# ECE 508
# Manycore Parallel Algorithms

## Lecture 10: Application Case Study: Deep Learning

# Objective

- to learn the implementation of a convolutional neural network
  - basic structure and levels of parallelism
  - inference
  - training
  - optimizations

(forward direction is review for those who took 408)

# Perspective is Important

**Chips are cheaper than ever.**

Unlike humans, digital systems offer

- **high-speed computation**,
- **low capital investment**
  (purchase vs. training a human), and
- **negligible operations cost** (no salary!).

**If computer outperforms** (or even matches)
a **human, use a computer**.

**Industry has done so for 40-50 years now.**

# What is Machine Learning?

**machine learning**: important method of building applications whose logic is not fully understood

**Usually** for problems for which **incorrect output** does **not** have **catastrophic** consequences.

Typically by **example**:

- **use labeled data** (matched input-output pairs)
- **to represent** desired **relationship**.

# Machine Learning Separates into Two Phases

**Two phases:**

- **training** / learning phase:
  **iteratively adjust program logic** to produce desired/approximate answers (called **training**)
- **inference** / deployment phase:
  apply learned function to solve new problems (for new input data)

# Types of Learning Tasks

- classification
  - Map each input to a category
  - Ex: object recognition, chip defect detection
- regression
  - Numerical prediction from a sequence
  - Ex: predict tomorrow's temperature
- transcription
  - Unstructured data into textual form
  - Ex: optical character recognition

# More Advanced Learning Tasks

- translation
  - Convert a sequence of symbols in one language to a sequence of symbols in another
- structured output
  - Convert an input to a vector with important relationships between elements
  - Ex: natural language sentence into grammatical structure
- others
  - Anomaly detection, synthesis, sampling, imputation, denoising, density estimation, genetic variant calling

# Test Cycle Time is Important

You've all written code…

- code, test, code, test, code, test
- integrate, test, test, test
- and test again!

**But how long is the code, test cycle?**

**Depends what you're building.**

**What's your longest?**

# Your Cycle Times are Probably Small

In college, **10k lines** took **½ hour** to compile on my PC.

In grad. school, **100k lines** took

- **½ hour** to compile on my workstation, or
- **2 minutes** on our cluster (research platform).

In ECE435 (networking lab), students needed

- **½ hour** to reinstall Linux after a bad bug.
- (Ever had a good bug?)

Gene sequencing / applications can take **two weeks**.

**We're all a little spoiled…**

# Why Machine Learning Again?

In 2007, **programmable GPUs accelerated the training cycle**.

- Today, new chip designs for learning applications have further accelerated.
- Led to a resurgence of interest
- in Computer Vision, Speech Recognition, Document Translation, Self Driving Cars, Data Science…
- all tasks that human brains solve regularly, but for which we have struggled to express solutions systematically.

# Many Problems are Still Hard

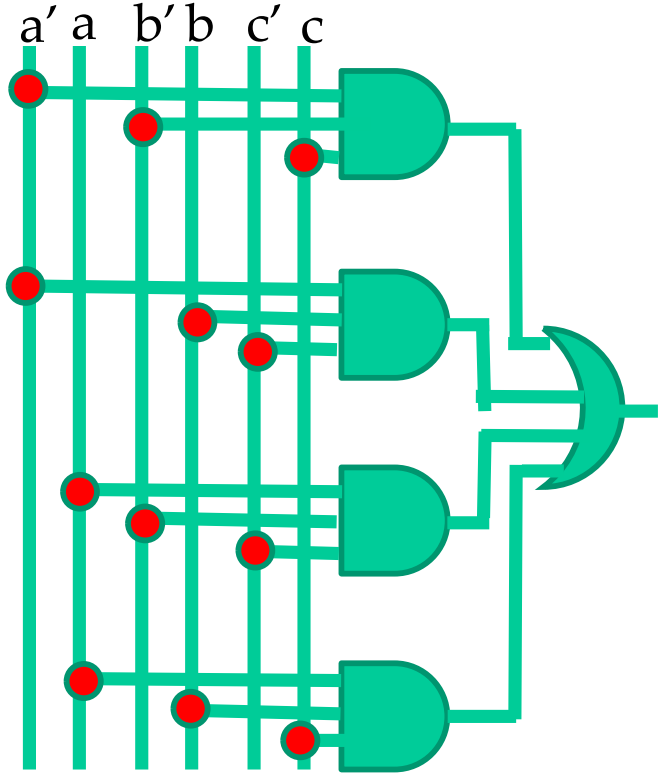
**Speed is not a panacea.**

- Many **tasks still require human insight**
  - for network structure and feature selection
  - for effective input and output formats, and
  - for production of high-quality labeled data.
- Other trends sometimes help: ubiquitous computing enables crowdsourcing, for example.

# Many Problems Have Systematic Solutions

Example: building a Boolean function from a truth table.

| Input | | | output |
|---|---|---|---|
| a | b | c | |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |



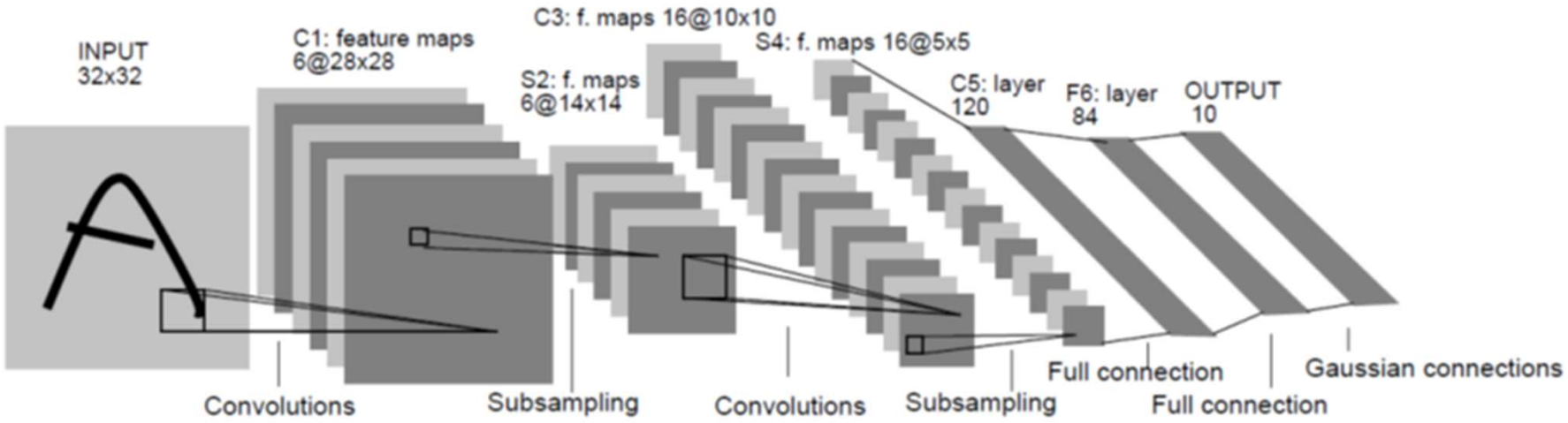
# What if We Lack a Truth Table?

- Make enough observations to construct a rule
  - 000 → 0
  - 011 → 0
  - 100 → 1
  - 110 → 0

- If we cover all input patterns,
  we can construct a truth table!

# Some Problems Too Large to Sample




LeNet-5, a convolutional neural network for hand-written digit recognition

One input is 32×32×8 bits. Truth table requires $2^{8192}$ rows!

1 billion ($2^{30}$) **images covers**… **0% of** the **input** space.

# Labeled Data Provide a Partial Specification

LeNet-5

- tries to look for features that indicate
- which digit is represented by an input image.

**Labeled data give** a **partial specification** …

- somewhat like random testing, but
- we need examples and
- counterexamples with "correct" labels.

# Inputs and Labels May be Biased

LeNet-5*, for example, used

- US Census Bureau and
- high school student samples.

**Does that include all versions shown above?**

(Actually it does, but you get the point.)

*See yann.lecun.com/exdb/mnist, or the original paper:
Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-Based Learning Applied to Document Recognition," Proceedings of the IEEE, 86(11):2278-2324, Nov. 1998.

# Who Defines Correctness of Labeled Data?

**Who provides the labels?**

**Grad students**!

16-wk intern. × 120 hrs/wk × 3600 s/hr × 0.1 labels/s

= **691,200 labeled inputs** = 1 paper!

Or maybe **undergraduates**.

These days … **crowdsourcing**?

# Output is a Probability for Each Category

**For** problems such as **classification**

- (LeNet-5: classify image as digit from 0 to 9)
- **networks** typically **produce** a **vector of probabilities** (one for each category).

Labels are usually 100% of some category,
but not always. Let's vote on this one…
(crowdsourcing may not produce binary answers).

# Difficulty Often Depends on Context

**Context is** also **important**:

- if you look carefully (at, say, the LeNet-5 paper),
- you'll see that one also has to capture 'non-digits.'

**Labeled examples of non-digits**

- **may** also **help** improve training
- if the inference use may include input
- other than the expected categories.

Analogy: it's easier to grade multiple choice than to grade numerical answers than to grade essay questions, as the space of possible answers grows.

# Focus on Neural Networks

**Machine learning** is a huge area

- with **many techniques and approaches**.
- We **focus on neural networks**
- mostly **because GPUs** have come to **dominate** the computation of **these systems**.

Both training and inference using neural networks

- requires **fairly regular computation** to which
- **we can apply what we have learned** so far.

# New Opportunities Have Arisen from Use of GPUs

Deep Neural Networks (**DNNs**) executing on GPUs

- **have** also **created** a variety of **opportunities**,
- **including simultaneous inference and training**,
- as when they are dynamically tuned
  - to a particular phone owner's voice and speech patterns
  - while simultaneously providing speech recognition services.

# Neural Networks Model Human Neural Activity

**What's a "neural" network?**

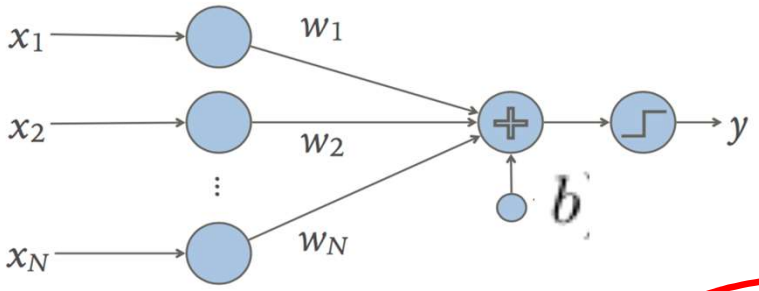
An **interconnection**—usually in layers—
**of** many individual **linear classifiers**
**modeled on human neurons**.

For training, we want continuous, piecewise differentiable functions with non-zero derivatives, so the neural "activation function" (`sign` in the next slide) must be chosen carefully.

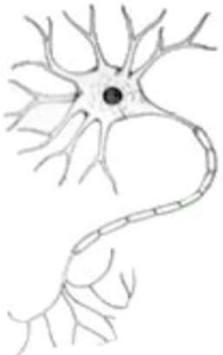
# Perceptron is a Simple Example

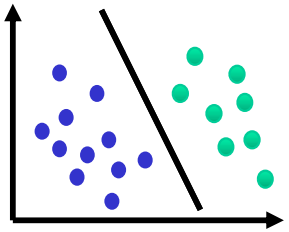
Example: **a perceptron**

$y = \text{sign}(W \bullet x + b)$ $\quad \theta = \{W, b\}$




$\approx$





Dot product: $y = sign\ (W \bullet x$

Scalar addition: $+ b)$

output

activation function

inputs

weights

bias

# One Choice: Sigmoid/Logistic Function

Until about 2017,

- **sigmoid / logistic function** most popular

$$\boldsymbol{f(x) = \frac{1}{1+e^{-x}}} \quad \text{(f: } \mathbb{R} \rightarrow (0,1)\text{ )}$$

for replacing the sign function.

- Once we have *f(x)*, finding *df/dx* is easy:

$$\boldsymbol{\frac{df(x)}{dx}} = \frac{e^{-x}}{(1+e^{-x})^2} = f(x)\frac{e^{-x}}{(1+e^{-x})} = \boldsymbol{f(x)(1-f(x))}$$

# Today's Choice: ReLU

In 2017, most common choice became

- **rectified linear unit / ReLU / ramp function**

  $$\boldsymbol{f(x) = \max(0, x)} \quad (\text{f: } \mathbb{R} \rightarrow \mathbb{R}^{+})$$

  which is much faster (no exponent required).
- A smooth approximation is **softplus/SmoothReLU**

  $$\boldsymbol{f(x) = \ln(1 + e^{x})} \quad (\text{f: } \mathbb{R} \rightarrow \mathbb{R}^{+})$$

  which is the integral of the logistic function.
- Lots of variations exist. See Wikipedia for an overview and discussion of tradeoffs.

# Use Softmax to Produce Probabilities

**How can sigmoid / ReLU produce probabilities?**

They can't.

- Instead, given output vector **Z = (z[0], …, z[C-1])***,
- we produce a second vector **K = (k[0], …, k[C-1])**
- using the **softmax function**

$$k[i] = \frac{e^{z[i]}}{\sum_{j=0}^{C-1} e^{z[j]}}$$

Notice that **the k[i] sum to 1.**

*Remember that we classify into one of C categories.

# Softmax Derivatives Needed to Train

We also need the **derivatives of softmax**,

$$\frac{dk[i]}{dz[m]} = k[i](\delta_{i,m} - k[m]),$$

where $\delta_{i,m}$ is the Kronecker delta (1 if i = m, and 0 otherwise).

# Fully Connected Layers Not Always Useful

Consider a **250×250 image** treated as 1D vector.

- **Fully connected** layer is huge:
  - **62,500** ($250^2$) **weights per node**!
  - Comparable number of nodes

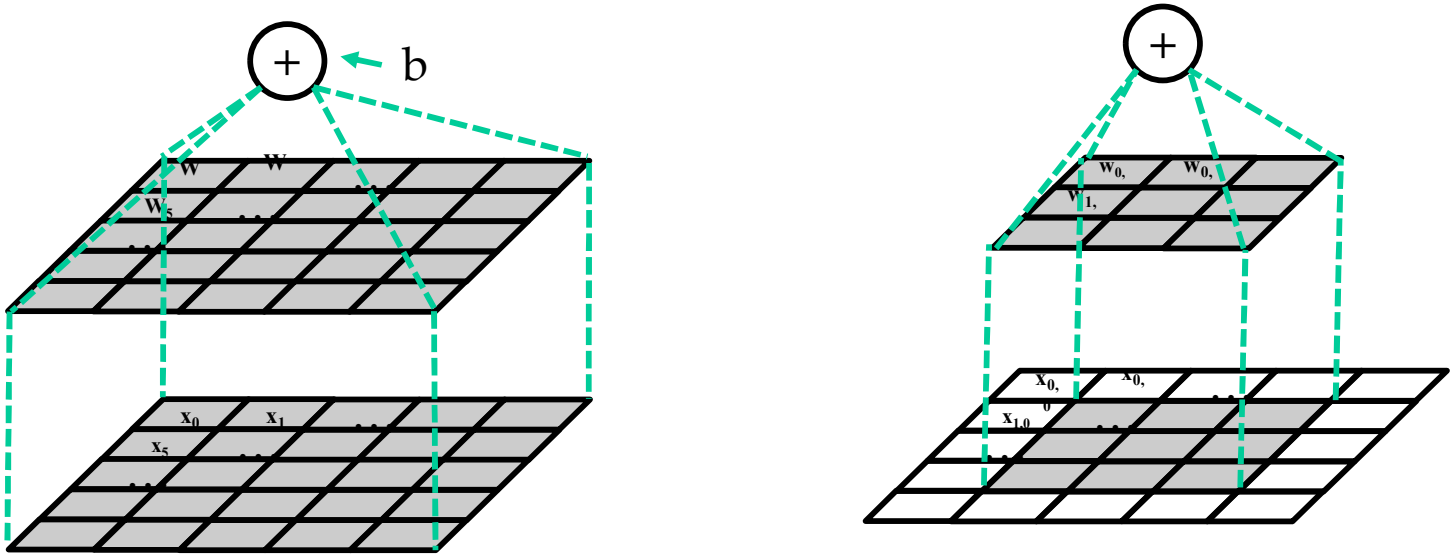
    gives **~4B weights total**!
- Need >1 hidden layer? Bigger images?

**Too much computation**, and **too much memory**.

# Apply Idea of Convolution to Neural Networks

Feature detection in image processing
uses convolution.

**Can we do the same in neural networks?**



# Differences Between CNN and Traditional Network

Convolutional neural networks (CNNs) offer three key differences from traditional layers:

- **spatial relationship between inputs and outputs** (usually squashed into 1D),
- any **output affected only by** spatially **local inputs** (within mask size), and
- **weights shared across** all **outputs** (common mask weights).

# Convolution Naturally Supports Varying Input Sizes

**Traditional** perceptron **layers**

- **have fixed structure**, so
- number of inputs / outputs is fixed.

**Convolution enables variably-sized inputs**

(observations of the same kind of thing)

- audio recordings of different lengths,
- images with more/fewer pixels, and so forth.

