6.5930/1
Hardware Architectures for Deep Learning

Overview of Deep Neural Network Components

February 4, 2026

Joel Emer and Vivienne Sze

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

Outline of Today's Lecture

- Accelerator Design Methodology: From Workload to Hardware
  – Einsums
  – Roofline Models

- DNN Workloads

From Workload to Hardware

Slides from “TeAAL and HiFiber: Precise and Concise Descriptions of (Sparse) Tensor Algebra Accelerators”
https://teaal.csail.mit.edu/

Accelerator Design Methodology

(described in TeAAL [**Nayak**, *MICRO* 2023])

(1) Describe the architecture

(2) Develop the workload

(3) Evaluate the workload

(4) Compare implementations

(5) Optimize the design

Describing the Hardware Architecture

(1) Describe the architecture

Select from a library of components and organize them by writing an accelerator specification

(2) Develop the workload

(3) Evaluate the workload

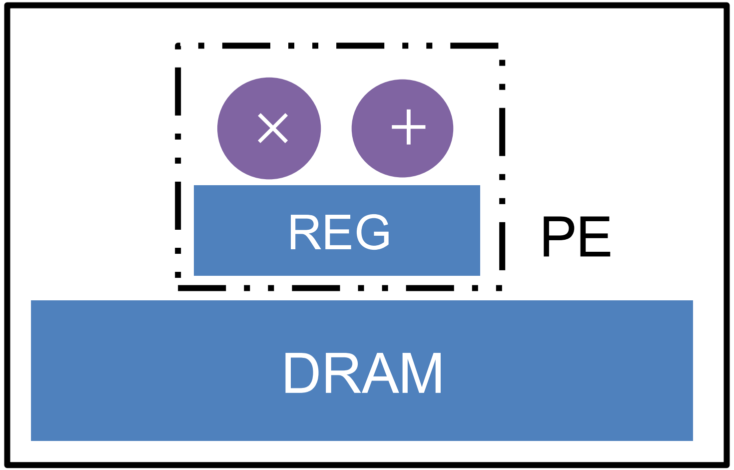
(4) Compare implementations

(5) Optimize the design

Architecture for the Simple End-to-End Example

Basic hardware architecture for tensor algebra operations:

- PE: ALU and local register files
- Memory: DRAM for global storage



Developing the Workload

(1) Describe the architecture

Select from a library of components and organize them by writing an accelerator specification

(2) Develop the workload

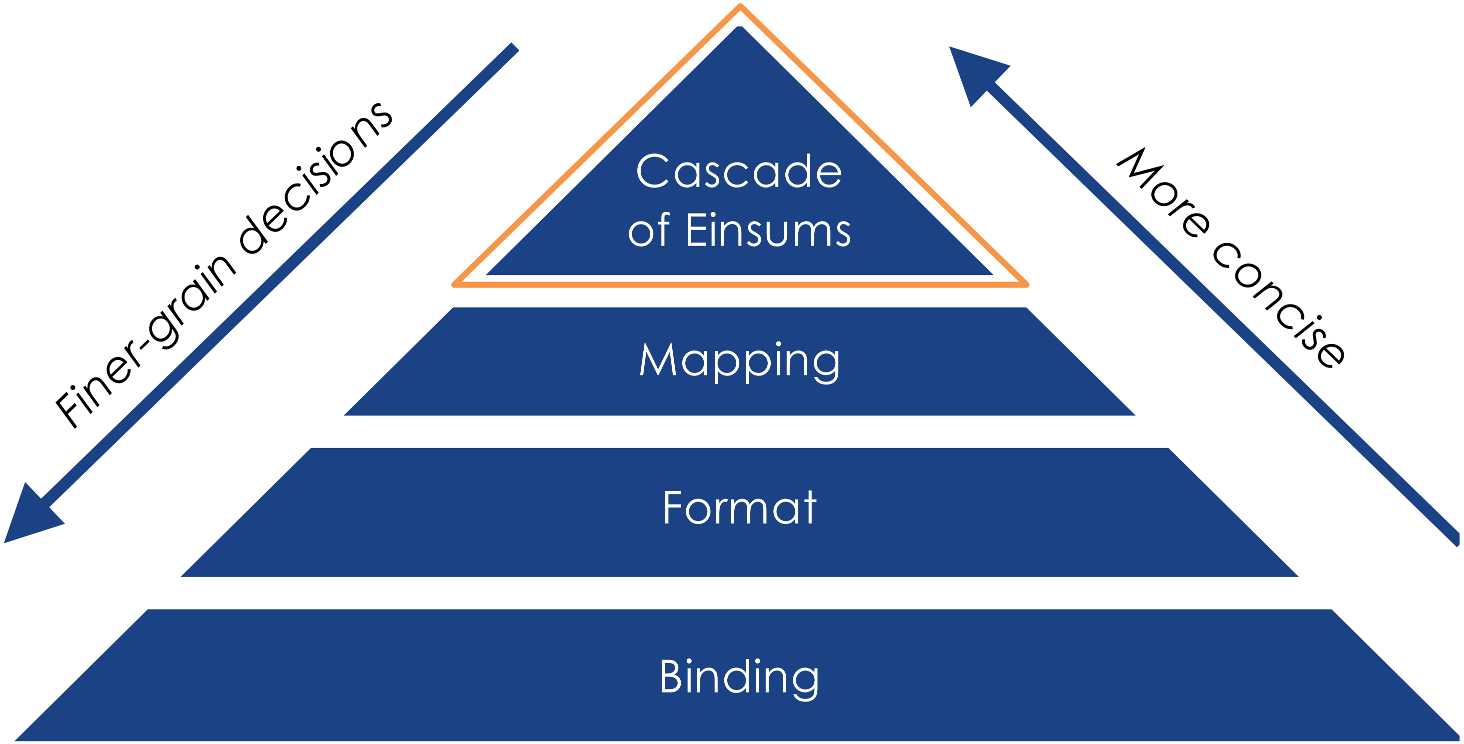
Write the cascade, mapping, format, and binding specifications

