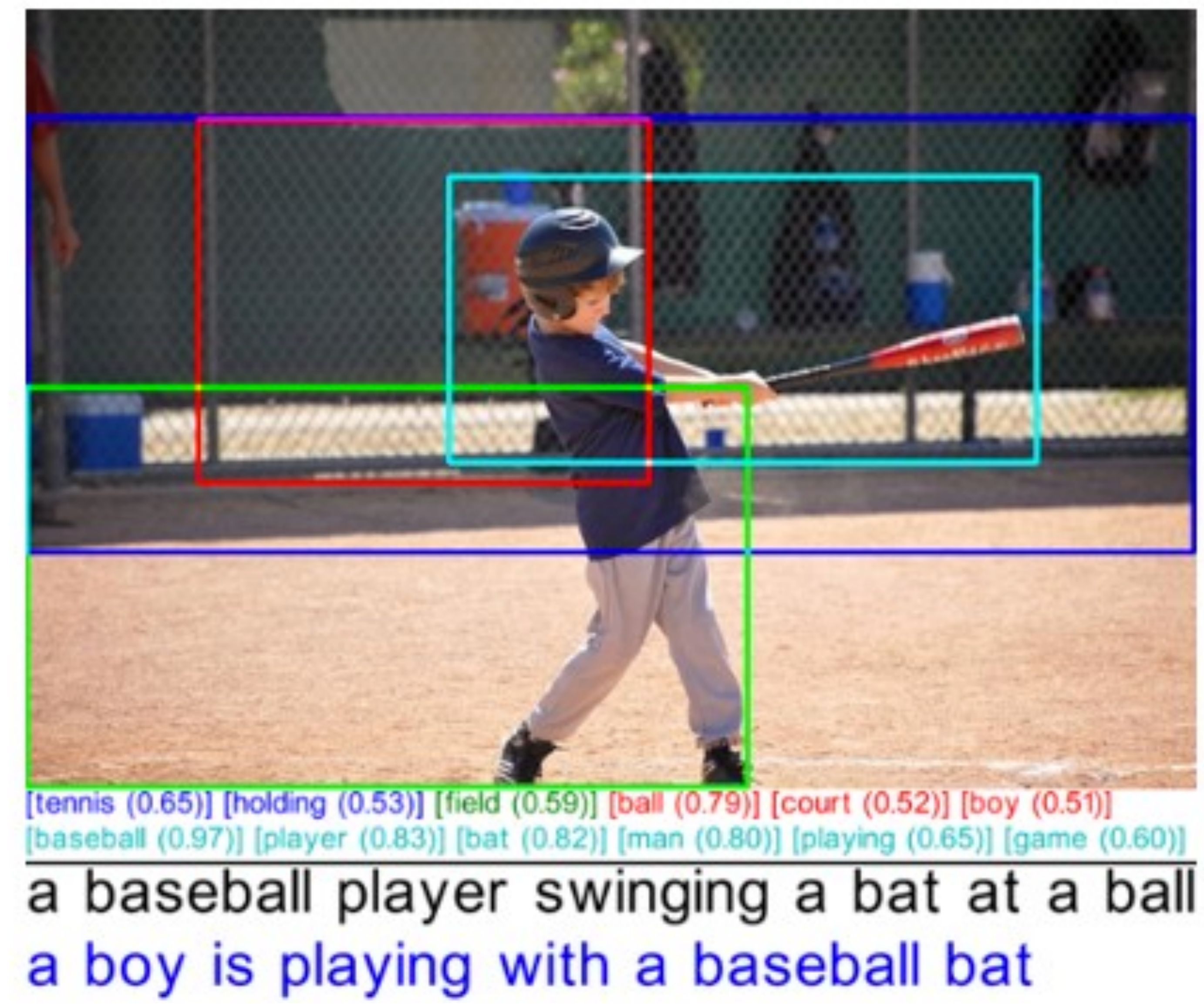
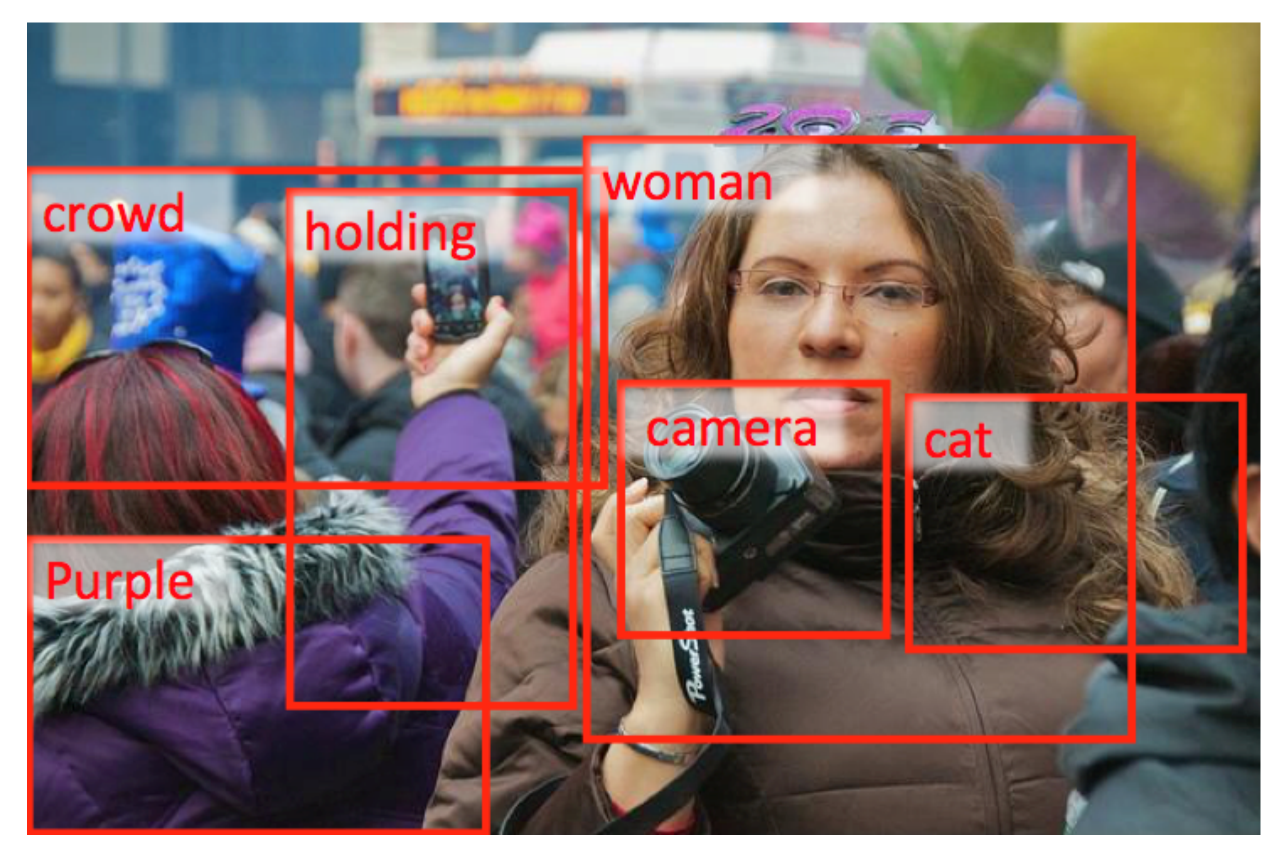



**Image interpretation and understanding**



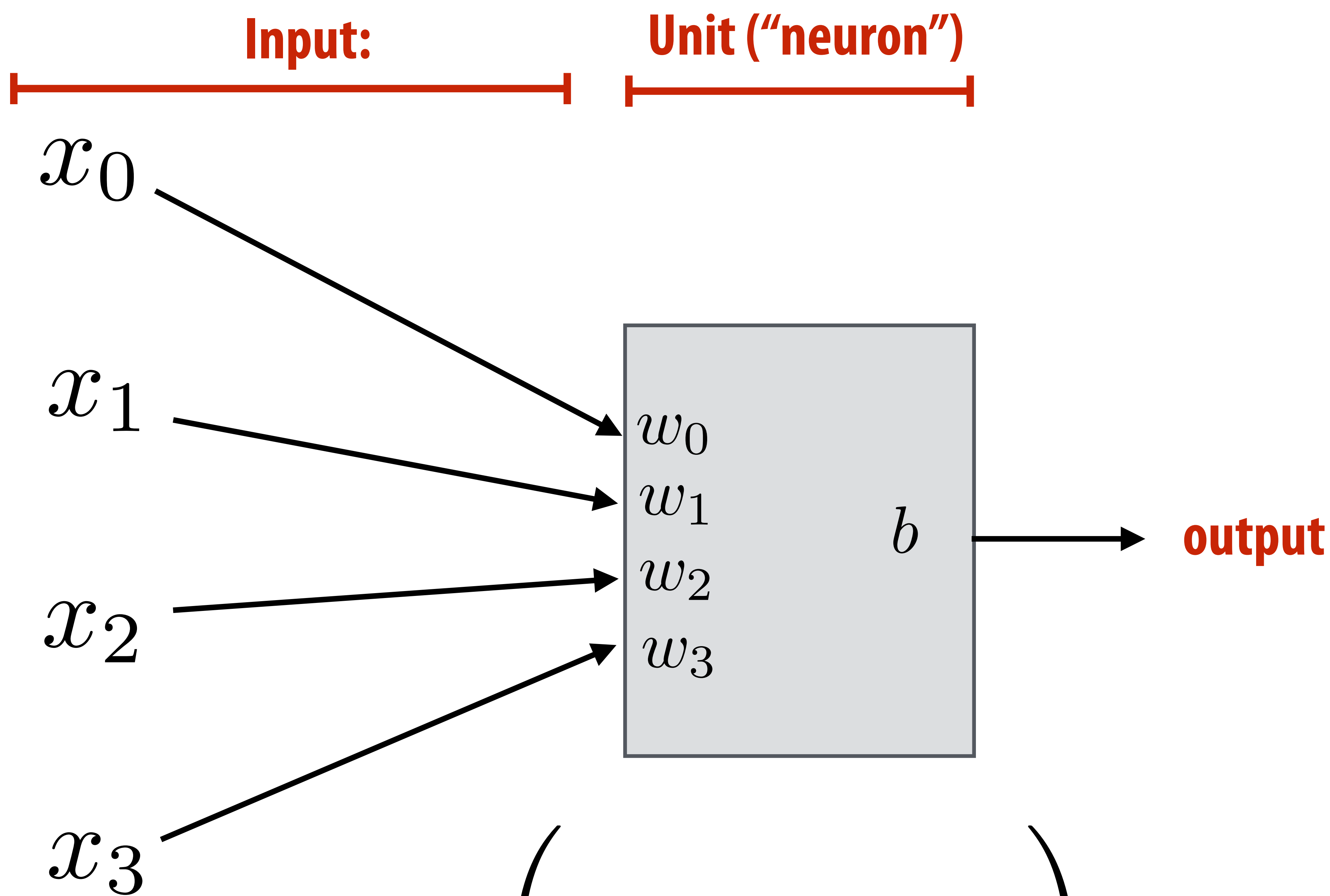




# What is a deep neural network?

## A basic unit:

**Unit with *n* inputs described by *n+1* parameters (weights + bias)**




$$f\left(\sum_i x_i w_i + b\right)$$

**Example: rectified linear unit (ReLU)**

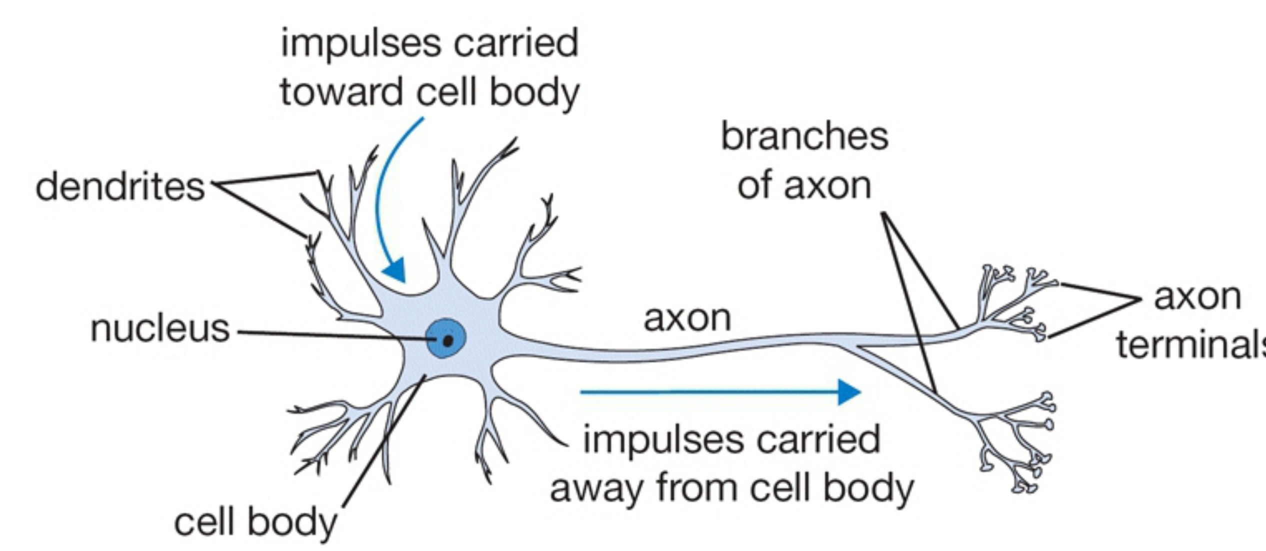
$$f(x) = max(0, x)$$

**Basic computational interpretation:**

**It's just a circuit!**

**Biological inspiration:**

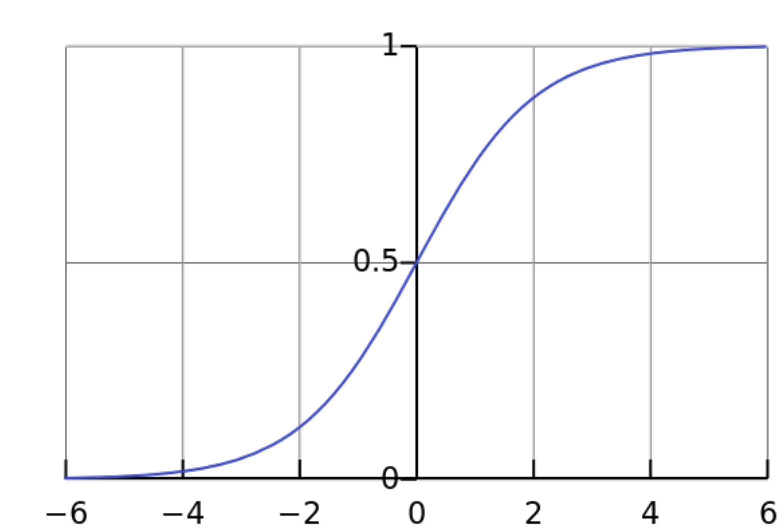
**unit output corresponds loosely to activation of neuron**




**Machine learning interpretation:**

**binary classifier: interpret output as the probability of one class**

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



# Two Distinct Issues with Deep Networks

- **Evaluation**
  - often takes milliseconds
- **Training**
  - often takes hours, days, weeks

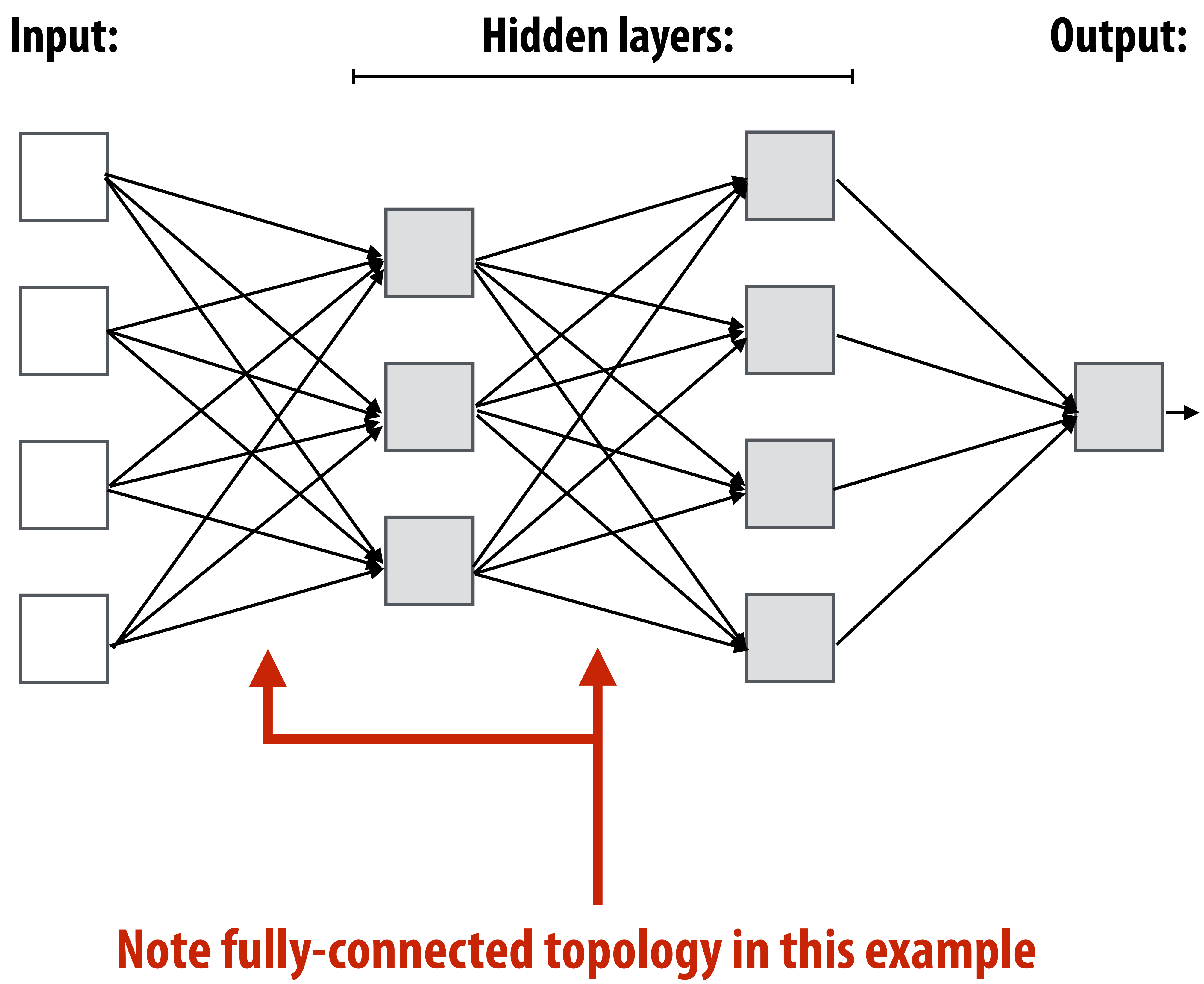
# What is a deep neural network? topology

This network has: 4 inputs, 1 output, 7 hidden units

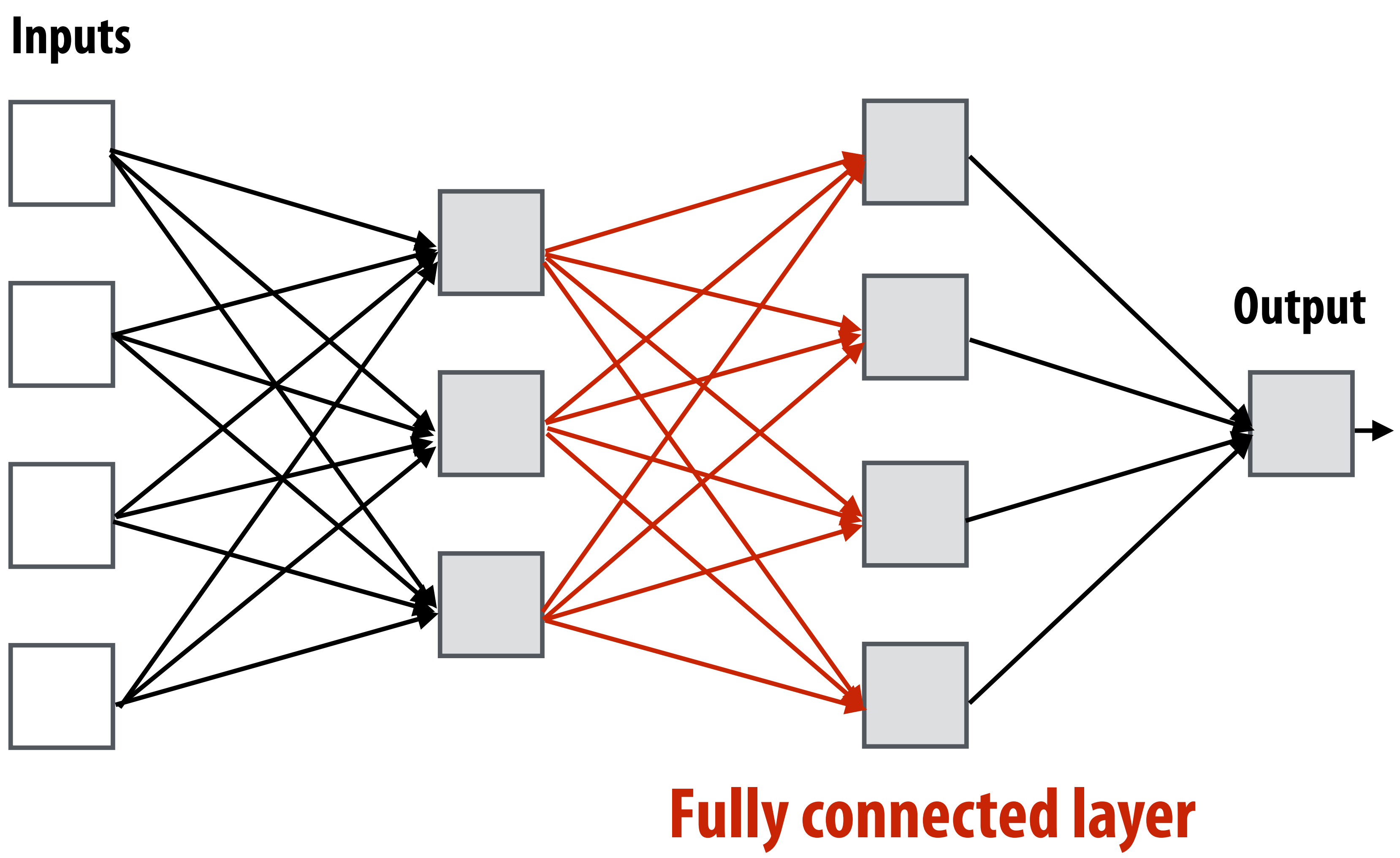
"Deep" = at least one hidden layer

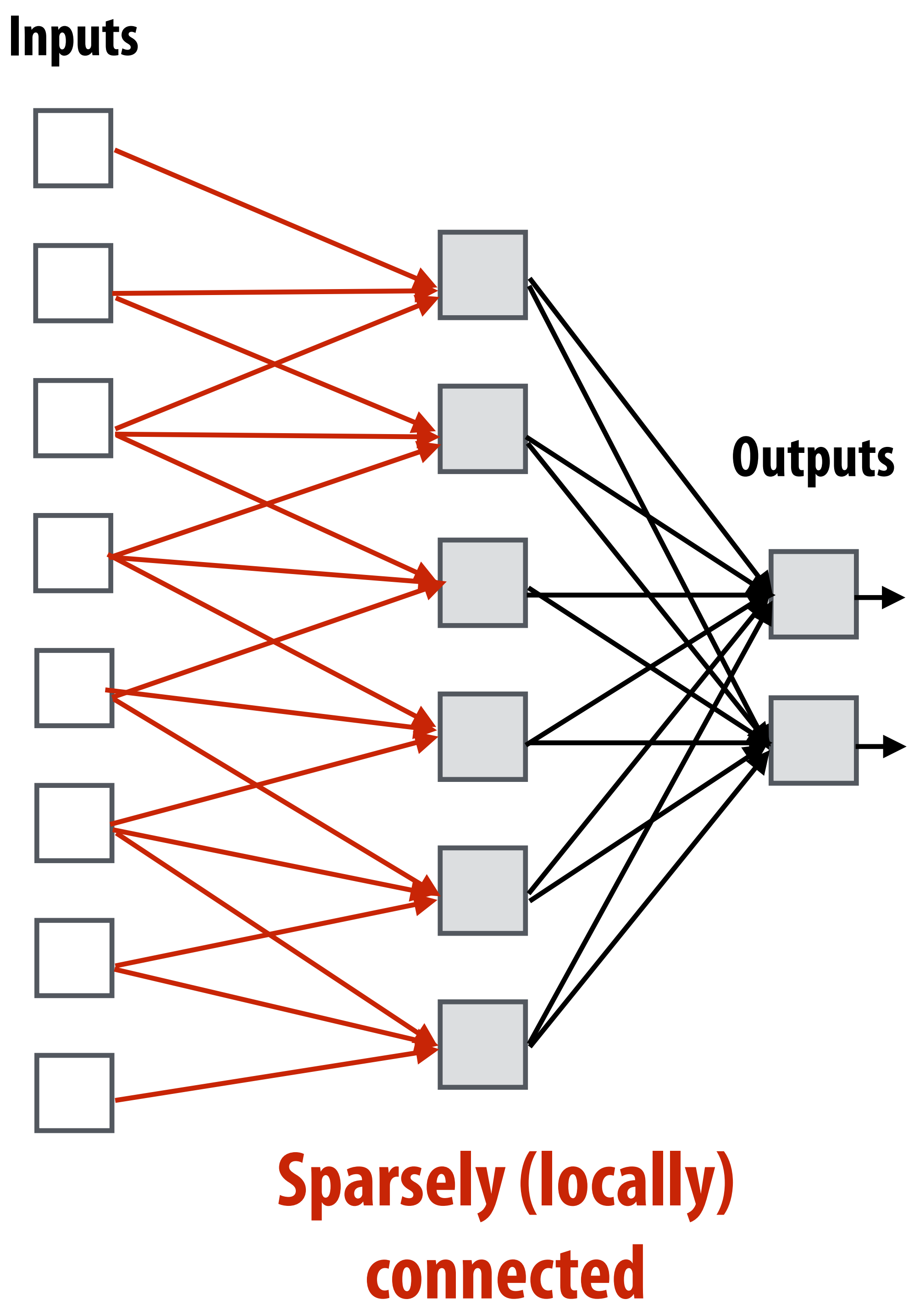
Hidden layer 1: 3 units x (4 weights + 1 bias) = 15 parameters