**We focus on CNN layers in Lab 7.**

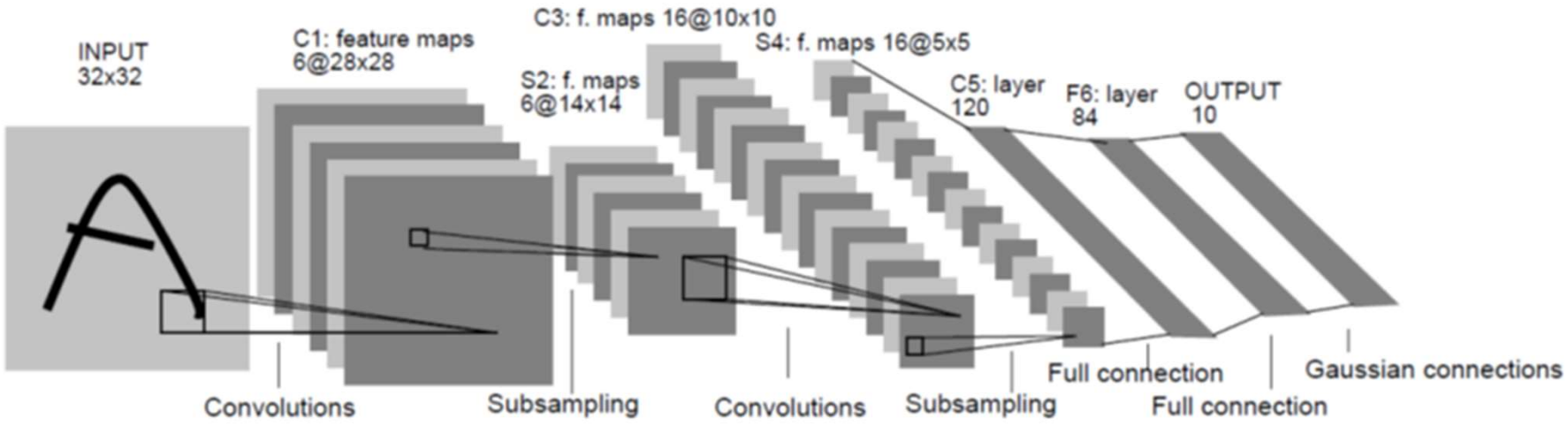
# Example Convolution Inputs

| | Single-channel | Multi-channel |
|---|---|---|
| 1D | audio waveform | Skeleton animation data:<br>1-D joint angles for each joint |
| 2D | Fourier-transformed audio data<br>Convolve over frequency axis:<br>invariant to frequency shifts<br>Convolve over time axis:<br>invariant to shifts in time | Color image data:<br>2D data for R,G,B channels |
| 3D | Volumetric data<br>(example: medical imaging) | Color video: 2D data across 1D time for R,G,B channels |

Deeplearningbook.org, ch 9, p 355

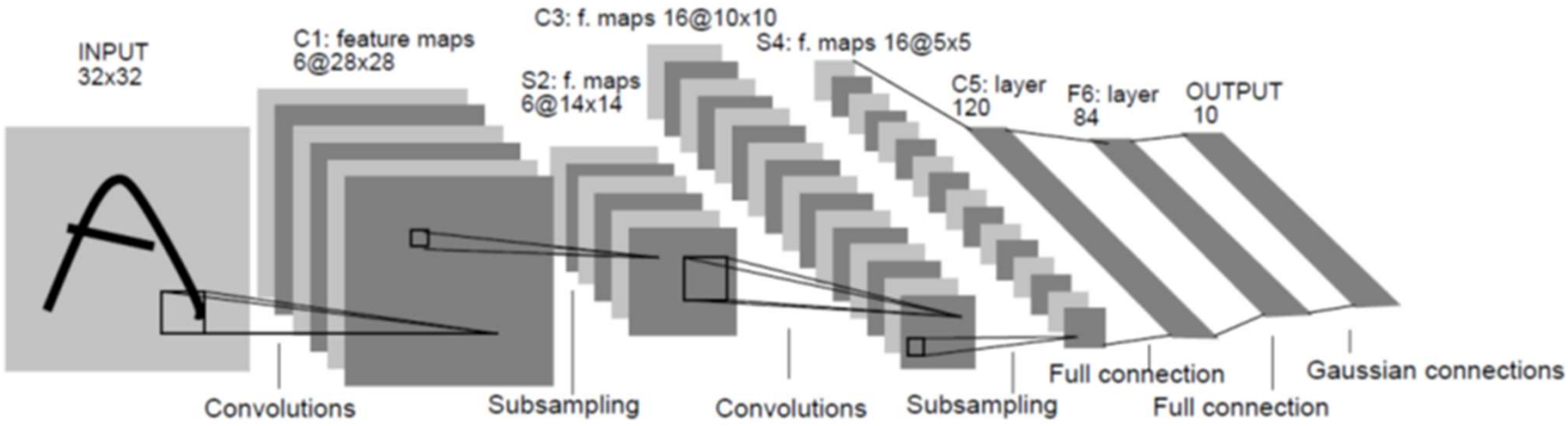
# Convolution Layers in LeNet-5

LeNet-5 has three convolution layers (C1,C3,C5) used to detect features (all) and break symmetry (C3).



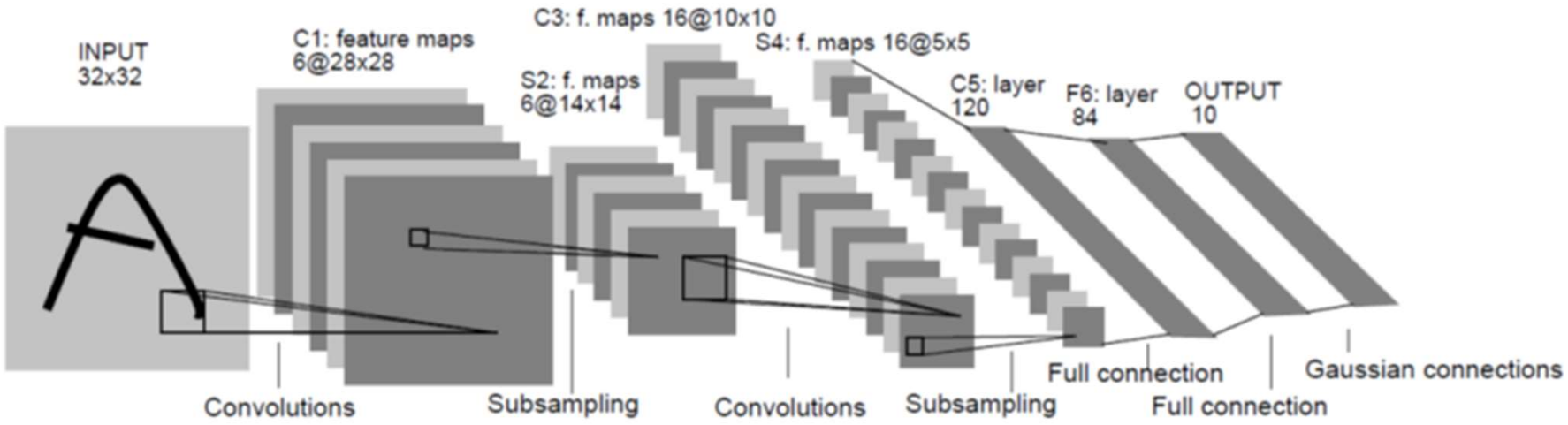
# Subsampling Layers in LeNet-5

The subsampling layers (S2,S4) help with scale invariance and reduce the amount of data without losing much information.



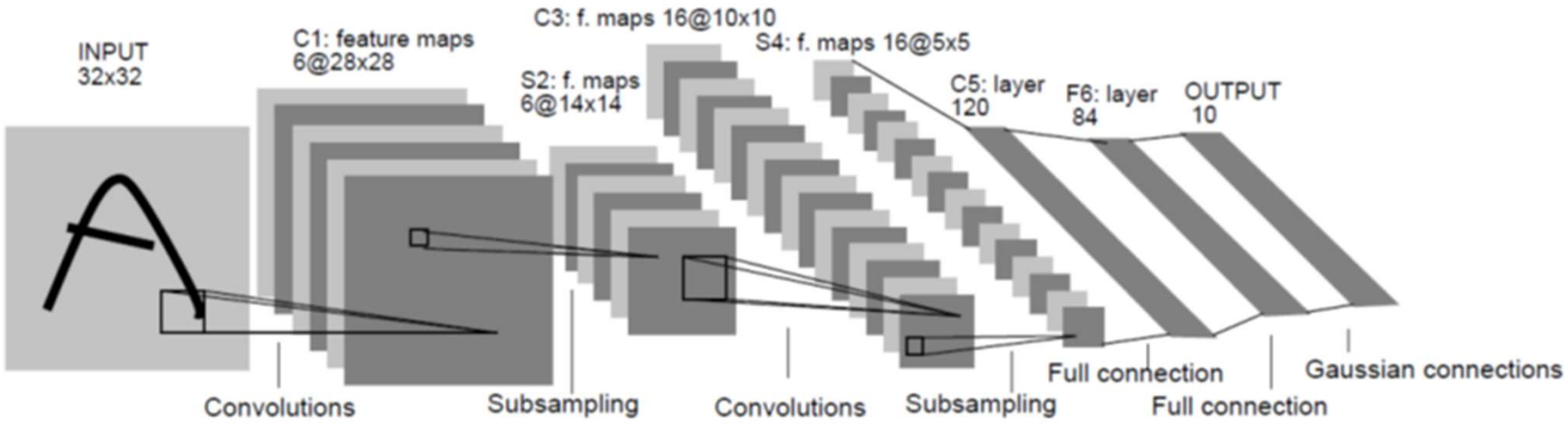
# Fully Connected Layer in LeNet-5

The fully connected layer (F6) is the "hidden" layer in a traditional neural network—implements the unknown function from features to digits.



# OUTPUT Layer in LeNet-5

The OUTPUT layer is (F6) uses "Radial Basis Functions" to push digit identification back into the network.



# Brains Evolve Architecturally to Learn Faster

**Neural network training is**
**like training and evolving brains.**

Over time, our brains have evolved

- to be good at learning things that are important,
- such as human language and physics.

One can actually identify connections across human languages to physical concepts, such as the differences in behavior between solids and liquids, spatial reasoning, and other important features of our universe.

# Neural Networks Train to Produce Output

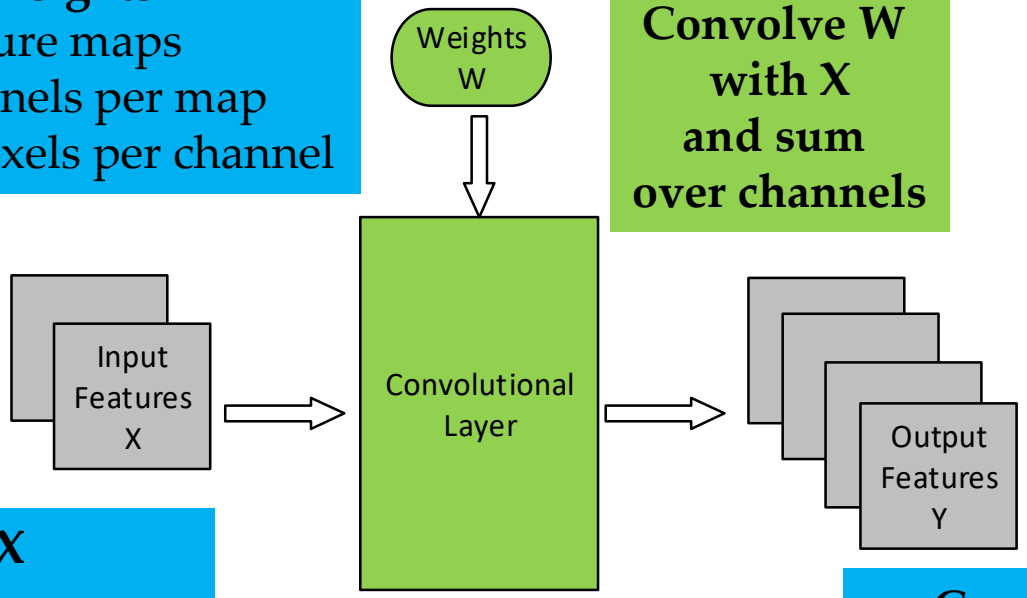
**In neural networks**,

- **creating a desired output** layer
- that represents information (as labels)
- then **allows training to produce that information**.

(Which then sometimes tells us about the network architecture needed for success, but we won't look at that.)

# Forward Propagation in a Convolution Layer

**Weights W**
M feature maps
C channels per map
K×K pixels per channel




**Convolve W with X and sum over channels**

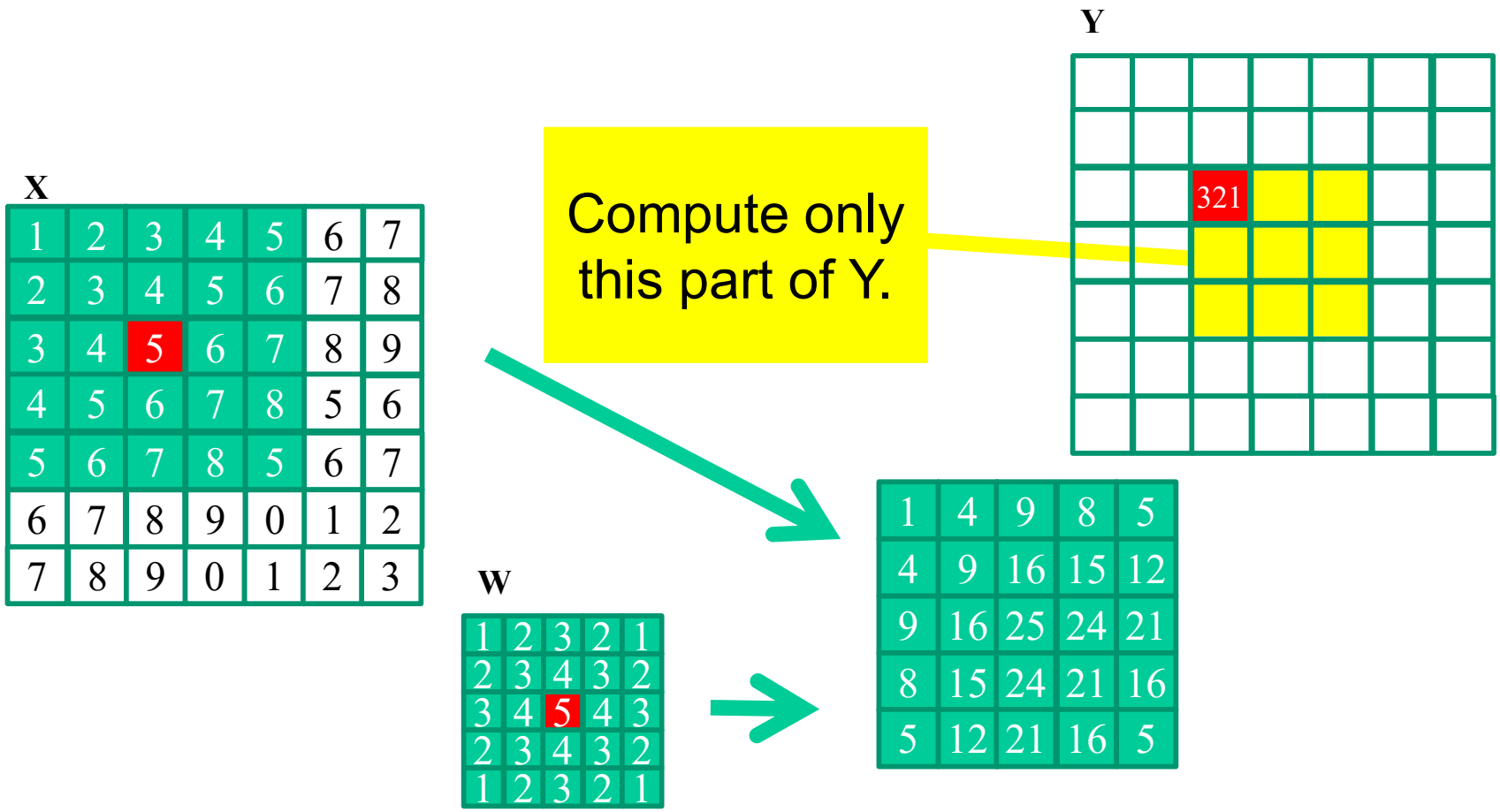
**Output Size**
$H_{out} = H - K + 1$
$W_{out} = W - K + 1$

**Input X**
B images
C channels per image
H×W pixels per channel

Input-output feature map connections may or may not be complete.

**Convolution Output Y**
B images
M features per image
$H_{out} \times W_{out}$ values per feature

# Only Compute with Full Mask/Kernel



# Inputs Inflated to Provide Padding

In practice, **inputs** are sometimes **inflated to prepare for** this type of **convolution**!

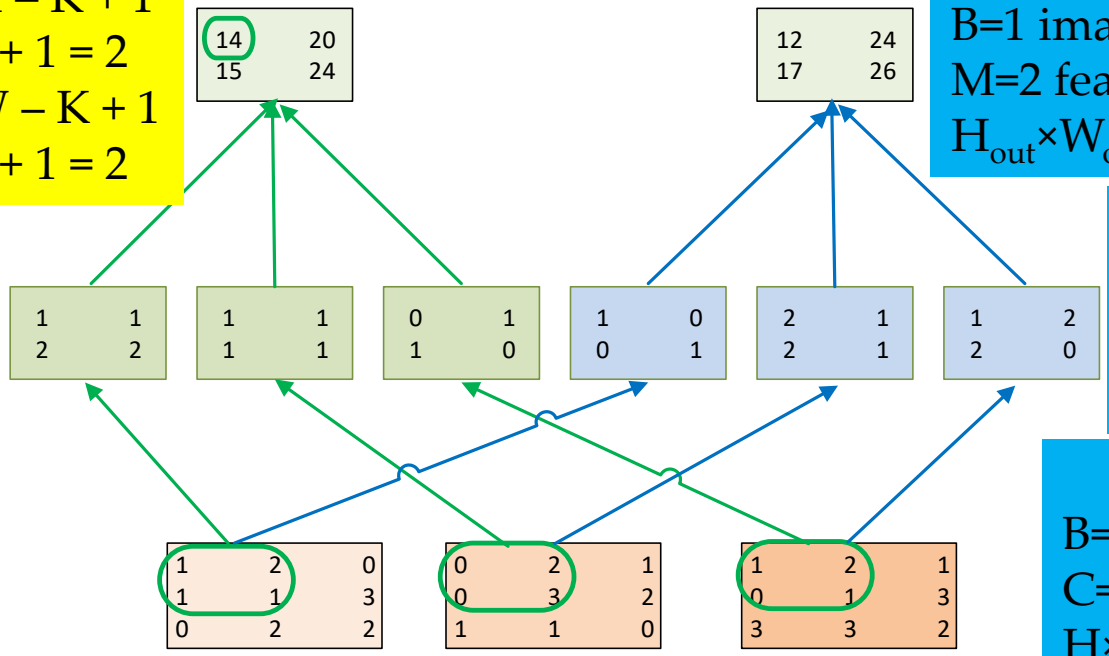
LeNet-5 input

- starts as **20×20 monochrome**, which are
- scaled to **28×28**, which introduces antialiasing and thus **8-bit grayscale**, and then
- **padded** to **32×32 to ensure** that **convolution** filters can **center on stroke data**.