(3) Evaluate the workload

(4) Compare implementations

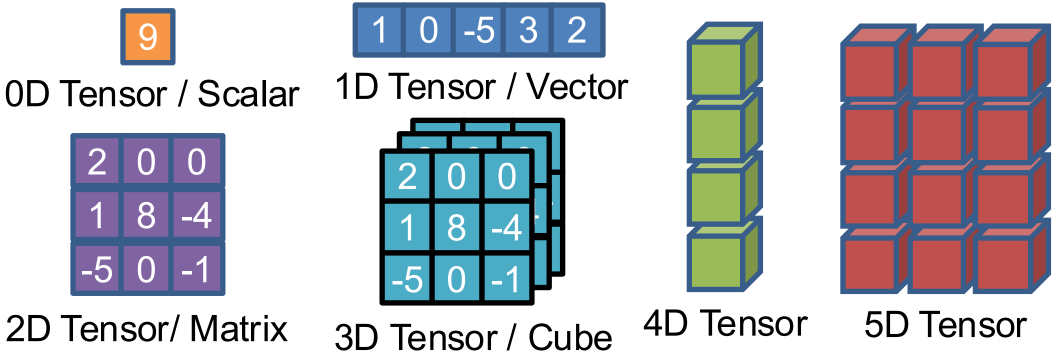
(5) Optimize the design

Separation of Concerns

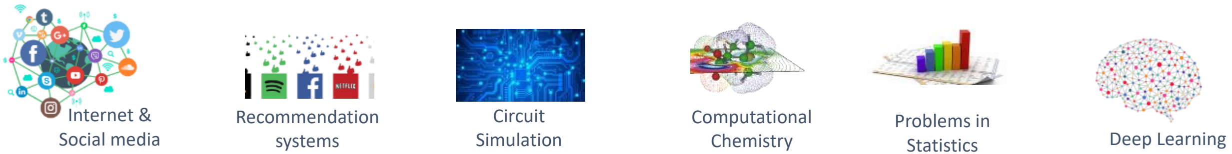


Tensor Algebra

Tensors are multi-dimensional arrays of data



Many applications can be framed as **tensor algebra**



Graphics courtesy of Hadi Asghari-Moghaddam

Tensor Terminology

In this class, we used the term “rank” to denote the dimension

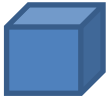
Properties of a Tensor:

Number of Ranks = Number of dimensions

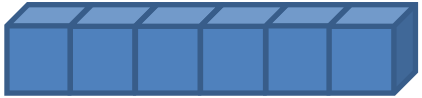
Rank Shape = Number of elements in each rank

Size of Tensor = Total number of elements in tensor (product of the shape of each rank)

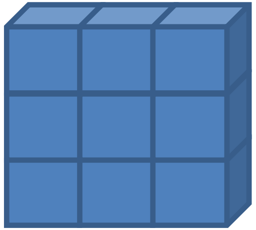
Scalar: 0 ranks



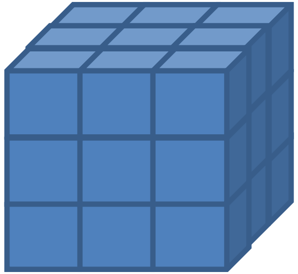
Vector: 1 rank



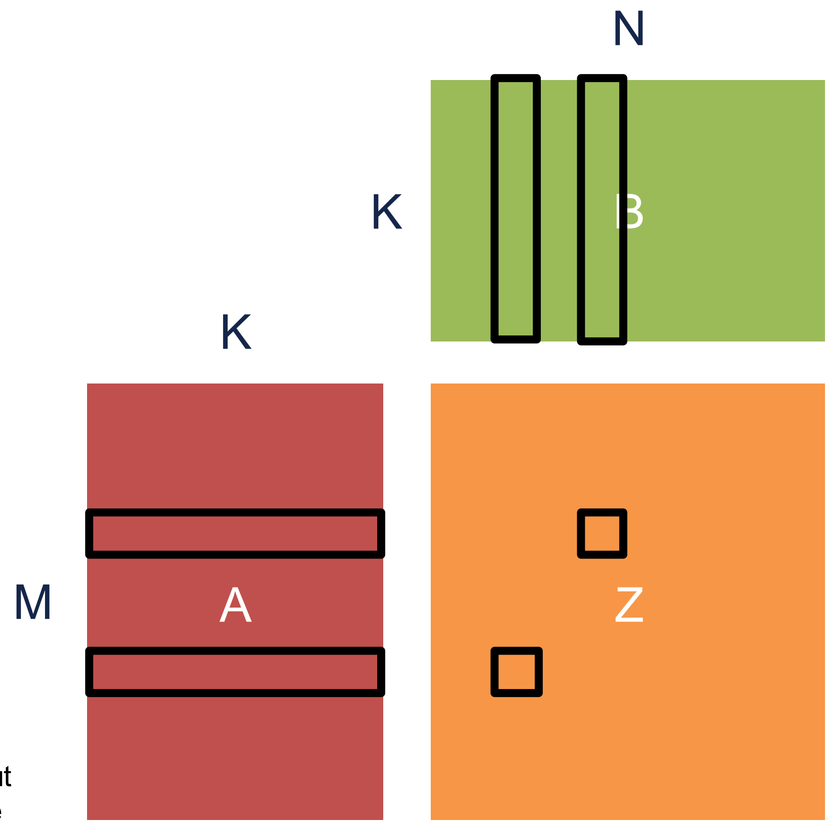
Matrix: 2 ranks



Cube: 3 ranks



Matrix Multiplication



Properties of B tensor:

Number of Ranks = 2
Rank names: N and K
Rank shape: N and K
Size of Tensor = N x K
Shape of Tensor = [N,K]

Properties of A tensor:

Number of Ranks = 2
Rank names: M and K
Rank shape*: M and K
Size of Tensor = M x K
Shape of Tensor = [M,K]

*In general shape and name same, but there are some exceptions we will see later (e.g., in attention of transformer)

Properties of Z tensor:

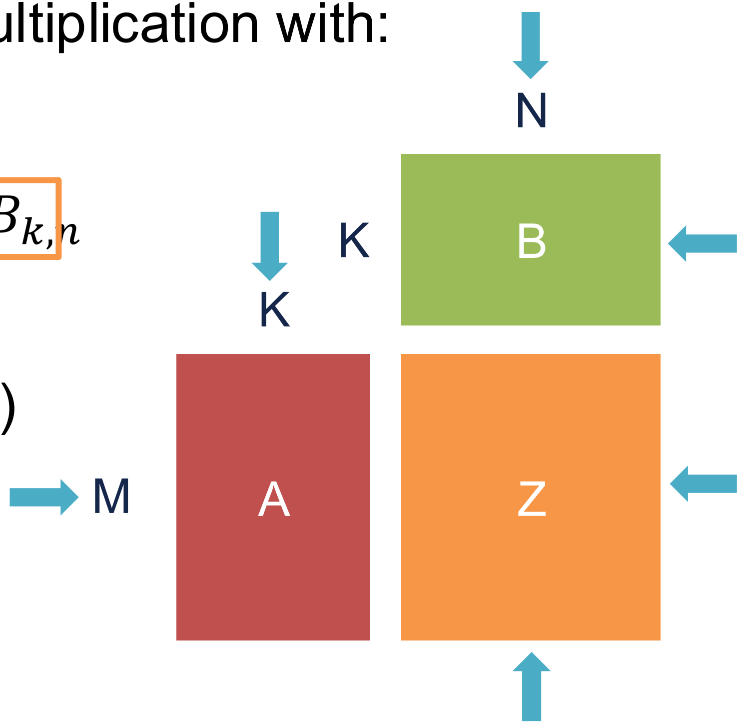
Number of Ranks = 2
Rank names: M and N
Rank shape: M and N
Size of Tensor = M x N
Shape of Tensor = [M,N]

Einstein Summation Notation (Einsums)

We can represent matrix multiplication with:

$$\boxed{Z_{m,n}} = \boxed{A_{k,m}}\boxed{\times}\boxed{B_{k,n}}$$

With implicit reduction (sum) over K

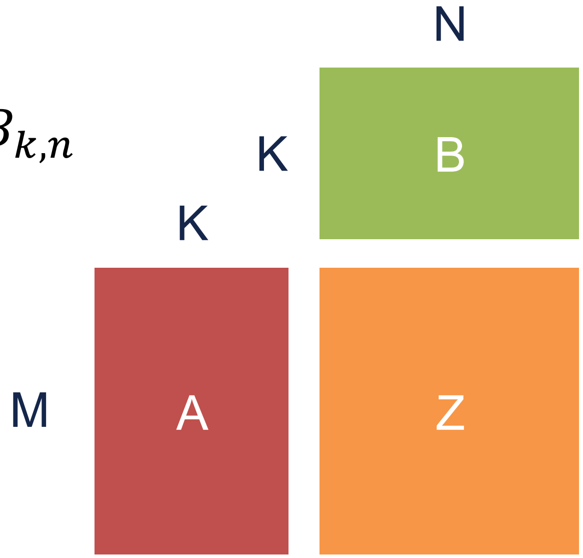


Einstein Summation Notation (Einsums)

We can represent matrix multiplication with:

$$Z_{m,n} = \cancel{\boxed{\sum_k}} A_{k,m} \times B_{k,n}$$

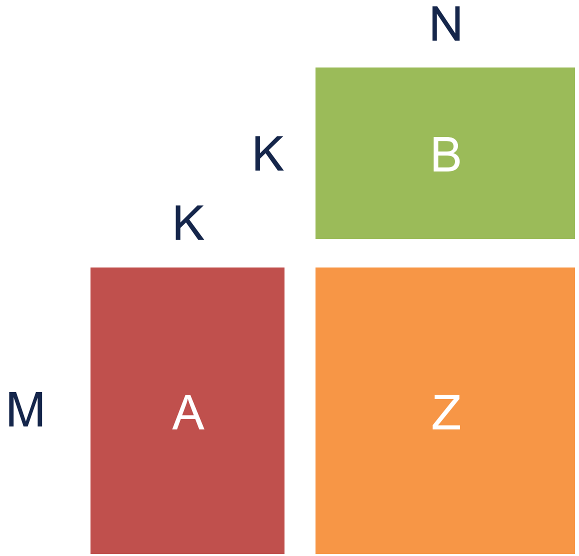
Explicit reduction is not necessary



Einstein Summation Notation (Einsums)

We can represent matrix multiplication with:

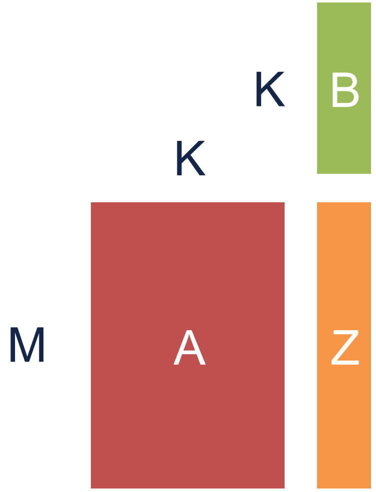
$$Z_{m,n} = A_{k,m} \times B_{k,n}$$



Operational Definition of an Einsum (ODE)

Simplifying to matrix-vector multiplication:

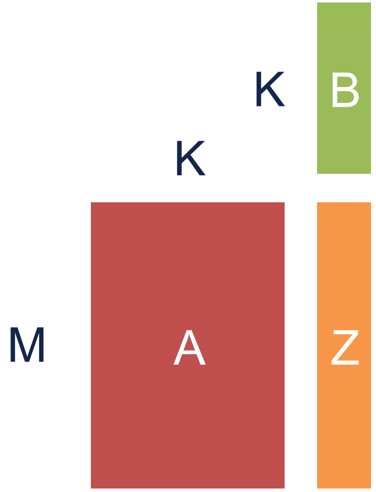
$$\boxed{Z_m} = \boxed{A_{k,m}} \times \boxed{B_k}$$



Operational Definition of an Einsum (ODE)

Einsum: $\boxed{Z_m} = \boxed{A_{k,m}} \times \boxed{B}_k$

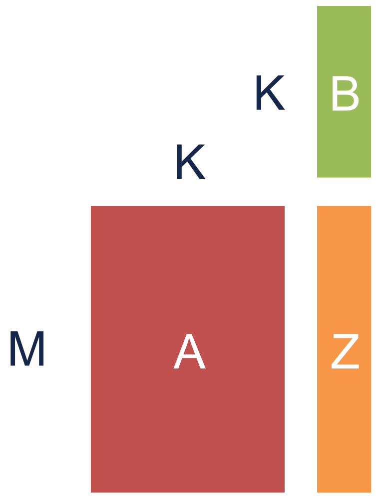
Iteration Space: Cartesian product of all legal coordinates in the Einsum