Hidden layer 2: 4 units x (3 weights + 1 bias) = 16 parameters



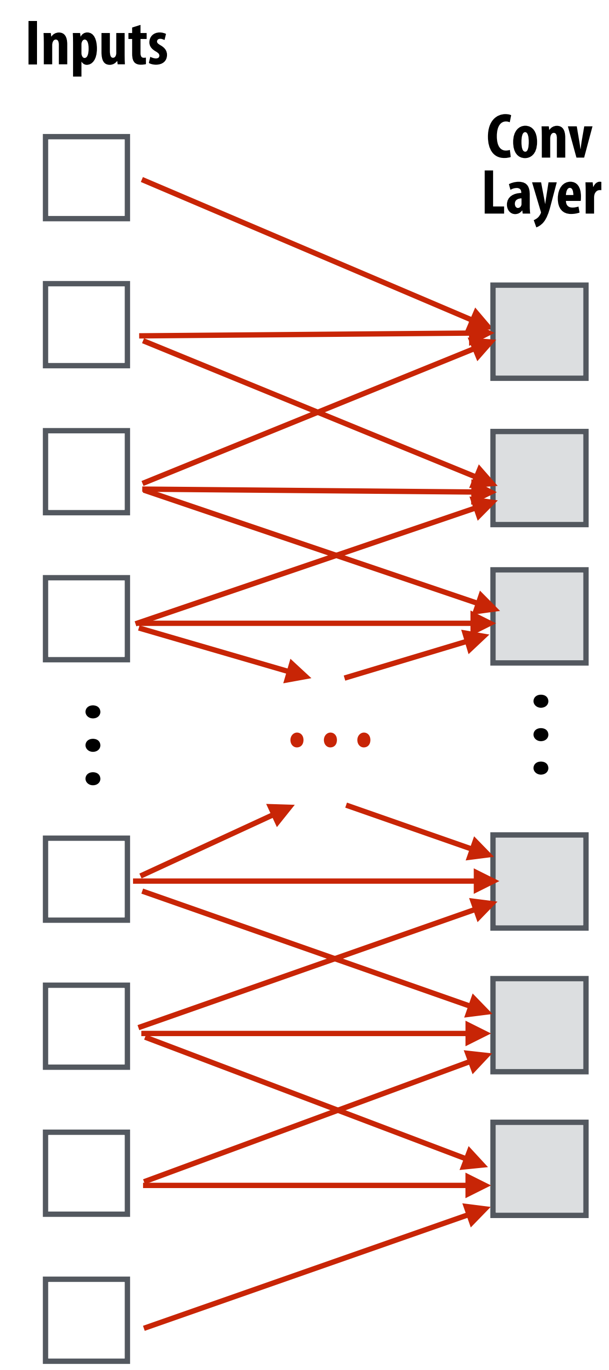
# What is a deep neural network? topology

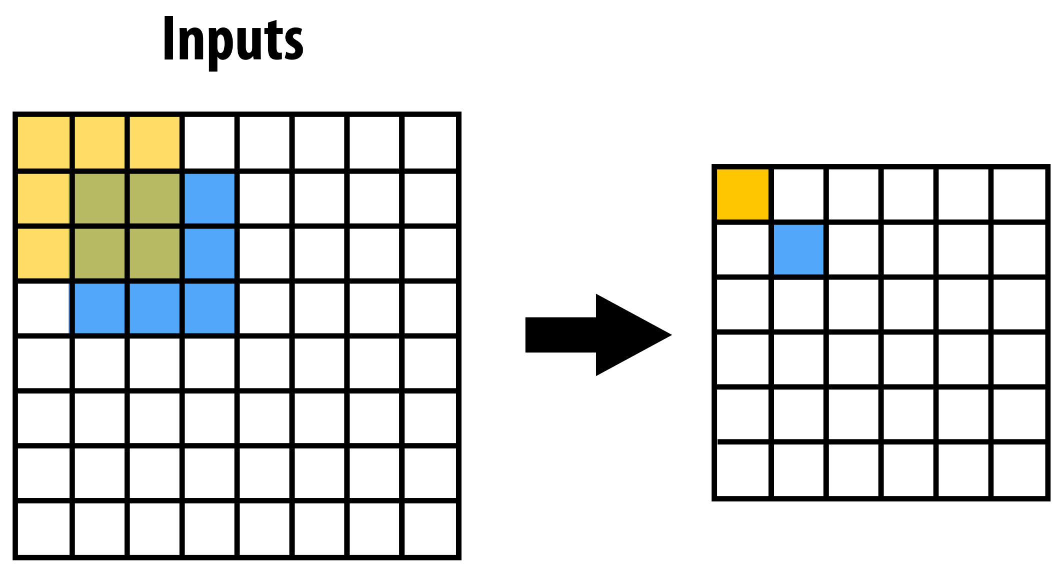


# Recall image convolution (3x3 conv)

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9};

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
    output[j*WIDTH + i] = tmp;
  }
}
```



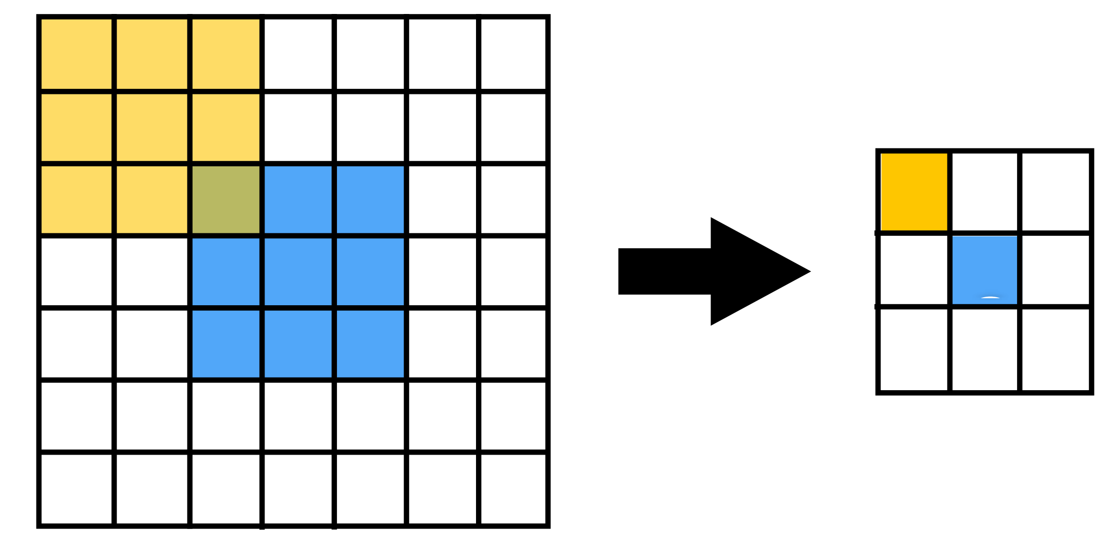




**Convolutional layer: locally connected AND all units in layer share the same parameters (same weights + same bias):**
**(note: network diagram only shows links due to one iteration of `ii` loop)**

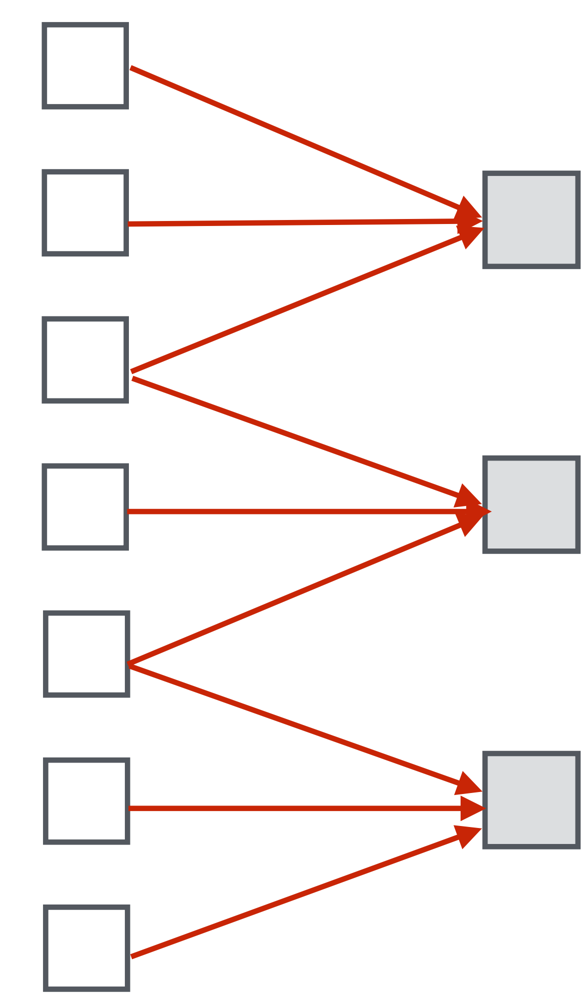
# Strided 3x3 convolution

```
int WIDTH = 1024;
int HEIGHT = 1024;
int STRIDE = 2;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[(WIDTH/STRIDE) * (HEIGHT/STRIDE)];


float weights[] = {1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9};


for (int j=0; j<HEIGHT; j+=STRIDE) {
  for (int i=0; i<WIDTH; i+=STRIDE) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      for (int ii=0; ii<3; ii++) {
         tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
      output[(j/STRIDE)*WIDTH + (i/STRIDE)] = tmp;
  }
}
```

Inputs



Inputs

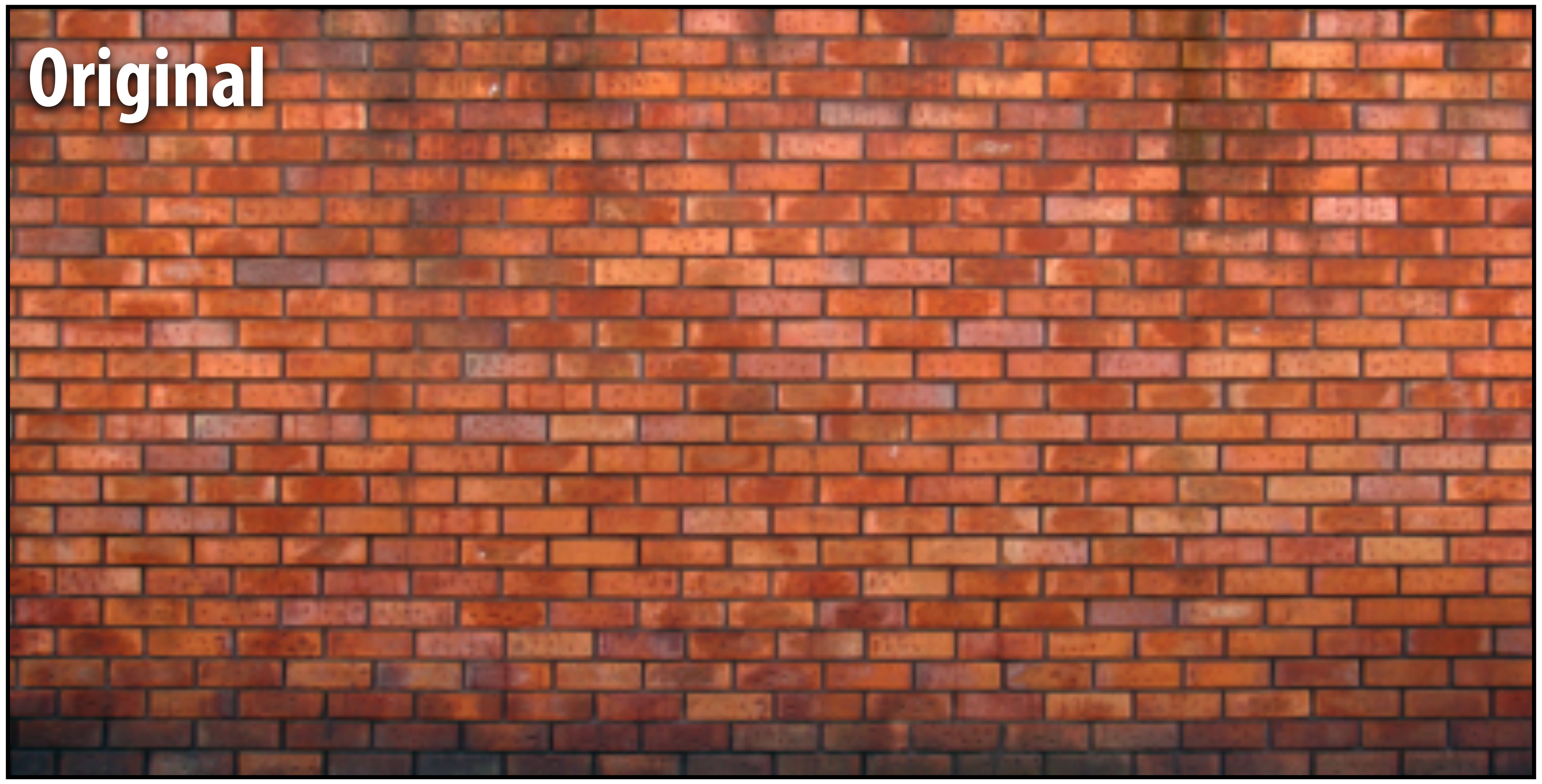


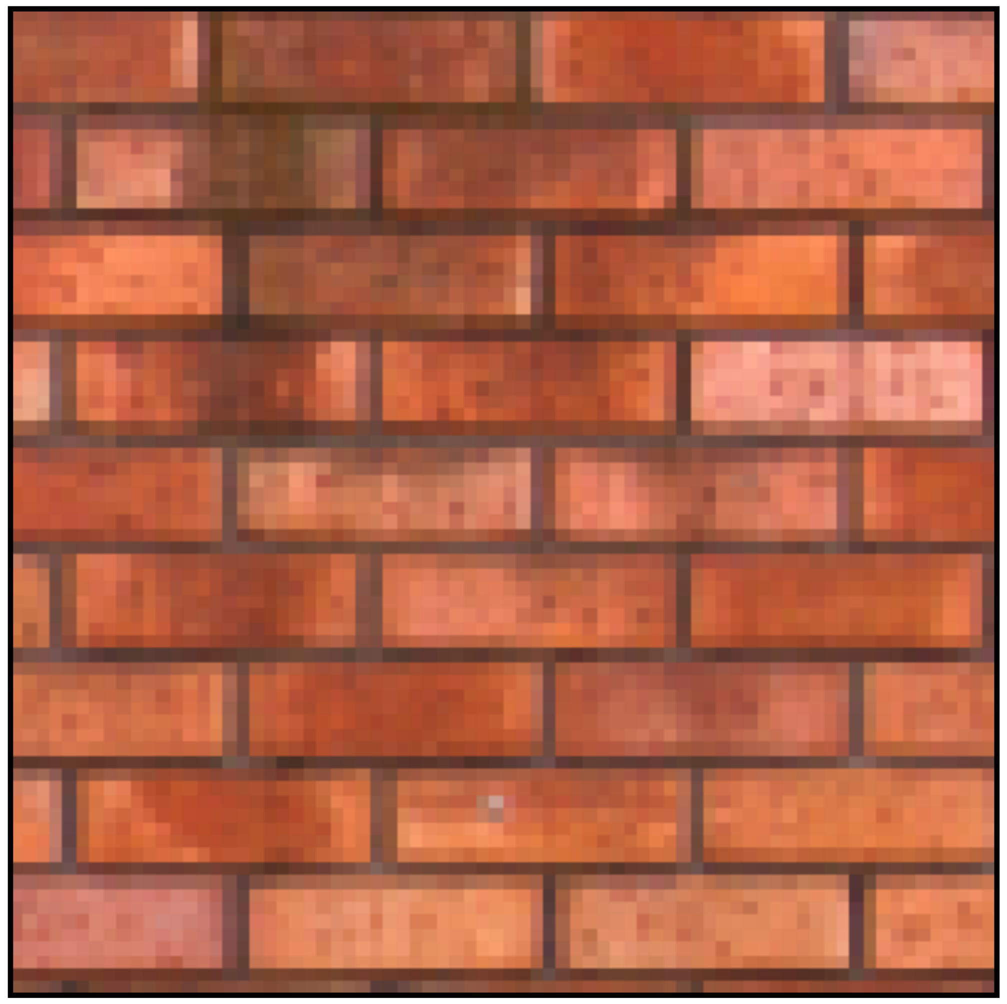
Convolutional layer with stride 2

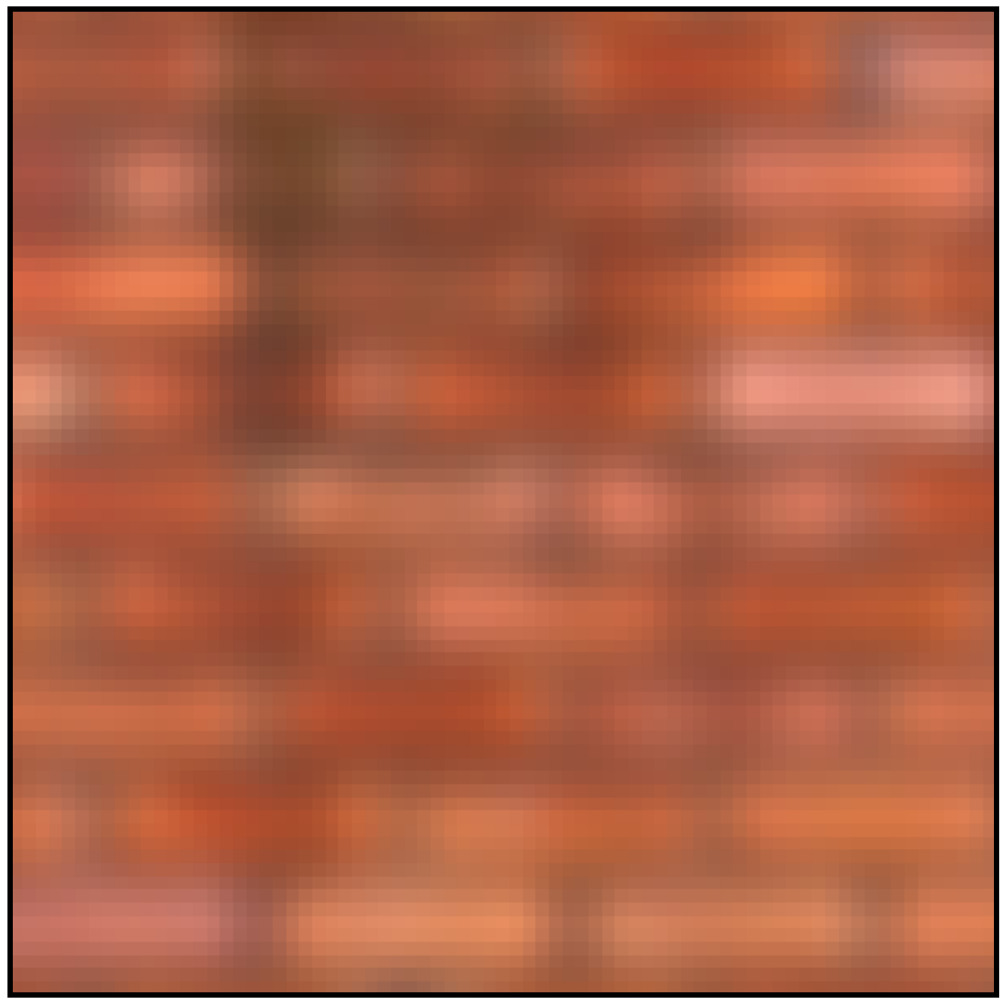
# What does convolution using these filter weights do?

$$\begin{bmatrix} .075 & .124 & .075 \\ .124 & .204 & .124 \\ .075 & .124 & .075 \end{bmatrix}$$

**“Gaussian Blur”**



# What does convolution with these filters do?

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

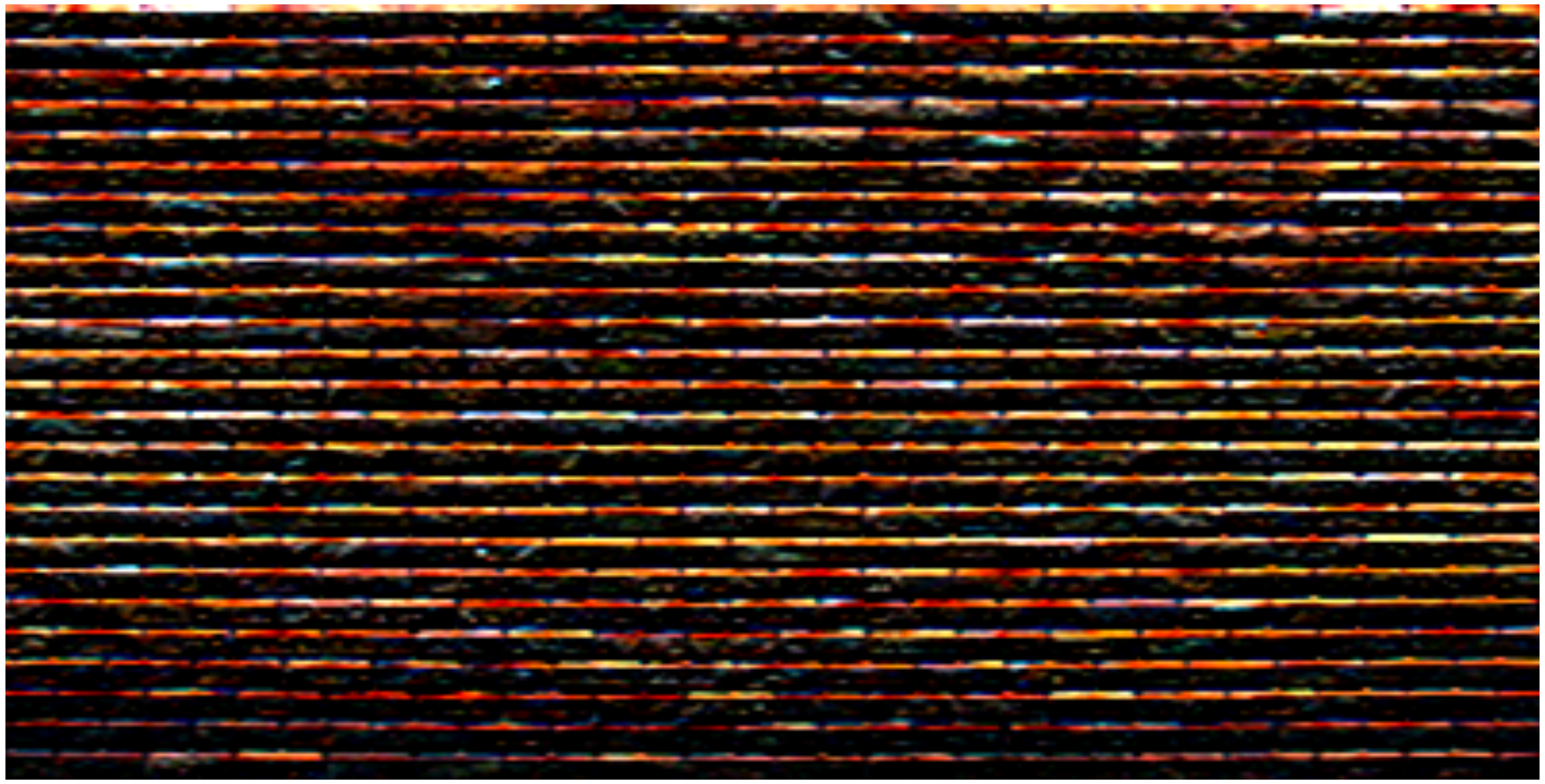
**Extracts horizontal gradients**

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

**Extracts vertical gradients**

# Gradient detection filters

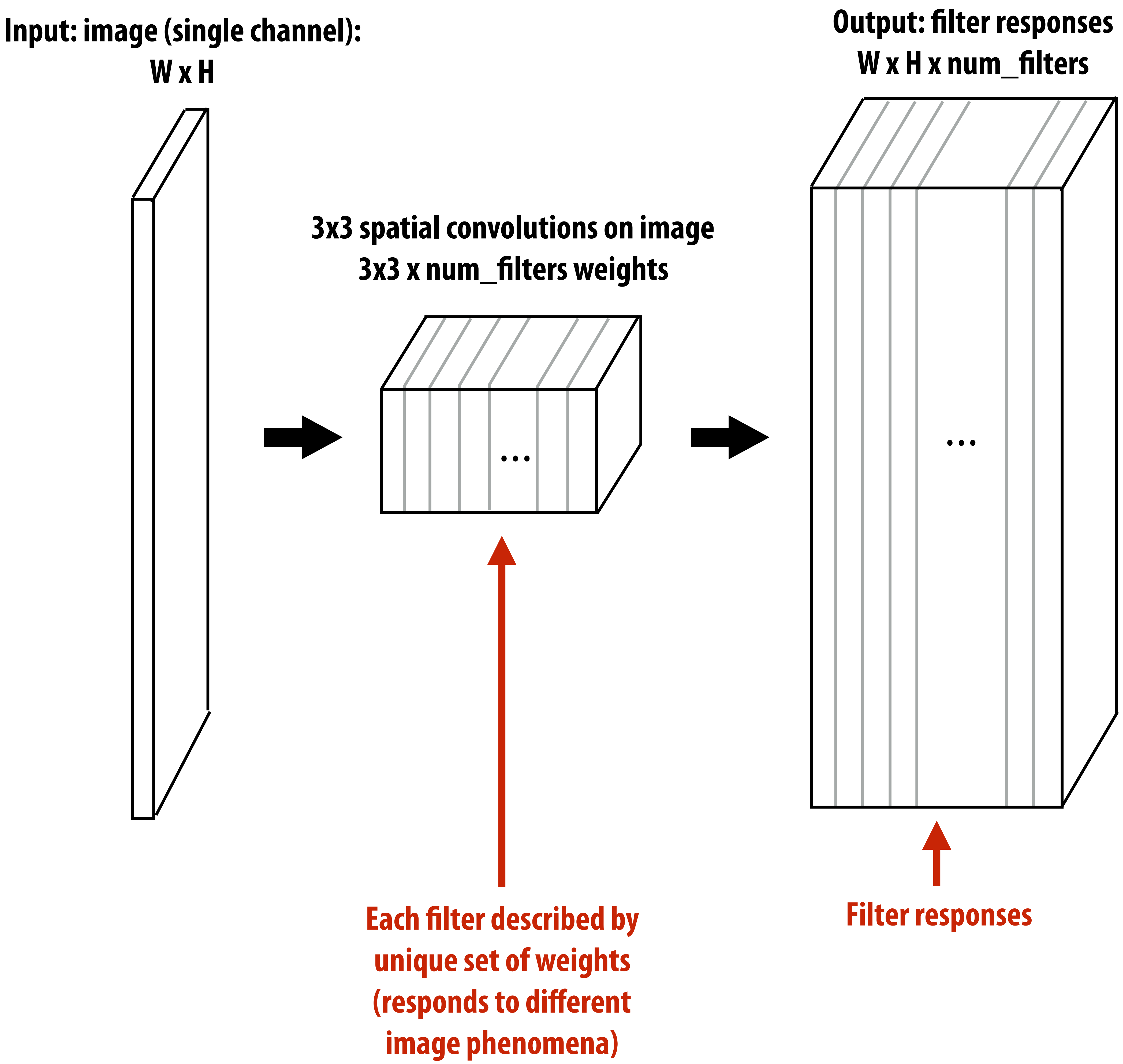
Horizontal gradients



Vertical gradients

Note: you can think of a filter as a "detector" of a pattern, and the magnitude of a pixel in the output image as the "response" of the filter to the region surrounding each pixel in the input image

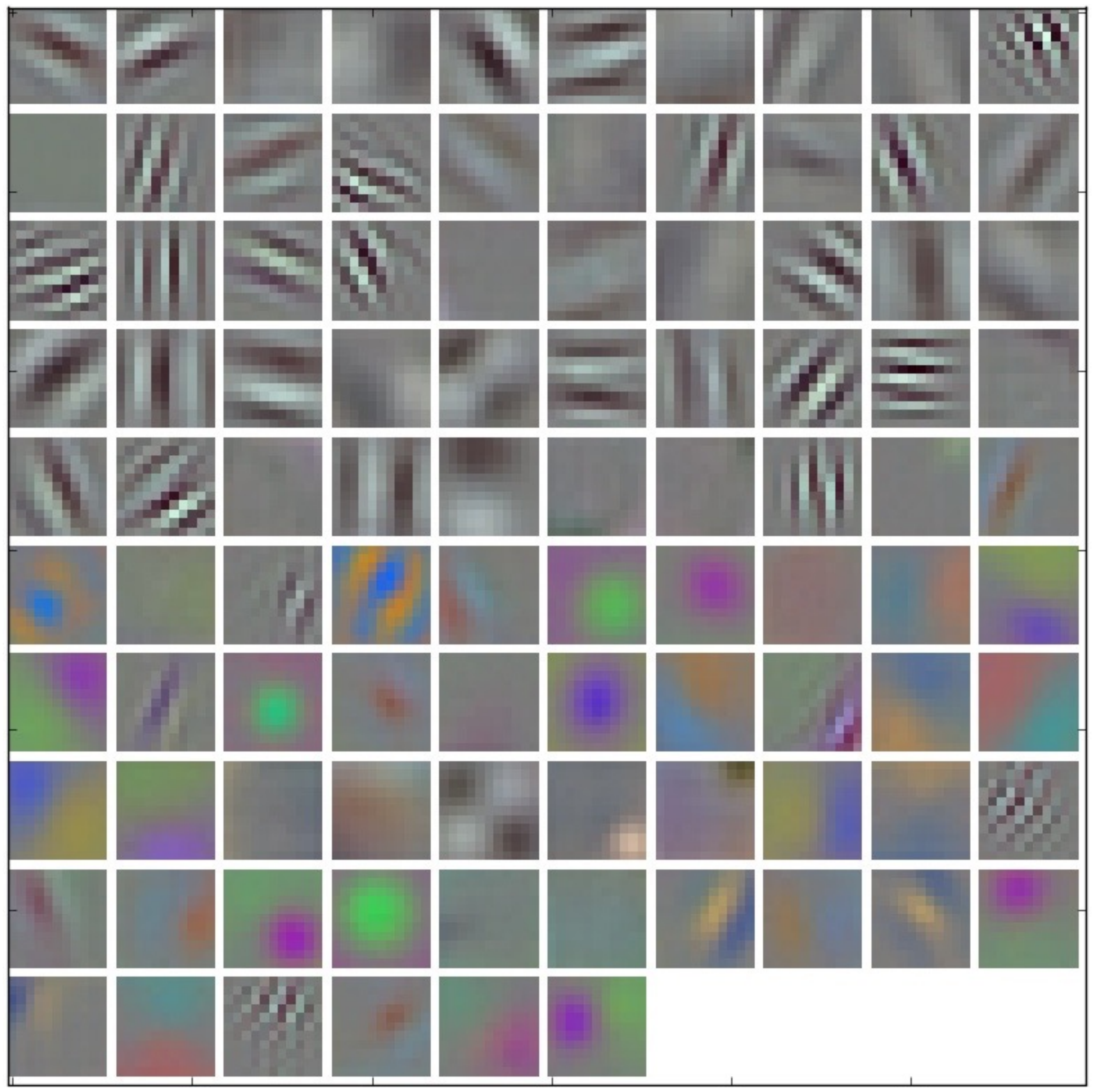
# Applying many filters to an image at once



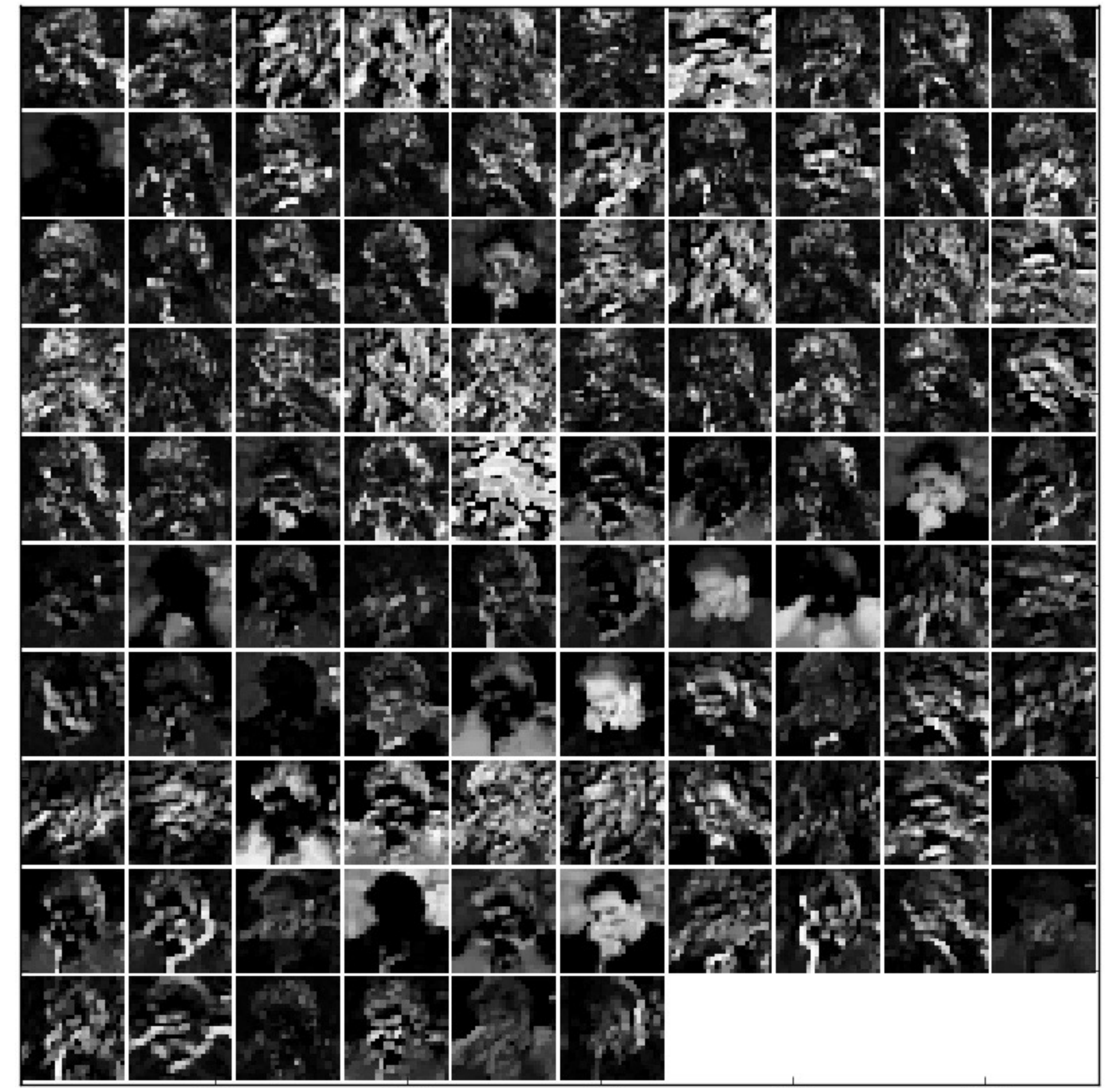
# Applying many filters to an image at once

Input RGB image (W x H x 3)

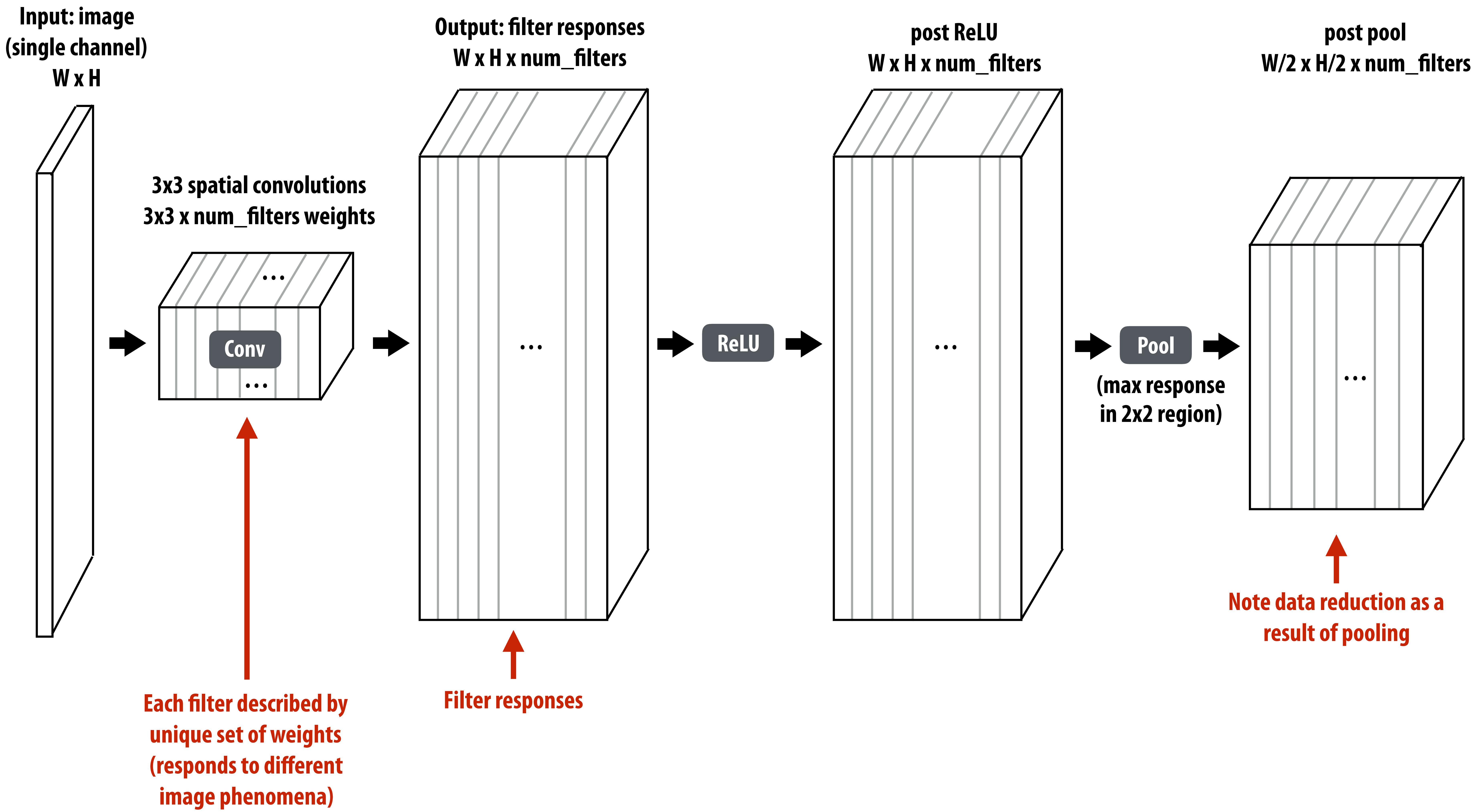
96 11x11x3 filters
(operate on RGB)



96 responses (normalized)



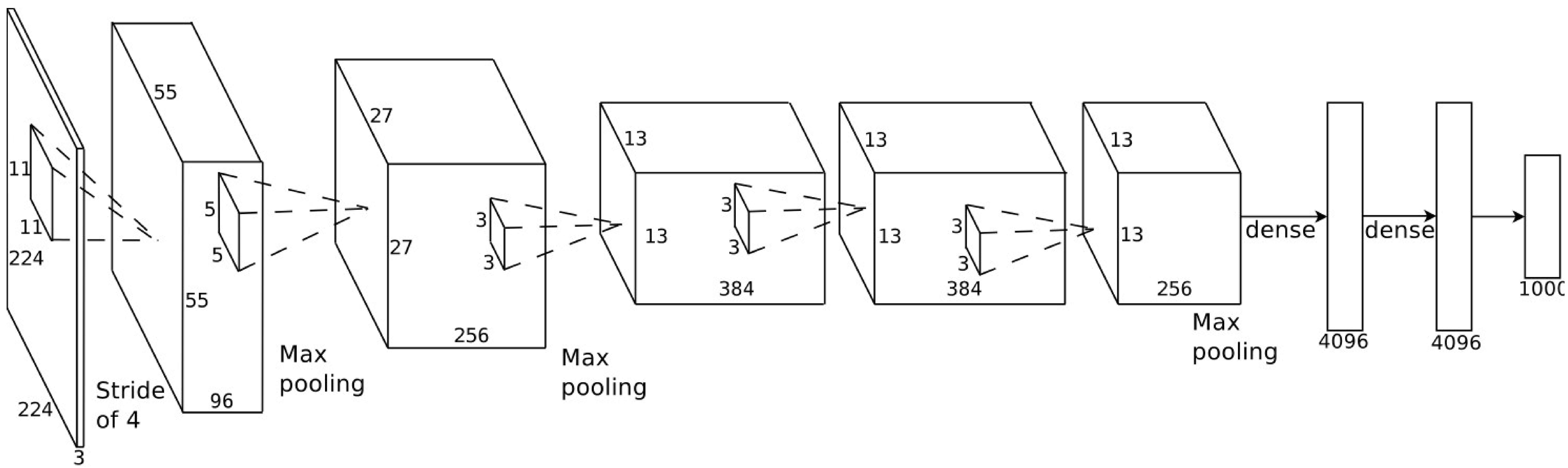
# Adding additional layers



# Modern object detection networks

## Sequences of conv + reLU + (optional) pool layers

### AlexNet [Krizhevsky12]: 5 convolutional layers + 3 fully connected




### VGG-16 [Simonyan15]: 13 convolutional layers

input: 224 x 224 RGB
conv/reLU: 3x3x3x64
conv/reLU: 3x3x64x64
maxpool
conv/reLU: 3x3x64x128
conv/reLU: 3x3x128x128
maxpool
conv/reLU: 3x3x128x256
conv/reLU: 3x3x256x256
conv/reLU: 3x3x256x256
maxpool
conv/reLU: 3x3x256x512
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
maxpool
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
maxpool
fully-connected 4096
fully-connected 4096
fully-connected 1000
soft-max

[VGG illustration credit: Yang et al.]

# Efficiently implementing convolution layers

# Direct implementation of conv layer

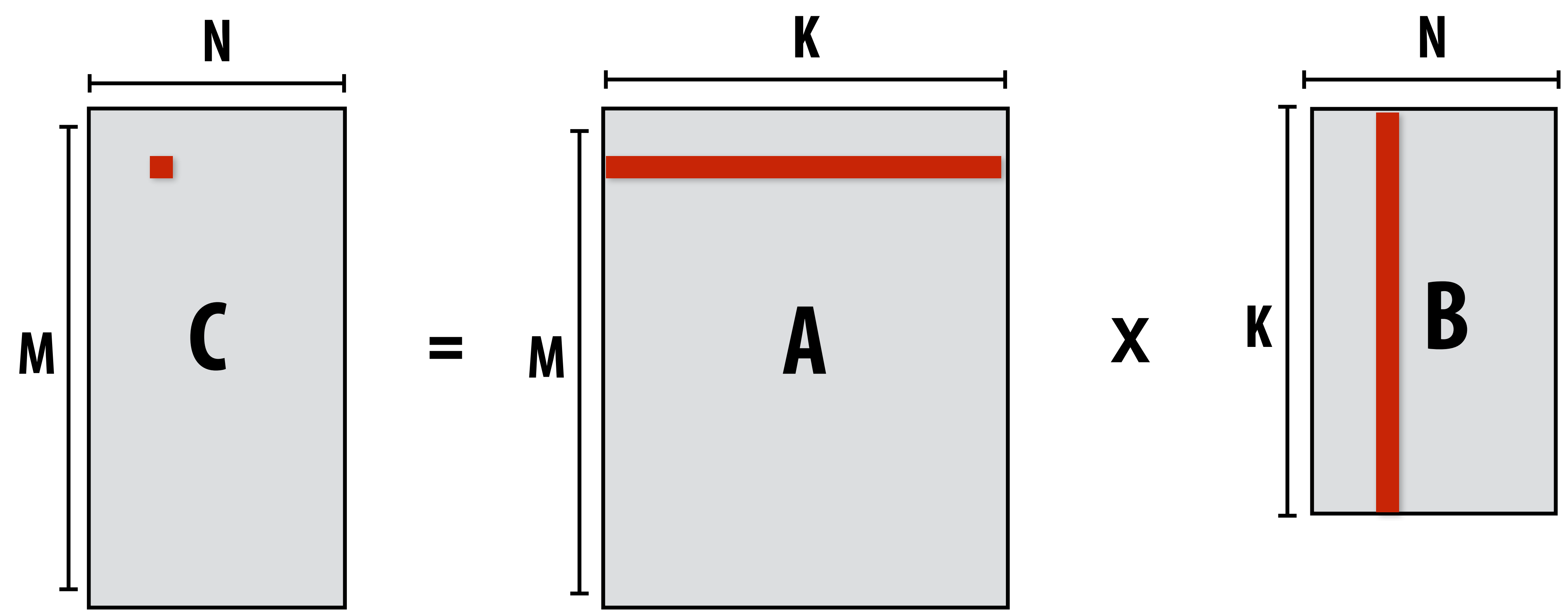
```
float input[INPUT_HEIGHT][INPUT_WIDTH][INPUT_DEPTH];
float output[INPUT_HEIGHT][INPUT_WIDTH][LAYER_NUM_FILTERS];
float layer_weights[LAYER_CONVY, LAYER_CONVX, INPUT_DEPTH];