# Example of the Forward Path of a Convolution Layer

**Output Size**

$H_{out} = H - K + 1$
$= 3 - 2 + 1 = 2$
$W_{out} = W - K + 1$
$= 3 - 2 + 1 = 2$




**Convolution Output Y**

B=1 image
M=2 features per image
$H_{out} \times W_{out} = 2 \times 2$ values per feature

**Weights W**

M=2 feature maps
C=3 channels per map
K×K=2×2 pixels per channel

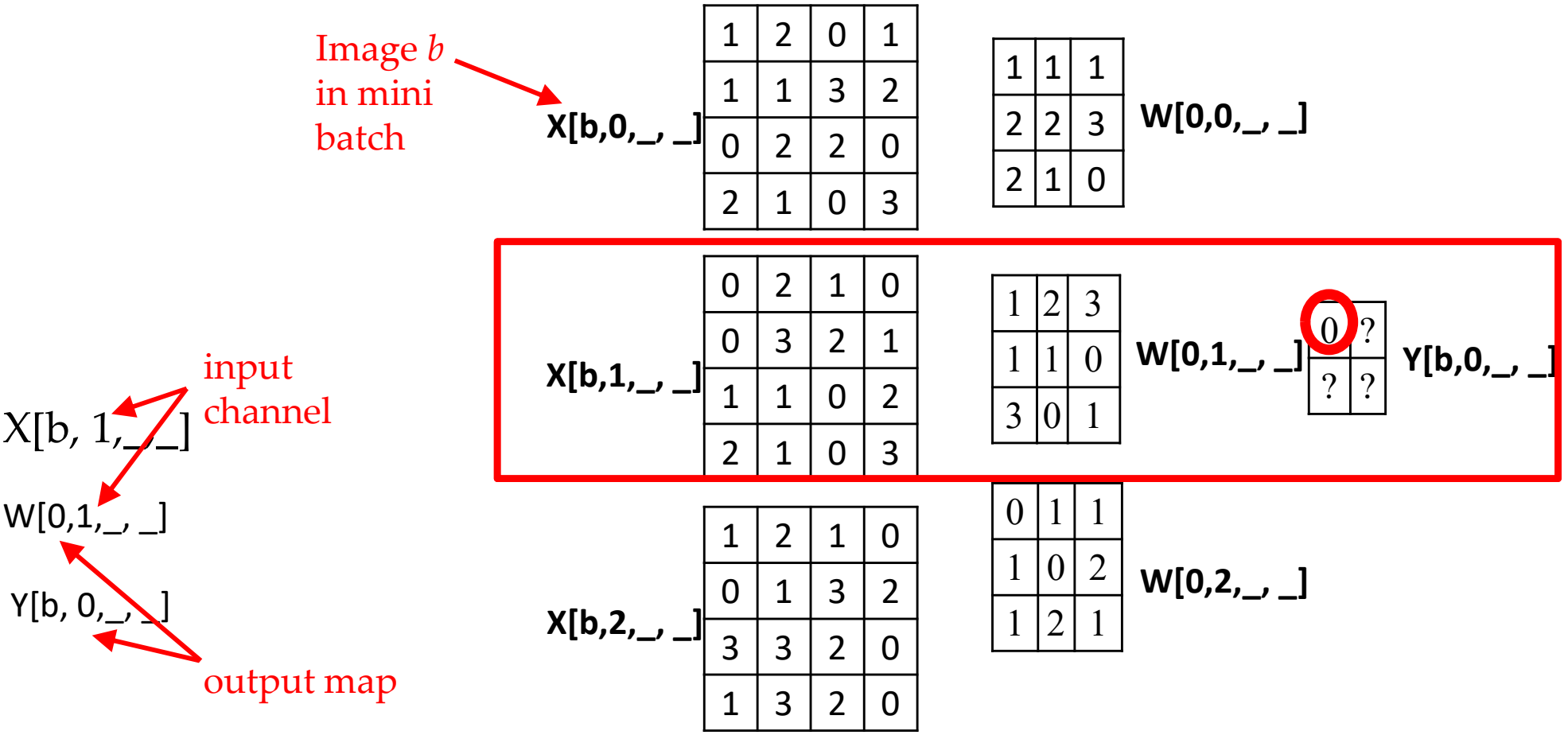
**Input X**

B=1 image
C=3 channels
H×W=3×3 pixels per channel

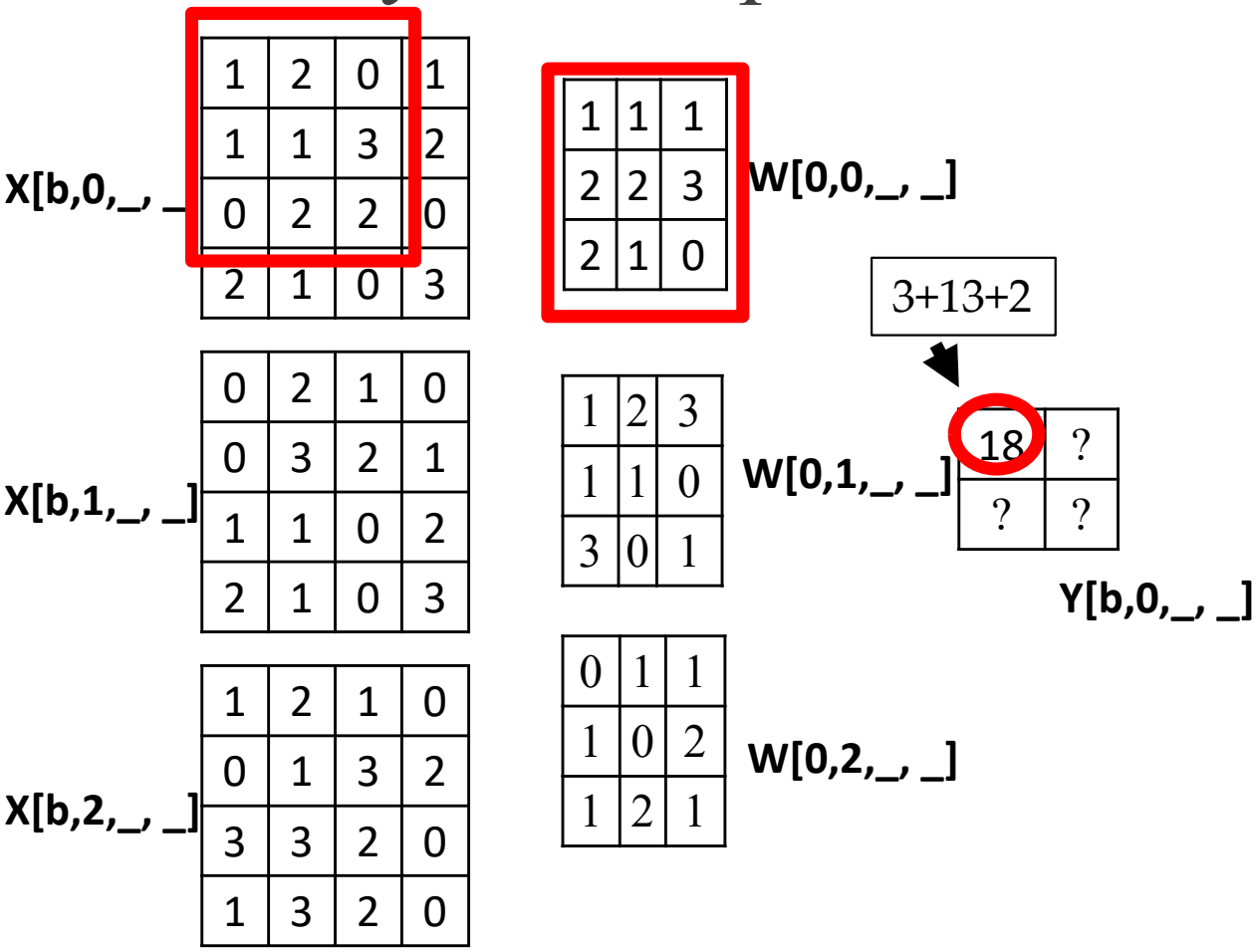
# Sequential Code: Forward Convolutional Layer

```
void convLayer_fwd(int B, int M, int C, int H, int W, int K,
                   float* X, float* W, float* Y) {
  int H_out = H - K + 1;                    // calculate H_out, W_out
  int W_out = W - K + 1;

  for (int b = 0; b < B; ++b)               // for each image
    for(int m = 0;  m < M;  m++)            // for each output feature map
      for(int h = 0; h < H_out; h++)        // for each output value (two loops)
        for(int w = 0; w < W_out; w++) {
          Y[b, m, h, w] = 0.0f;             // initialize sum to 0
          for(int c = 0; c < C; c++)        // sum over all input channels
            for(int p = 0; p < K; p++)      // KxK filter
              for(int q = 0; q < K; q++)
                Y[b, m, h, w] += X[b, c, h + p, w + q] * W[m, c, p, q];
       }
}
```

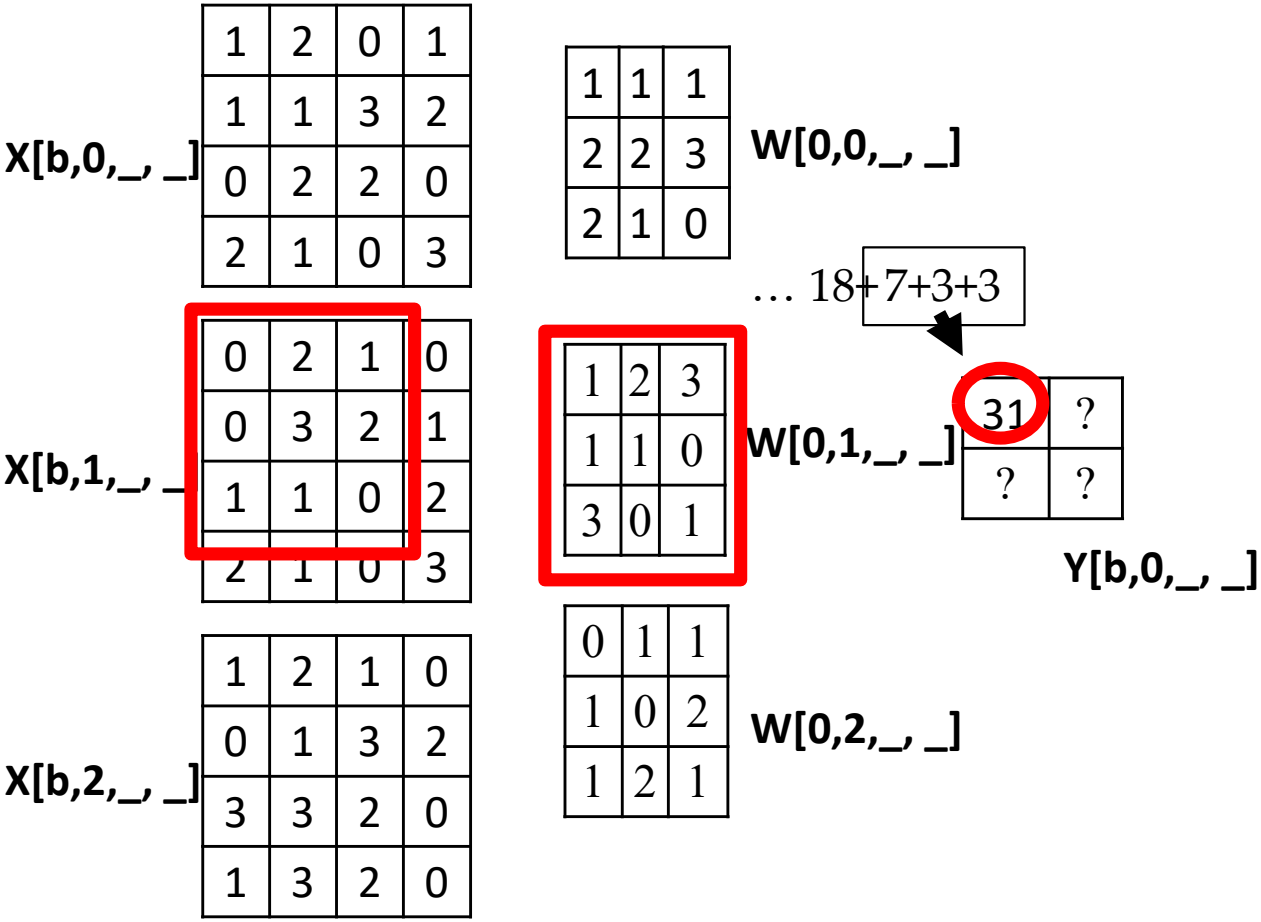
# A Small Convolution Layer Example



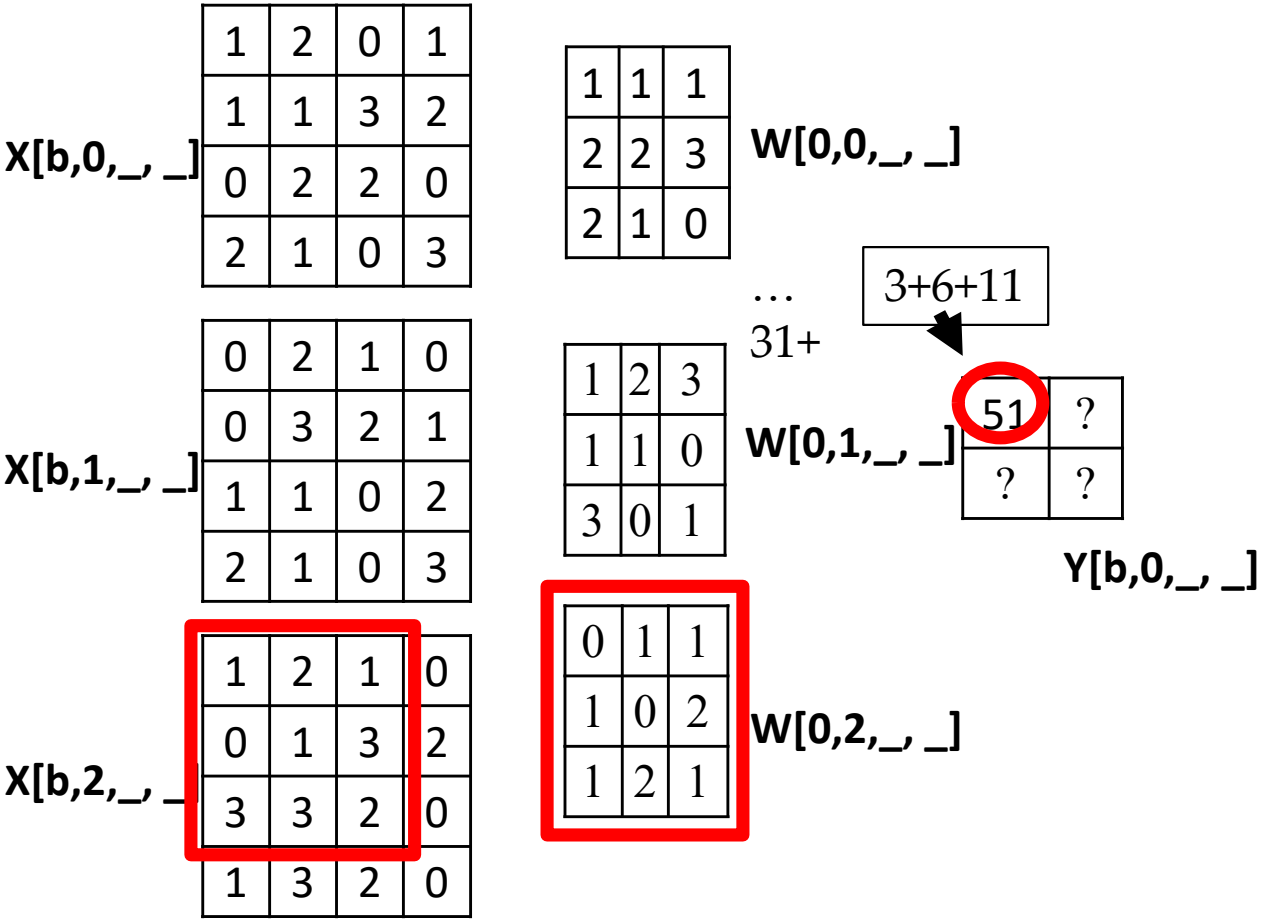
# A Small Convolution Layer Example c = 0



# A Small Convolution Layer Example c = 1



# A Small Convolution Layer Example c = 2



# Parallelism in a Convolution Layer

**Output feature maps** can be calculated in parallel, but usually a small number—not sufficient to utilize GPU.

All **output** feature map **pixels** can be calculated in parallel:

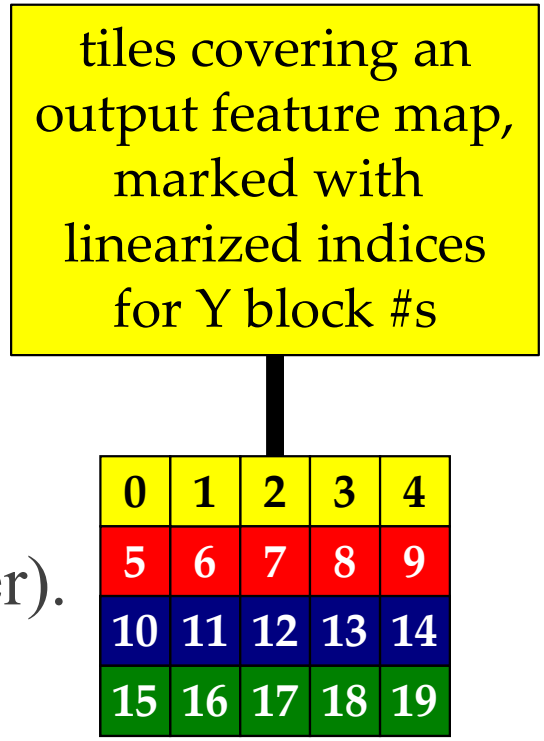
- all rows in parallel, and
- all pixels in each row in parallel;
- a large number, but diminishes in deeper layers.

All **input feature maps** can be processed in parallel, but need atomic operations or tree reduction.

**Different layers may demand different strategies.**

# Design of a Basic Kernel

- Each block computes
  - a tile of output pixels for one feature
  - **TILE_WIDTH** pixels in each dimension.
- Grid's X dimension maps to **M** output feature maps.
- Grid's Y dimension maps to the tiles in the output feature maps (linearized order).
- (Grid's Z dimension is used for images in batch, which we omit from slides.)



# Example of Block Y Dimension Mapping

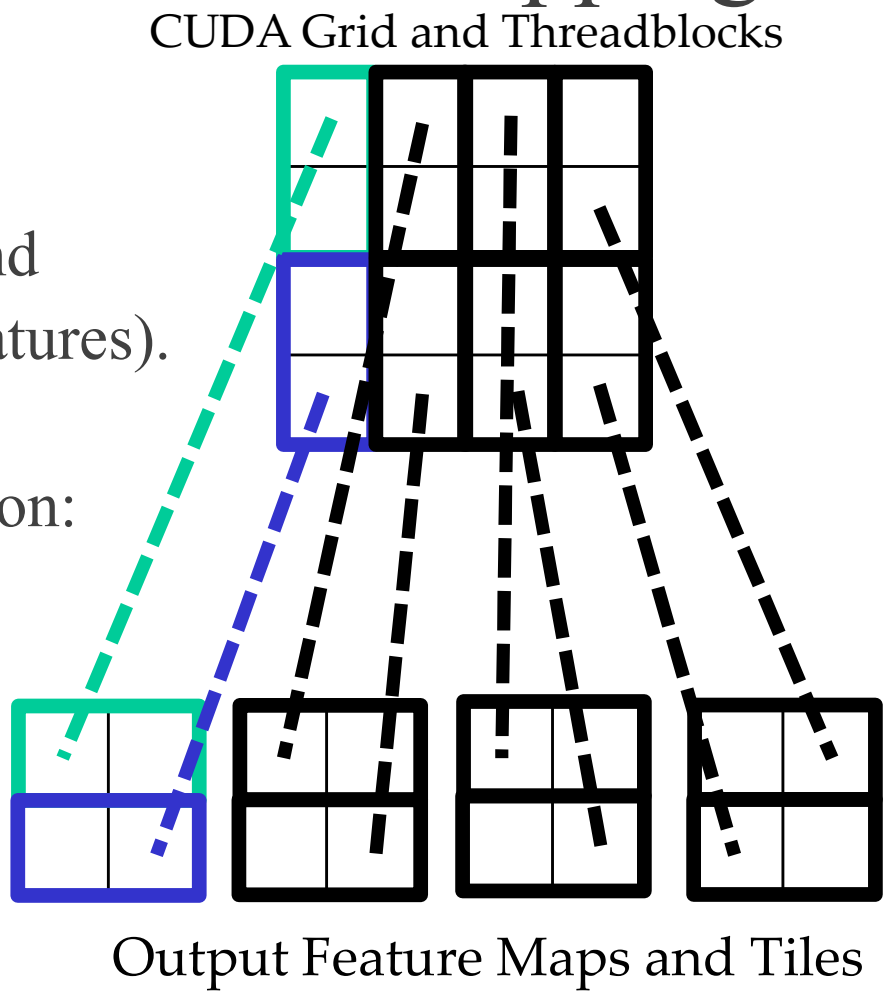
Assume

- **M = 4** (4 output feature maps),
- thus 4 blocks in the X dimension, and
- **W_out = H_out = 8** (8×8 output features).