Operational Definition of an Einsum (ODE)

Einsum: $Z_m = A_{k,m} \times B_k$

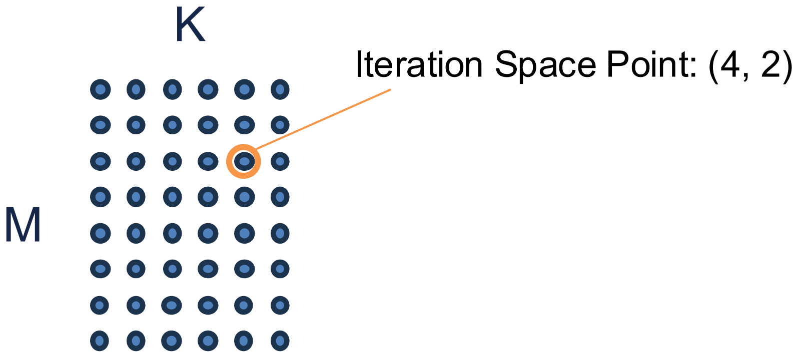
Iteration Space: $[0, K) \times [0, M)$

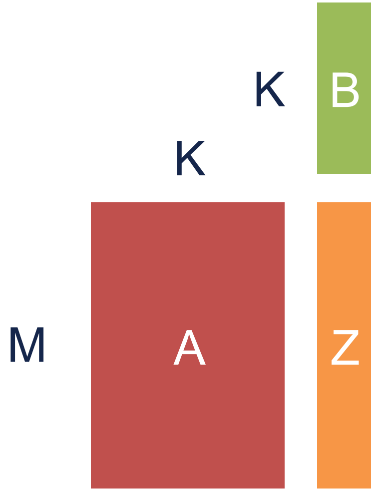


Operational Definition of an Einsum (ODE)

Einsum: $Z_m = A_{k,m} \times B_k$

Iteration Space: $K \times M$





Many ways to traverse iteration space
(processing order)

Operational Definition of an Einsum (ODE)

Einsum: $Z_m = A_{k,m} \times B_k$

Iteration Space: $K \times M$

For each point (k, m) in the iteration space:

- **Select the input values $A_{k,m}$ and B_k**
- **Multiply ($\times$) them together**
- **Update the output value Z_m**
- **Reduce ($+$) if necessary**

Operational Definition of an Einsum (ODE)

$$Z_m = A_{k,m} \times B_k$$

- Einsum defines
  - an iteration space over tensors
  - what computation is done on and between tensors at each point in the iteration space
- Traverse all points in space of all legal index values (iteration space)
  - The size of space is the Cartesian product of number of values of the unique indices (e.g., K*M) → amount of work that needs to be done!
- At each point in iteration space:
  - Calculate value on right hand side at specified indices for each operand (tensor)
  - Assign value to operand at specified indices on left hand side
  - Perform reduction across indices that appear on right-hand side but not left-hand side
- ***Note: Einsum will be the input format of the workload to the modeling tools for this class***

Evaluating the Workload

(1) Describe the architecture

Select from a library of components and organize them by writing an accelerator specification

(2) Develop the workload

Write the cascade, mapping, format, and binding specifications

(3) Evaluate the workload

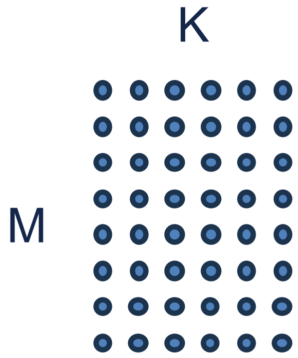
Model the workload and analyze with metrics like number of computes, memory traffic, and compute intensity

(4) Compare implementations

(5) Optimize the design

Analysis: What Compute is Required?

Einsum: $Z_m = A_{k,m} \times B_k$



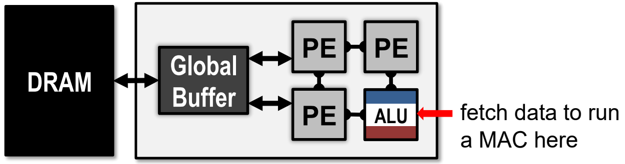
Iteration Space: $K \times M$

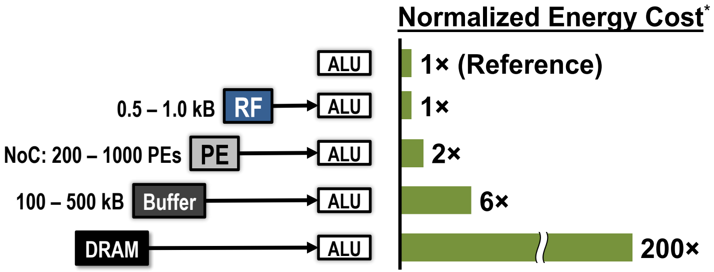
One multiply ($\times$) and reduce ($+$) per point in the iteration space (excluding edge effects)

- $K \times \mathrm{M}$ **multiplies**
- $(K-1) \times \mathrm{M}$ **adds**

Analysis: What is the Best-Case Compute Intensity?

- **Compute Intensity** is a measure of how much **data reuse** is theoretically possible
  - Higher compute intensity implies more data reuse feasible → potentially less data movement required





* measured from a commercial 65nm process

Defining Compute Intensity (CI)

(Standard) Compute Intensity: FLOPs / byte

However, this definition introduces questions:

- Is the multiply-accumulate (MAC) one operation or two?
- What is the bitwidth of our values?

Compute Intensity: Multiplications / value

Analysis: What is the Best-Case CI?

Compute Intensity: Multiplications / value

Multiplications : $K \times M$

Best-case memory traffic:

- $K \times M$ loads of $A_{k,m}$
- K loads of B_k
- M stores of Z_m

Analysis: What is the Best-Case CI?