// assumes convolution stride is 1
for (int img=0; img<IMAGE_BATCH_SIZE; img++)
   for (int j=0; j<INPUT_HEIGHT; j++)
      for (int i=0; i<INPUT_WIDTH; i++)
         for (int f=0; f<LAYER_NUM_FILTERS; f++) {
            output[j][i][f] = 0.f;
            for (int kk=0; kk<INPUT_DEPTH; kk++)  // sum over filter responses of input channels
               for (int jj=0; jj<LAYER_CONVY; jj++) // spatial convolution
                  for (int ii=0; ii<LAYER_CONVX; ii+)  // spatial convolution
                      output[j][i][f] += layer_weights[f][jj][ii][kk] * input[j+jj][i+ii][kk];
         }
```

**Seven loops with significant input data reuse: reuse of filter weights (during convolution), and reuse of input values (across different filters)**

**But must roll your own highly optimized implementation of a complicated loop nest.**

# Dense matrix multiplication

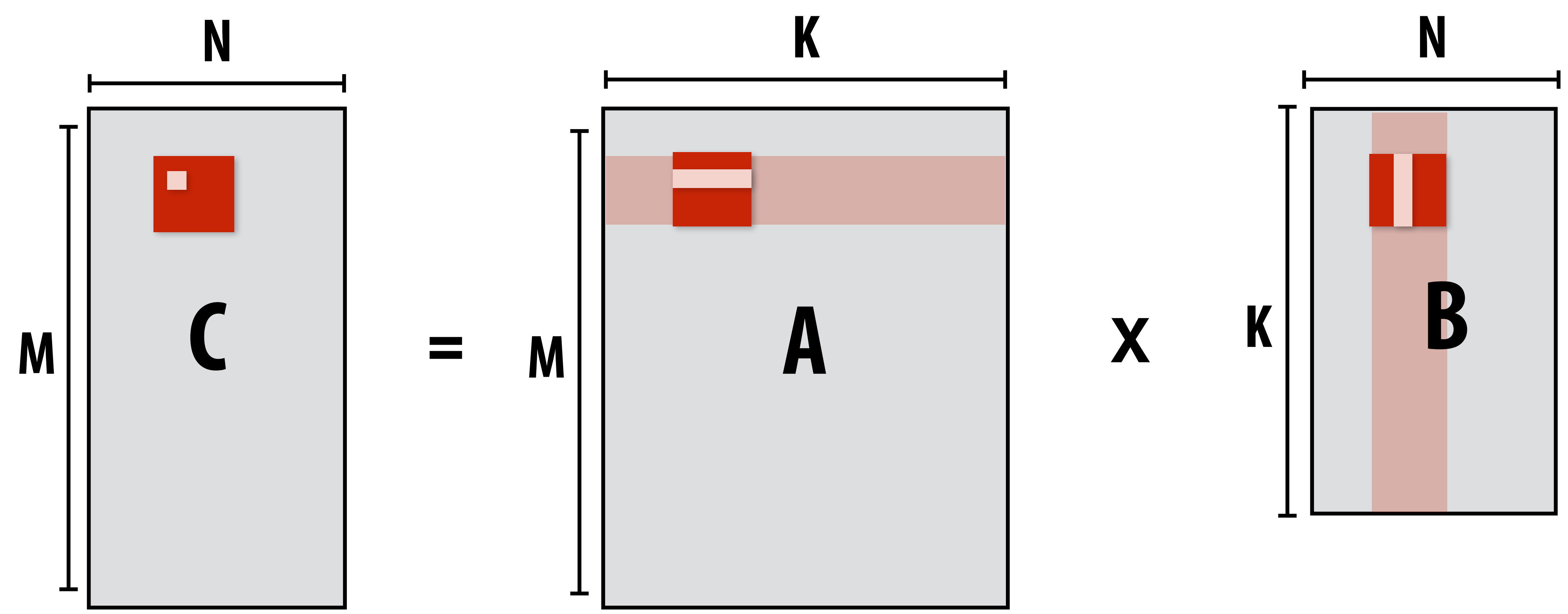
```
float A[M][K];
float B[K][N];
float C[M][N];

// compute C += A * B
#pragma omp parallel for
for (int j=0; j<M; j++)
  for (int i=0; i<N; i++)
     for (int k=0; k<K; k++)
        C[j][i] += A[j][k] * B[k][i];
```




**What is the problem with this implementation?**

**Low arithmetic intensity (does not exploit temporal locality in access to A and B)**

# Blocked dense matrix multiplication



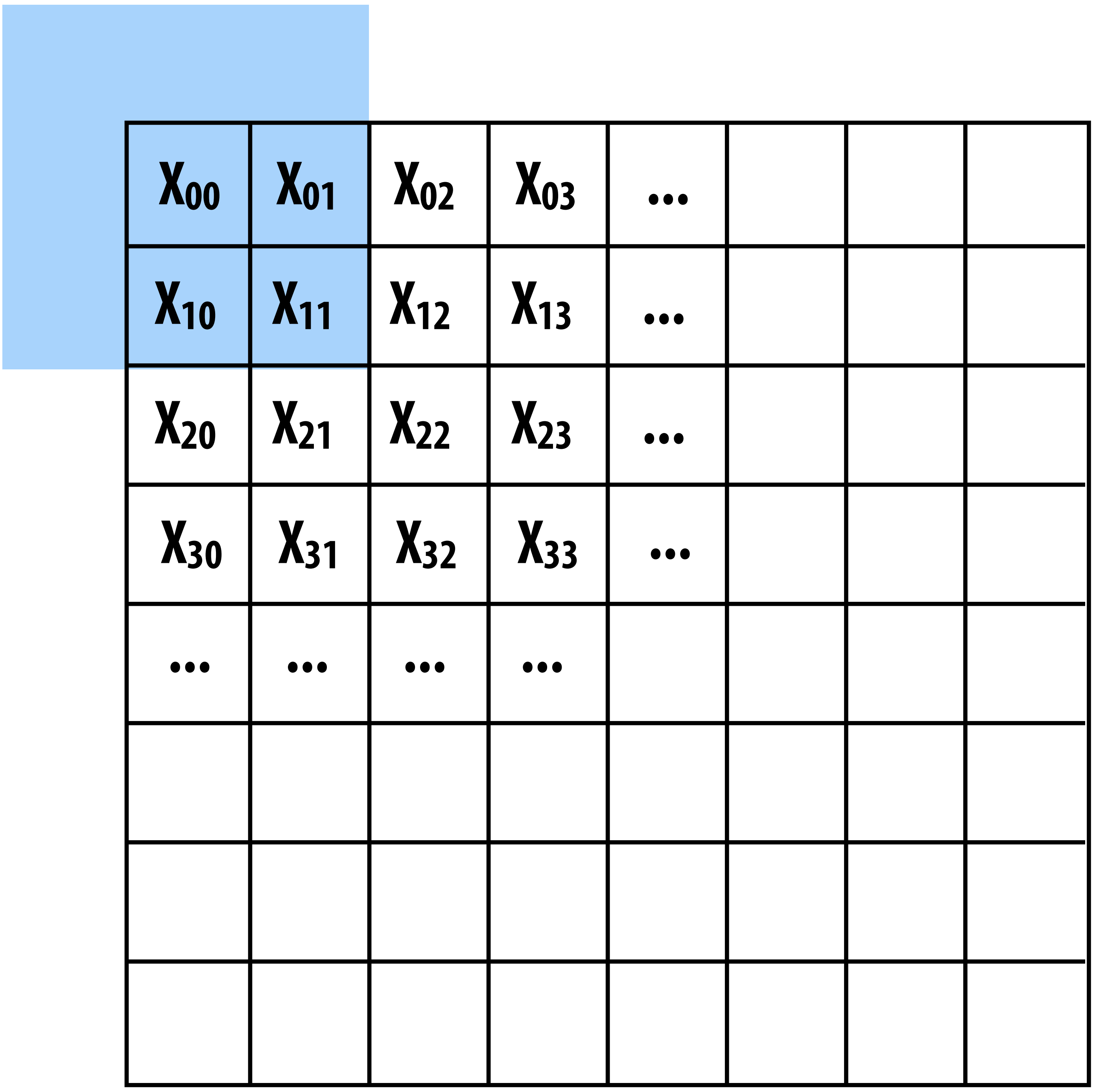

```
float A[M][K];
float B[K][N];
float C[M][N];