If **TILE_WIDTH = 4**,

we also need 4 blocks in the Y dimension:

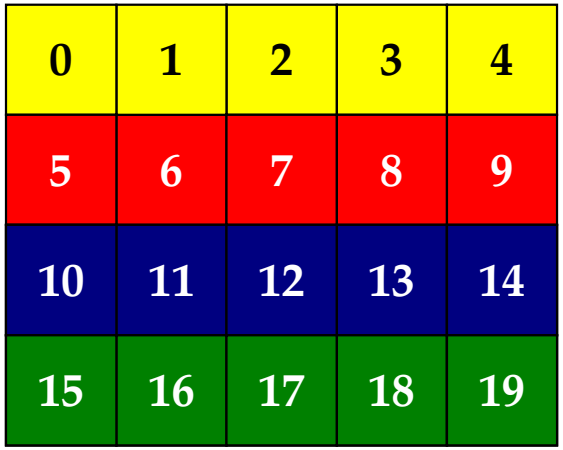
- for each output feature,
- top two blocks in each column calculates the top row of tiles, and
- bottom two calculate the bottom row.



# Host Code for a Basic Kernel: CUDA Grid

Consider an output feature map:

- width is **W_out**, and
- height is **H_out**.
- Assume these are multiples of **TILE_WIDTH**.




Let **X_grid** be the number of blocks needed in X dim (5 above).

Let **Y_grid** be the number of blocks needed in Y dim (4 above).

# Host Code for a Basic Kernel: CUDA Grid

(Assuming W_out and H_out are multiples of TILE_WIDTH.)

```
// 4 for small examples.
#define TILE_WIDTH 16
// horiz. and vert. tiles per output map
W_grid = W_out/TILE_WIDTH;
H_grid = H_out/TILE_WIDTH;
Y = H_grid * W_grid;
// output tile for untiled code
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(M, Y, 1);
ConvLayerFwd_Kernel<<<gridDim,blockDim>>>(...);
```

# Partial Kernel Code for a Convolution Layer

```
__global__ void ConvLayerFwd_Basic_Kernel
    (int C, int W_grid, int K, float* X, float* W, float* Y)
{
    int m = blockIdx.x;
    int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.y;
    int w = (blockIdx.y % W_grid) * TILE_WIDTH + threadIdx.x;
    float acc = 0.0f;
    for (int c = 0;  c < C; c++) {  // sum over all input channels
       for (int p = 0; p < K; p++)  // loop over KxK filter
          for (int q = 0; q < K; q++)
             acc += X[c, h + p, w + q] * W[m, c, p, q];
    }
    Y[m, h, w] = acc;
}
```

# Some Observations

**Enough parallelism**

- if the total number of pixels
- across all output feature maps is large
- (often the case for CNN layers).

Each input tile

- loaded M times (number of output features), so
- **not efficient in global memory bandwidth,**
- but block scheduling in X dimension should give cache benefits.

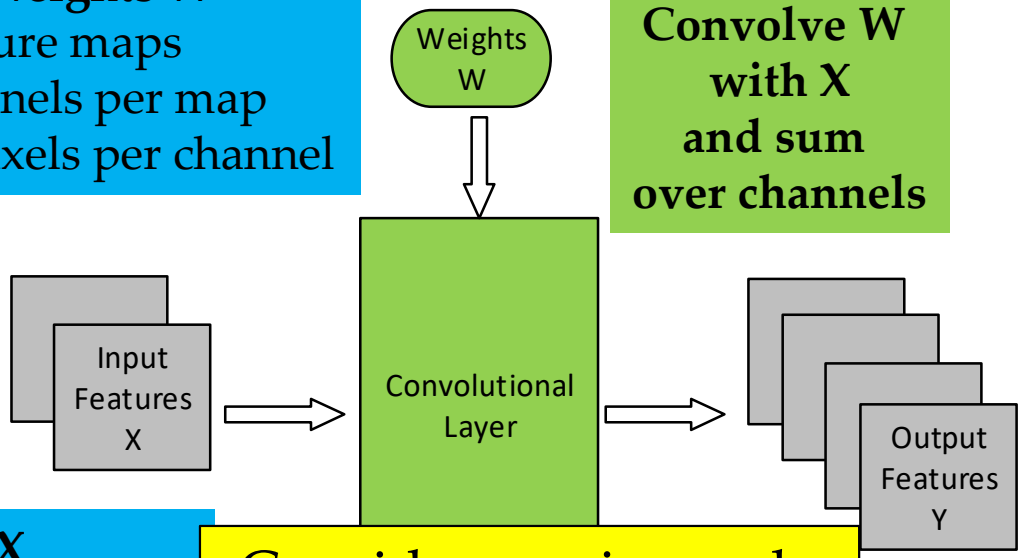
# Strategies for Lab 7

1. Use tiled 2D convolution with masks in constant memory.
2. Switch to matrix multiply (explanation follows):
   - "unroll" input to enable use of matrix multiplication and
   - use shared memory tiling to perform multiplication.
3. Unroll during load:
   - integrate "unrolling" into loading a matrix tile
   - (unrolled matrix is only conceptual).
4. Use joint register-shared memory tiling.

Code must work for any credit,
but (only) last step needed for full credit.

# Thread Computes One Pixel of One Output Feature

**Weights W**
M feature maps
C channels per map
K×K pixels per channel




**Convolve W with X and sum over channels**

**Output Size**
$H_{out} = H - K + 1$
$W_{out} = W - K + 1$

**Input X**
B images
C channels per image
H×W pixels per channel

Consider one image b. One thread computes one value of Y for one feature.

**Convolution Output Y**
B images
M features per image
$H_{out} \times W_{out}$ values per feature

# Reorganize Input X for Matrix Multiplication

**Input X**
B images
C channels per image
H×W pixels per channel

**Weights W**
M feature maps
C channels per map
K×K pixels per channel

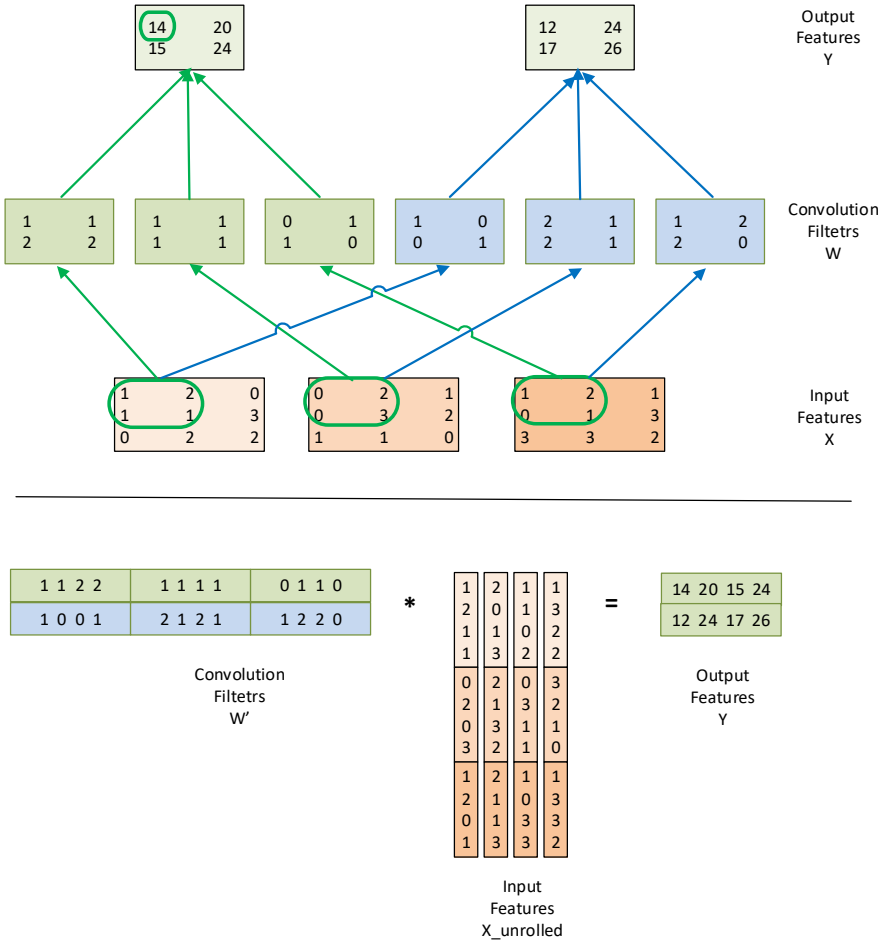
**Multiply W by unrolled X**

Each thread needs a K×K block of X for each input channel.

**Unrolled Input X**
B images
$H_{out}$×$W_{out}$ pixels per image
C×K×K values per pixel

Note: Likely to be larger than X, since values are replicated.

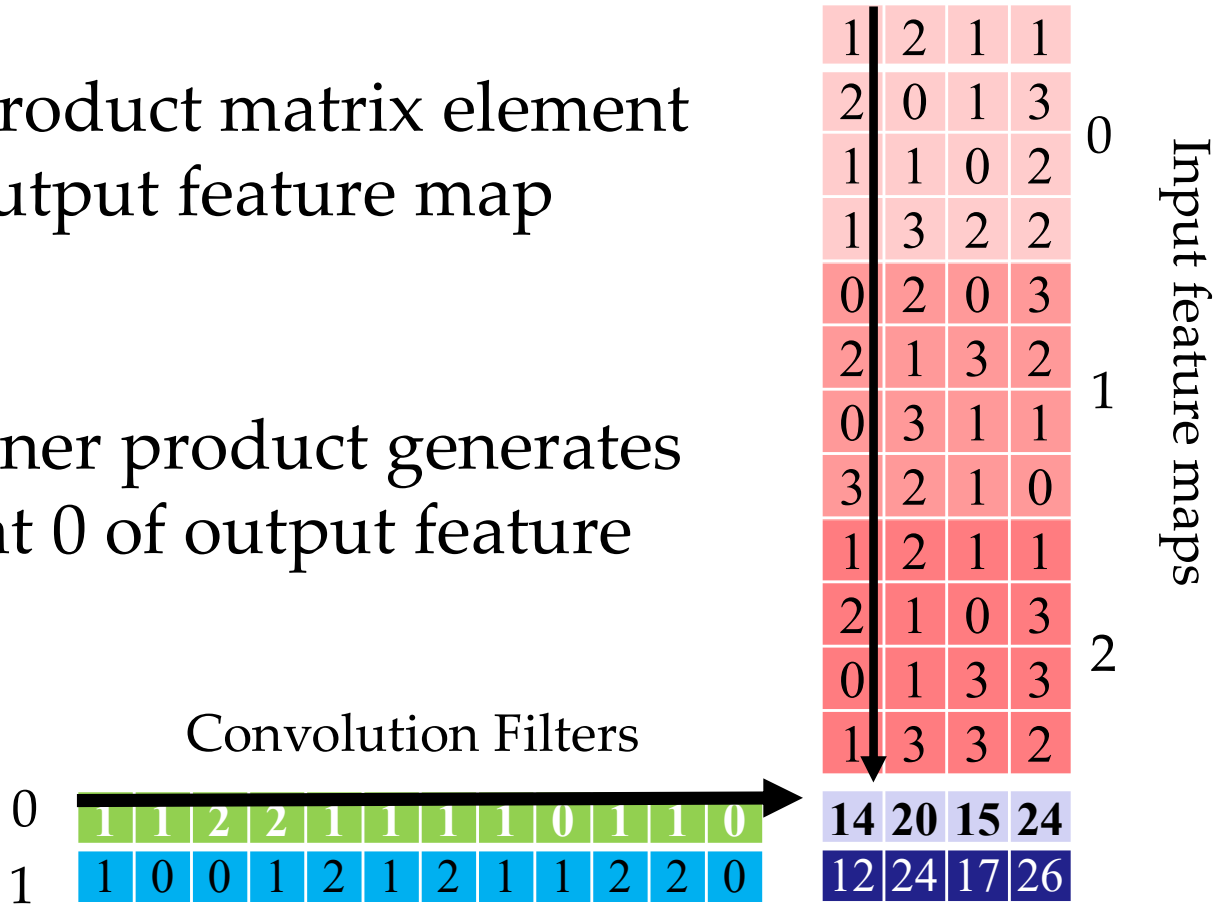
# Matrix Multiplication for a Convolution Layer



# Simple Matrix Multiplication

Each product matrix element is an output feature map pixel.

This inner product generates element 0 of output feature map 0.



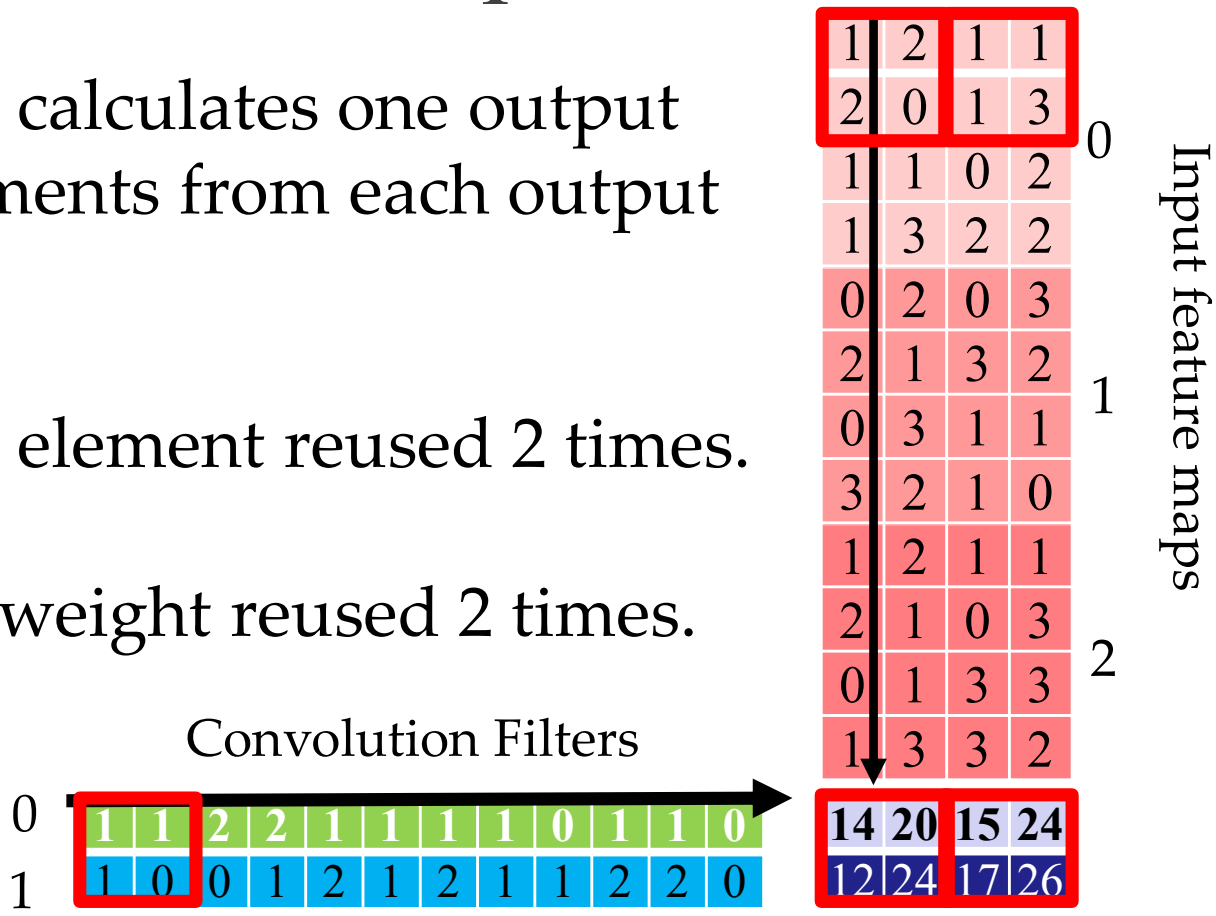

 

# Tiled Matrix Multiplication 2×2 example

Each block calculates one output tile – 2 elements from each output map,

Each input element reused 2 times.

Each filter weight reused 2 times.

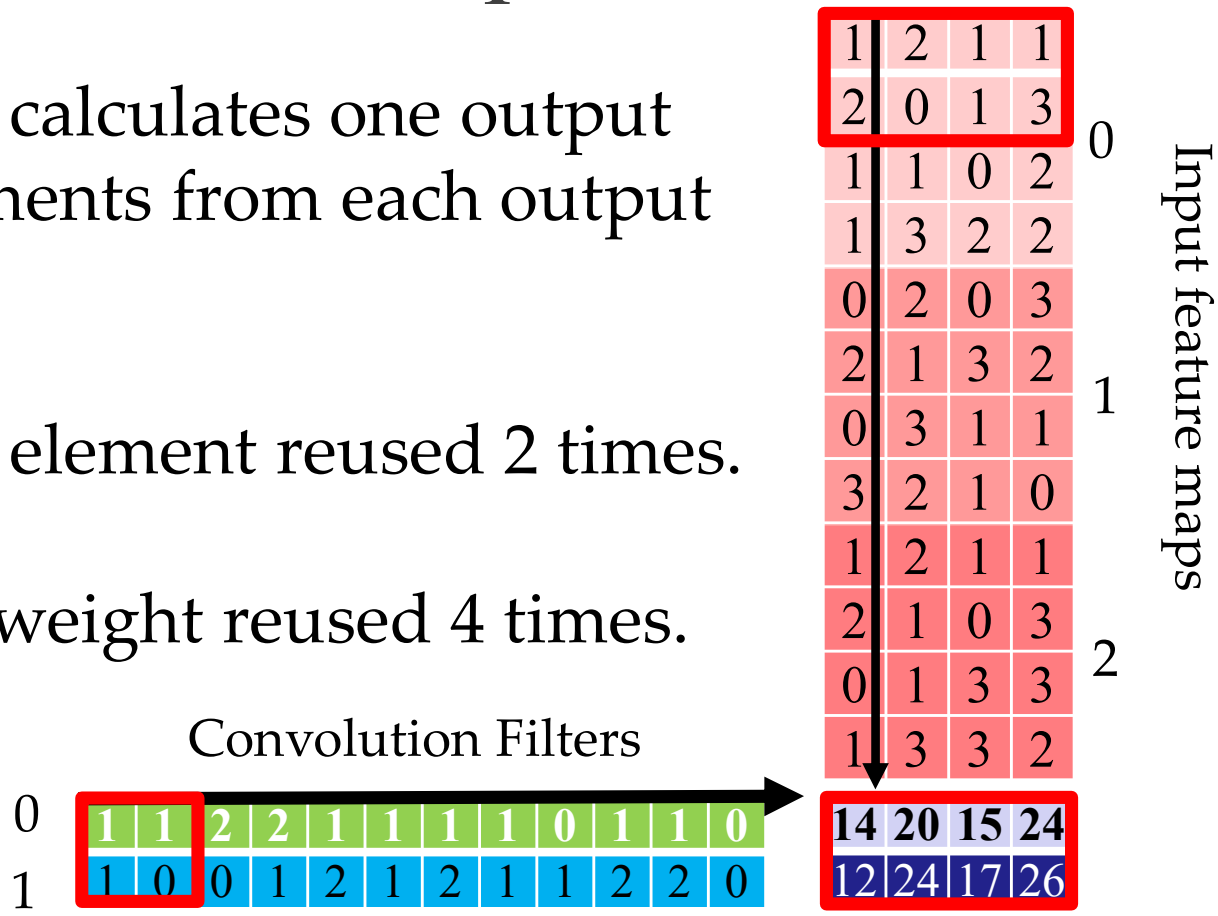


# Tiled Matrix Multiplication 2×4 example

Each block calculates one output tile – 4 elements from each output map.