Compute Intensity: Multiplications / value

Multiplications : $K \times M$

Best-case memory traffic: $K \times M + \mathrm{K} + \mathrm{M}$ values

Best-case compute intensity: $$\frac{K \times M}{K \times M + \mathrm{K} + \mathrm{M}}$$

Lab 1 focuses on this type of analysis of workloads (Einsum)

Developing the Workload

(1) Describe the architecture

Select from a library of components and organize them by writing an accelerator specification

(2) Develop the workload

Write the cascade, mapping, format, and binding specifications

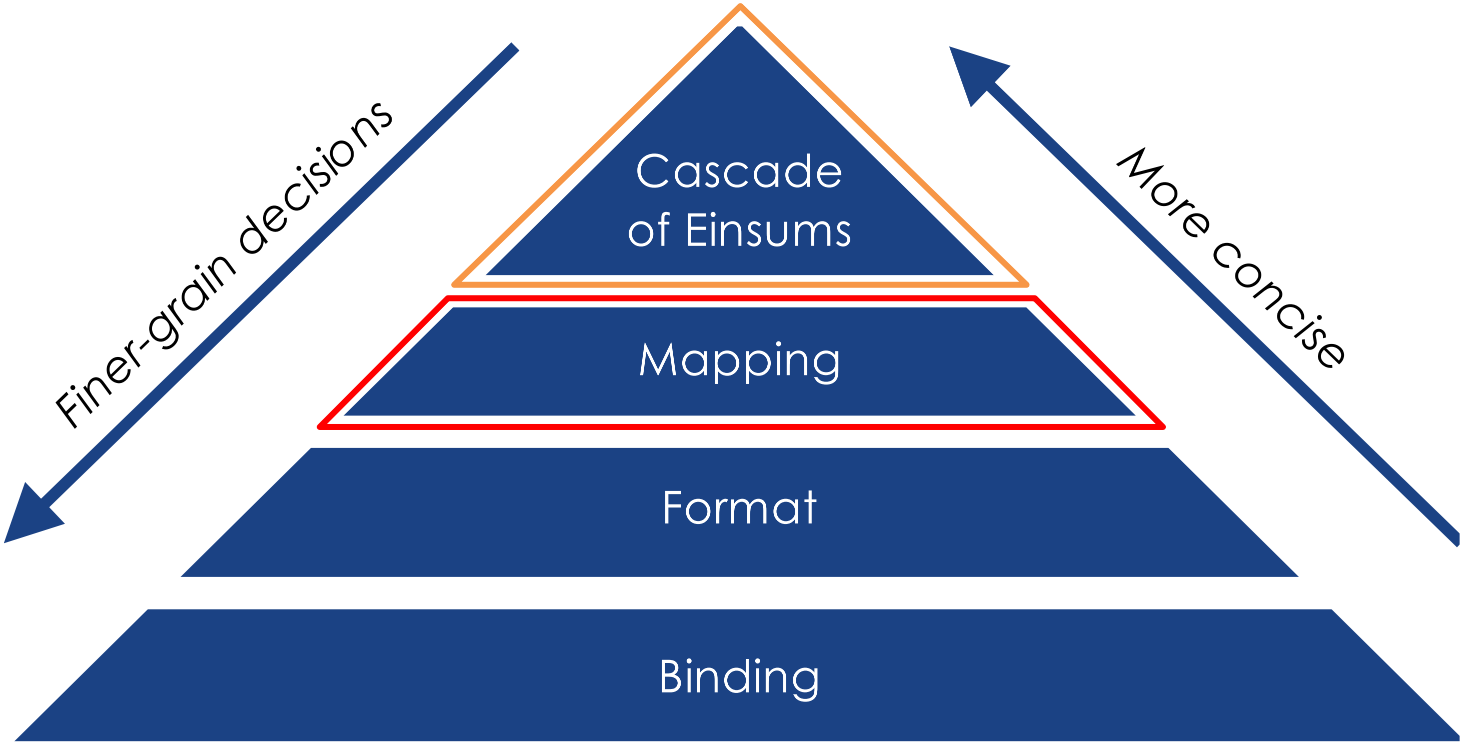
(3) Evaluate the workload

Model the workload and analyze with metrics like number of computes, memory traffic, and AI

(4) Compare implementations

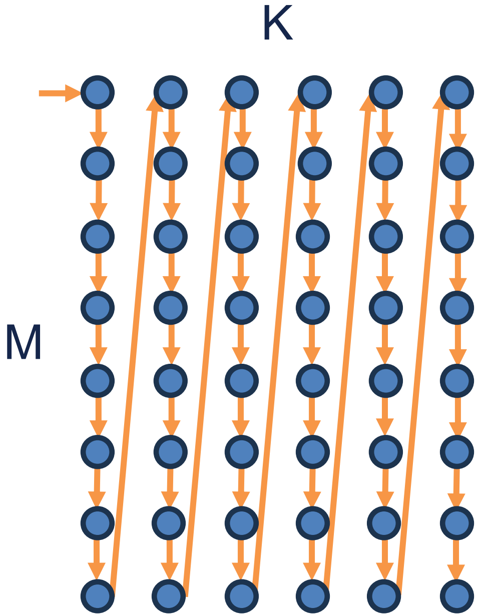
(5) Optimize the design

Separation of Concerns



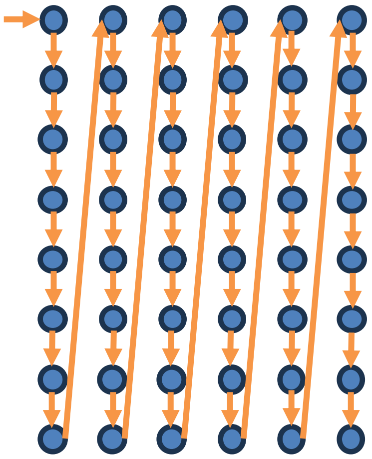
Traversing the Iteration Space

Can do so in any order



Traverse with Loop Nests

```
for k in range(K):
    for m in range(M):
        Z[m] += A[k, m] * B[k]
```



Lab 2 & 3 focuses on traverse order of iteration space (mapping)

Evaluating the Workload

(1) Describe the architecture

Select from a library of components and organize them by writing an accelerator specification