// compute C += A * B
#pragma omp parallel for
for (int jblock=0; jblock<M; jblock+=BLOCKSIZE_J)
  for (int iblock=0; iblock<N; iblock+=BLOCKSIZE_I)
     for (int kblock=0; kblock<K; kblock+=BLOCKSIZE_K)
        for (int j=0; j<BLOCKSIZE_J; j++)
           for (int i=0; i<BLOCKSIZE_I; i++)
              for (int k=0; k<BLOCKSIZE_K; k++)
                 C[jblock+j][iblock+i] += A[jblock+j][kblock+k] * B[kblock+k][iblock+i];
```

**Idea: compute partial result for block of C while required blocks of A and B remain in cache**
**(Assumes BLOCKSIZE chosen to allow block of A, B, and C to remain resident)**

**Self check: do you want as big a BLOCKSIZE as possible? Why?**

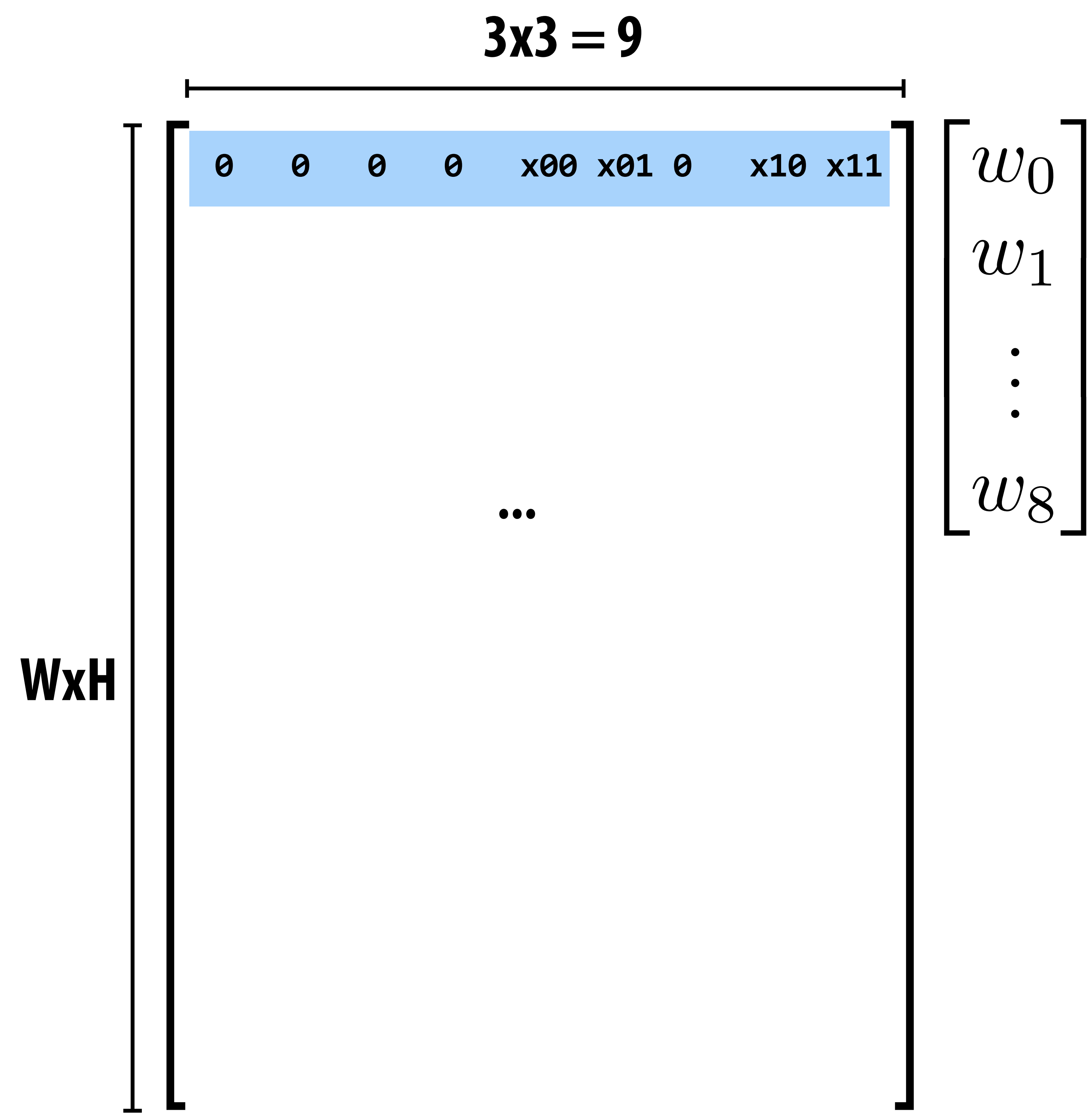
# Convolution as matrix-vector product

## Construct matrix from elements of input image




**O(N) storage overhead for filter with N elements**

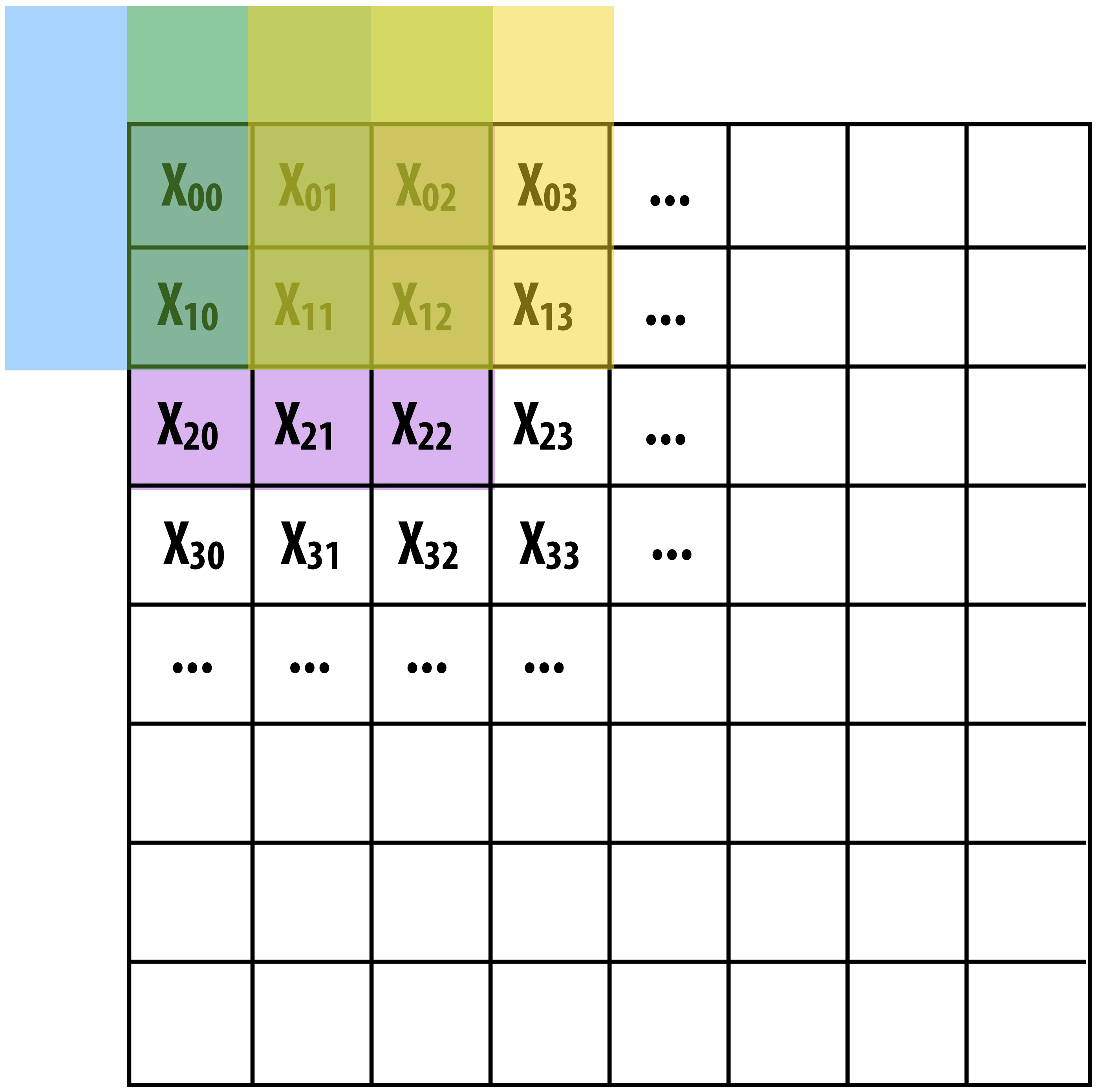
**Must construct input data matrix**




**Note: 0-pad matrix**

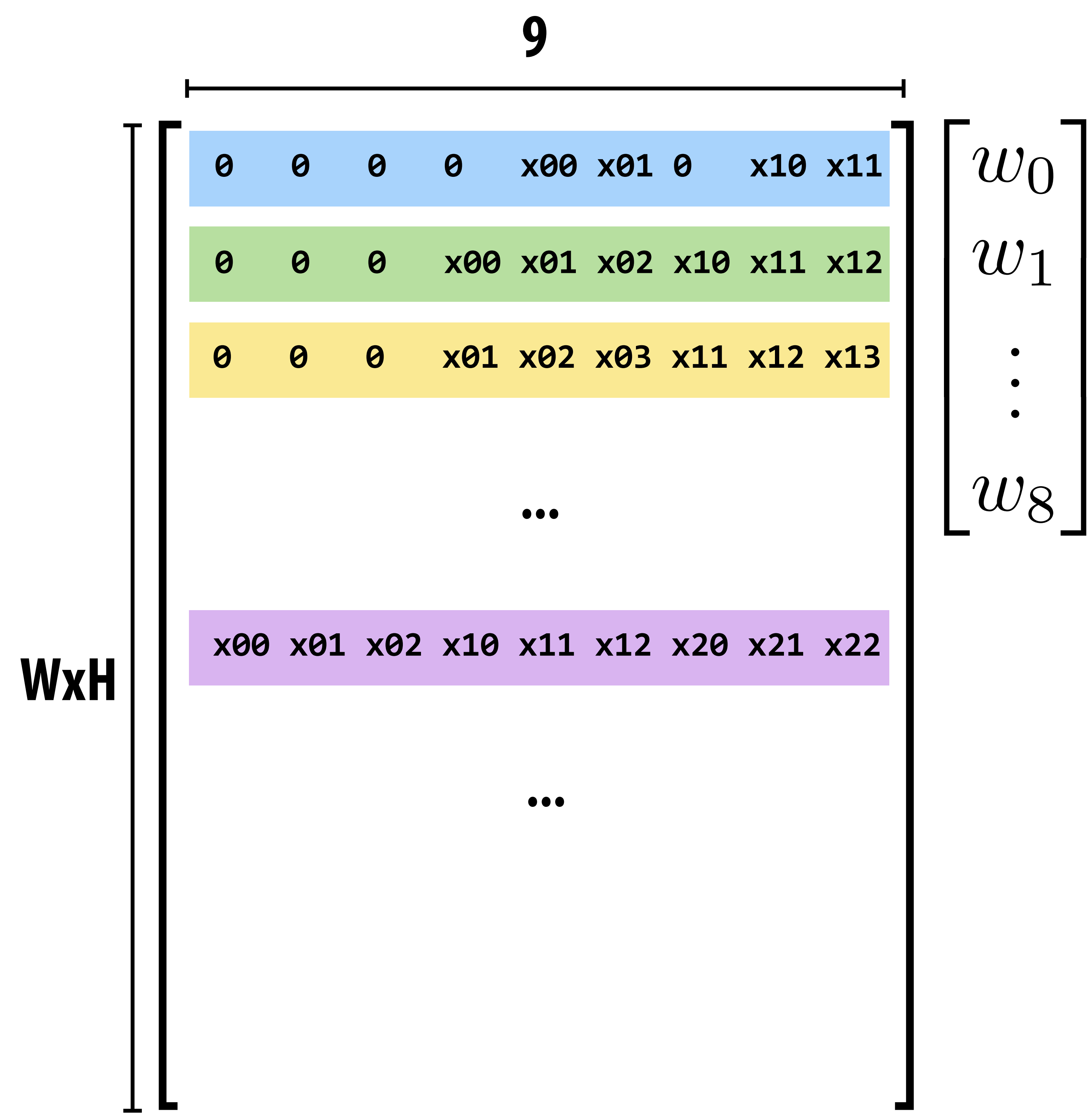
# 3x3 convolution as matrix-vector product

## Construct matrix from elements of input image



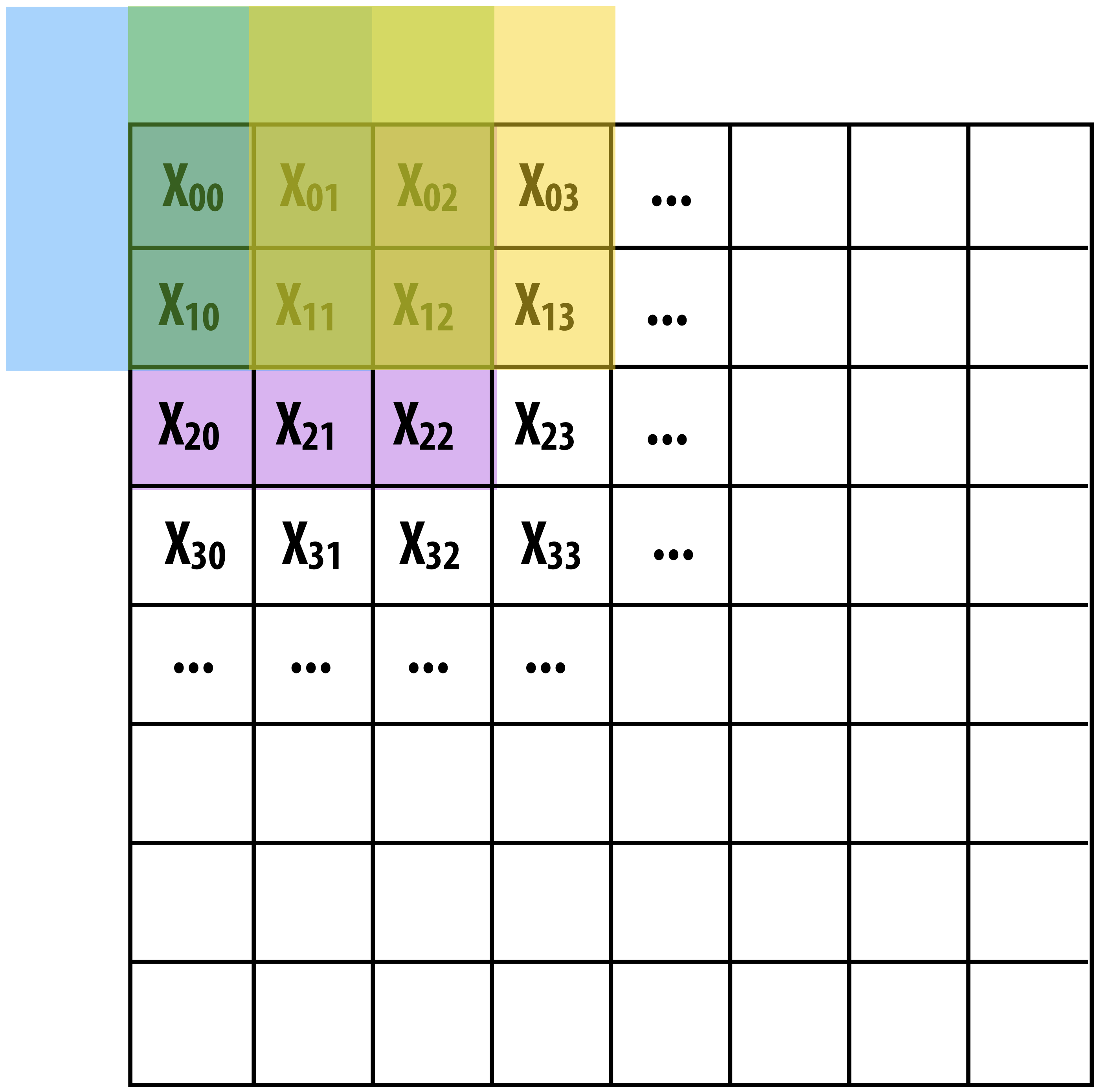

O(N) storage overhead for filter with N elements

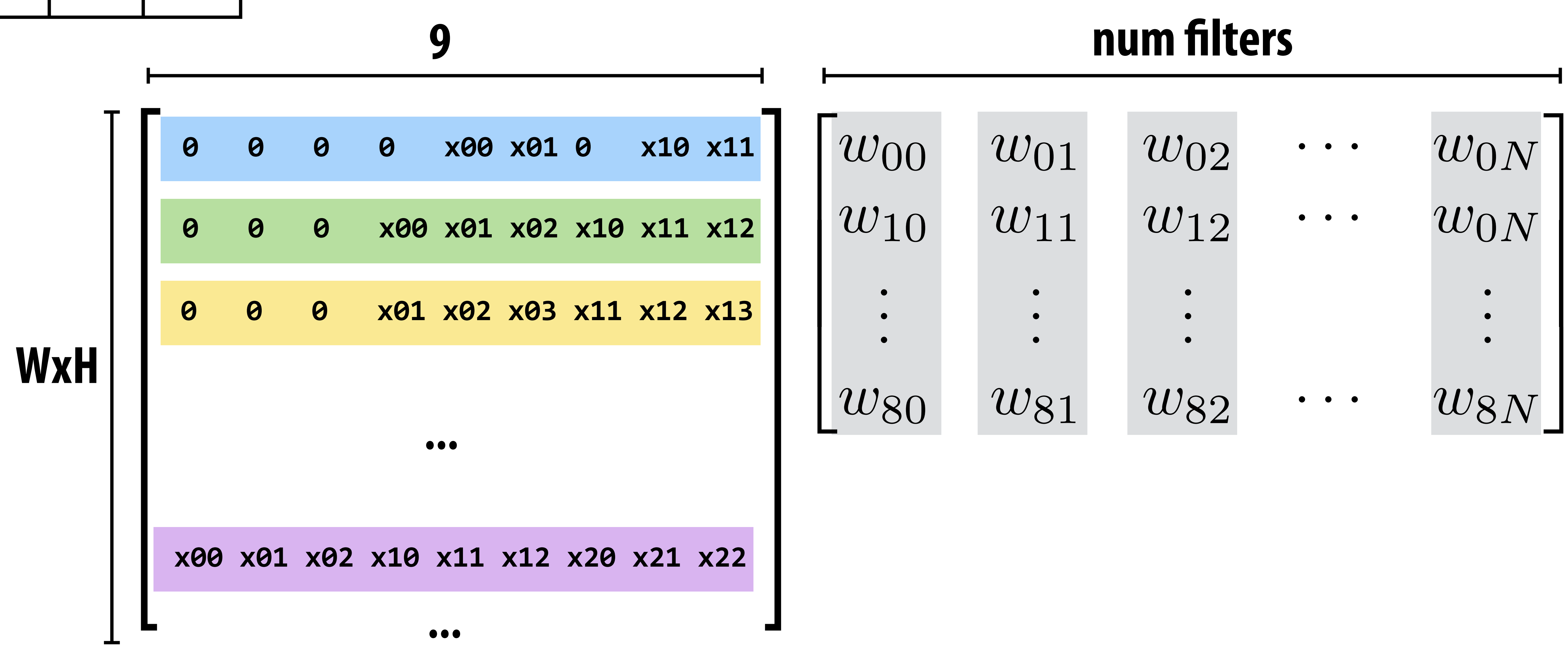
Must construct input data matrix



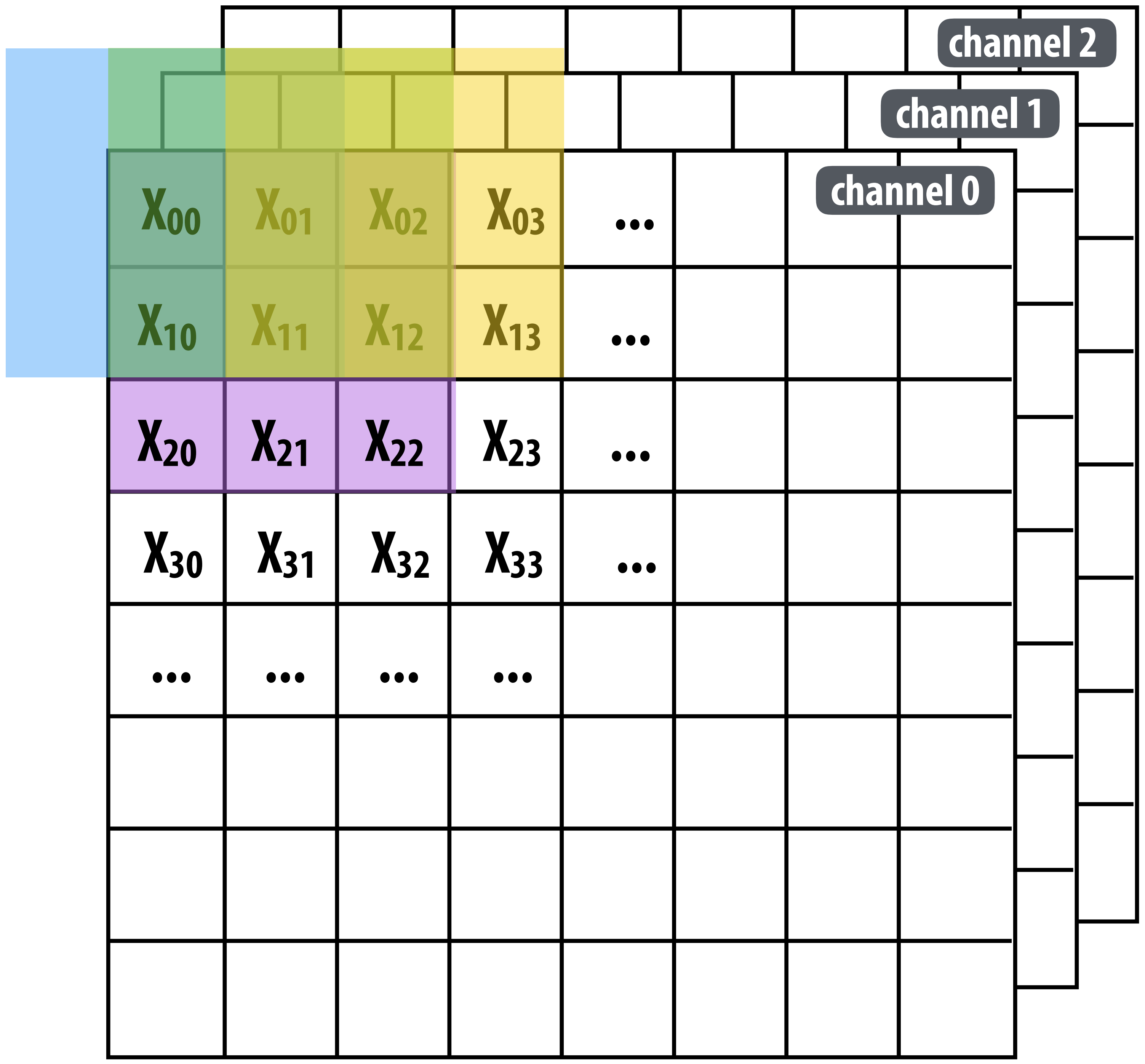

**Note: 0-pad matrix**

# Multiple convolutions as matrix-matrix mult



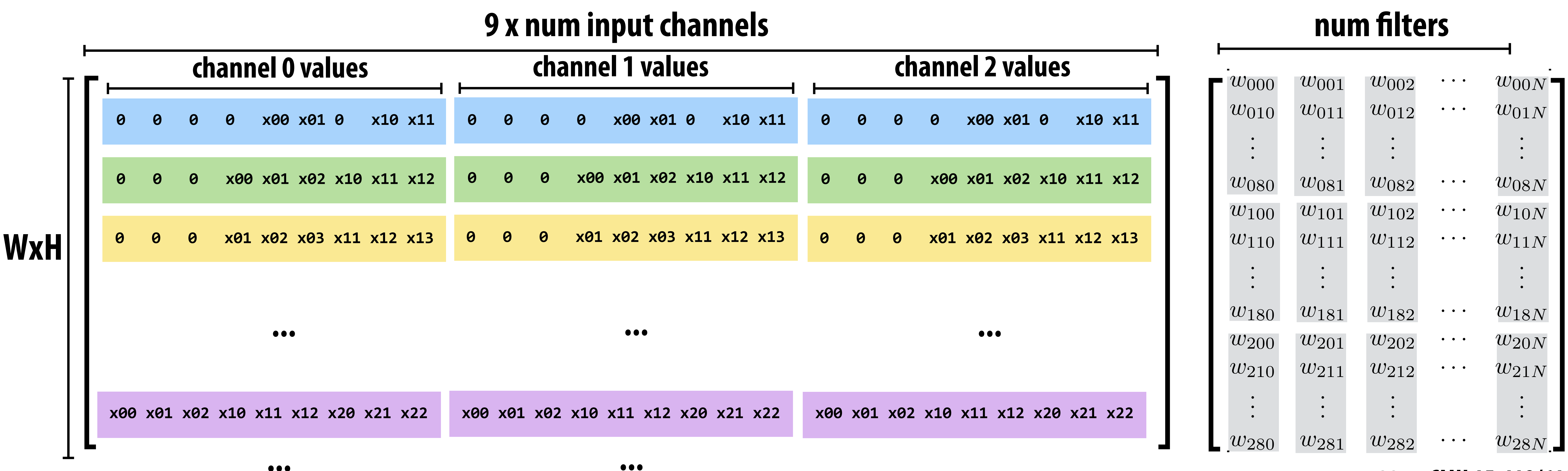



# Multiple convolutions on multiple input channels




For each filter, sum responses over input channels

Equivalent to (3 x 3 x num_channels) convolution on (W x H x num_channels) input data



# VGG memory footprint

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

inputs/outputs get multiplied by image batch size

multiply by next layer's conv window size to form input matrix to next conv layer!!! (for VGG, this is a 9x data amplification)

| | 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 |

# Reducing network footprint

- **Large storage cost for model parameters**
  - AlexNet model: ~200 MB
  - VGG-16 model: ~500 MB
  - This doesn't even account for intermediates during evaluation


- **Footprint: cumbersome to store, download, etc.**
  - 500 MB app downloads make users unhappy!

- **Consider energy cost of 1B parameter network**
  - Running on input stream at 20 Hz
  - 640 pJ per 32-bit DRAM access
  - (20 x 1B x 640pJ) = 12.8W for DRAM access

    (more than power budget of any modern smartphone)

# Compressing a network

**Step 1**: **prune low-weight links** (iteratively retrain network, then prune)

- Over 90% of weights can be removed without significant loss of accuracy
- Store weight matrices in compressed sparse row (CSR) format

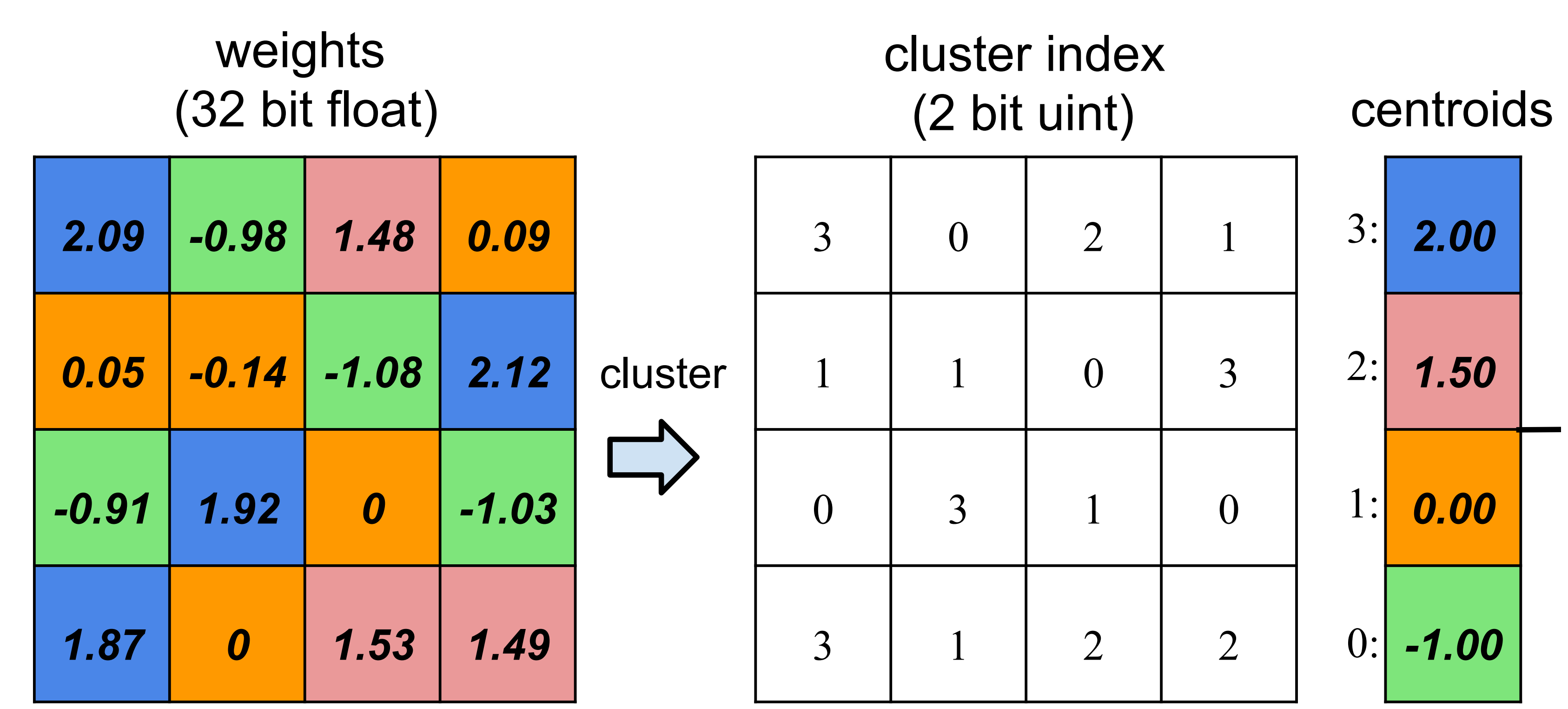
| | | | | |
|---|---|---|---|---|
| Indicies | 1 | 4 | 9 | ... |
| Value | 1.8 | 0.5 | 2.1 | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1.8 | 0 | 0 | 0.5 | 0 | 0 | 0 | 0 | 2.1 | ... |

**Step 2**: **weight sharing**: make surviving connects share a small set of weights

- Cluster weights via k-means clustering (irregular ("learned") quantization)
- Compress weights by only storing cluster index (lg(k) bits)
- Retrain network to improve quality of cluster centroids

**Step 3**: **Huffman encode** quantized weights and CSR indices



# VGG-16 compression

## Substantial savings due to combination of pruning, quantization, Huffman encoding

| Layer | #Weights | Weights% (P) | Weigh bits (P+Q) | Weight bits (P+Q+H) | Index bits (P+Q) | Index bits (P+Q+H) | Compress rate (P+Q) | Compress rate (P+Q+H) |
|---|---|---|---|---|---|---|---|---|
| conv1_1 | 2K | 58% | 8 | 6.8 | 5 | 1.7 | 40.0% | 29.97% |
| conv1_2 | 37K | 22% | 8 | 6.5 | 5 | 2.6 | 9.8% | 6.99% |
| conv2_1 | 74K | 34% | 8 | 5.6 | 5 | 2.4 | 14.3% | 8.91% |
| conv2_2 | 148K | 36% | 8 | 5.9 | 5 | 2.3 | 14.7% | 9.31% |
| conv3_1 | 295K | 53% | 8 | 4.8 | 5 | 1.8 | 21.7% | 11.15% |
| conv3_2 | 590K | 24% | 8 | 4.6 | 5 | 2.9 | 9.7% | 5.67% |
| conv3_3 | 590K | 42% | 8 | 4.6 | 5 | 2.2 | 17.0% | 8.96% |
| conv4_1 | 1M | 32% | 8 | 4.6 | 5 | 2.6 | 13.1% | 7.29% |
| conv4_2 | 2M | 27% | 8 | 4.2 | 5 | 2.9 | 10.9% | 5.93% |
| conv4_3 | 2M | 34% | 8 | 4.4 | 5 | 2.5 | 14.0% | 7.47% |
| conv5_1 | 2M | 35% | 8 | 4.7 | 5 | 2.5 | 14.3% | 8.00% |
| conv5_2 | 2M | 29% | 8 | 4.6 | 5 | 2.7 | 11.7% | 6.52% |
| conv5_3 | 2M | 36% | 8 | 4.6 | 5 | 2.3 | 14.8% | 7.79% |
| fc6 | 103M | 4% | 5 | 3.6 | 5 | 3.5 | 1.6% | 1.10% |
| fc7 | 17M | 4% | 5 | 4 | 5 | 4.3 | 1.5% | 1.25% |
| fc8 | 4M | 23% | 5 | 4 | 5 | 3.4 | 7.1% | 5.24% |
| Total | 138M | 7.5%(13×) | 6.4 | 4.1 | 5 | 3.1 | 3.2% (**31**×) | 2.05% (**49**×) |

**P = connection pruning (prune low weight connections)**

**Q = quantize surviving weights (using shared weights)**

**H = Huffman encode**

**ImageNet Image Classification Performance**

| | Top-1 Error | Top-5 Error | Model size | |
|---|---|---|---|---|
| VGG-16 Ref | 31.50% | 11.32% | 552 MB | |
| VGG-16 Compressed | 31.17% | 10.91% | **11.3 MB** | 49× |

# Deep neural networks on GPUs

- Today, best performing DNN implementations target GPUs
  - High arithmetic intensity computations (computational characteristics similar to dense matrix-matrix multiplication)
  - Benefit from flop-rich architectures
  - Highly-optimized library of kernels exist for GPUs (cuDNN)
    - Most CPU-based implementations use basic matrix-multiplication-based formulation (good implementations could run faster!)



Facebook's Big Sur

# Summary: Efficiently Evaluating DNNs

- **Computational structure**
  - Convlayers: high arithmetic intensity, significant portion of cost of evaluating a network
  - Similar data access patterns to dense-matrix multiplication (exploiting temporal reuse is key)
  - But straight reduction to matrix-matrix multiplication is often sub-optimal
  - Work-efficient techniques for convolutional layers (FFT-based, Wingrad convolutions)

- **Large numbers of parameters**: significant interest in reducing size of networks for both training and evaluation
  - Pruning: remove least important network links
  - Quantization: low-precision parameter representations often suffice

- Many ongoing studies of specialized hardware architectures for efficient evaluation
  - Future CPUs/GPUs, ASICs, FPGS, …
  - Specialization will be important to achieving "always on" applications

# Two Distinct Issues with Deep Networks

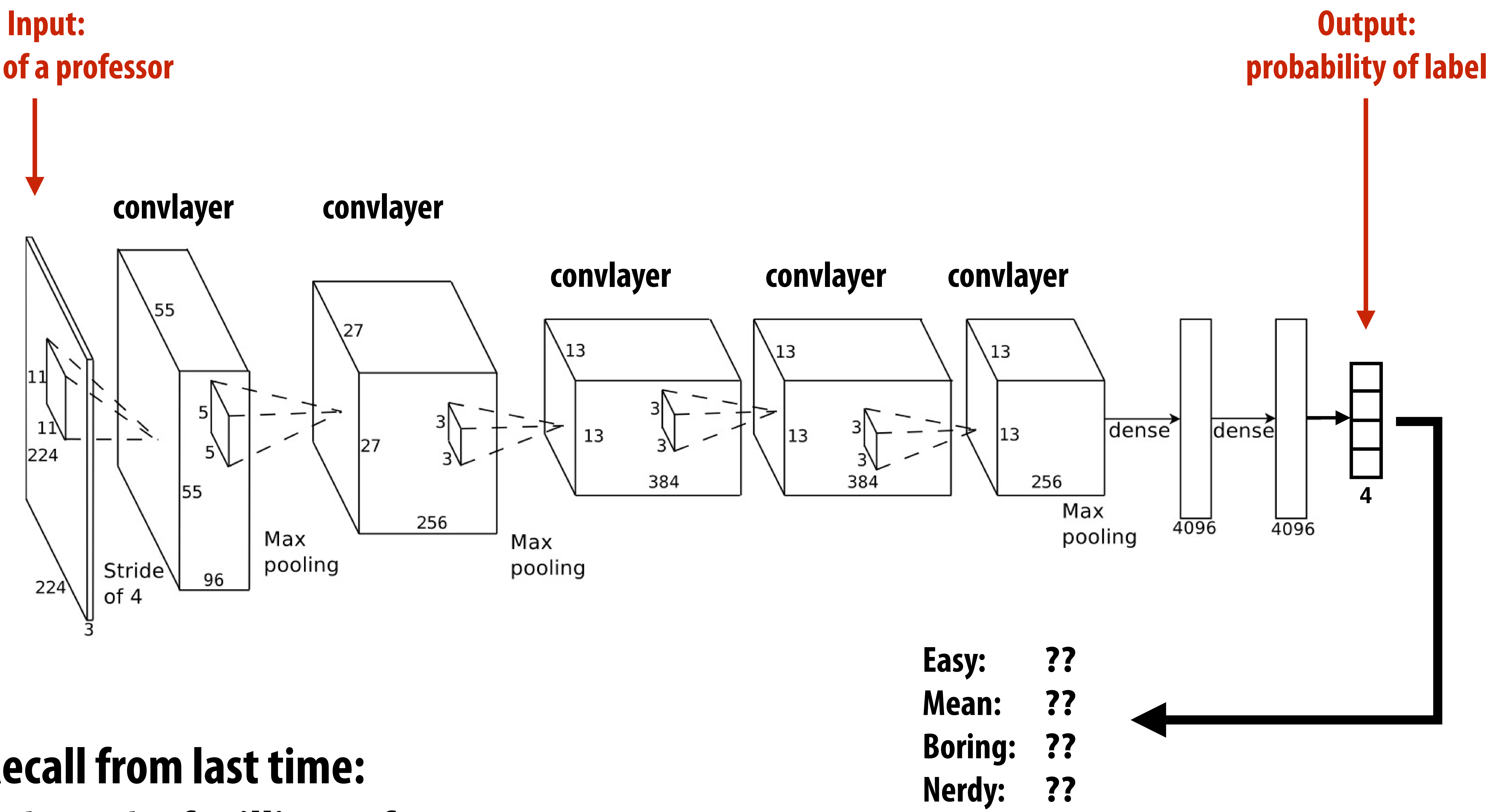
- **Evaluation**
  - often takes milliseconds
- **Training**
  - often takes hours, days, weeks

# "Training a network"

- **Training a network is the process of learning the value of network parameters so that output of the network provides the desired result for a task**
  - [Krizhevsky12] task = object classification
    - input 224 x 224 x 3 RGB image
    - output probability of 1000 ImageNet object classes: "dog", "cat", etc…
    - ~ 60M weights

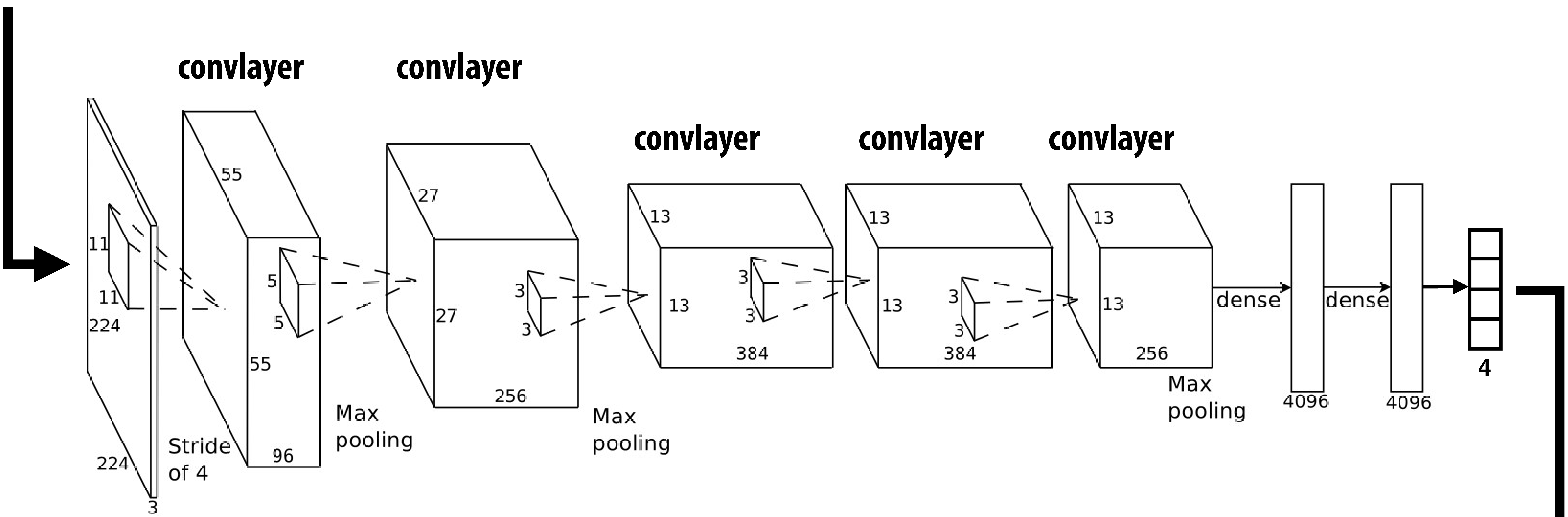
# Professor classification network

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




**Recall from last time:**
**10's-100's of millions of parameters**

# Professor classification network




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

# Where did the parameters come from?

# Training data (ground truth answers)


[label omitted]


[label omitted]


[label omitted]


Nerdy


[label omitted]


[label omitted]


[label omitted]

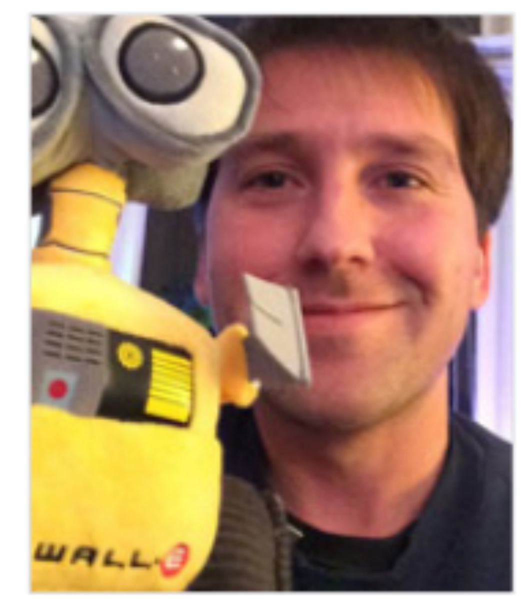

[label omitted]


[label omitted]


Nerdy


[label omitted]


[label omitted]


Nerdy


[label omitted]


[label omitted]


[label omitted]


Nerdy

[label omitted]

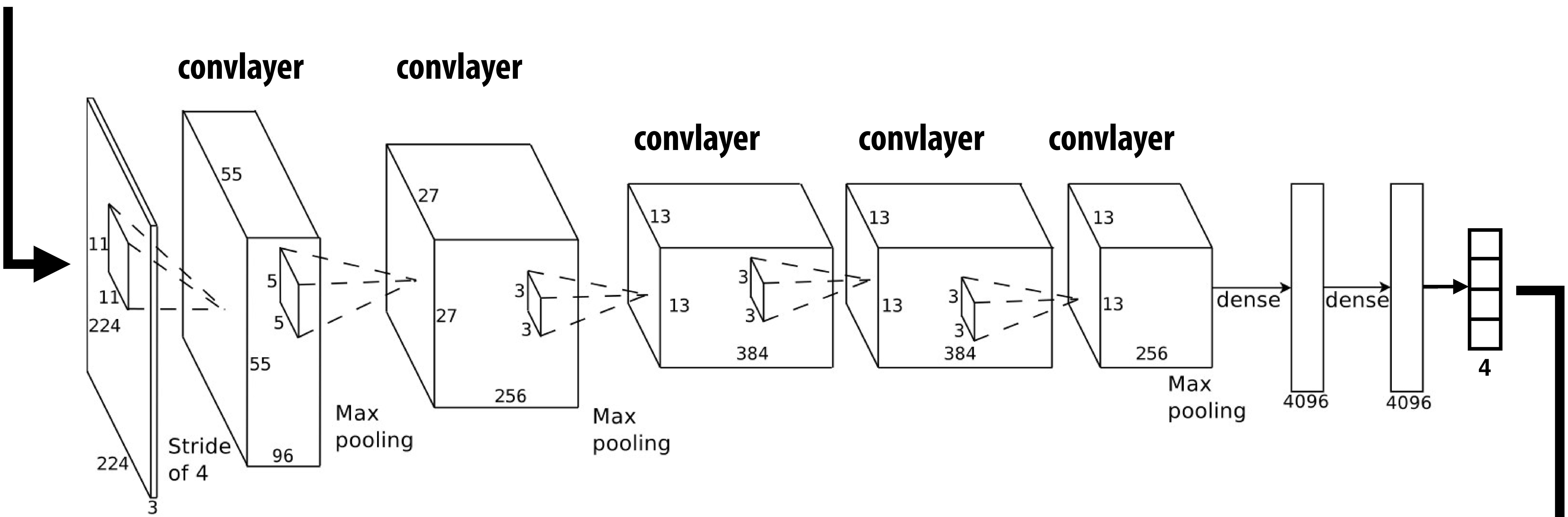

[label omitted]


[label omitted]


Nerdy

# Professor classification network

New image of Kayvon
(not in training set)




| | | | | |
|---|---|---|---|---|
| Easy: | 0.0 | | Easy: | 0.26 |
| Mean: | 0.0 | | Mean: | 0.08 |
| Boring: | 0.0 | | Boring: | 0.14 |
| Nerdy: | 1.0 | | Nerdy: | 0.52 |

Ground truth
(what the answer should be)

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:**

$$L = -log\left(\frac{e^{f_c}}{\sum_j e^{f_j}}\right)$$

Output of network for correct category ($e^{f_c}$)

Output of network for all categories ($\sum_j e^{f_j}$)

* 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

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

**<u>Idea</u>: 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 $x_i$)**

**<u>Intuition</u>: 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 a 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.**

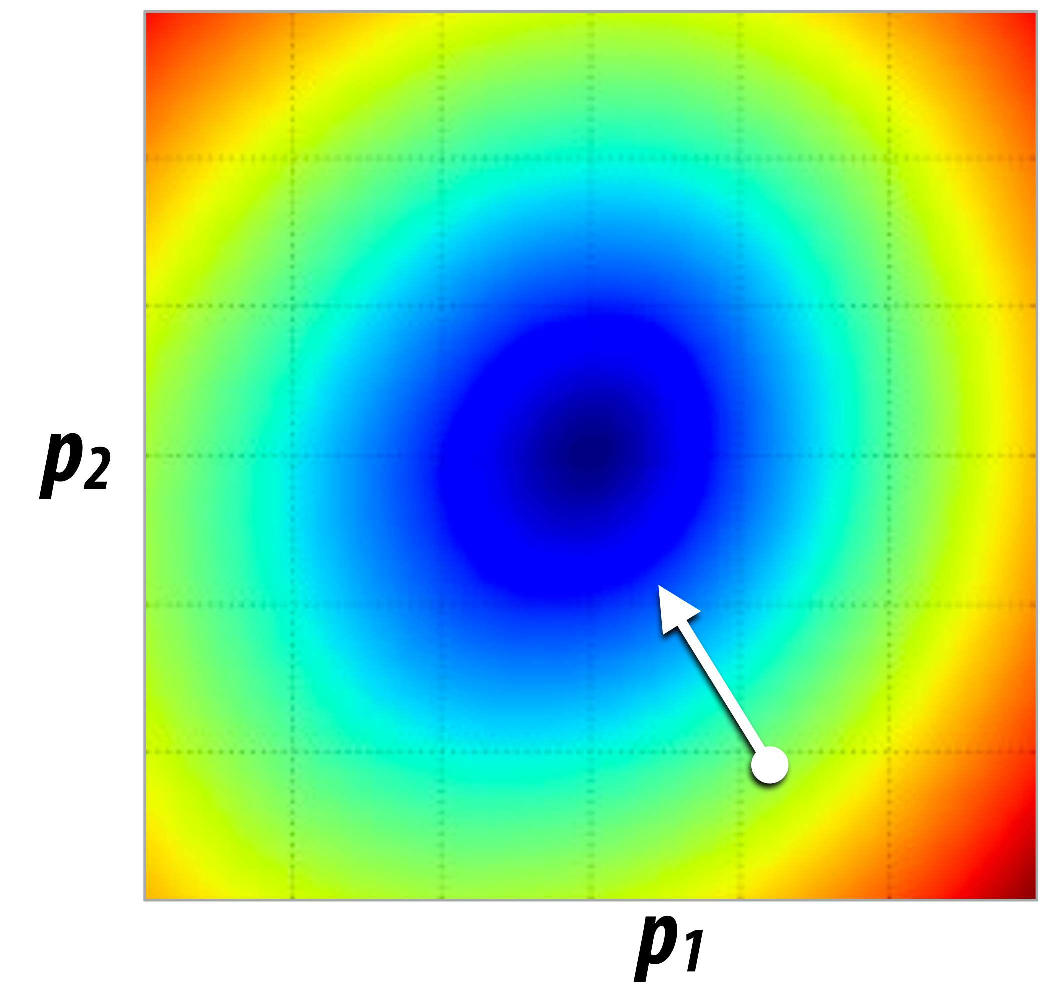
**But for the current values of $p_1$ *and* $p_2$, it currently outputs 10.**

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

**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 item x_i in training set:
      grad += evaluate_loss_gradient(f, loss_func, params, x_i)

   params += -grad * step_size;
```