Each input element reused 2 times.

Each filter weight reused 4 times.



# Unrolling May Boost Your Performance

As we will show, the **advantages of "unrolling"** are

- to **leverage reuse** of input values **across output features**
- without the complexity of 3D convolution, and
- to fit **joint tiling** into a **known pattern**.

(You may be able to do as well or better by solving the general convolution, but previous student teams have not succeeded.)

# Analyzing the Amount of Input Replication

An output feature map has $\mathbf{H_{out}W_{out}}$ elements, and

- each element requires a copy of $\mathbf{K^2}$ input elements.
- Each copy is used across all **M** output feature maps.

The **number of unrolled input elements** is then

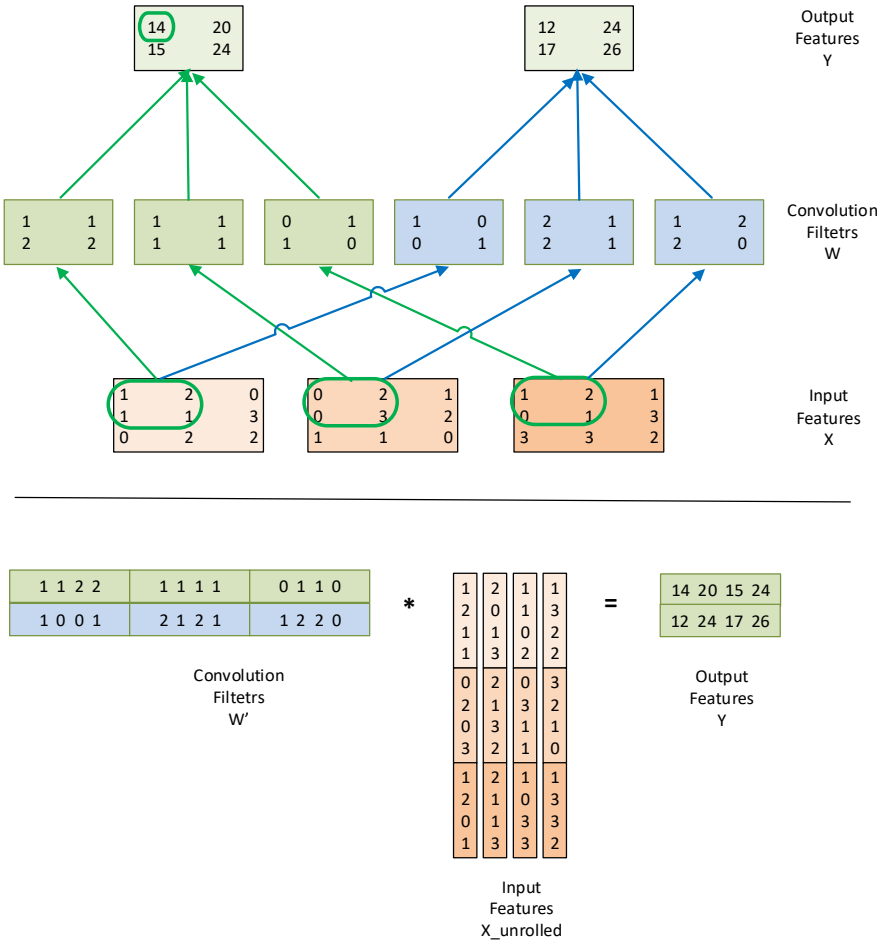
- $\mathbf{H_{out}W_{out}K^2}$ for each of the **C** input feature maps.
- The original input feature maps contain a total of $\mathbf{C(H_{out} + K - 1)(W_{out} + K - 1)}$ elements, so $\mathbf{H_{out}W_{out}K^2 / [(H_{out}+K-1)(W_{out}+K-1)]}\times$ **replication**.

# Analysis of Small Example

For our toy example,

- $H_{out} = 2$, $W_{out} = 2$, and K = 2.
- C=3 input features.
- "Unrolled" input contains $3 \times 2^4 = 48$ elements.
- Replication factor is $48/3^3 = 1.78\times$.



# LeNet-5 C1 Layer Unrolling Replication: 19.1×

LeNet-5 C1

- **32×32** input, **C=1** feature
- **5×5** filters (**K=5**)
- **28×28** outputs, **M=6** features

Filter matrix: $\mathbf{M \times CK^2 = 6 \times 25}$

Unrolled input matrix: $\mathbf{CK^2 \times H_{out}W_{out} = 25 \times 784}$

Input **replication**: $\mathbf{25 \times 784/1024 = 19.1\times}$

# LeNet-5 C3 Layer Unrolling Replication: 12.8×

LeNet-5 C3

- **14×14** input, **C=6** features
- **5×5** filters (**K=5**)
- **10×10** outputs, **M=16** features

Filter matrix: $\mathbf{M \times CK^2 = 16 \times 150}$

Unrolled input matrix: $\mathbf{CK^2 \times H_{out}W_{out} = 150 \times 100}$

Input **replication**: $\mathbf{150 \times 100/(6 \times 14^2) = 12.8\times}$

# LeNet-5 C5 Layer Unrolling Replication: 1×

LeNet-5 C5

- **5×5** input, **C=16** features
- **5×5** filters (**K=5**)
- **1×1** outputs, **M=120** features

Filter matrix: $\mathbf{M \times CK^2 = 120 \times 400}$

Unrolled input matrix: $\mathbf{CK^2 \times H_{out}W_{out} = 400 \times 1}$

Input **replication**: $\mathbf{400 \times 1/(16 \times 5^2) = 1\times}$

# Reuse in 2D Tiled Convolution

**Consider 2D convolution** with shared memory tiling.

- Each **output tile** has **TILE_WIDTH$^2$** elements.
- Each **input tile** has **(TILE_WIDTH + K – 1)$^2$** elements.
- Number of input element **accesses** is **TILE_WIDTH$^2$ × K$^2$**.
- Input value **reuse** is then

**K$^2$ × TILE_WIDTH$^2$ / (TILE_WIDTH + K – 1)$^2$**.

# Convolution Reuse Matches Unrolling Replication!

Theoretical reuse of input values for LeNet-5 is then

- C1: **19.1×** (32×32 input tiles)
- C2: **12.8×** (14×14 input tiles)
- C3: **1×** (5×5 input tiles)

**Remember those numbers?**

**Those are** the **replication** factors **for** the **unrolled inputs**. If one unrolls conceptually, one obtains most/all of this reuse as well. If one actually replicates, this reuse is lost.

# Matrix-Multiplication Enables Additional Reuse

**What about input reuse with matrix multiplication?**

Filter matrix is **$M \times CK^2$**,

- so **theoretical reuse is M**
- for each value from the input feature map matrix
- (also limited by output tile size).

In LeNet-5, C1 gives 6, C3 gives 16, and C5 gives 120,

- which are the **numbers M of output features**.
- Note that C5 is actually a matrix-vector multiply.

# Conceptual Unrolling May Help Performance

So while **actually unrolling** the **inputs**

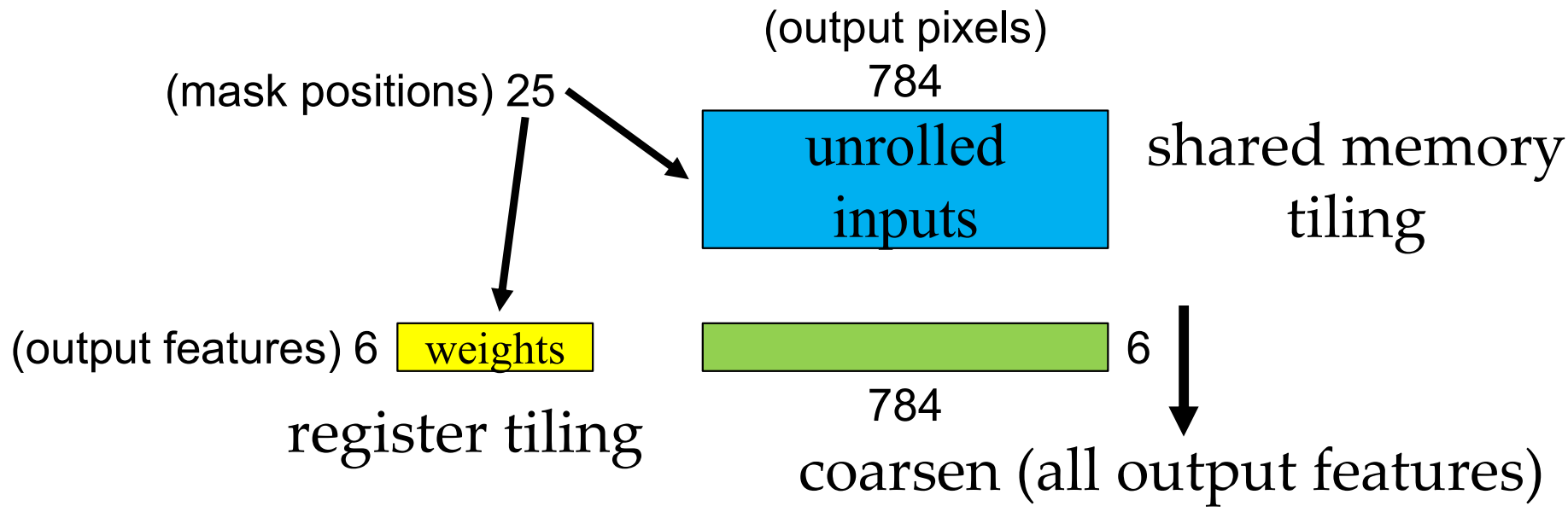
- **may** provide a **boost** in **coalesced accesses**,
- the **cost** of doing so **is high**, both
  - in terms of preprocessing overhead
  - as well as loss of reuse.

However, performing **conceptual unrolling**

- during matrix tile load
- to transform convolution into matrix multiply
- often **produces** a **better** end **result**,
- particularly **when coupled with joint tiling**.

# Specialization for the LeNet-5 C1 Layer

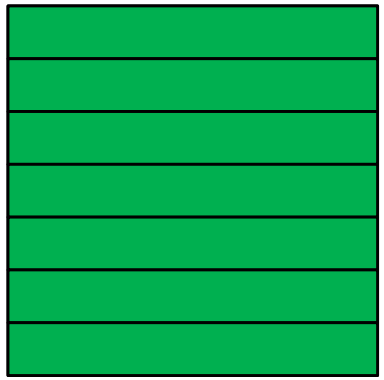
Let's see how we might write **specialized code for** the **LeNet-5 C1** layer. C1 looks like this…



# Block Organization for LeNet-5 C1 Layer

**How many threads in a block?**

**Maybe 4 × 28 = 112?**

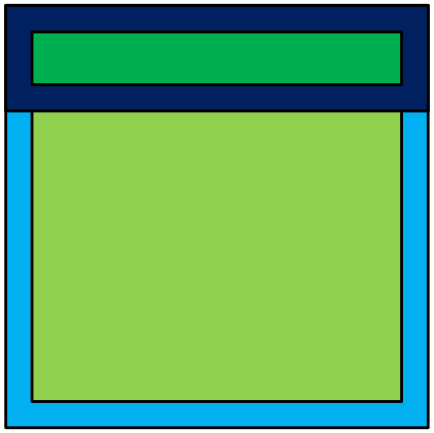


So a **block produces four rows** of each output feature (coarsened across all output features).

And we need **7 thread blocks**.

# Shared Tile for LeNet-5 C1 Layer

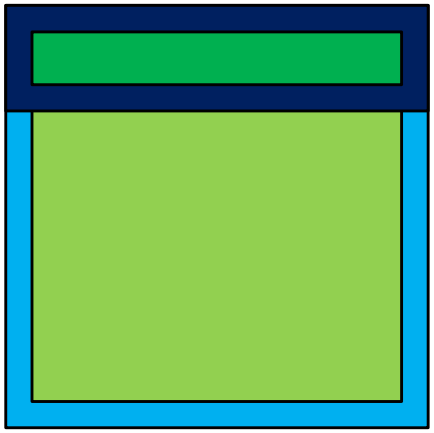
To do so, the **block needs 8 rows of input**.
That's **8 × 32 = 256** elements.



For simplicity, let's use **one tile** and **128 threads** doing **two loads**, with only **112 threads computing** output.

# Pseudo-Code for LeNet-5 C1 Layer

```
Initialize 6 output element sums
Load to shared memory
Load to shared memory
Barrier
If computing (tx < 112)
    loop over 25 mask positions
        loop over 6 output features
            read a weight
            multiply-and-add to an output
    loop over 6 output features
        store element
```



# Coarsening Does Not Match Tiling Choices

**Oops!**

We coarsened the wrong way!

**Every thread loads every weight!**

But those are **in constant memory**, and

- used in exactly the same order
- by every thread
- in every block—should be ok.

# Mini-Batching May be Needed to Fill GPU

**Mini-batching increases work done by** each **kernel** launch:

- group **B** independent inputs, then
- **compose B × C** input features **into** a **larger input** matrix, and
- **solve in parallel**.

Earlier example used Z dimension for mini-batch.

# Some Other Optimizations

- Use streams to overlap
  - reading next set of input feature maps,
  - with processing the current set,
  - and writing the previous set of outputs.
- Create unrolled matrix elements only when they are loaded into shared memory.
- For large inputs, use algorithms such as FFT to implement convolution.

# Rules for Lab 7 Performance Boosting

- In Lab 7, you **MAY optimize** for Titan V **based on** the **test input parameters** given (including the number of images)—one can imagine doing so with C++ templates, although it sounds a little painful.
- You **MAY NOT optimize for** all **weights** being **equal to 1**; your code MUST use stored weights.
- You **PROBABLY want to copy** the **weights to constant memory** for your kernel.

# Training Optimizes Weights to Produce Outcomes

**Now it's time to look at training.**

**Where do the weights come from?**

Training a neural network is

- a fancy version of **curve fitting**.
- Try to **pick weights** (coefficients)
- **to** best **approximate** a **desired** (but discrete and sparse) set of **outcomes**.

# Iterative Tuning Relies on Derivatives

**How do we pick weights?**

**Newton's method in 100,000 dimensions.**

The **network defines a function**.

Optimizing weights is much easier if

- the function produced is
- **continuous,** piecewise **differentiable, and**
- has **non-zero derivatives**.

Remember discussing activation functions?

# Choose a Simple Error Function

**We also have to define the "best" approximation.**
**How do we quantify the difference**
**between desired and actual outcomes?**

Typically, we do so by

- **cast**ing an "**answer**" **into** a **vector space**, then
- using a **simple measure**—
  - sum of absolute differences,
  - sum of squared differences,
  - or something similar—
- **to quantify error**.

# Training Data Used to Evaluate Weights

Recall that **training data** is **labeled**

- with "correct" answers,
- such as a digit from 0 to 9 in LeNet-5.

**Labels** are used

- to **generate** "correct" **vectors**,
- such as a 10-value probability vector in LeNet-5.

Then one **iteratively refine**s weights

**until one finds an acceptable solution**.

# Choosing an Error Function

Many error functions are possible.

For example, **given label *T*** (digit ***T***),

- ***E = 1 – k[T]***,
- the **probability of not classifying as *t***.

**Alternatively**, since our categories are numeric,

we can **penalize quadratically**:

$$E = \sum_{j=0}^{C-1} k[j](j - T)^2$$

Let's **go with the latter**.

# Loss Function Measures Proximity to Training Data

Once we have

- computed vectors for training data and
- chosen an error function,
- the network can be used to compute error per training input.

Given a sequence of training inputs, the **loss function averages the error** (typically as arithmetic mean).

Loss function used **to define an acceptable solution**.

# Why Not Use the Intuitive Measure?

**Intuitive** definition of classifier **success**:

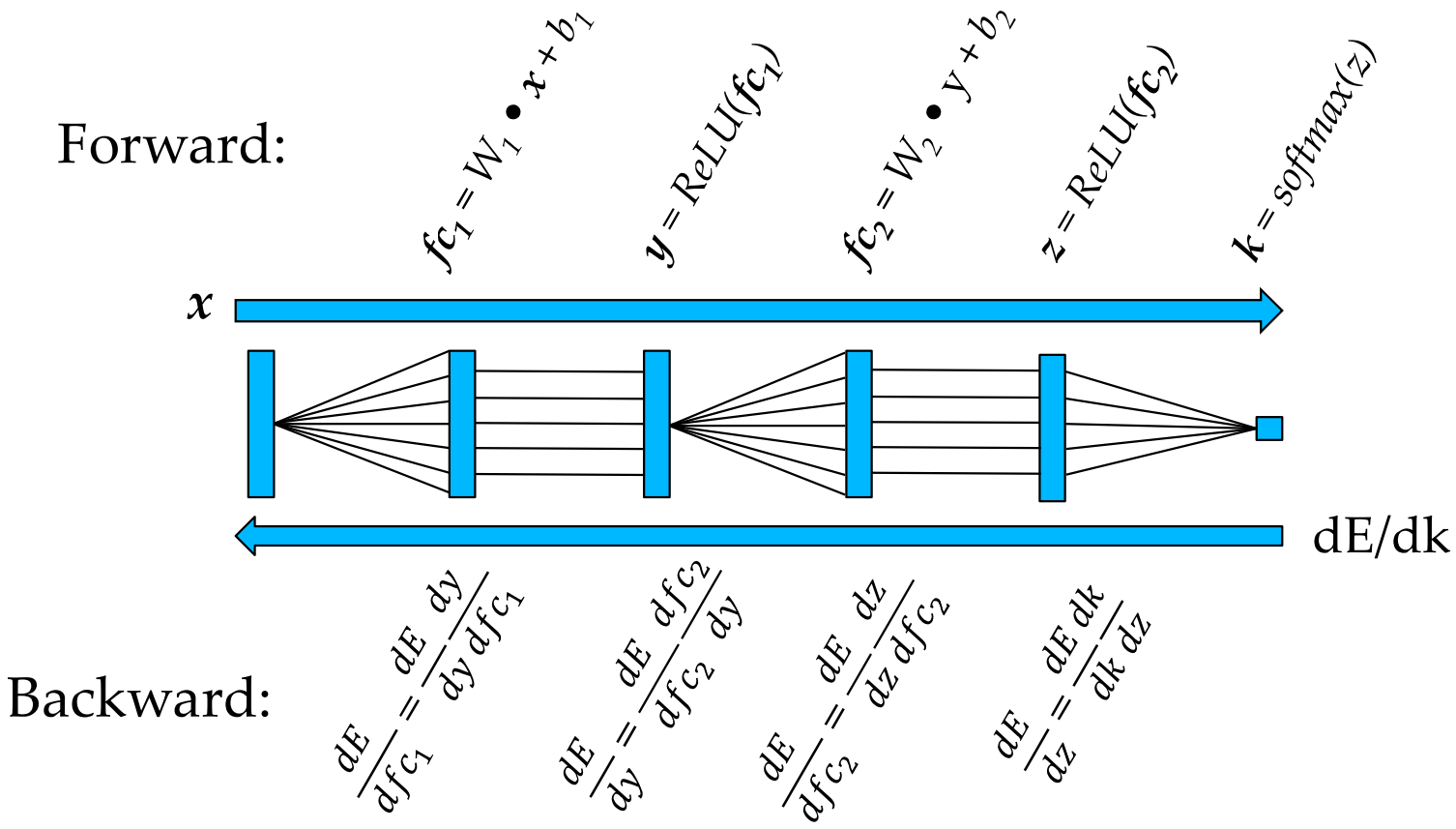
- output **matches** the (unique) **human answer**;
- Report as a percentage across a test set.