(2) Develop the workload

Write the cascade, mapping, format, and binding specifications

(3) Evaluate the workload

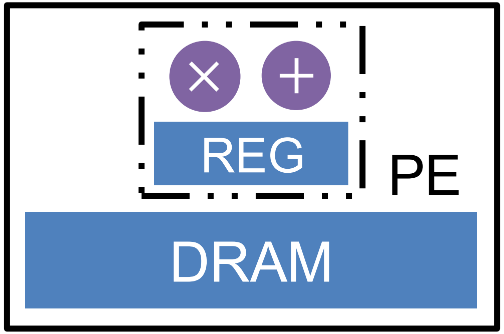
Model the workload and analyze with metrics like number of computes, memory traffic, and compute intensity

(4) Compare implementations

(5) Optimize the design

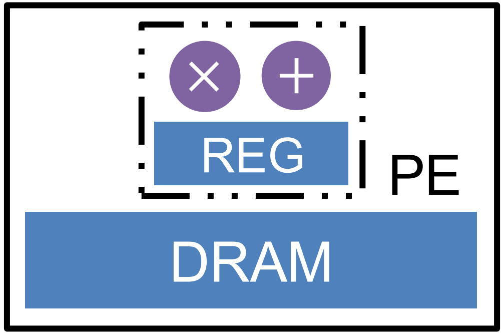
Analysis: What is the Achieved Traffic?

```
for k in range(K):

    for m in range(M):
        a_reg = A[k, m]
        b_reg = B[k]
        z_reg = Z[m]
        Z[m] += A[k, m] * B[k]
        Z[m] = z_reg
```



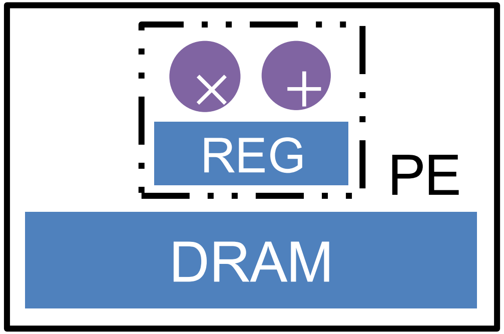
Analysis: What is the Achieved Traffic?

```
for k in range(K):

    for m in range(M):
        a_reg = A[k, m]
        b_reg = B[k]
        z_reg = Z[m]
        Z[m] += a_reg * B[k]
        Z[m] = z_reg
```



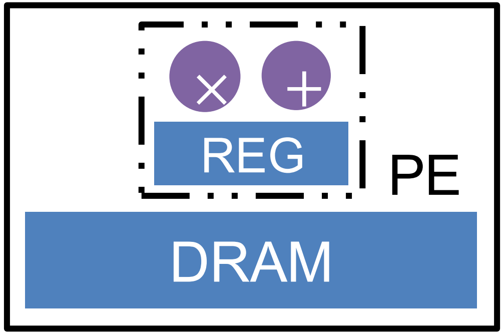
Analysis: What is the Achieved Traffic?

```
for k in range(K):

    for m in range(M):
        a_reg = A[k, m]
        b_reg = B[k]
        z_reg = Z[m]
        Z[m] += a_reg * b_reg
        Z[m] = z_reg
```



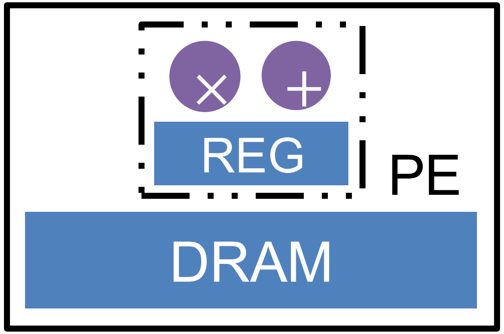
Analysis: What is the Achieved Traffic?

```
for k in range(K):
    for m in range(M):
        a_reg = A[k, m]
        b_reg = B[k]
        z_reg = Z[m]
        z_reg += a_reg * b_reg
        Z[m] = z_reg
```



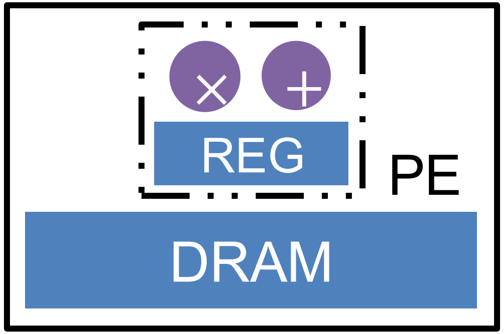
Exploit Stationarity

```
for k in range(K):

    for m in range(M):
        a_reg = A[k, m]
        b_reg = B[k]
        z_reg = Z[m]
        z_reg += a_reg * b_reg
        Z[m] = z_reg
```



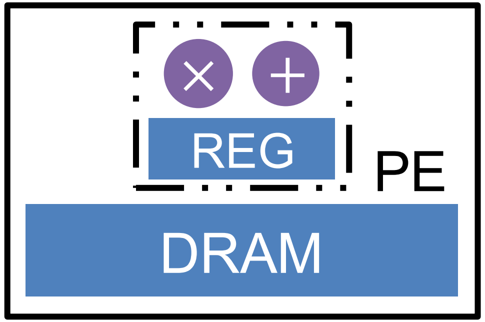
Exploit Stationarity

```
for k in range(K):
    b_reg = B[k]
    for m in range(M):
        a_reg = A[k, m]
        b_reg = B[k]
        z_reg = Z[m]
        z_reg += a_reg * b_reg
        Z[m] = z_reg
```



Analysis: What is the Achieved Traffic?

```
for k in range(K):
  b_reg = B[k]
  for m in range(M):
    a_reg = A[k, m]
    z_reg = Z[m]
    z_reg += a_reg * b_reg
    Z[m] = z_reg
```



Achieved memory traffic:

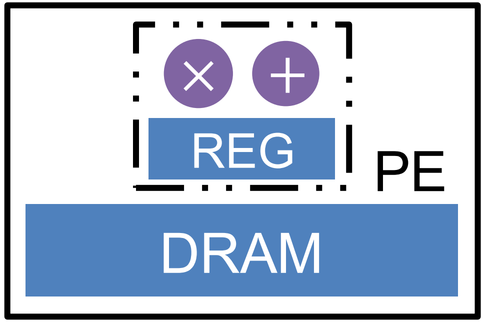
- $K \times$ M loads of $A_{k,m}$
- K loads of B_k
- $(K-1) \times M$ loads of Z_m
- $K \times M$ stores of Z_m

Analysis: What is the Achieved Traffic?

```
for k in range(K):
  for m in range(M):
    Z[m] += A[k,m] * B[k]
```

Achieved memory traffic:

- $K \times$ M loads of $A_{k,m}$
- K loads of B_k
- $(K-1) \times M$ loads of Z_m
- $K \times M$ stores of Z_m



Loads and stores are always derivable from the loop order

Analysis: What is the Achieved CI?