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

**How to compute df/dp for a complex neural network with millions of parameters?**

# Derivatives using the chain rule

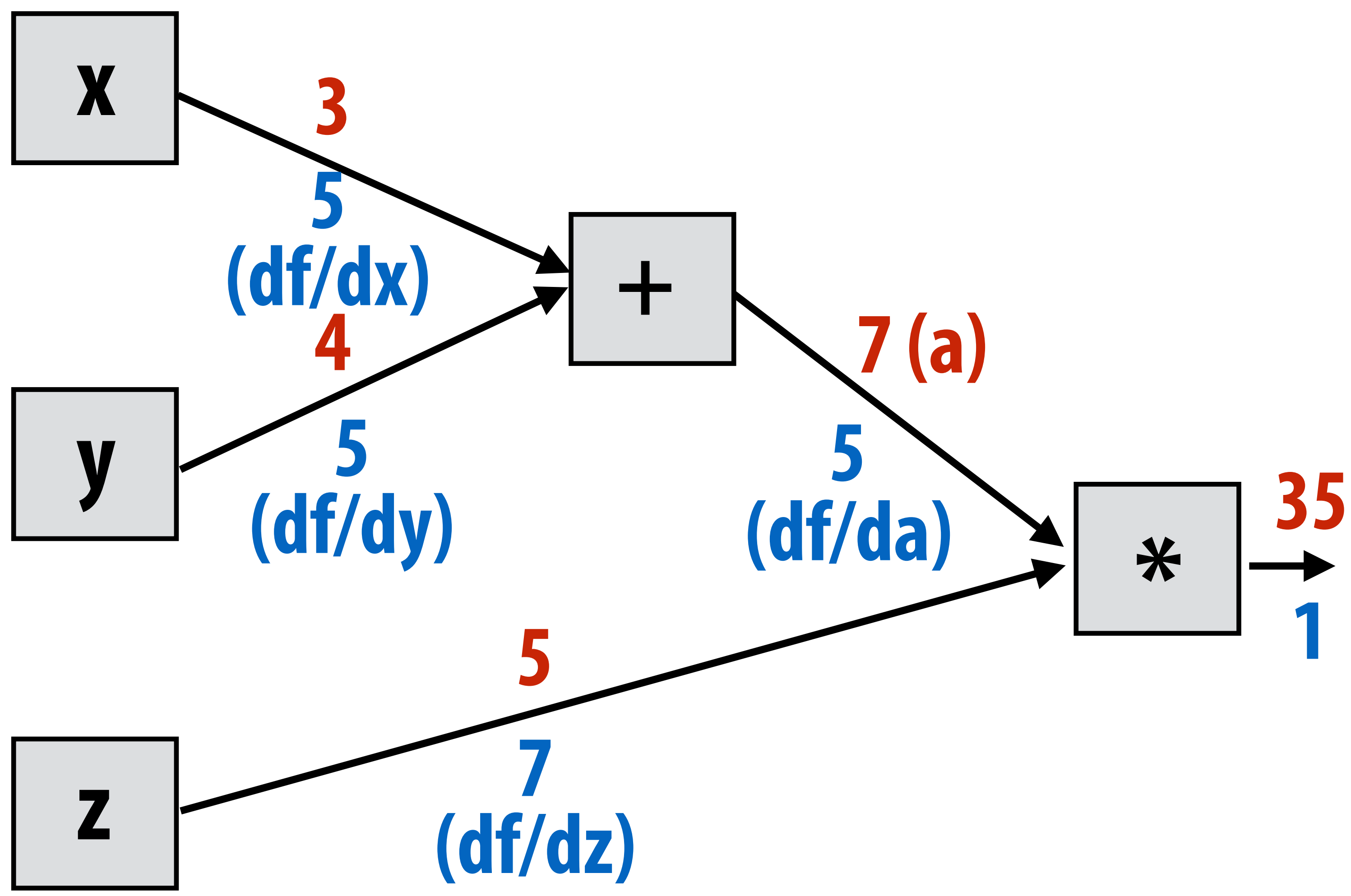
$$f(x, y, z) = (x + y)z = az$$

**Where:** $a = x + y$

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

**So, by the derivative chain rule:**

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



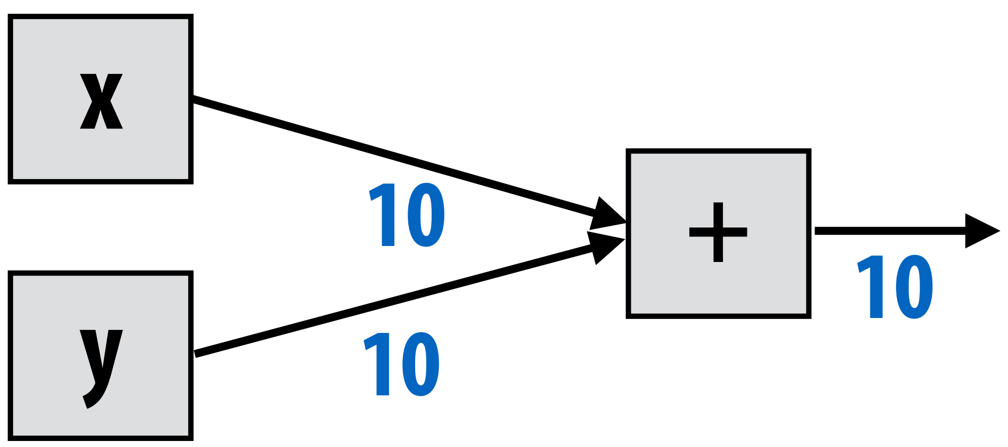

Red = output of node
Blue = df/dnode

# Backpropagation

Red = output of node

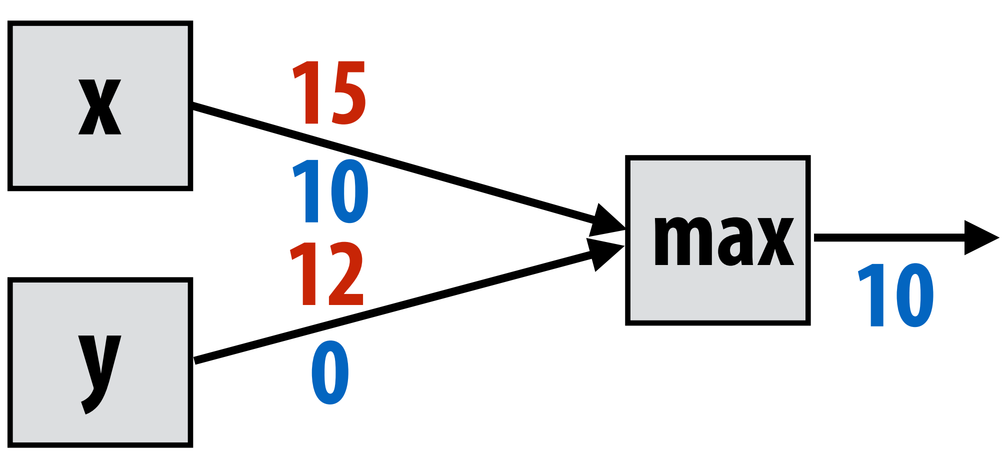
Blue = df/dnode

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



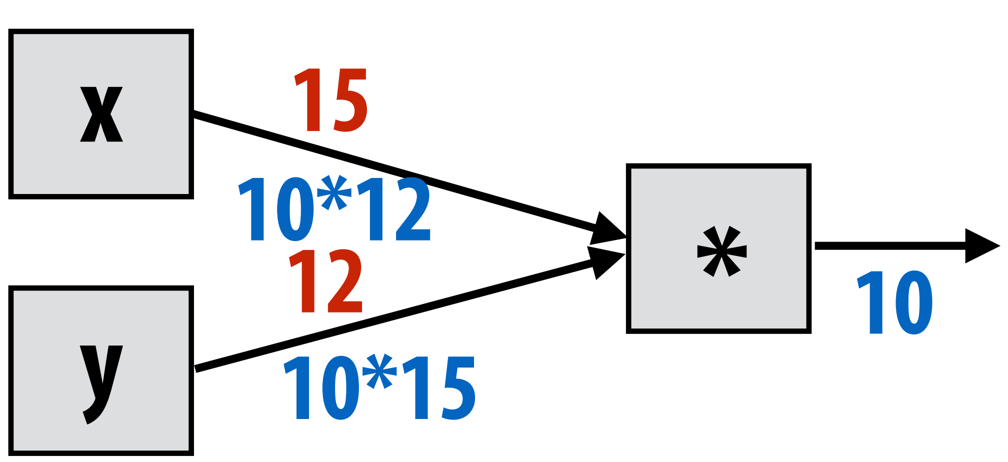

$$g(x,y) = x + y$$

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




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

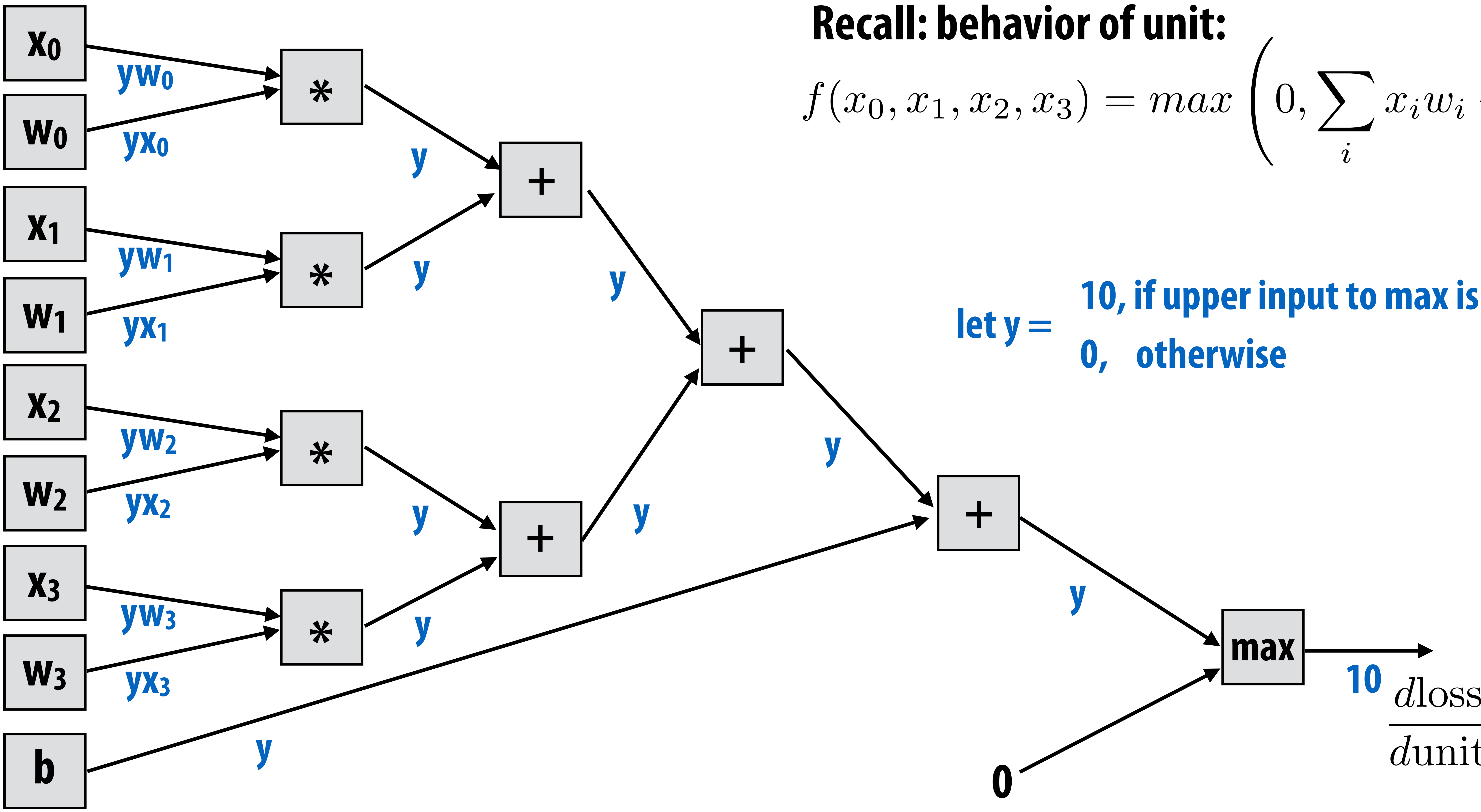
$\frac{dg}{dx} =$ **1, if x > y**
**0, otherwise**