That function is a **hard max** (also known as **argmax**):

- **what discrete category has highest probability?**
- Function is piecewise differentiable, but
- has **zero derivative** wherever a derivative is defined,
- thus can**not easily** be **used for training**.

**Error/loss functions** will be different,
but **need to correlate** with intuitive metric.

# Forward and Backward Propagation



# Stochastic Gradient Descent

**How do we calculate the weights?**

One common answer: **stochastic gradient descent**.

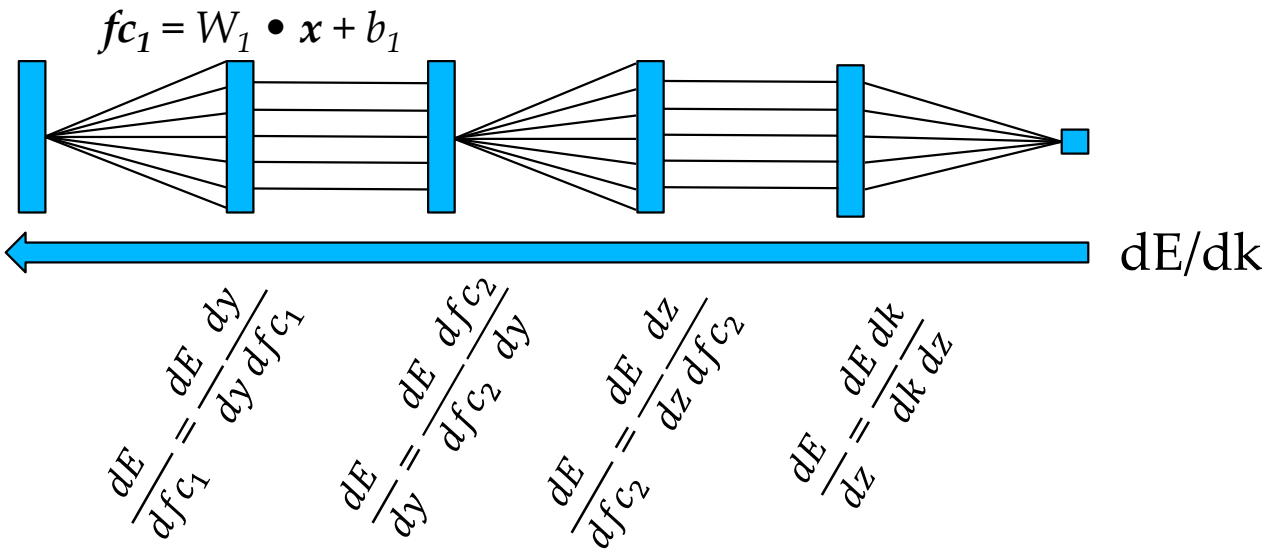
1. **Calculate**
   - **derivative** of sum **of error** ***E***
   - **over all** training **inputs**
   - **for** all network parameters ***θ***.
2. **Change *θ* slightly** in the opposite direction (to decrease error).
3. **Repeat**.

# Stochastic Gradient Descent

- More precisely:
  - **for every input X,**
  - evaluate network to **compute k[i]** (forward),
  - then **use k[i] and label T** (target digit) **to compute error E.**
  - Backpropagate error derivative to **find derivatives for each parameter**.
  - **Adjust Θ to reduce total E: $\Theta_{i+1} = \Theta_i - \varepsilon\Delta\Theta$**
- **(Update ε uses most accurate minima estimation.)**

# Parameter Updates and Propagation




Need propagated error gradient (from backward pass)

Weight update

$$\frac{dE}{dW_1} = \frac{dE}{dfc_1}\frac{dfc_1}{dW_1} = \frac{dE}{dfc_1}x$$

Need input (from forward pass)

# Overview of Gradient Back-Propagation

- Back-propagation is done for all images in the sequence for which the loss function value was generated
- We put each input image through the forward path. This generates input feature maps Xi and output feature maps Yi for all layers.
- We will just use X and Y for simplicity for the rest of this lecture

# Overview of Gradient Back-Propagation

- The process propagates the gradient of the error with respect to the output from the last layer ($\partial E/\partial Y$) towards the first layer through the network.

- Each layer starts with $\partial E/\partial Y$ – gradient with respect to its output feature maps and computes:
  - $\partial E/\partial W$ – gradient of error with respect to its weights
  - $\partial E/\partial X$ – gradient of error with respect to its input feature maps, this becomes the $\partial E/\partial Y$ of the previous layer

# Back Propagation using Chain Rule

- To understand how the error changes with respect to the weights (∂E/∂W) or the inputs (∂E/∂X) in each layer, we can apply the Chain Rule in calculus:

$z = f(y)$ and $y = g(x)$, then

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$$

# Back Propagation using Chain Rule (Example)

- G = W * X + B //Network
- Y = ReLU(G) //Final activation layer
- k[j] = softmax (Y)
- $E = \frac{1}{B}\sum_{b=0}^{B-1}\sum_{j=0}^{C-1} k[b,j](j - T(b))^2$ //Loss function
- B is the number of images, T(b) is the true label, C=10 is the number of categories

- $\frac{\partial E}{\partial W} = \frac{\partial E}{\partial k} \times \frac{\partial k}{\partial Y} \times \frac{\partial Y}{\partial G} \times \frac{\partial G}{\partial W} = \frac{2}{B}\sum_{b=0}^{B-1}\sum_{j=0}^{C-1}(j - T(b)) \times \sum_{m=0}^{C-1} k[b,j](\delta_{j,m} - k[b,m]) \times$

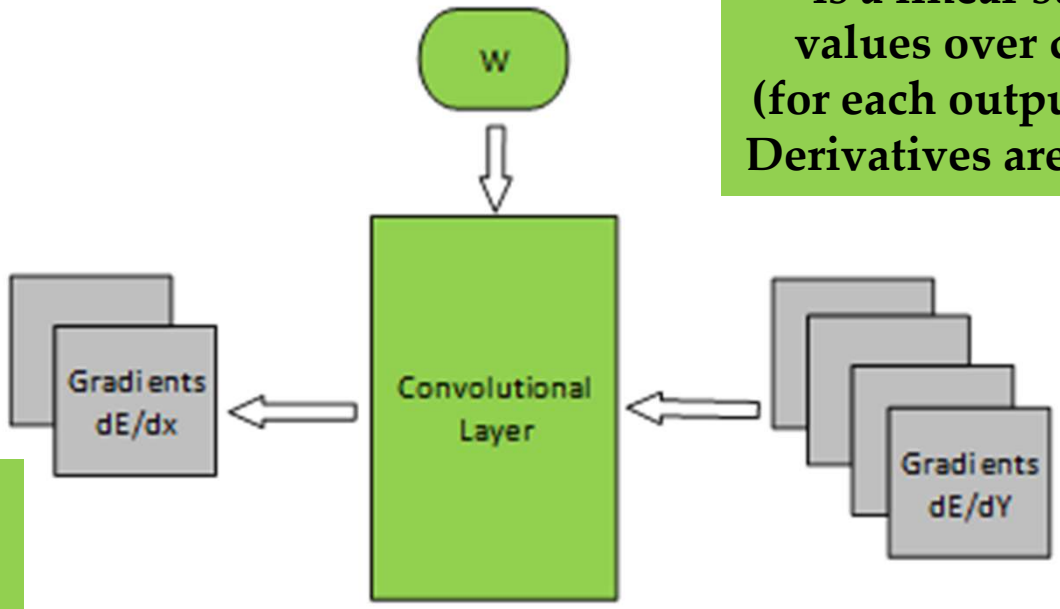
$$(Y(m) > 0\ ?\ Y(m) : 0) \times \sum_{w=0}^{N} W(w)$$

- Where N is the number of inputs in X.

- $\frac{\partial E}{\partial X} = \frac{\partial E}{\partial k} \times \frac{\partial k}{\partial Y} \times \frac{\partial Y}{\partial G} \times \frac{\partial G}{\partial X}$

# Backpropagation

**Remember that Y is a linear sum of X values over channels (for each output feature). Derivatives are W values.**




$$Y = W \cdot X$$

$$\frac{dE}{dX} = \frac{dE}{dY}\frac{dY}{dX} = W \cdot \frac{dE}{dY}$$

# Calculating dE/dX from dE/dY

```
void convLayer_backward_dgrad(int B, int M, int C, int H, int W, int K, float *dE_dY, float *W, float *dE_dX)
{
  int H_out = H - K + 1;              // calculate H_out, W_out
  int W_out = W - K + 1;
  for (int b = 0; b < B; ++b) {       // for each image
    for (int c = 0;  c < C; ++c)      // for each input channel
      for (int h = 0; h < H; ++h)     // for each input pixel (two loops)
       for (int w = 0; w < W; ++w)
          dE_dX[b, c, h, w] = 0.0f;  // initialize to 0

    for (int m = 0;  m < M;  ++m)           // for each output feature map
      for (int h = 0; h < H_out; ++h)       // for each output value (two loops)
        for (int w = 0; w < W_out; ++w)
          for (int c = 0;  c < C; ++c)      // for each input channel
            for (int p = 0; p < K; p)       // for each element of KxK filter (two loops)
              for (int q = 0; q < K; ++q)
                dE_dX[b, c, h + p, w + q] += dE_dY[b, m, h, w] * W[m, c, p, q];
  }
}
```

# What is ∂E/∂W for a conv layer?

- It is a matrix that gives the influence of each W element on the error.
  - It is the same dimension as the W matrix
- The calculation is about generating each element of the ∂E/∂W matrix
- For any given p and q values, ∂E/∂W[p,q] is the sum of ∂E/∂Y[h,w] * ∂Y[h,w]/∂W[p,q] for all relevant h and w values

# Back-propagation for a convolutional layer

- Convolutional network:

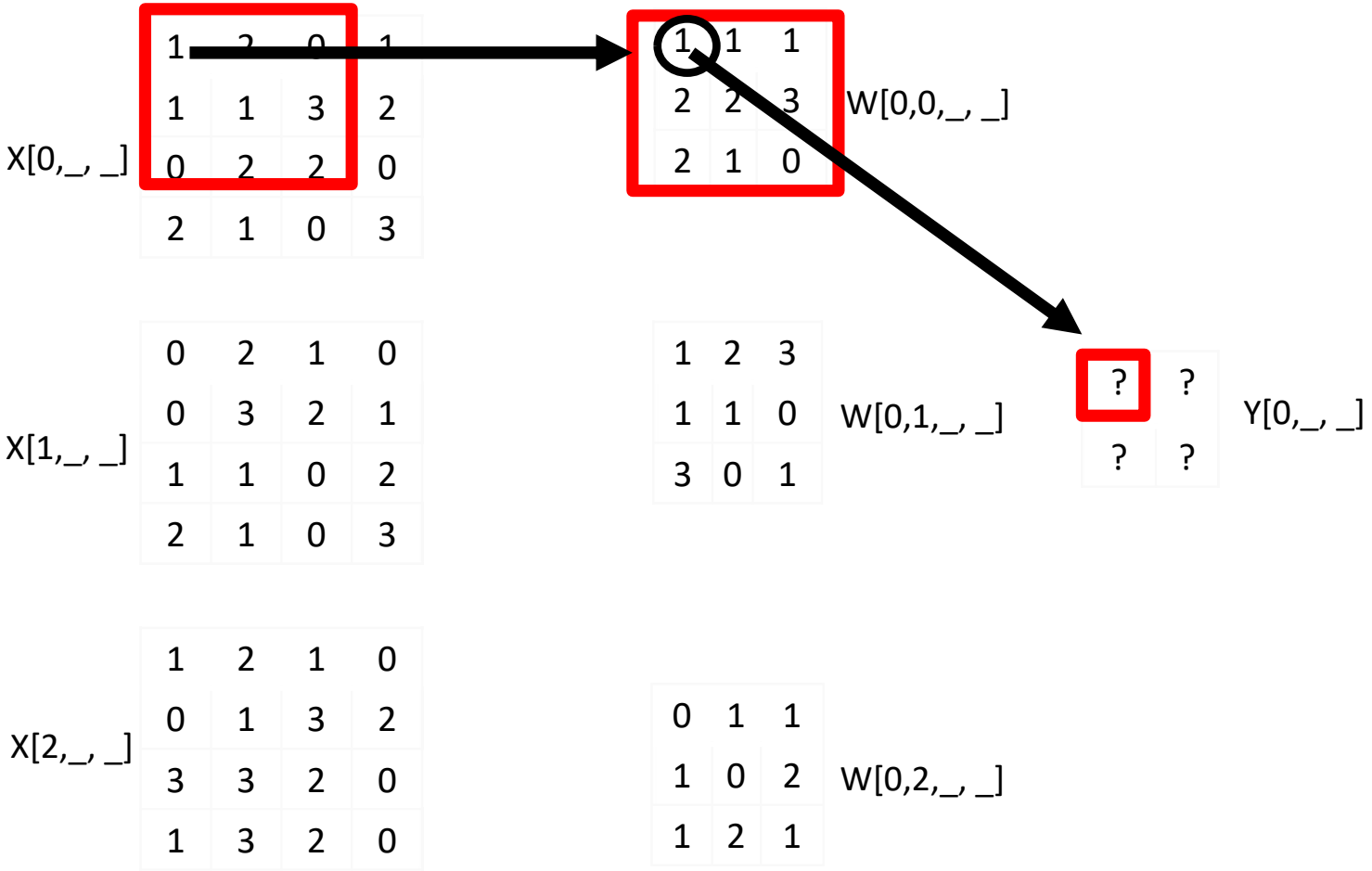
$$Y(h,w) = \sum_{p=1}^{P_{out}} \sum_{q=1}^{Q_{out}} (X(c, h+p, w+q) \times W(c,p,q))$$

- ∂E/∂W :

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial Y} \times \frac{\partial Y}{\partial W}$$

$$\frac{\partial E}{\partial W}(c,p,q) = \sum_{h=1}^{H_{out}} \sum_{w=1}^{W_{out}} (\frac{\partial E}{\partial Y}(h,w) \times X(c, h+p, w+q)$$

# A Small Convolution Layer Example Generating Y[0,0,0], c=0 (review)



# Example

- In our example, W[0, 0, 0, 0] contribute to output
  - Y[0,0,0] with X[0,0,0] * W[0,0,0,0]
  - Y[0,0,1] with X[0,0,1] * W[0,0,0,0]
  - Y[0,1,0] with X[0,1,0] * W[0,0,0,0]
  - Y[0,1,1] with X[0,1,1] * W[0,0,0,0]
  - In general, the total number of terms is Hout×Wout
- So, ∂E/∂W[0,0,0,0] can be calculated as the sum of
  - ∂E/∂Y[0,0,0] * ∂Y[0,0,0]/∂W[0,0,0,0]= ∂E/∂Y[0,0,0] * X[0,0,0]
  - ∂E/∂Y[0,0,1] * ∂Y[0,0,1]/∂W[0,0,0,0]= ∂E/∂Y[0,0,1] * X[0,0,1]
  - ∂E/∂Y[0,1,0] * ∂Y[0,1,0]/∂W[0,0,0,0]= ∂E/∂Y[0,1,0] * X[0,1,0]
  - ∂E/∂Y[0,1,1] * ∂Y[0,1,1]/∂W[0,0,0,0]= ∂E/∂Y[0,1,0] * X[0,1,1]

# Back-propagation for a convolutional layer

- Convolutional layer for a single image

$$Y^{m}(h,w) = \sum_{p=1}^{P_{out}} \sum_{q=1}^{Q_{out}} (X(c, h+p, w+q) \times W^{m}(c,p,q))$$

- Since each W(c, m, p, q) affects all elements of output Y(m, h, w), we should accumulate gradients over all pixels in the corresponding output feature

$$\frac{\partial E}{\partial W}^{m}(c,p,q) = \sum_{h=1}^{H_{out}} \sum_{w=1}^{W_{out}} (\frac{\partial E}{\partial Y}^{m}(h,w) \times X(c, h+p, w+q)$$

# Calculating ∂E/∂W

```
void convLayer_backward_wgrad(int M, int C, int H, int W, int K,
        float* dE_dY, float* X, float* dE_dW)
{
  int m, c, h, w, p, q;
  int H_out = H – K + 1;
  int W_out = W – K + 1;
  for(m = 0; m < M; m++)
   for(c = 0; c < C; c++)
    for(p = 0; p < K; p++)
     for(q = 0; q < K; q++)
      dE_dW[m, c, p, q] = 0;

  for(m = 0;  m < M;  m++)
   for(h = 0; h < H_out; h++)
    for(w = 0; w < W_out; w++)
     for(c = 0;  c  <  C;  c++)
      for(p = 0; p < K; p++)
       for(q = 0; q < K; q++)
        dE_dW[m, c, p, q] += X[c, h + p, w + q] * dE_dY[m, h, w];
}
```

# A Small Convolution Layer Example Generating dE/dW[0,0,_,_]

X[0,_, _]

| | | | |
|---|---|---|---|
| 1 | 2 | 0 | 1 |
| 1 | 1 | 3 | 2 |
| 0 | 2 | 2 | 0 |
| 2 | 1 | 0 | 3 |

X[1,_, _]

| | | | |
|---|---|---|---|
| 0 | 2 | 1 | 0 |
| 0 | 3 | 2 | 1 |
| 1 | 1 | 0 | 2 |
| 2 | 1 | 0 | 3 |

X[2,_, _]

| | | | |
|---|---|---|---|
| 1 | 2 | 1 | 0 |
| 0 | 1 | 3 | 2 |
| 3 | 3 | 2 | 0 |
| 1 | 3 | 2 | 0 |

∂E/∂Y[0,_, _]

| | |
|---|---|
| 1 | 1 |
| 1 | 1 |

∂E/∂W[0,0,_, _ ]

| | | |
|---|---|---|
| ? | ? | ? |
| ? | ? | ? |
| ? | ? | ? |

∂E/∂W[0,1,_, _ ]

| | | |
|---|---|---|
| ? | ? | ? |
| ? | ? | ? |
| ? | ? | ? |

∂E/∂W[0,2,_, _ ]

| | | |
|---|---|---|
| ? | ? | ? |
| ? | ? | ? |
| ? | ? | ? |

# A Small Convolution Layer Example Generating dE/dW[0,0-2,_,_]

X[0,_, _]

| | | | |
|---|---|---|---|
| 1 | 2 | 0 | 1 |
| 1 | 1 | 3 | 2 |
| 0 | 2 | 2 | 0 |
| 2 | 1 | 0 | 3 |