Multiplications: $K \times \mathrm{M}$

Achieved memory traffic: $3 \times K \times M - M + K$

Achieved compute intensity: $\dfrac{K \times \mathrm{M}}{3 \times K \times M - M + K}$

Example: Best Case vs Achieved CI

$$K = 250; \;\; M = 100$$

- Best Case CI

$$\frac{K \times \mathrm{M}}{K \times \mathrm{M} + \mathrm{K} + \mathrm{M}} =$$

$$\frac{250 \times 100}{250 \times 100 + 250 + 100} =$$

$$\mathbf{0.99}\ \boldsymbol{Multiplications/value}$$

- Achieved CI

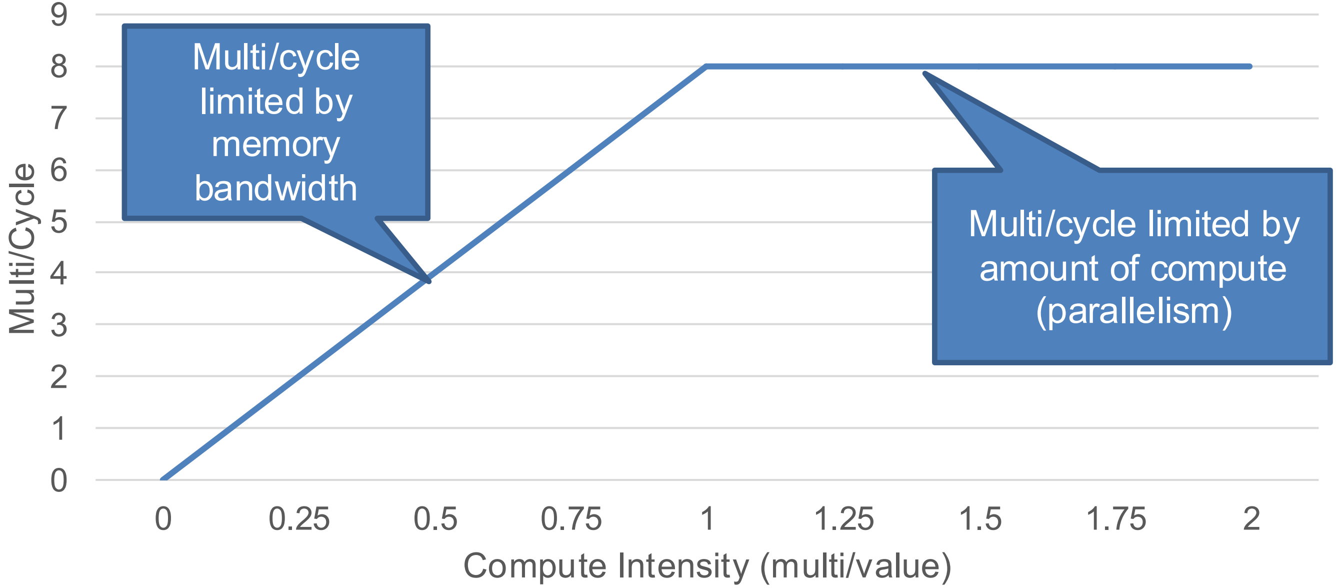
$$\frac{K \times \mathrm{M}}{3 \times K \times M - M + K} =$$

$$\frac{250 \times 100}{3 \times 250 \times 100 - 100 + 250} =$$

$$\mathbf{0.33}\ \boldsymbol{Multiplications/value}$$

Roofline Model

L=8
(8 MACs/cycle)



Williams, Samuel, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." Communications of the ACM 52.4 (2009): 65-76.

Roofline Model

- Roofline Model is a way to visualize throughput given
  - Memory bandwidth, amount of parallelism, and computational intensity
  - Tells you if more parallelism would help, or more memory bandwidth
    - When memory bound, increasing number of lanes will not increase throughput → parallelism does not always equal speed up in throughput
  - Tells you how far you are from limit
    - Away from limit due to overhead (e.g., stalls, instruction overhead, mapping limitations)
- Compute intensity
  - Theoretical upper bound [max reuse] (best-case compute intensity) (computed in Lab 1)
  - Actual implementation depends on processing order (amount of reuse exploited by hardware)
- Roofline model can be draw for each level of the memory hierarchy (though typically for DRAM)

Accelerator Design Methodology

(1) Describe the architecture

Select from a library of components and organize them by writing an accelerator specification

(2) Develop the workload

Write the cascade, mapping, format, and binding specifications

(3) Evaluate the workload

Model the workload and analyze with metrics like number of computes, memory traffic, and compute intensity

(4) Compare implementations

Write corresponding specifications, normalize hardware parameters, and reevaluate

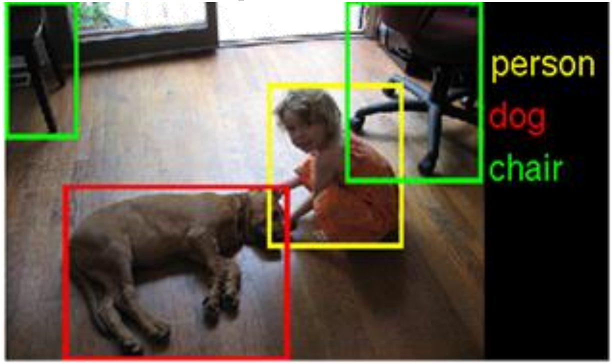
(5) Optimize the design

Incrementally modify one or more specifications

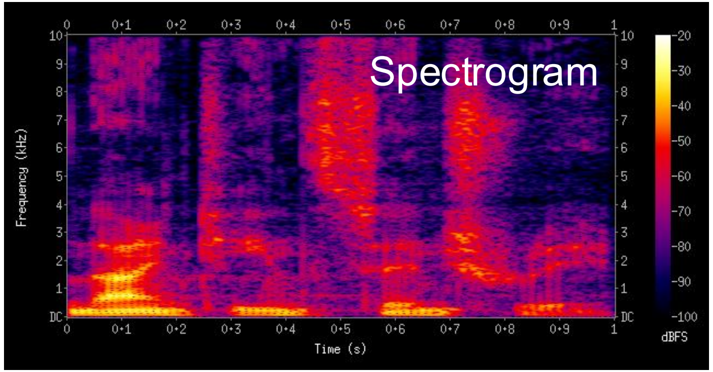
Convolutional Neural Networks (CNNs)