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

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

# Backpropagating through single unit



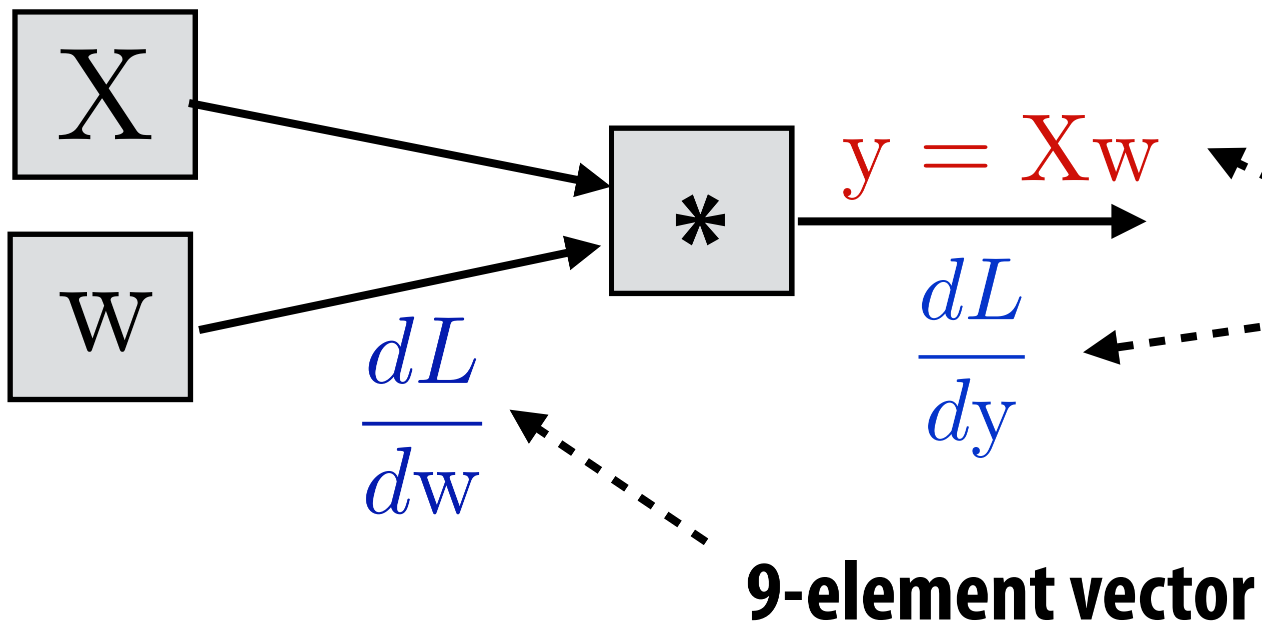

**Recall: behavior of unit:**

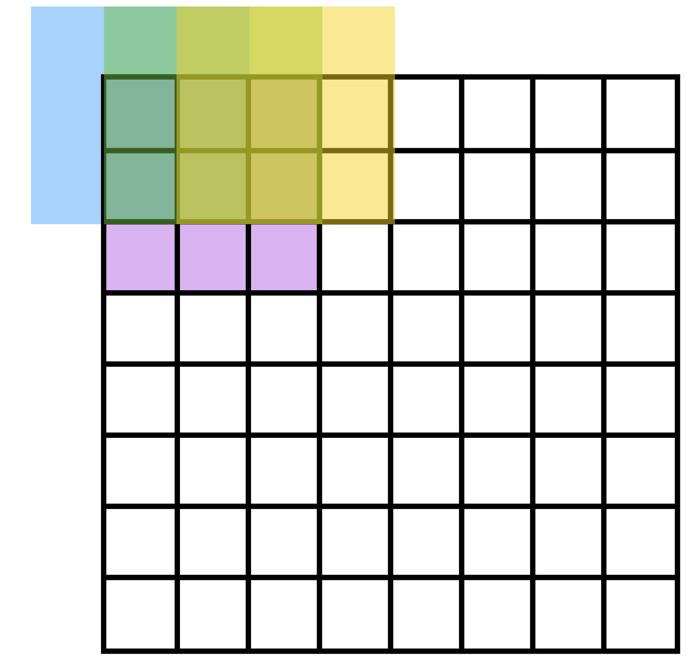
$$f(x_0, x_1, x_2, x_3) = max\left(0, \sum_i x_i w_i + b\right)$$

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

**Observe: output of prior layer ($x_i$'s) and output of this unit must be retained in order to compute weight gradients for this unit during backprop.**

# Backpropagation: matrix form

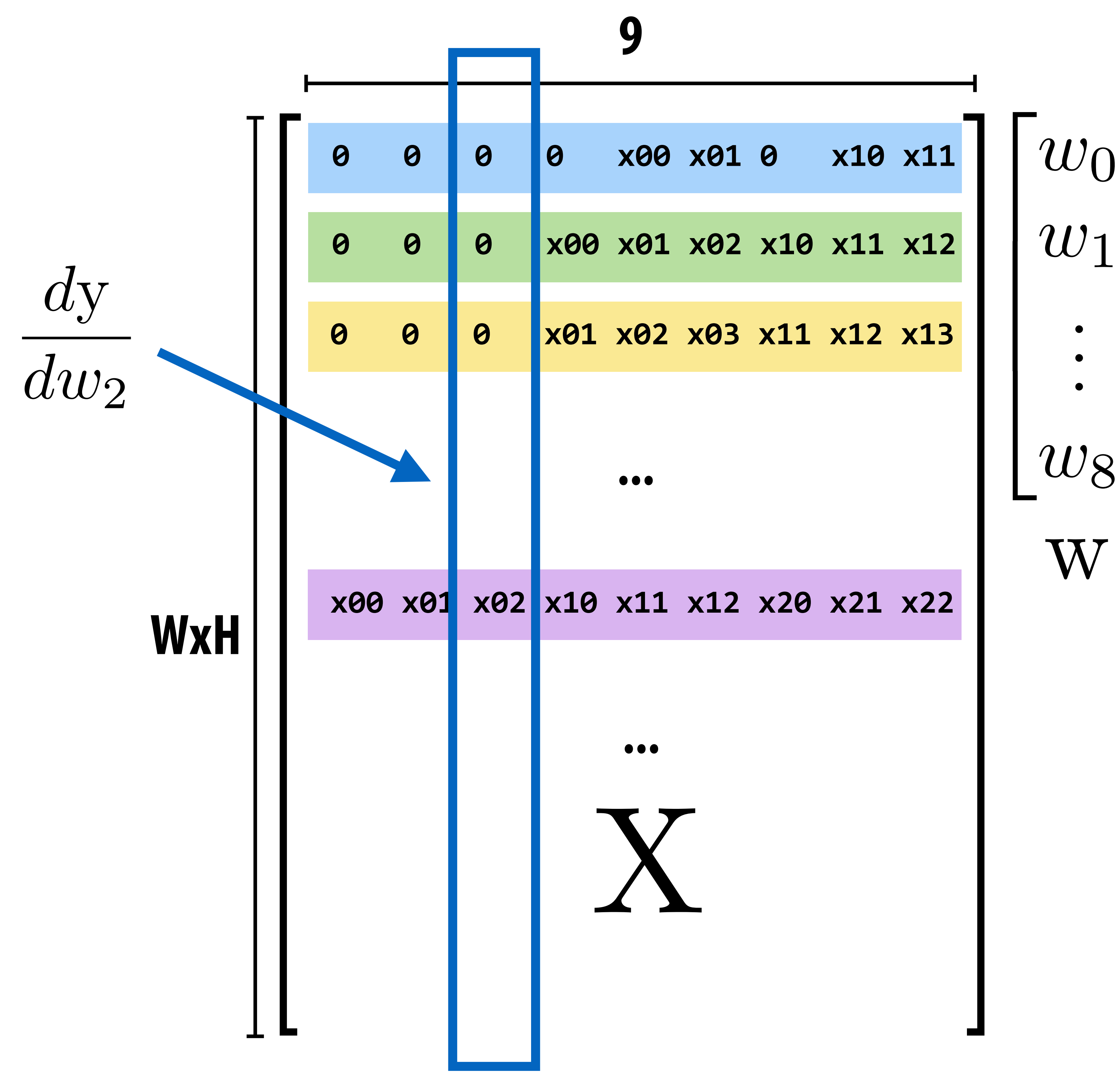


$$\frac{dy_j}{dw_i} = X_{ji}$$

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

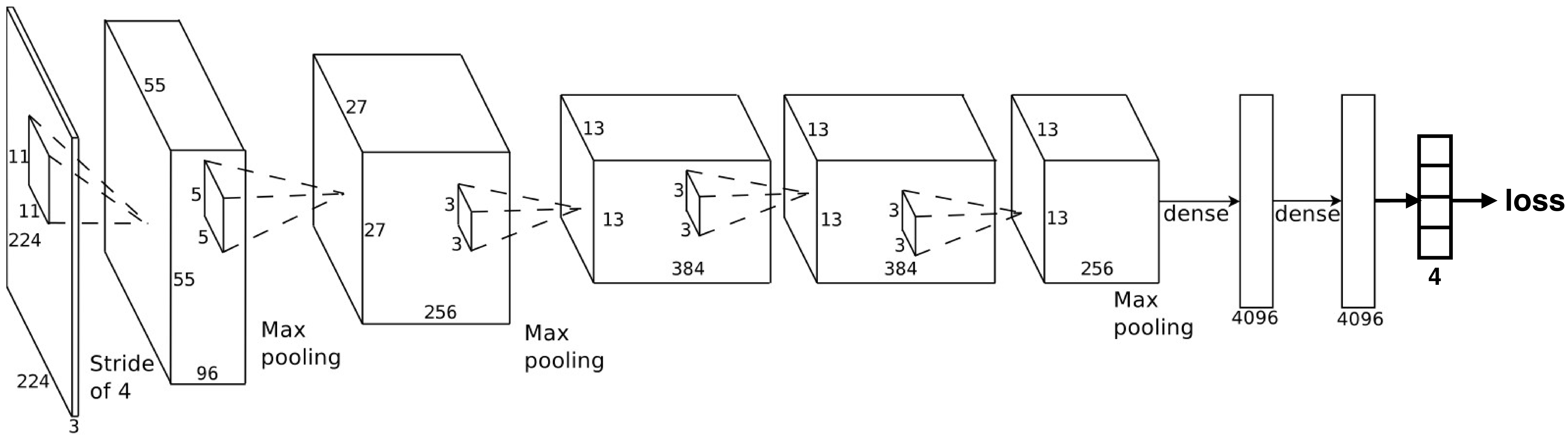
$$= \sum_j \frac{dL}{dy_j} X_{ji}$$




**Therefore:**

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

# Back-propagation through the entire professor classification network




**For each training example $x_i$ in mini-batch:**

- **Perform forward evaluation to compute loss for $x_i$**
  - **Note: must retain all layer outputs + output gradients (needed to compute weight gradients during backpropagation)**
- **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 batch)**

**Update all parameter values: wi_new = wi_old - step_size * gradi**

# VGG memory footprint

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

inputs/outputs get multiplied by mini-batch size

Unlike forward evaluation:

1. must store outputs and gradient of outputs
2. 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

sum reduction

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

trivial data-parallel over parameters

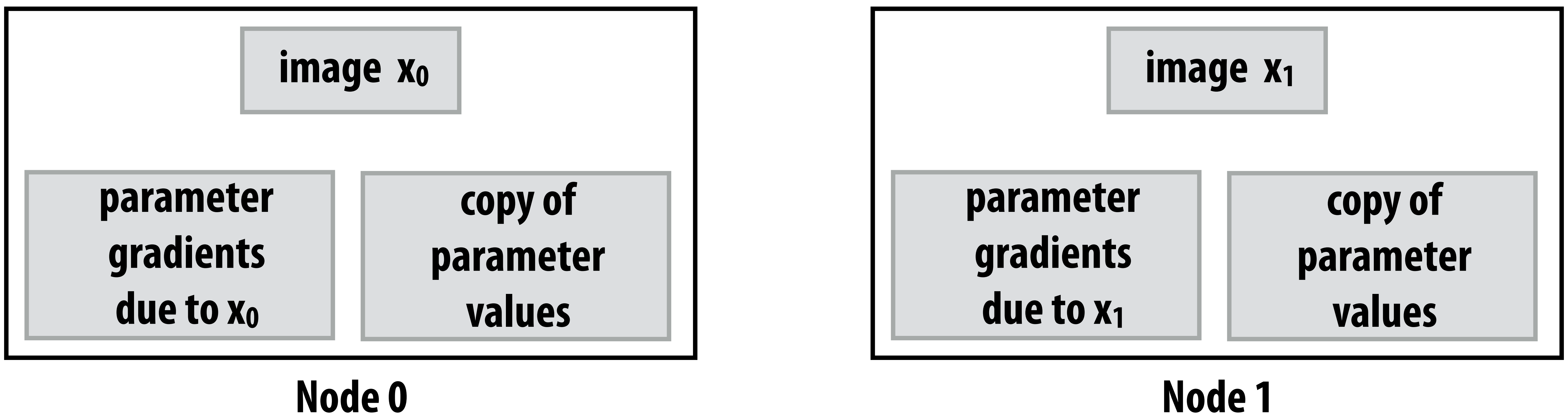
# Deep network training workload

- **Huge 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 takes days
- **Large memory footprint**
  - Must maintain network layer outputs from forward pass
  - Additional memory to store gradients 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 network across nodes to keep network + 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

# 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 * step_size;
```

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