X[1,_, _]

| | | | |
|---|---|---|---|
| 0 | 2 | 1 | 0 |
| 0 | 3 | 2 | 1 |
| 1 | 1 | 0 | 2 |
| 2 | 1 | 0 | 3 |


dE/dY[0,_, _]

dE/dW[0,0,_, _ ]

| | | |
|---|---|---|
| 5 | 6 | 5 |
| 3 | 5 | 5 |
| 3 | 3 | 5 |

dE/dW[0,1,_, _ ]

| | | |
|---|---|---|
| 6 | ? | ? |
| ? | ? | ? |
| ? | ? | ? |

0*1 + 2*1 + 0*1 + 3*1 = 6

# A Small Convolution Layer ExampleGenerating dE/dW[0,0-2,_,_]

X[0,_, _]

| | | | |
|---|---|---|---|
| 1 | 2 | 0 | 1 |
| 1 | 1 | 3 | 2 |
| 0 | 2 | 2 | 0 |
| 2 | 1 | 0 | 3 |

X[1,_, _]

| | | | |
|---|---|---|---|
| 0 | 2 | 1 | 0 |
| 0 | 3 | 2 | 1 |
| 1 | 1 | 0 | 2 |
| 2 | 1 | 0 | 3 |

| | |
|---|---|
| 1 | 1 |
| 1 | 1 |

dE/dY[0,_, _]

dE/dW[0,0,_, _ ]

| | | |
|---|---|---|
| 5 | 6 | 5 |
| 3 | 5 | 5 |
| 3 | 3 | 5 |

dE/dW[0,1,_, _ ]

| | | |
|---|---|---|
| 6 | 8 | ? |
| ? | ? | ? |
| ? | ? | ? |

2*1 + 1*1 + 3*1 + 2*1 = 8

# A Small Convolution Layer Example Generating dE/dW[0,0-2,_,_]

X[0,_, _]

| | | | |
|---|---|---|---|
| 1 | 2 | 0 | 1 |
| 1 | 1 | 3 | 2 |
| 0 | 2 | 2 | 0 |
| 2 | 1 | 0 | 3 |

X[1,_, _]

| | | | |
|---|---|---|---|
| 0 | 2 | 1 | 0 |
| 0 | 3 | 2 | 1 |
| 1 | 1 | 0 | 2 |
| 2 | 1 | 0 | 3 |

| | |
|---|---|
| 1 | 1 |
| 1 | 1 |

dE/dY[0,_, _]

dE/dW[0,0,_, _ ]

| | | |
|---|---|---|
| 5 | 6 | 5 |
| 3 | 5 | 5 |
| 3 | 3 | 5 |

dE/dW[0,1,_, _ ]

| | | |
|---|---|---|
| 6 | 8 | 4 |
| ? | ? | ? |
| ? | ? | ? |

1*1 + 0*1 + 2*1 + 1*1 = 4

We can picture this computation in a more familiar way

# A Small Convolution Layer Example Generating dE/dW[0,0-2,_,_]

X[0,_, _]

| | | | |
|---|---|---|---|
| 1 | 2 | 0 | 1 |
| 1 | 1 | 3 | 2 |
| 0 | 2 | 2 | 0 |
| 2 | 1 | 0 | 3 |

X[1,_, _]

| | | | |
|---|---|---|---|
| 0 | 2 | 1 | 0 |
| 0 | 3 | 2 | 1 |
| 1 | 1 | 0 | 2 |
| 2 | 1 | 0 | 3 |

| | |
|---|---|
| 1 | 1 |
| 1 | 1 |

dE/dY[0,_, _]

dE/dW[0,0,_, _ ]

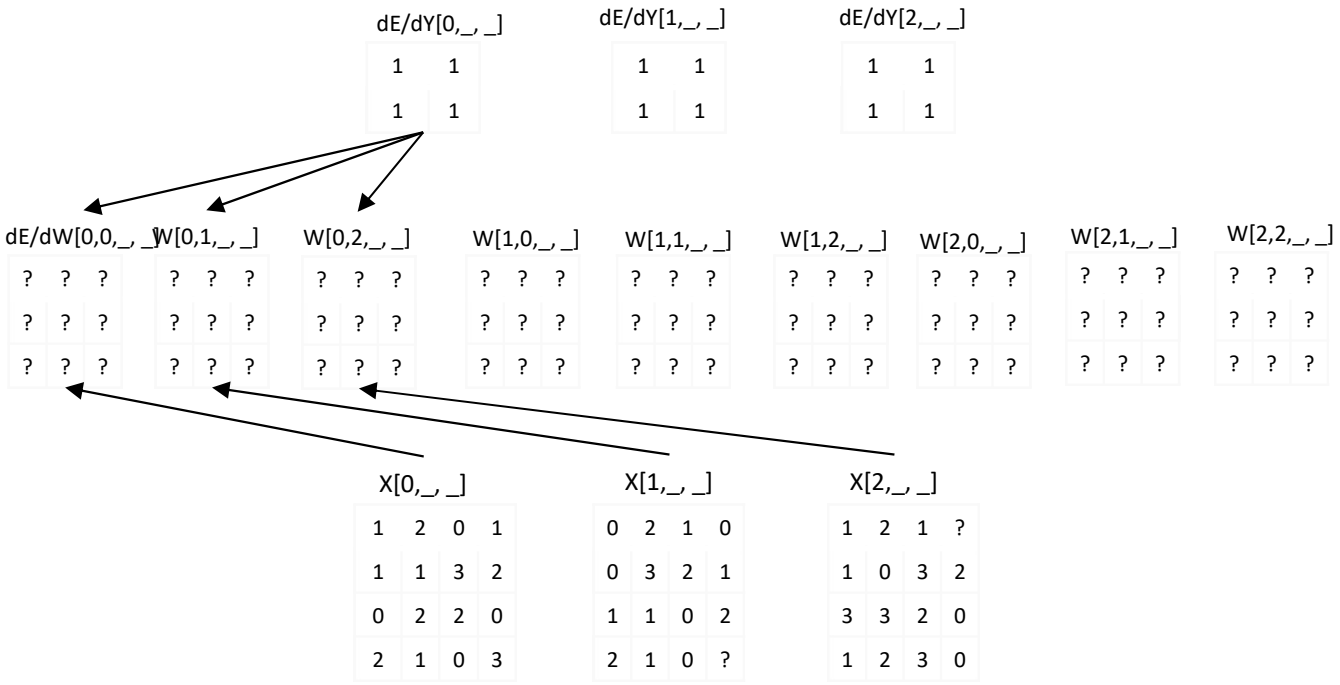
| | | |
|---|---|---|
| 5 | 6 | 5 |
| 3 | 5 | 5 |
| 3 | 3 | 5 |

dE/dW[0,1,_, _ ]

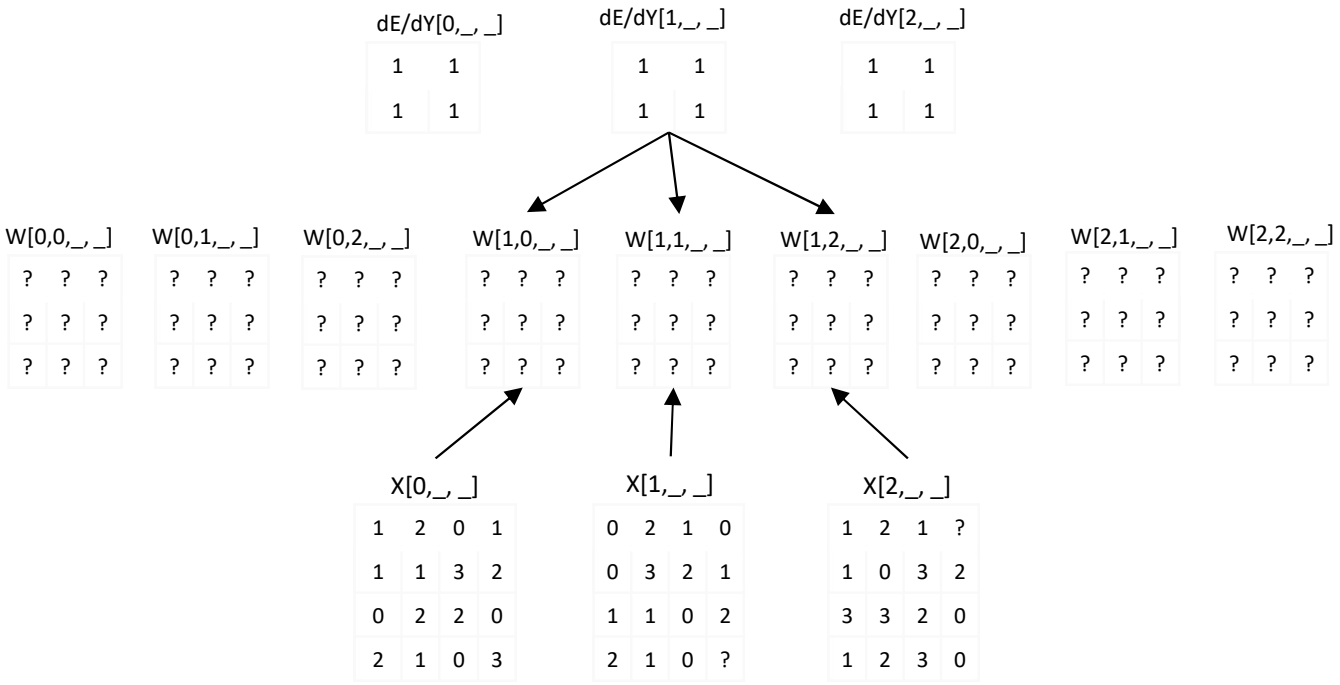
| | | |
|---|---|---|
| 6 | 8 | 4 |
| ? | ? | ? |
| ? | ? | ? |

1*1 + 0*1 + 2*1 + 1*1 = 4

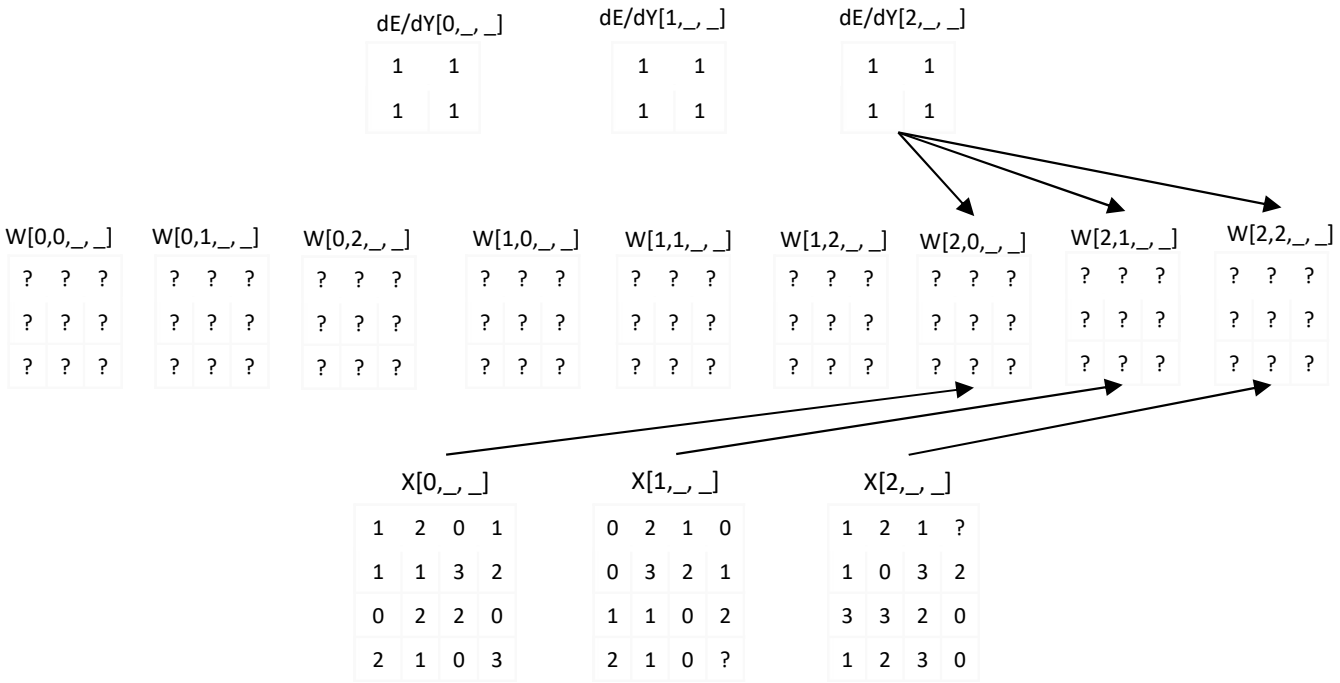
# A Small Convolution Layer Example Generating dE/dW[0,0-2,_,_]



# A Small Convolution Layer Example Generating dE/dW[0,0-2,_,_]

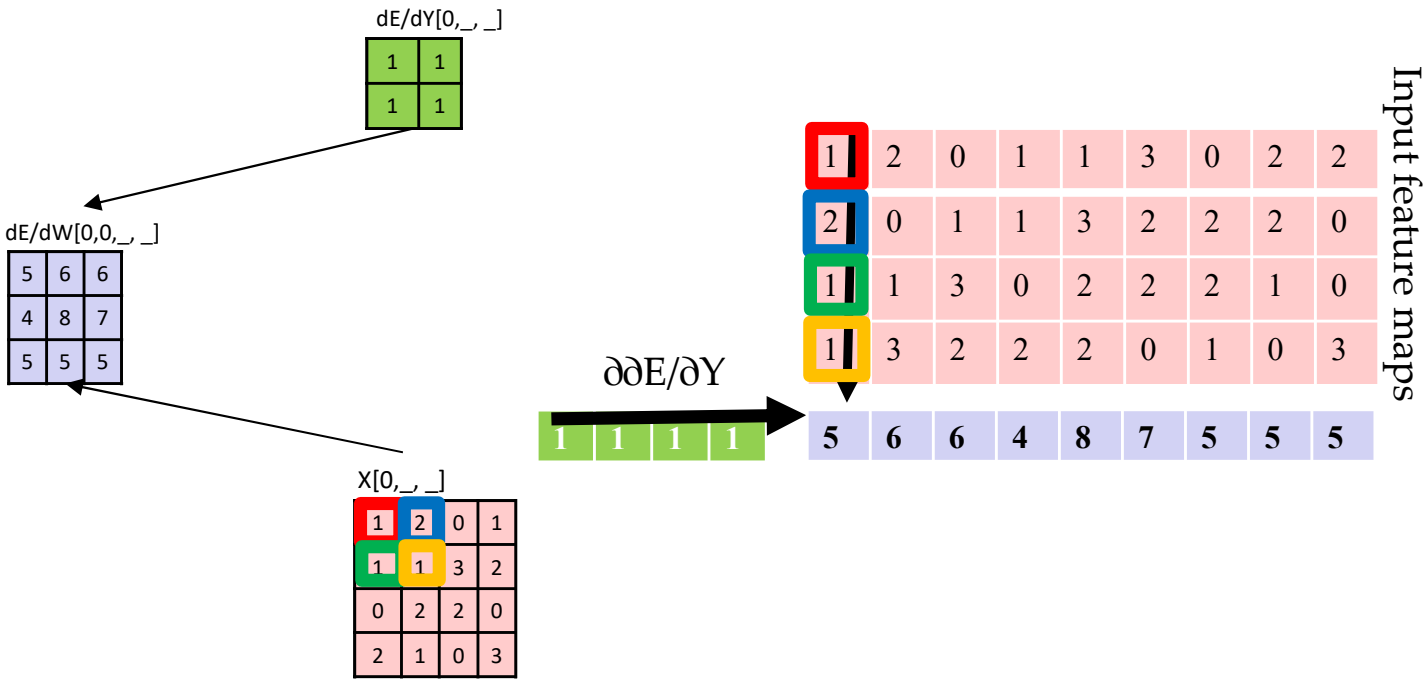


# A Small Convolution Layer Example Generating dE/dW[0,0-2,_,_]




- A single output feature and a single input channel are needed
- Matrix-vector multiplication can be use to compute each dE/dW channel per output

# A Small Convolution Layer Example Generating dE/dW[0,0-2,_,_]



# Adjusting Weights

- After the dE/dW values at all feature map element positions are computed, weights are updated:

- For each weight value
- $w(t+1) = w(t) - \lambda\ (dE/dw)$

- where $\lambda$ is the learning rate (hyper-parameter).

$W(t) - \lambda W = W(t+1)$

W[0,0,_, _]

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 2 | 3 |
| 2 | 1 | 0 |

$- 0.01 *$ dE/dW[0,0,_, _ ]

| | | |
|---|---|---|
| 3 | 2 | 1 |
| 2 | 4 | 5 |
| 2 | 4 | 2 |

$=$

| | | |
|---|---|---|
| 0.97 | 0.98 | 0.99 |
| 1.98 | 1.96 | 2.95 |
| 1.98 | 0.96 | 0 |

$\lambda = 0.01$

W[0,1,_, _]

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 1 | 0 |
| 3 | 0 | 1 |

$- 0.01 *$ dE/dW[0,1,_, _ ]

| | | |
|---|---|---|
| 2 | 3 | 1 |
| 3 | 5 | 3 |
| 2 | 1 | 3 |

$=$

| | | |
|---|---|---|
| 0.98 | 1.97 | 2.99 |
| 0.97 | 0.95 | 0 |
| 2.98 | 0 | 0.97 |

W[0,2,_, _]

| | | |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 2 |
| 1 | 2 | 1 |

$- 0.01 *$ dE/dW[0,2,_, _ ]

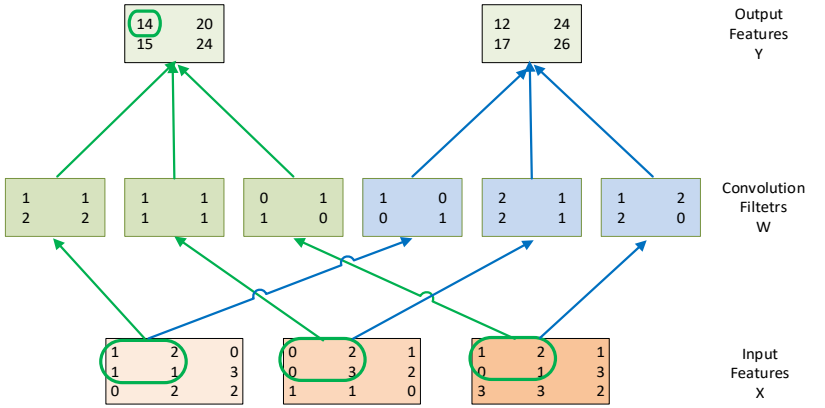
| | | |
|---|---|---|
| 3 | 3 | 1 |
| 1 | 4 | 5 |
| 6 | 5 | 2 |

$=$

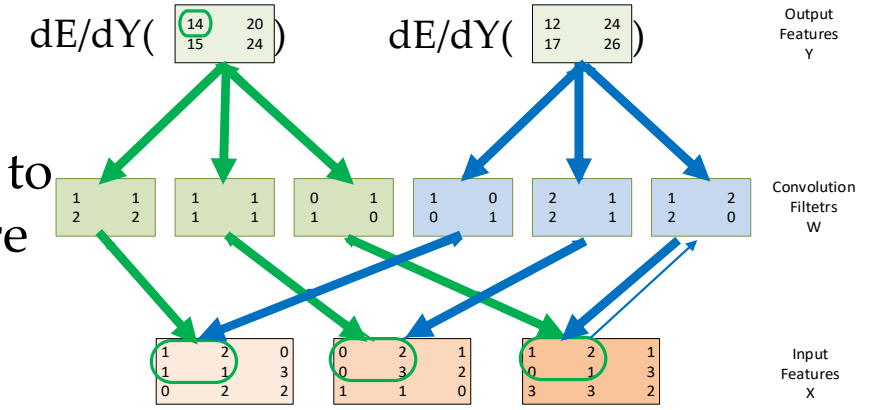
| | | |
|---|---|---|
| 0 | 0.97 | 0.99 |
| 0.99 | 0 | 1.95 |
| 0.94 | 1.95 | 0.98 |

# Back-propagation dE/dX for a convolutional layer

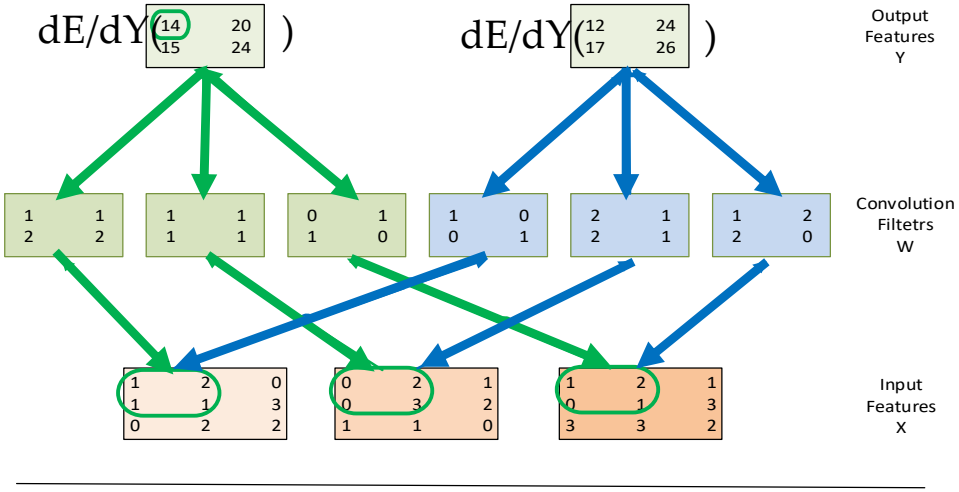
- Convolution Layer:
  - Multiple *input* features affect a single *output* feature
  - Each input feature affects multiple output features




- dE/dX:
  - Multiple *output* features propagate to a single *input* feature
  - Each one through a different W



# Back-propagation dE/dX for a convolutional layer




$$\frac{\partial E}{\partial X}(c,h,w) = \sum_{m=1}^{M}\sum_{p=1}^{K}\sum_{q=1}^{K}\left(W(p,q) * \frac{\partial E}{\partial Y}(h-p,w-q)\right)$$

Now we need to iterate through all output features

# Example

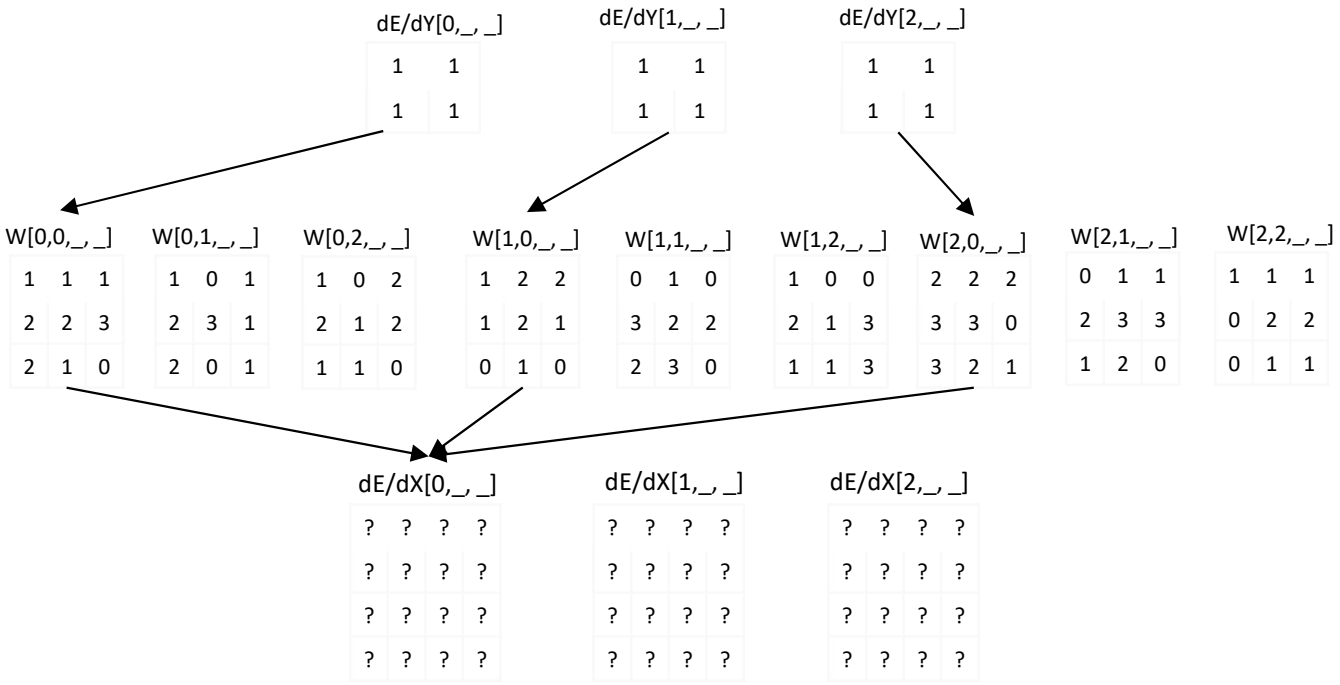
- In our example, X[0, 1, 1] contribute to output
  - Y[0,1,1] with X[0,1,1] * W[0,0,0,0]
  - Y[0,1,0] with X[0,1,1] * W[0,0,0,1]
  - Y[0,0,1] with X[0,1,1] * W[0,0,1,0]
  - Y[0,0,0] with X[0,1,1] * W[0,0,1,1]
  - In general, the total number of terms is K×K
- So, ∂E/∂X[0,1,1] can be calculated as the sum of
  - ∂E/∂Y[0,1,1] * ∂Y[0,1,1]/∂X[0,1,1]= ∂E/∂Y[0,1,1] * W[0,0,0,0]
  - ∂E/∂Y[0,1,0] * ∂Y[0,1,0]/∂X[0,1,1]= ∂E/∂Y[0,1,0] * W[0,0,0,1]
  - ∂E/∂Y[0,0,1] * ∂Y[0,0,1]/∂X[0,1,1]= ∂E/∂Y[0,0,1] * W[0,0,1,0]
  - ∂E/∂Y[0,0,0] * ∂Y[0,0,0]/∂X[0,1,1]= ∂E/∂Y[0,0,0] * W[0,0,1,1]

# Calculating dE/dX

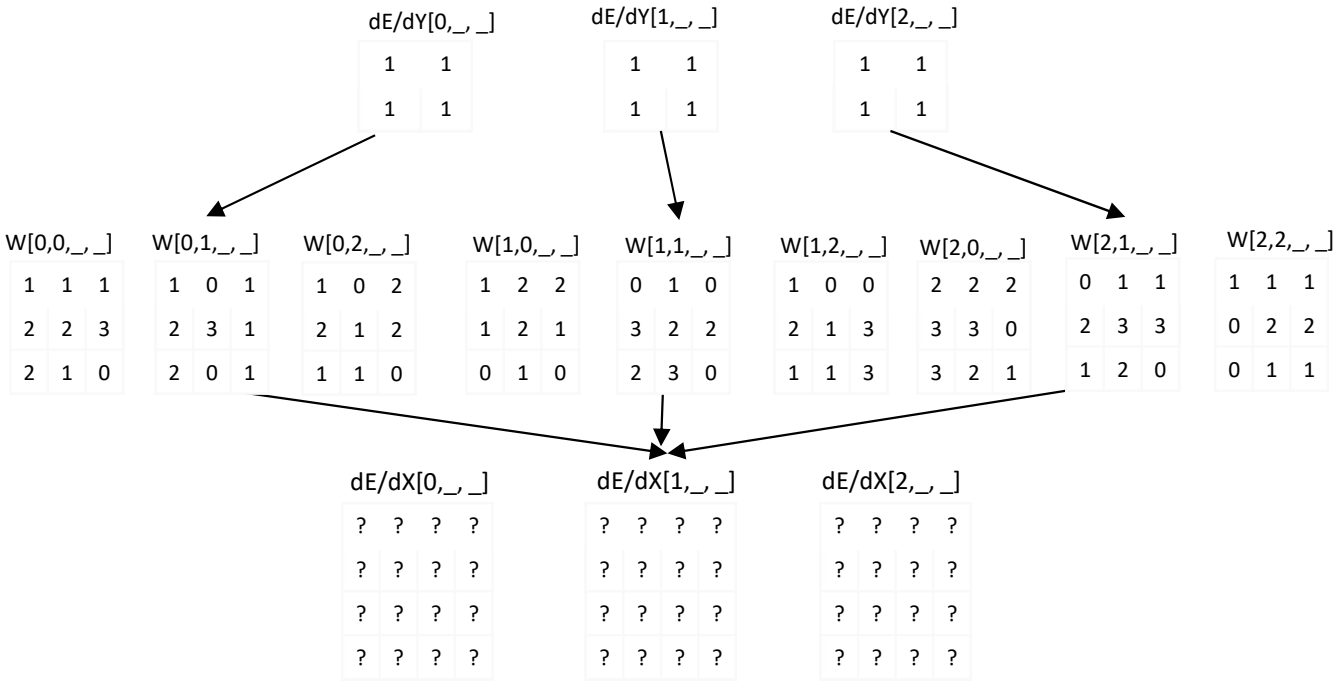
```
void convLayer_backward_dgrad(int M, int C, int H, int W, int K,
      float* dE_dY, float* W, float* dE_dX)
{
  int m, c, h, w, p, q;
  int H_out = H – K + 1;
  int W_out = W – K + 1;
  for(c = 0;  c < C; c++)
   for(h = 0; h < H; h++)
    for(w = 0; w < W; w++)
     dE_dX[c, h, w] = 0.;

  for(m = 0;  m < M;  m++)
   for(h = 0; h < H_out; h++)
    for(w = 0; w < W_out; w++)
     for(c = 0;  c < C; c++)
      for(p = 0; p < K; p++)
       for(q = 0; q < K; q++)
        dE_dX[c, h + p, w + q] += W[m, c, p, q] * dE_dY[m, h, w] ;
}
```

# A Small Convolution Layer Example Generating dE/dX[0,_,_]



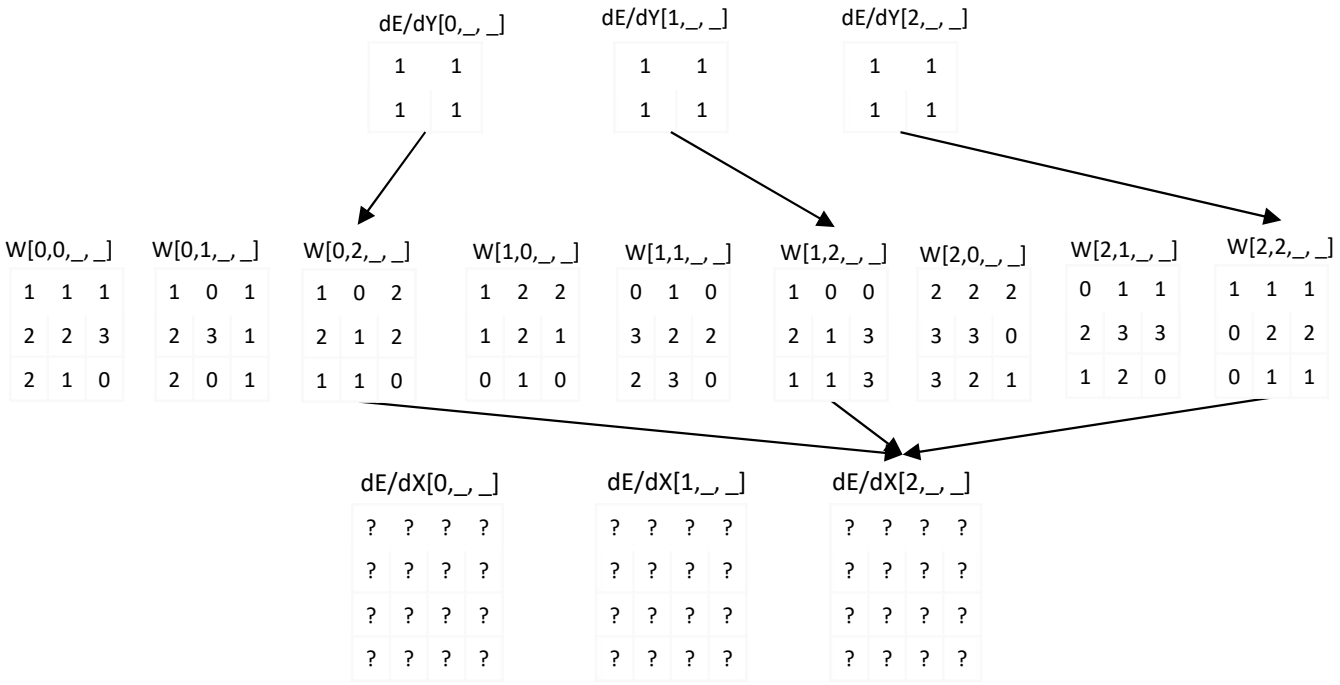

 

# A Small Convolution Layer Example Generating dE/dX[0,_,_]

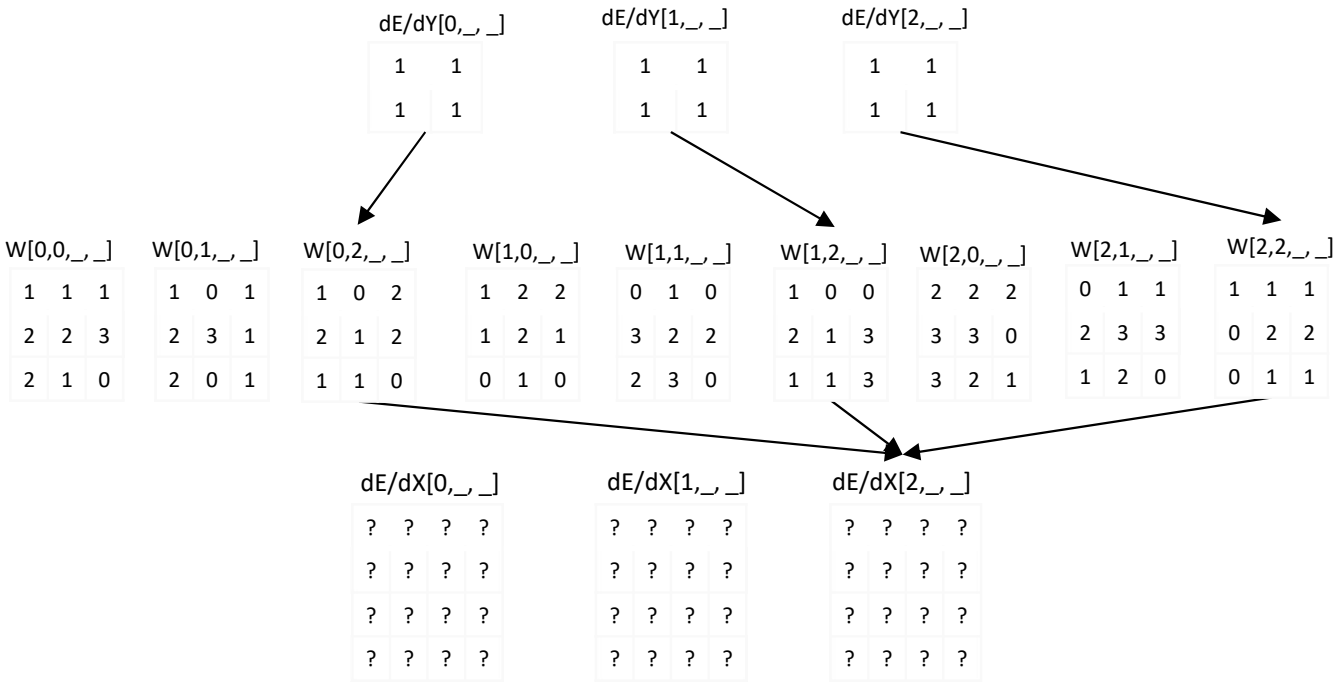


# A Small Convolution Layer Example Generating dE/dX[0,_,_]

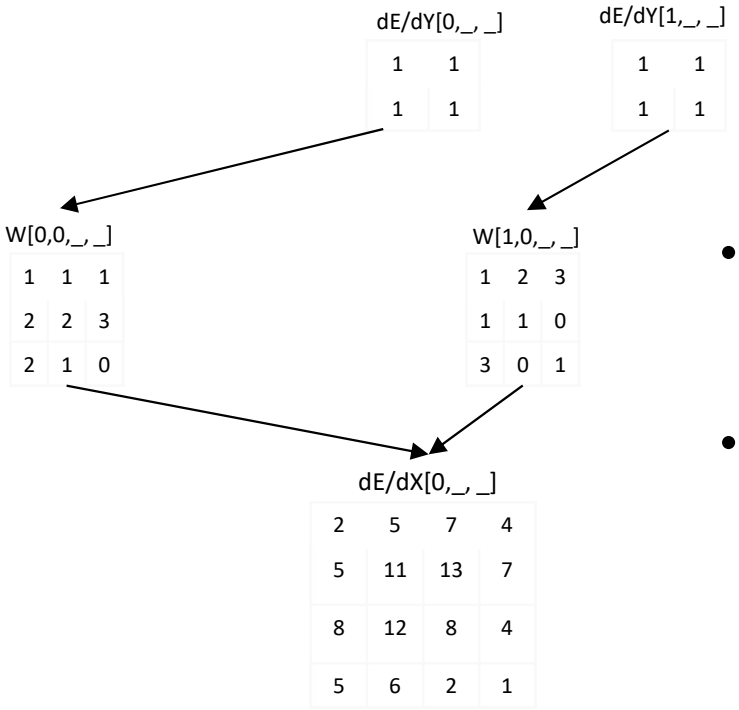


# A Small Convolution Layer Example Generating dE/dX[0,_,_]




- Let's go back to our previous example that has only two dE/dY contributions

# A Small Convolution Layer Example Generating dE/dX[0,_,_]




- We can design this computation as convolution
- But first we need to ready both dE/dY and W which are the inputs to dE/dX

# A Small Convolution Layer Example Generating dE/dX[0,_,_]

W[0,0,_, _]

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 2 | 3 |
| 2 | 1 | 0 |

The values of W need to be transposed

Wr[0,0,_, _]

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 2 | 2 |
| 1 | 1 | 1 |

dE/dY[0,_, _]

| | |
|---|---|
| 1 | 1 |
| 1 | 1 |

dE/dY needs to be padded with zeros

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 |

dE/dX[0,_, _]

| | | | |
|---|---|---|---|
| 2 | 5 | 7 | 4 |
| 5 | 11 | 13 | 7 |
| 8 | 12 | 8 | 4 |
| 5 | 6 | 2 | 1 |

# ANY QUESTIONS?