```
partition mini-batch 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
```

# Challenges of computing at cluster scale

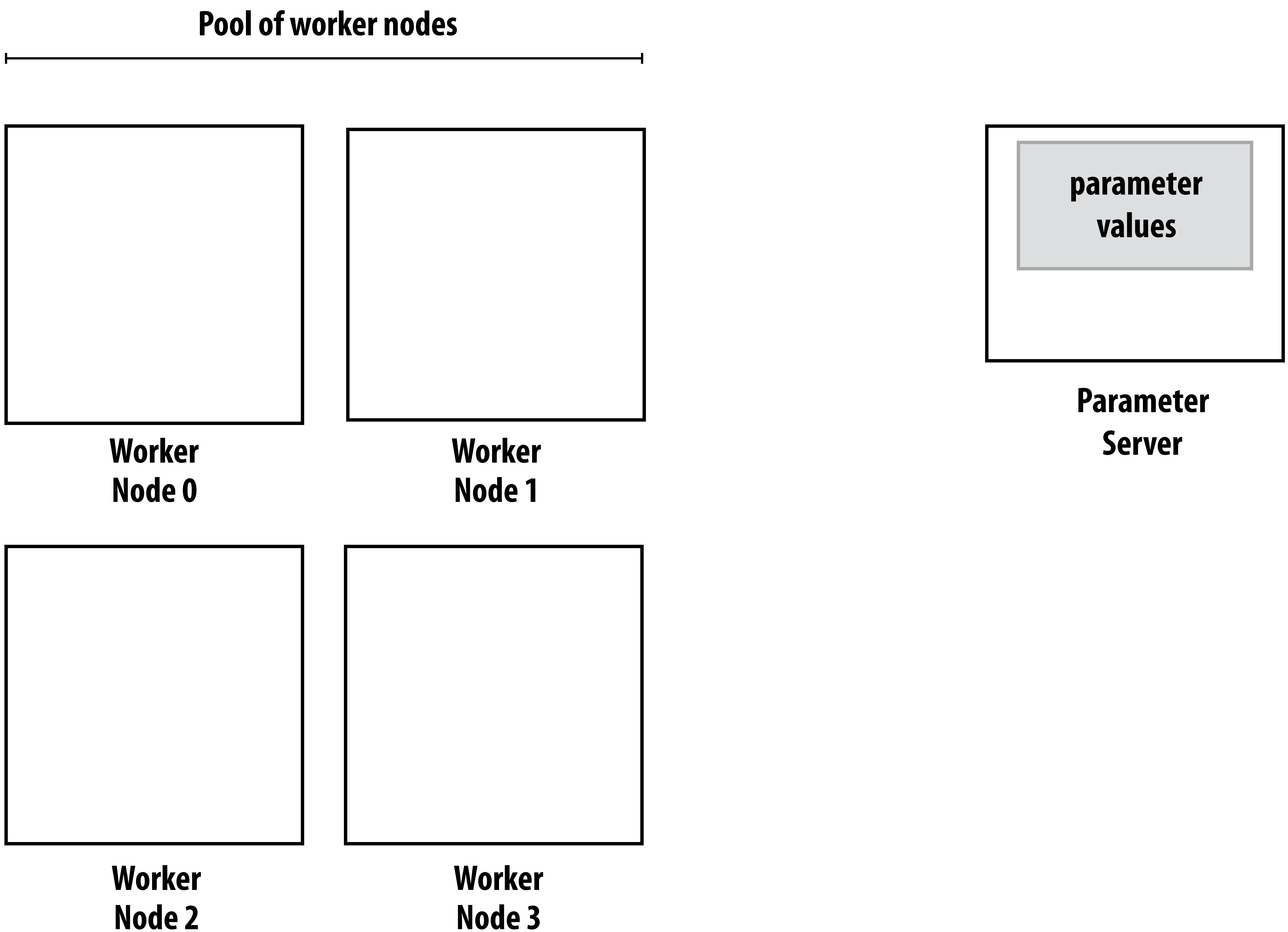
- **Slow communication between nodes**
  - Clusters do not feature high-performance interconnects typical of supercomputers

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

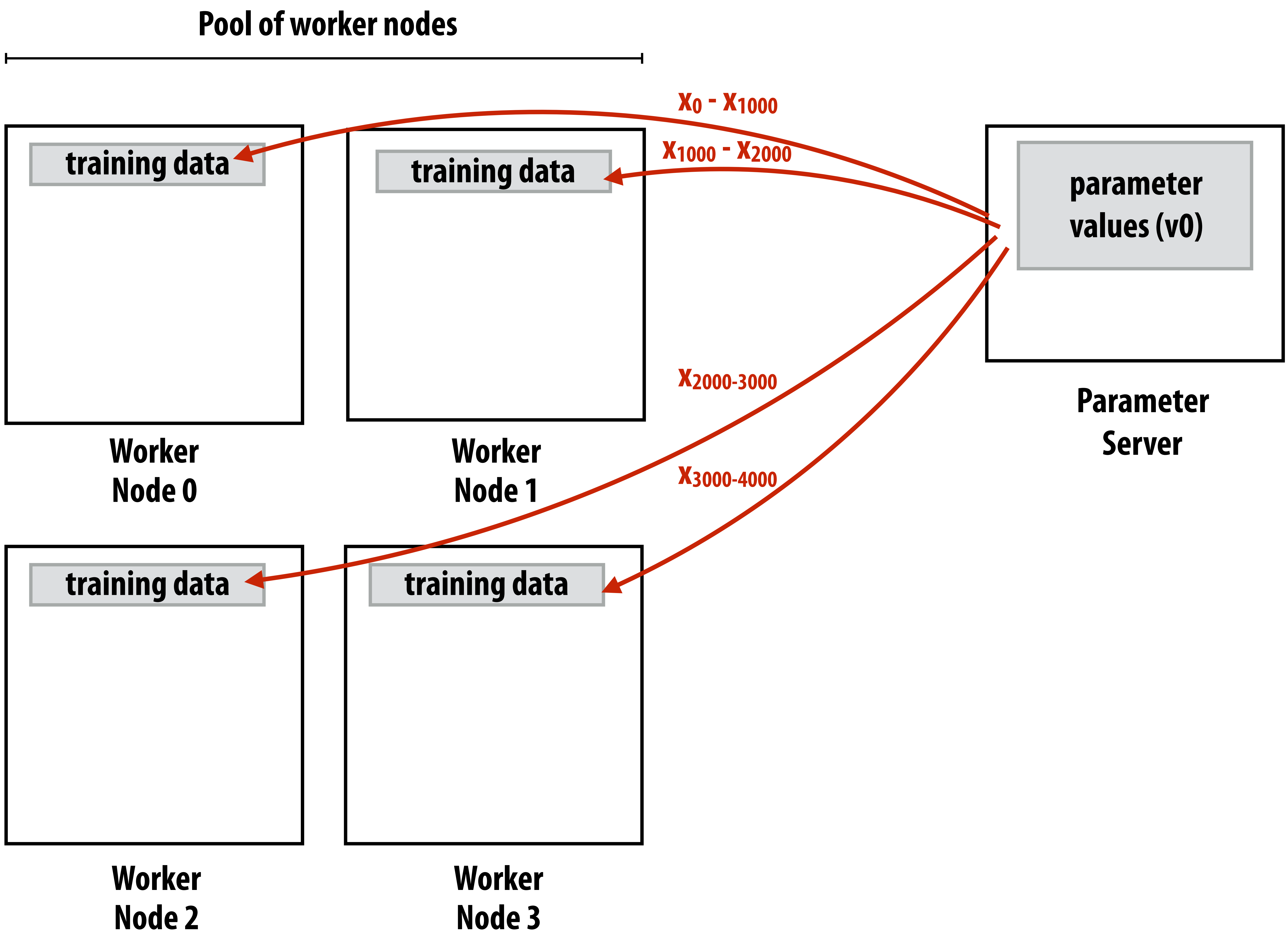
**Modern solution: exploit characteristics of SGD using asynchronous execution!**

# Parameter server design

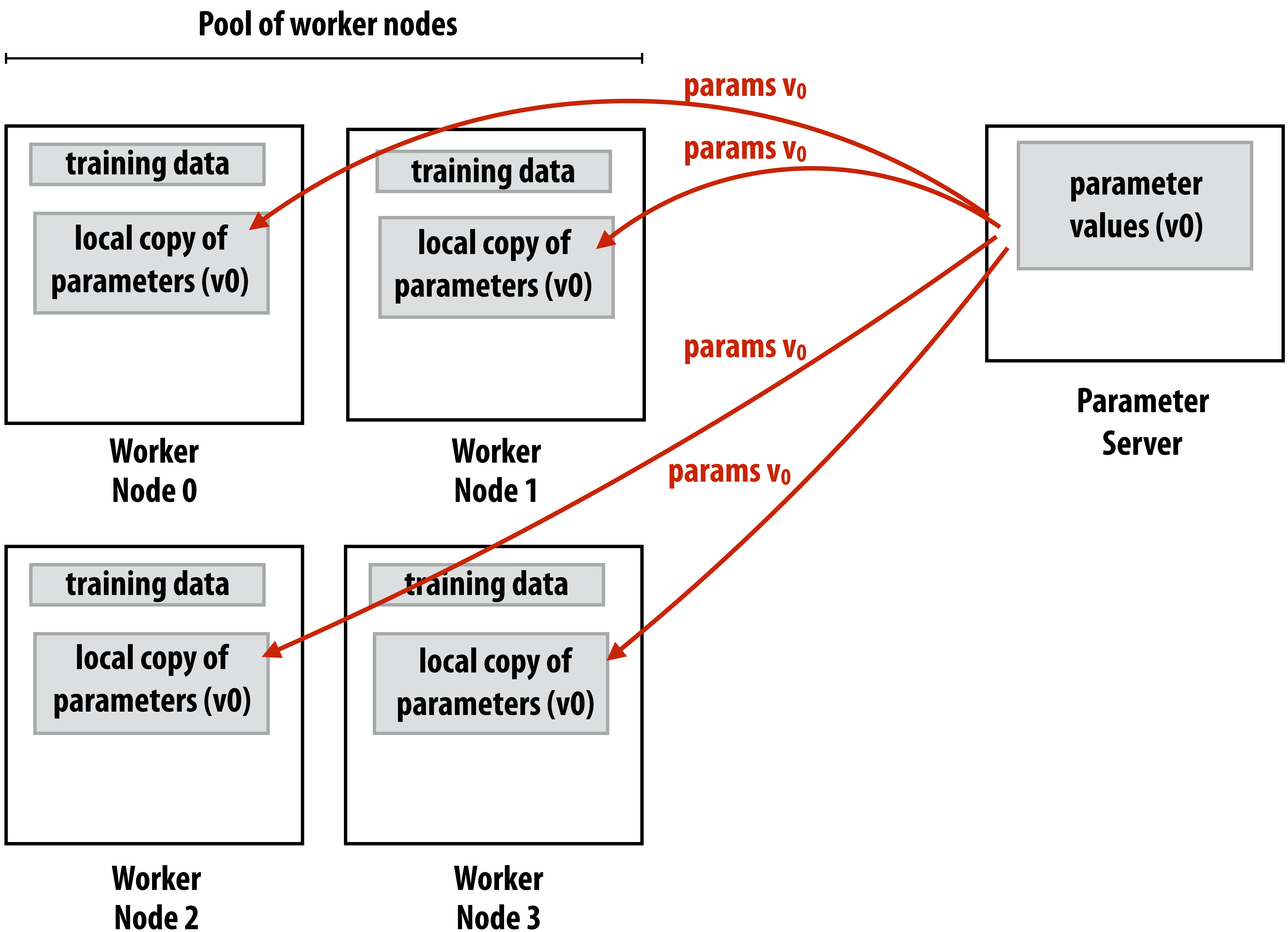
Parameter Server [Li OSDI14]
Google's DistBelief [Dean NIPS12]
Microsoft's Project Adam [Chilimbi OSDI14]



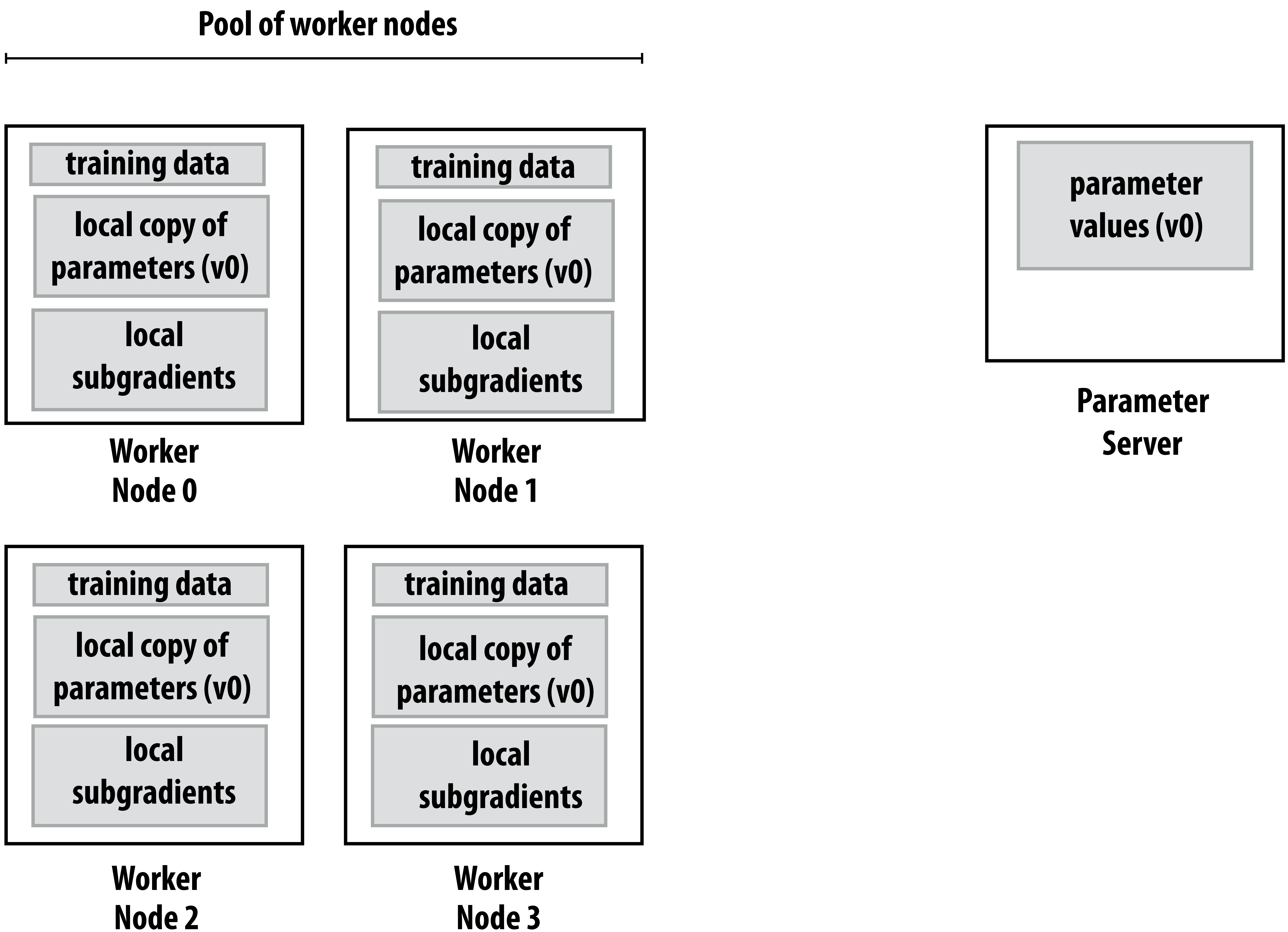
# Training data partitioned among workers



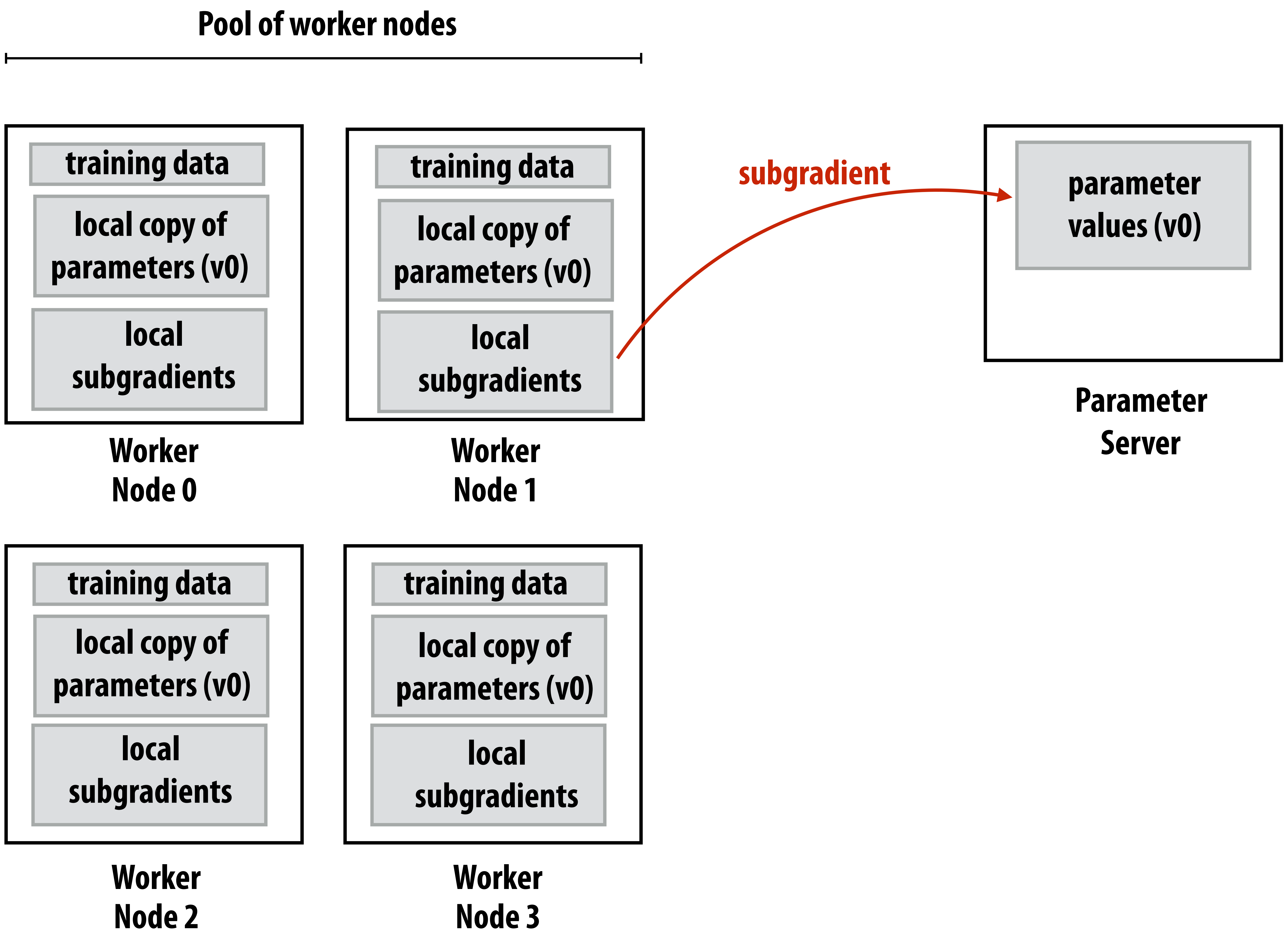
# Copy of parameters sent to workers



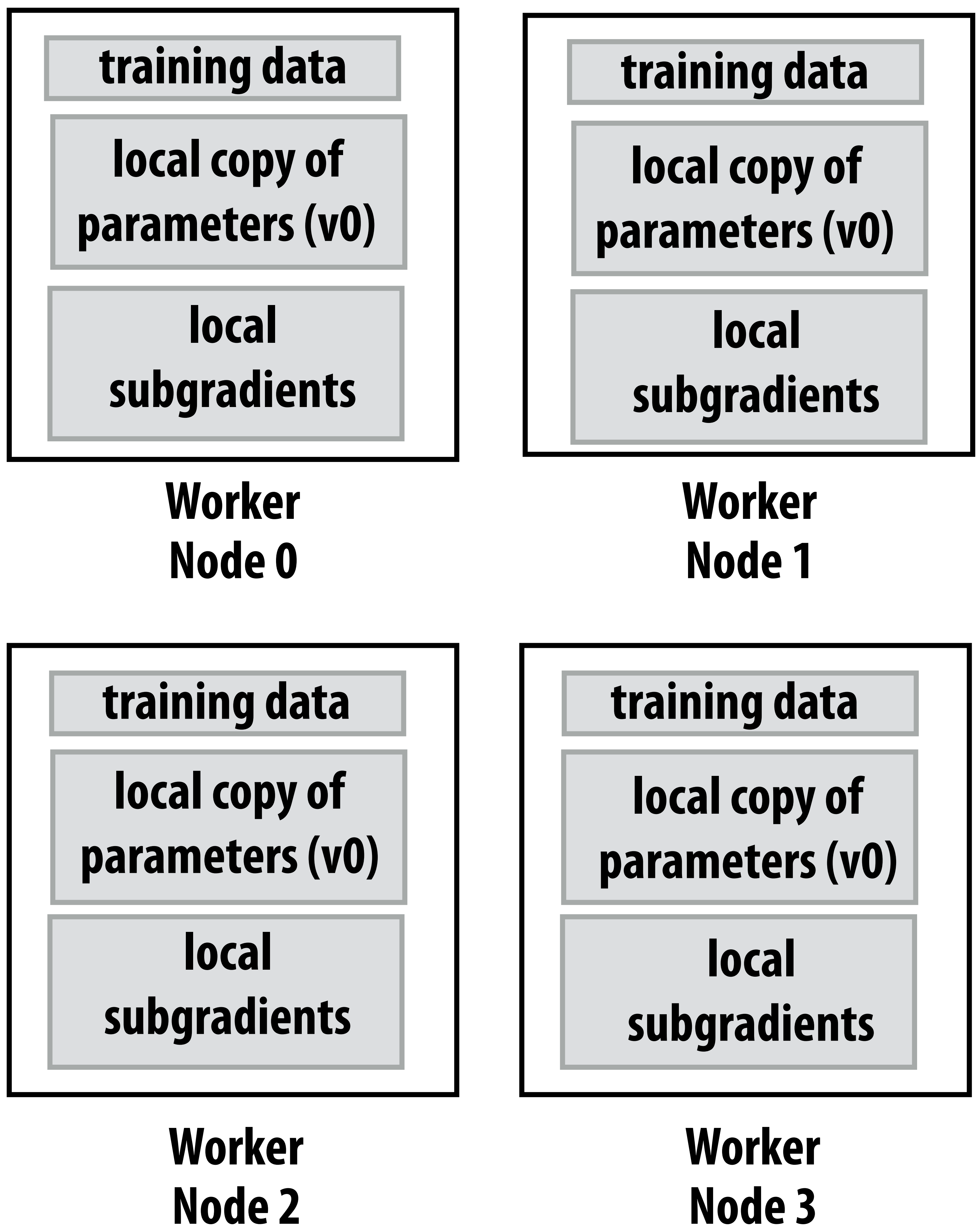
# Workers independently compute local "subgradients"

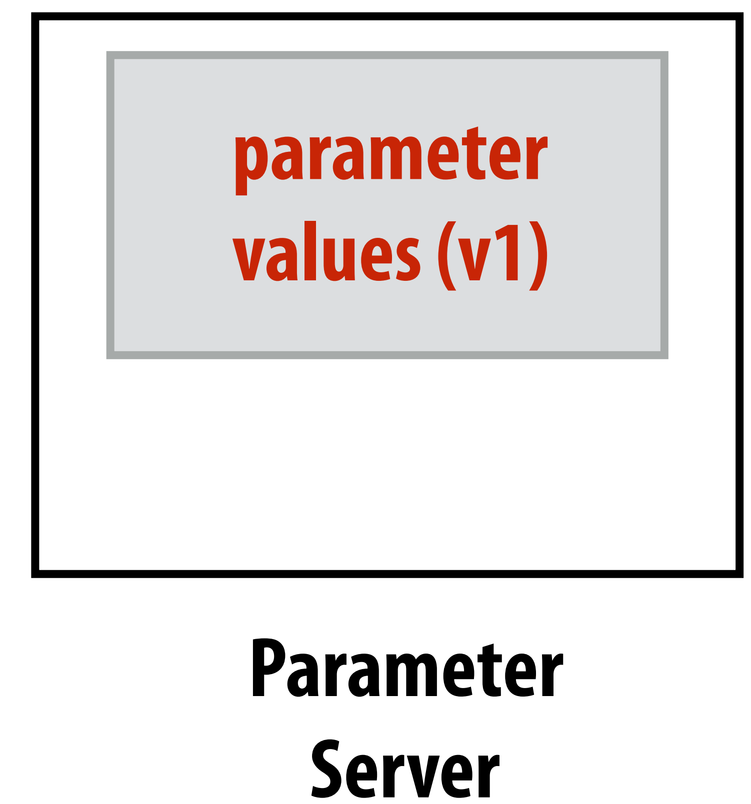


# Worker sends subgradient to parameter server



# Server updates global parameter values based on subgradient

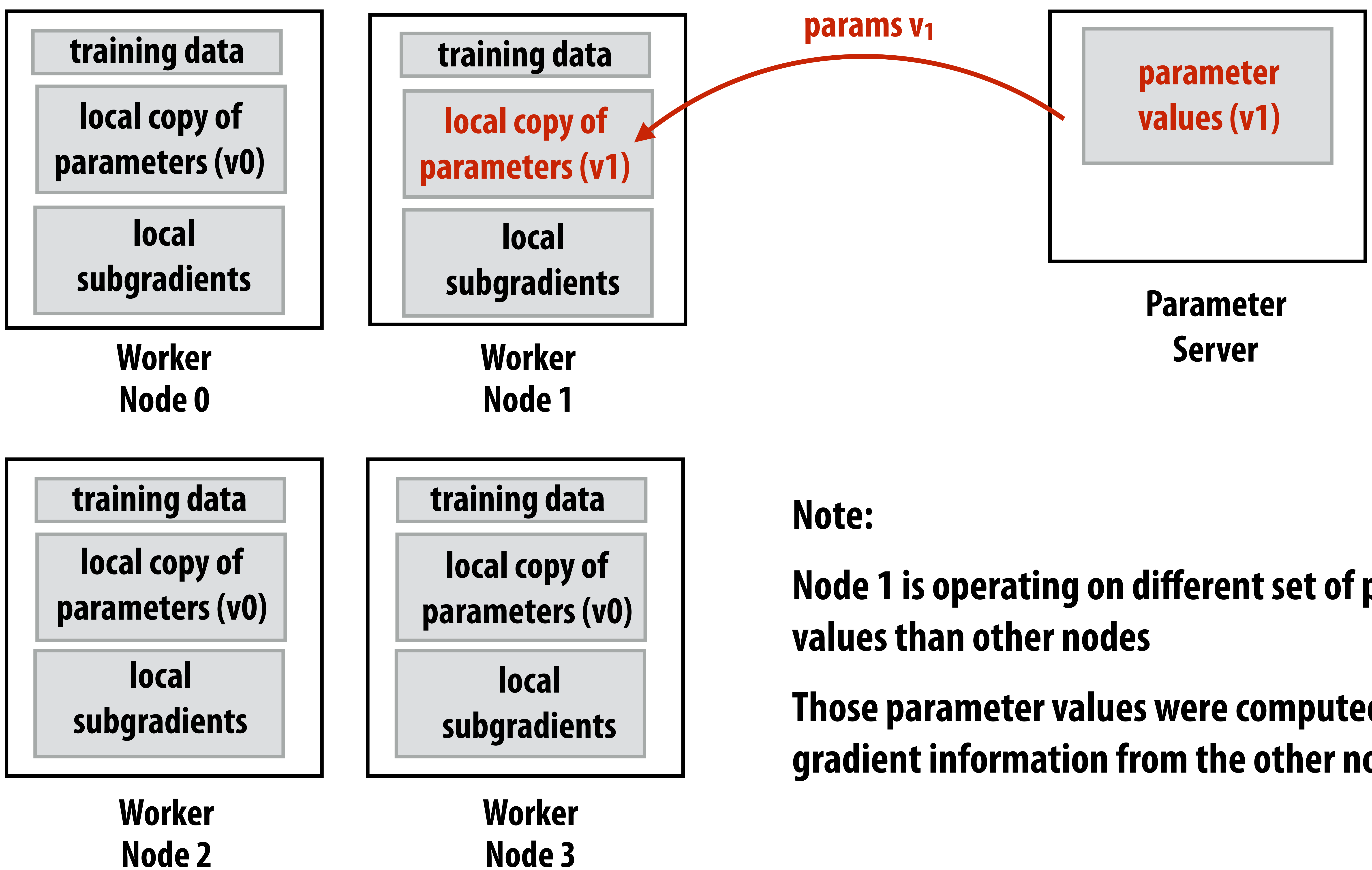



```
params += -subgrad * step_size;
```

# Updated parameters sent to worker

## Worker proceeds with another gradient computation step

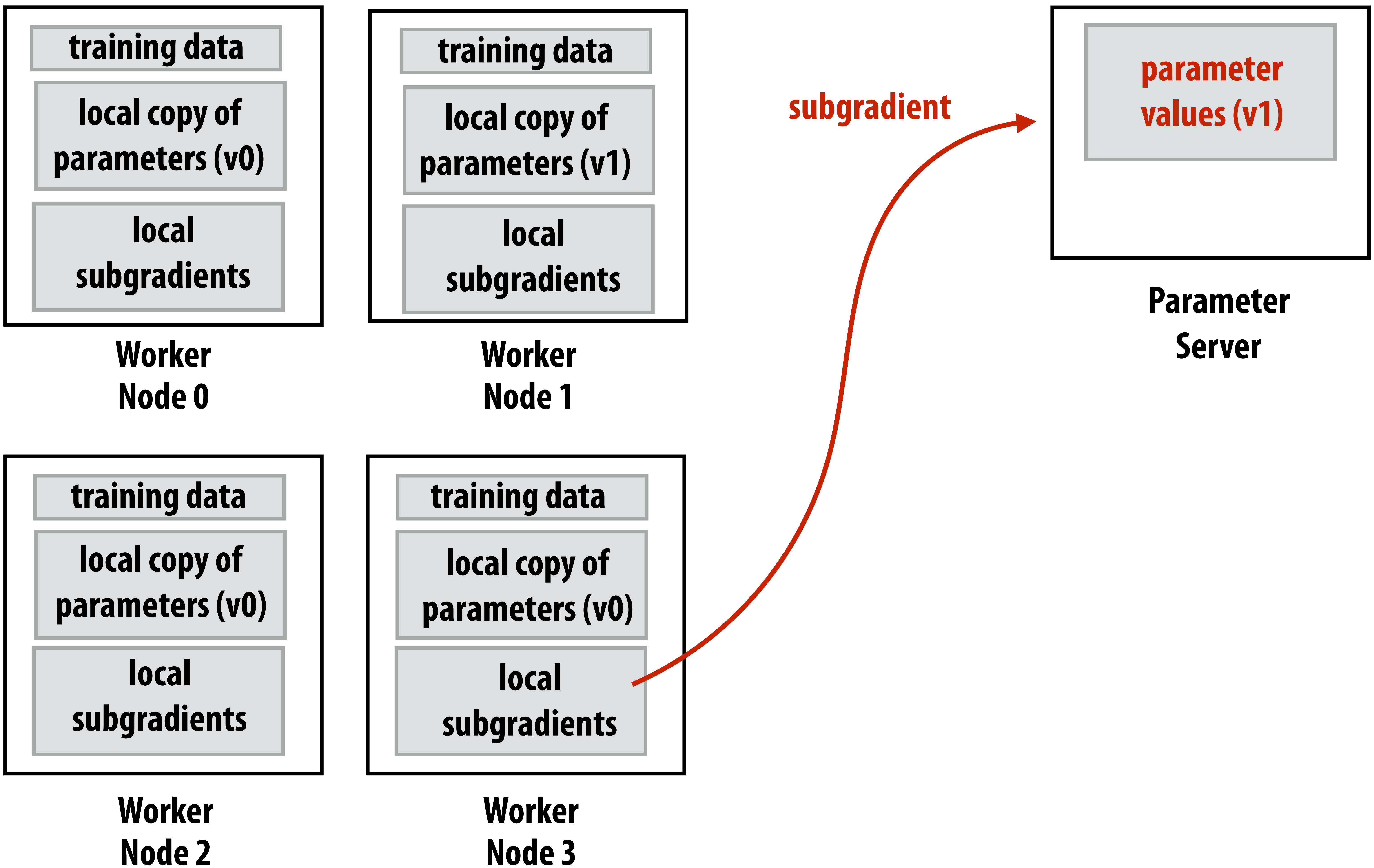



**Note:**

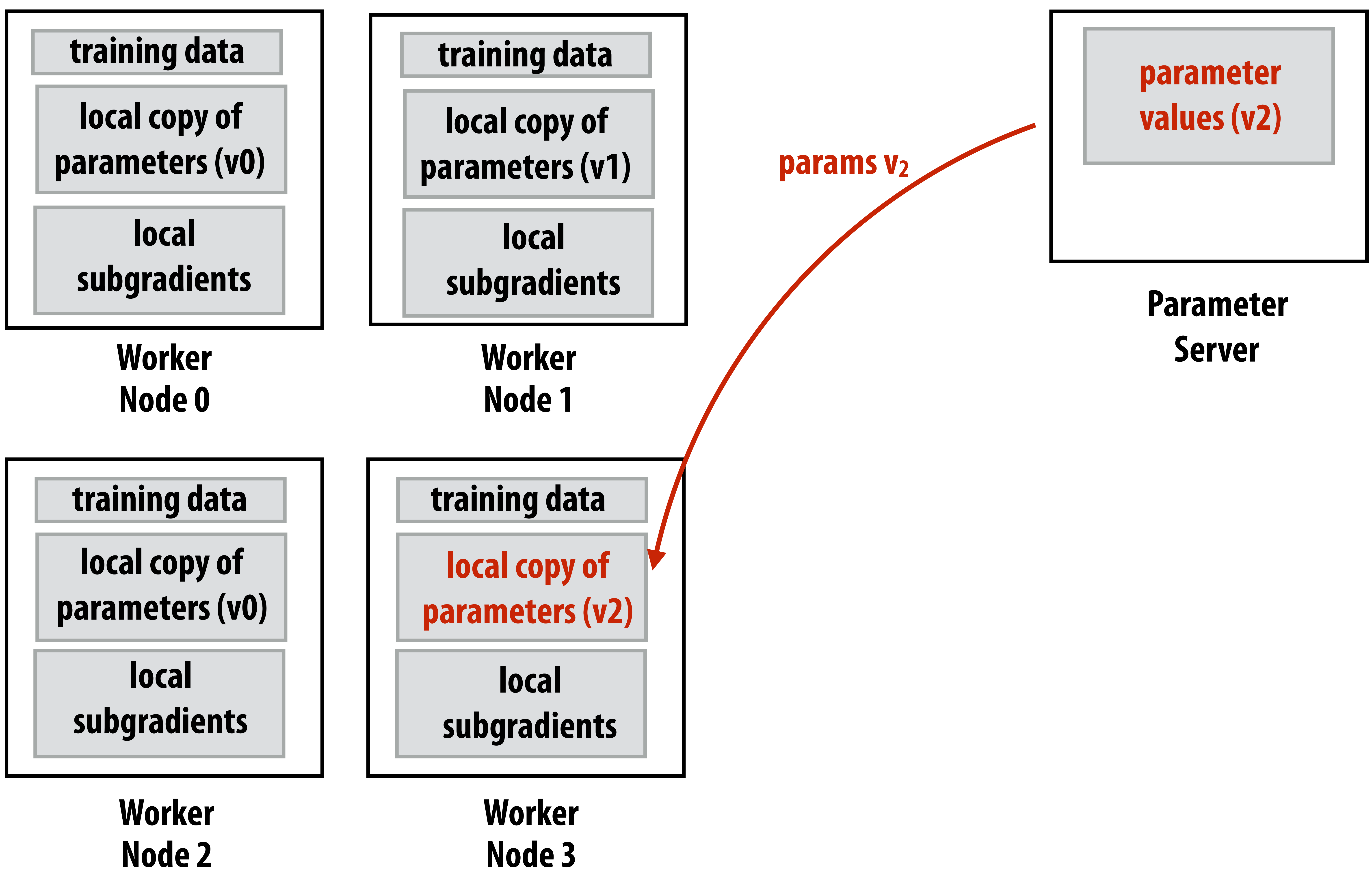
**Node 1 is operating on different set of parameter values than other nodes**

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

# Updated parameters sent to worker (again)



# Worker continues with updated parameters

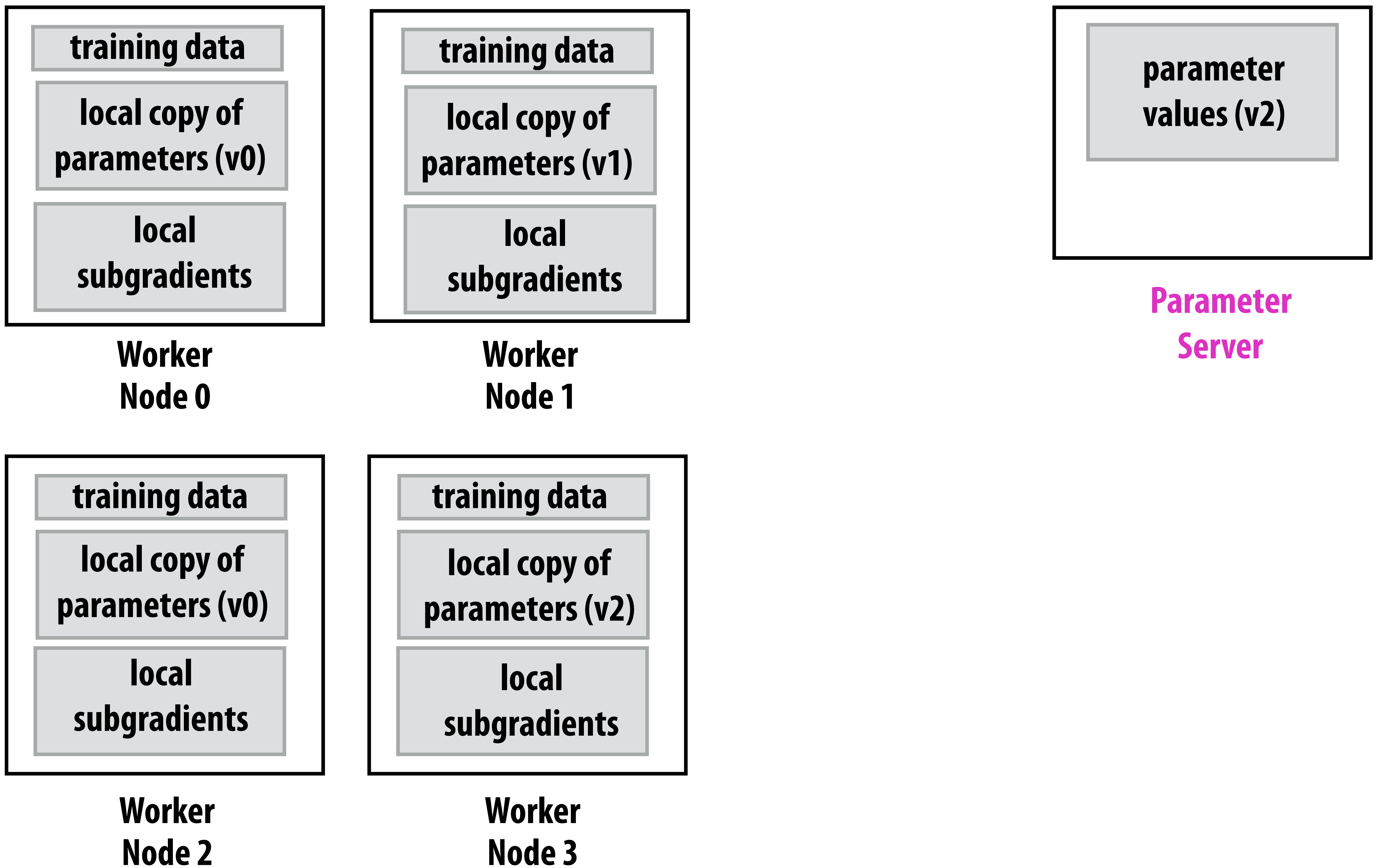


# Summary: asynchronous parameter update

- 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?

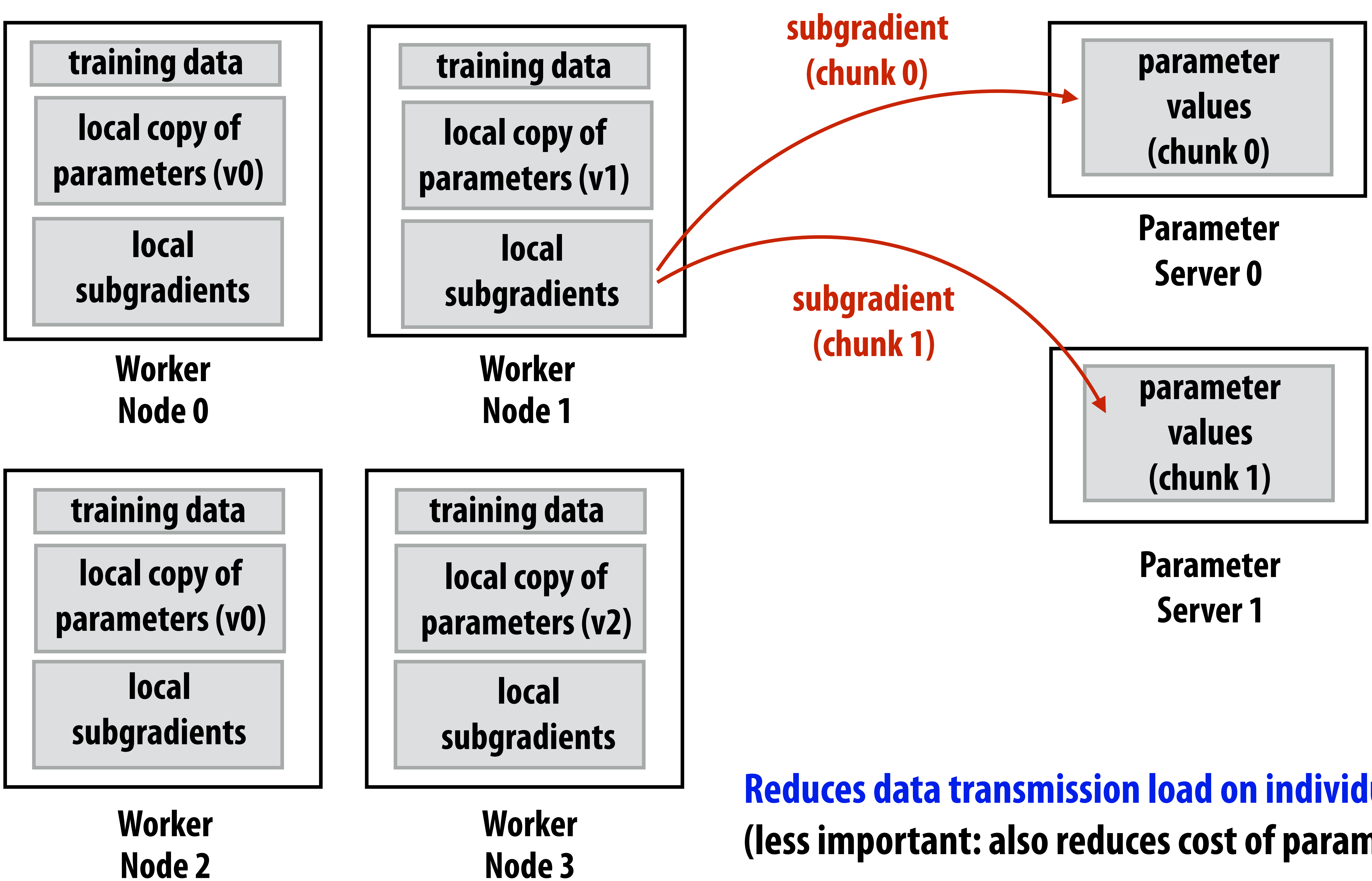
## What if there is heavy contention for parameter server?



# Shard the parameter server

## Partition parameters across servers

## Worker sends chunk of subgradients to owning parameter server

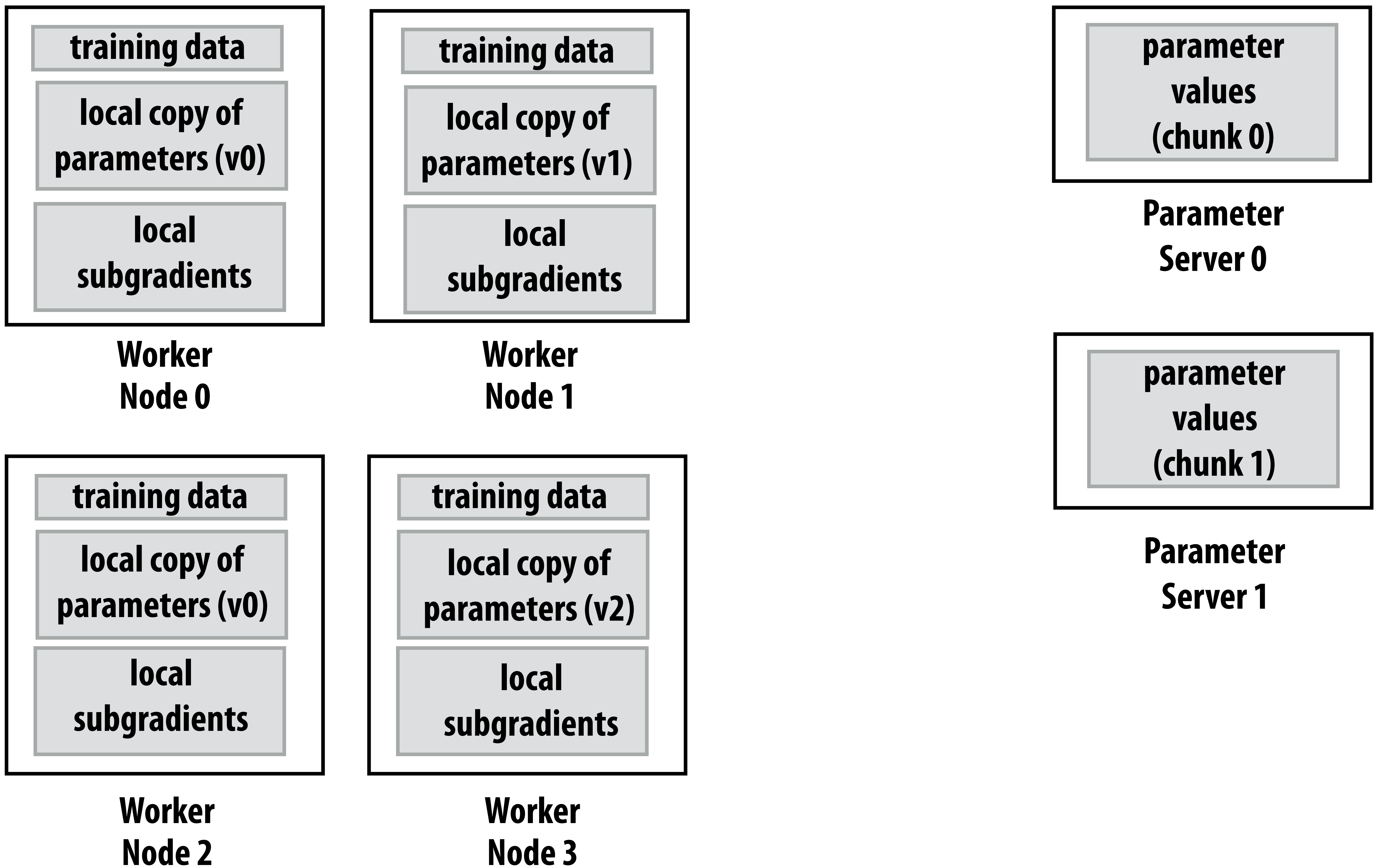



**Reduces data transmission load on individual servers**

(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)

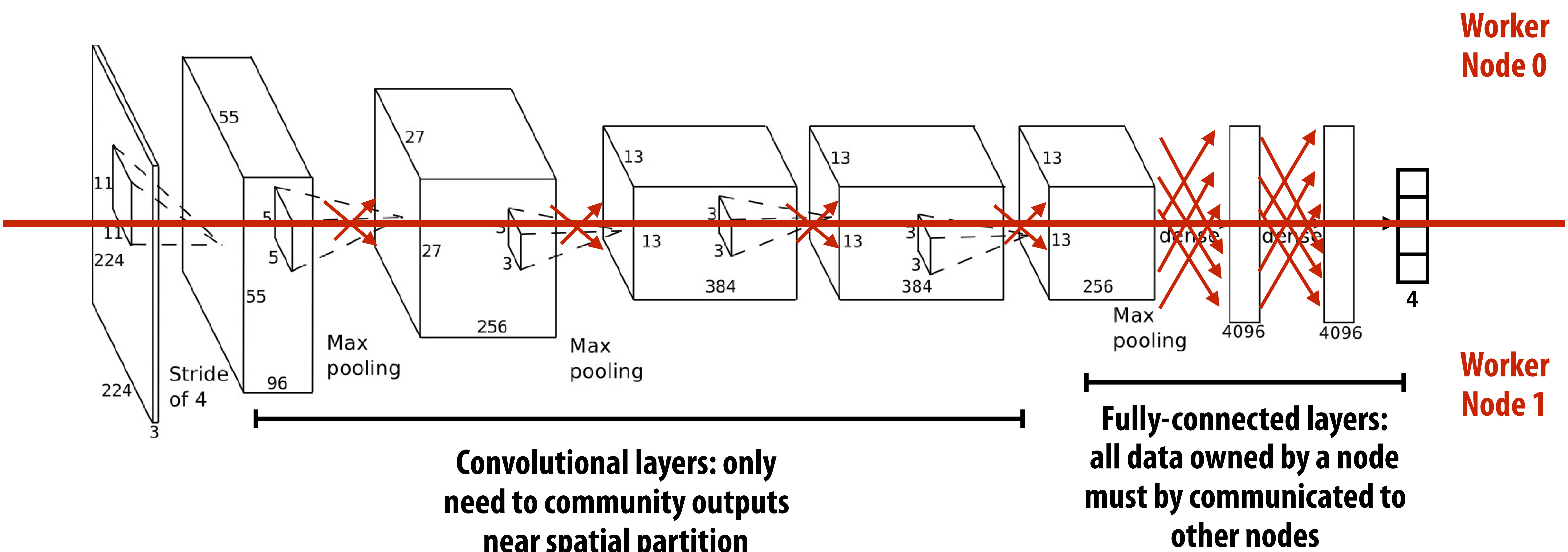


# Model parallelism

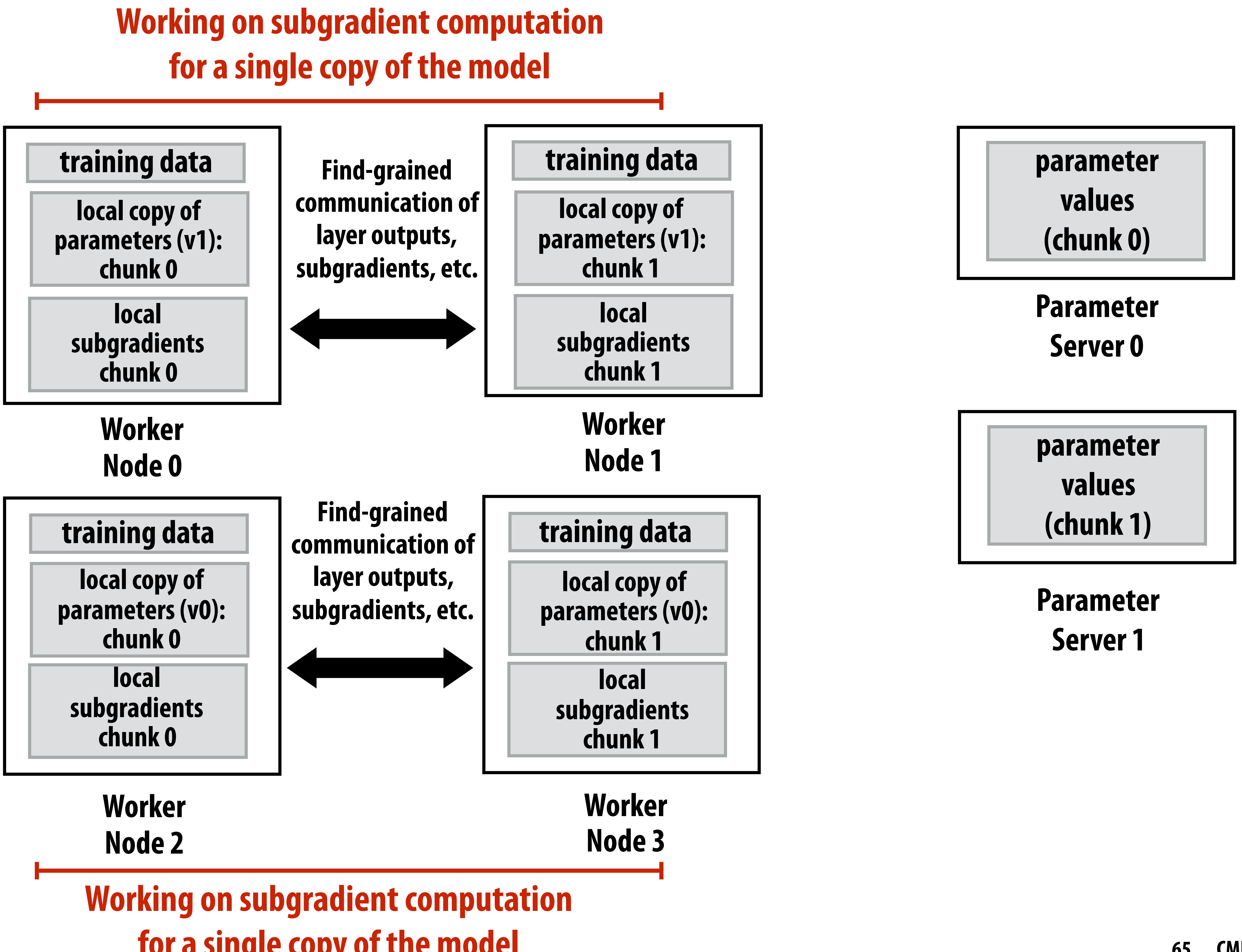
## Partition network parameters across nodes (spatial partitioning to reduce communication)

## Reduce internode communication through network design:

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



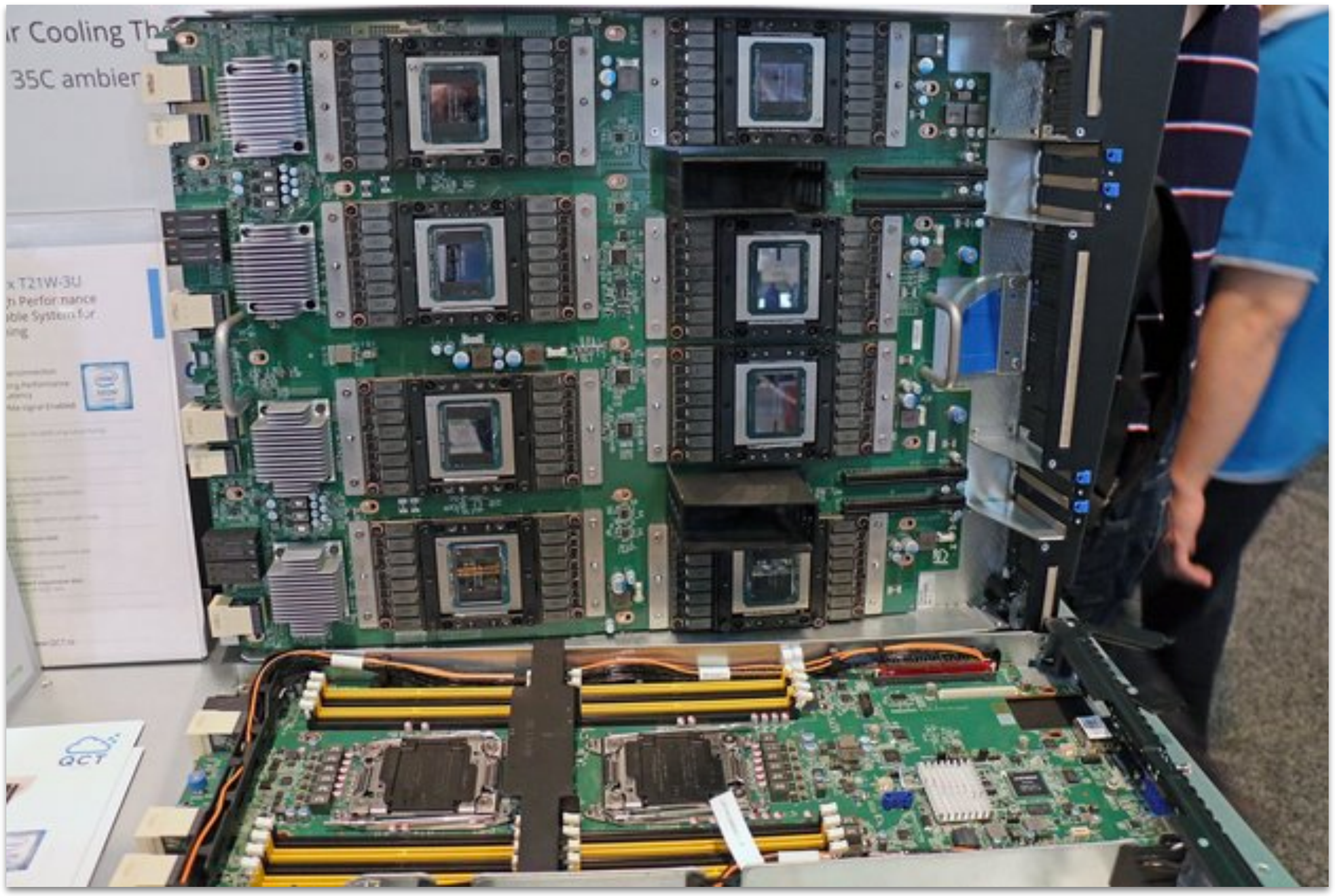
# Training data-parallel and model-parallel execution



# Using supercomputers for training?

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

OakRidge Titan Supercomputer

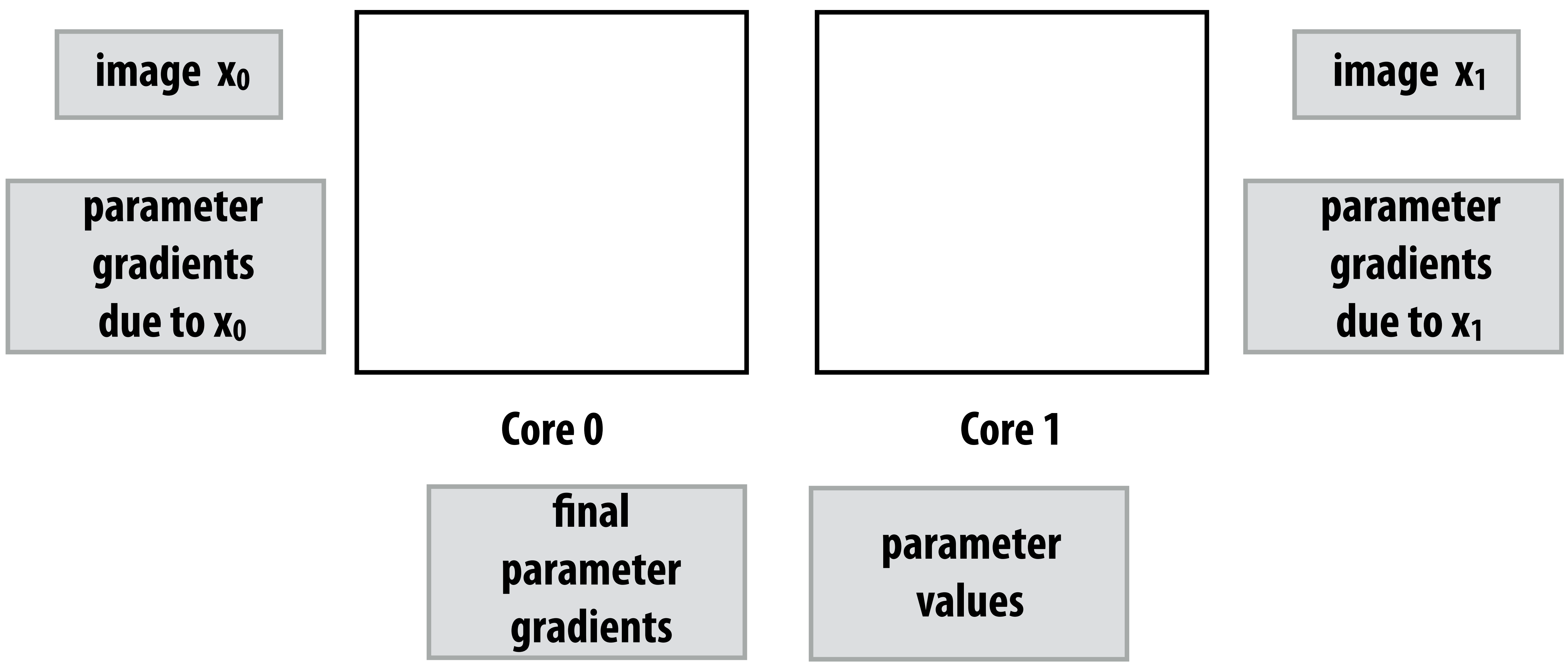


NVIDIA DGX-1: 8 Pascal GPUs connected via high speed NV-Link interconnect

# Parallelizing mini-batch on one machine

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

**Consider parallelization of the outer for loop across cores**




**<u>Good</u>: completely independent computations (until gradient reduction)**

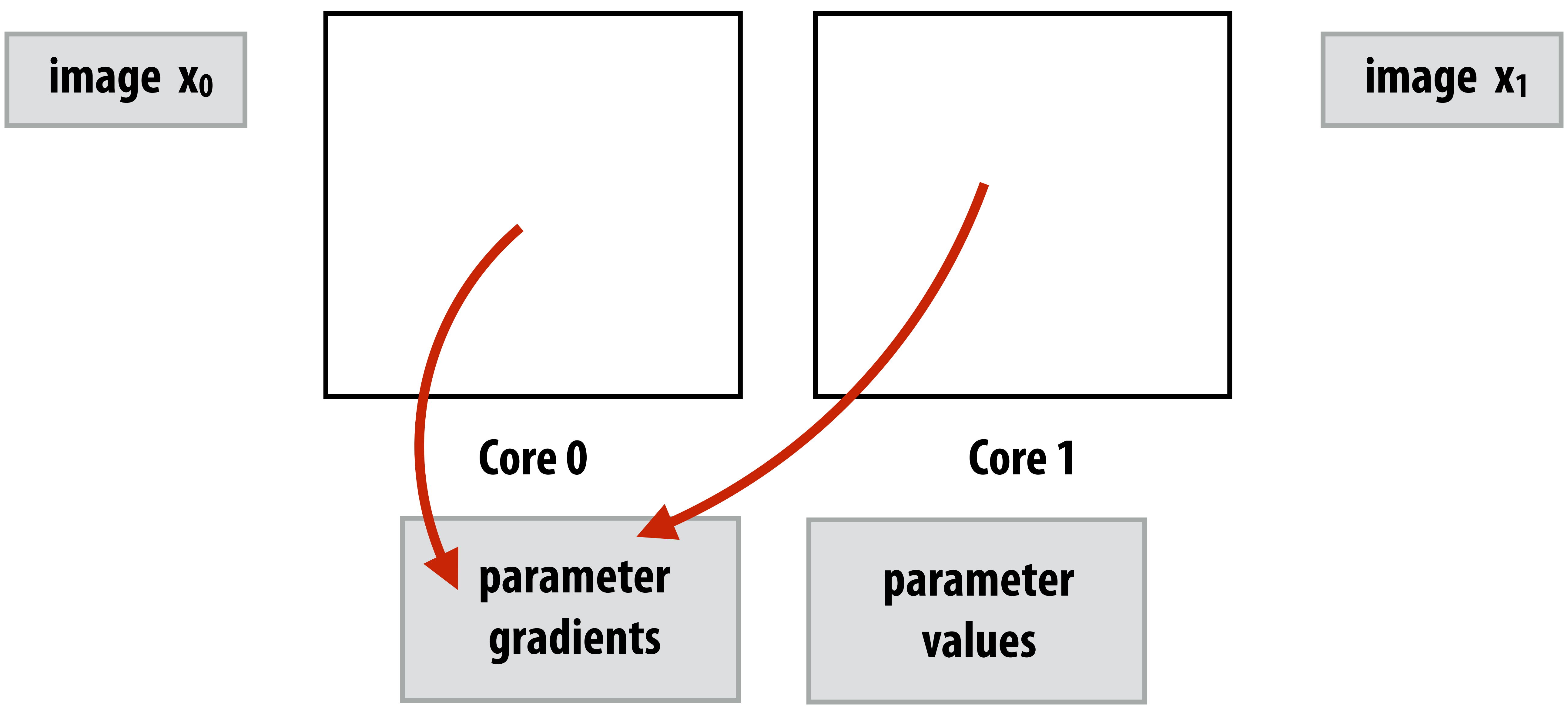
**<u>Bad</u>: complete duplication of parameter gradient state (100's MB per core)**

# Asynchronous update on one node

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

**Cores update shared set of gradients.**

**Skip taking locks / synchronizing across cores: perform "approximate reduction"**




Project Adam [Chilimbi OSDI14]

# Summary: training large networks in parallel

- **Most systems rely on asynchronous update to efficiently used clusters of commodity machines**
  - Modification of SGD algorithm to meet constraints of modern parallel systems
  - Open question: effects on convergence are problem dependent and not particularly well understood
  - Tighter integration / faster interconnects may provide alternative to these methods (facilitate tightly orchestrated solutions much like supercomputing applications)

- **Open question: how big of networks are needed?**
  - >90% of connections could be removed without significant impact on quality of network
  - 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! recall the original grid solver example!)