Applications of CNN

Computer Vision

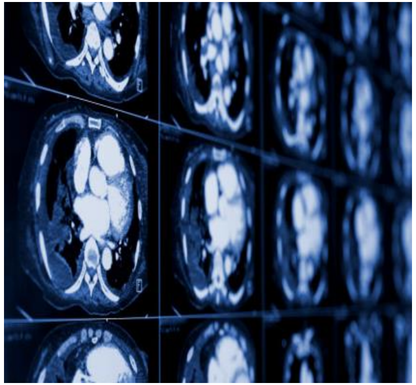


Speech Recognition

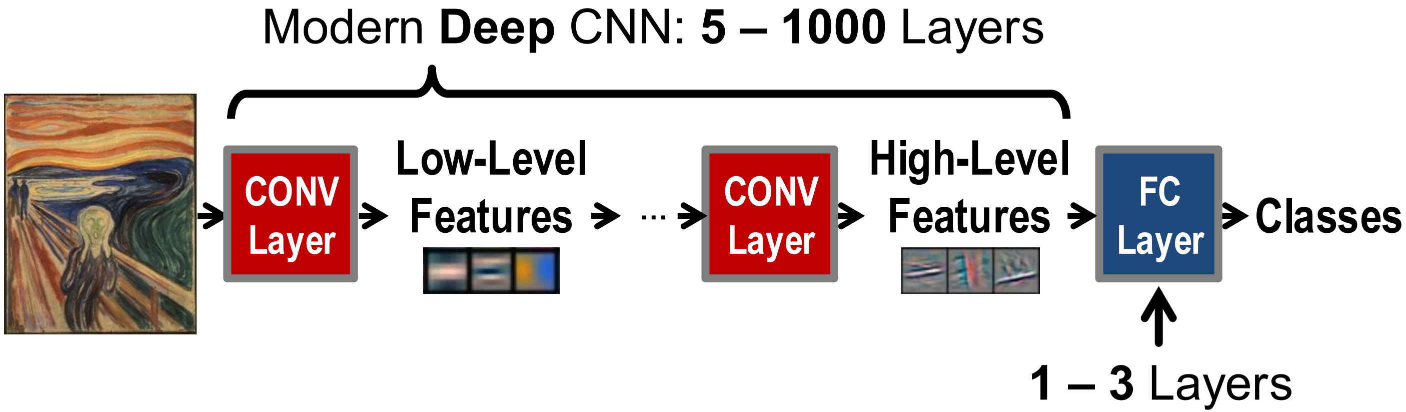


Game Play

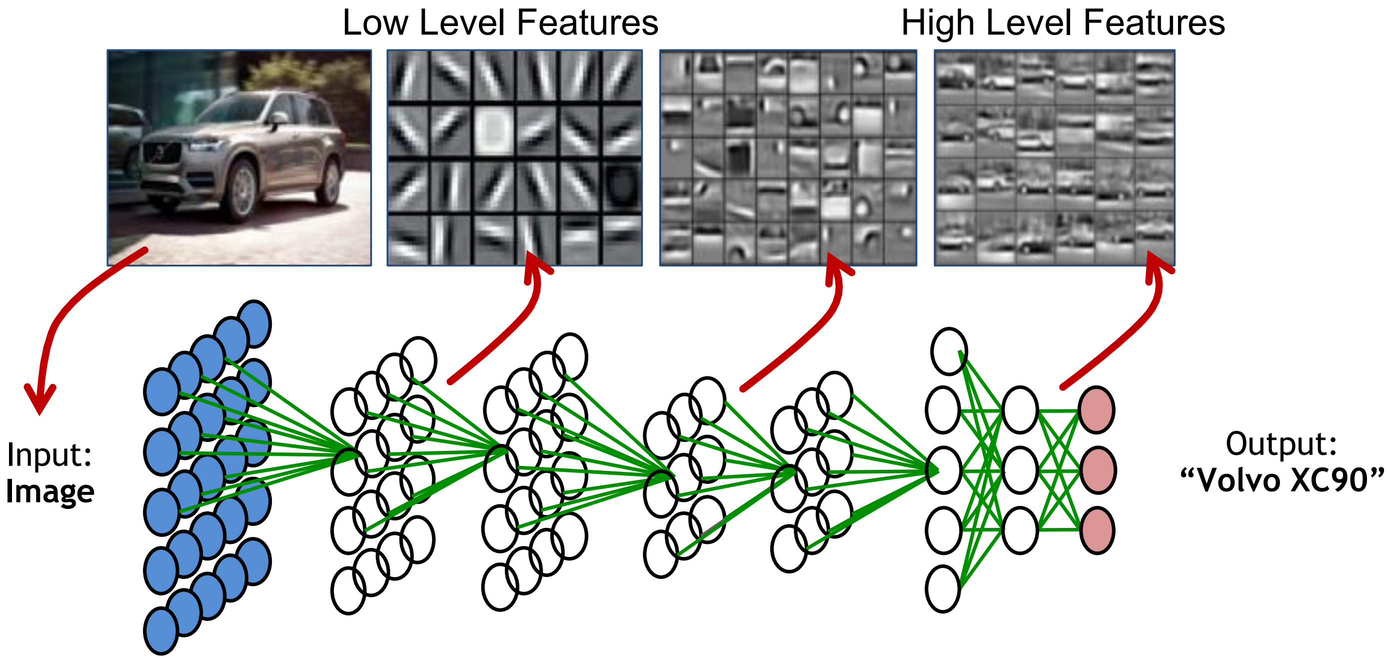
Medical



Convolutional Neural Networks

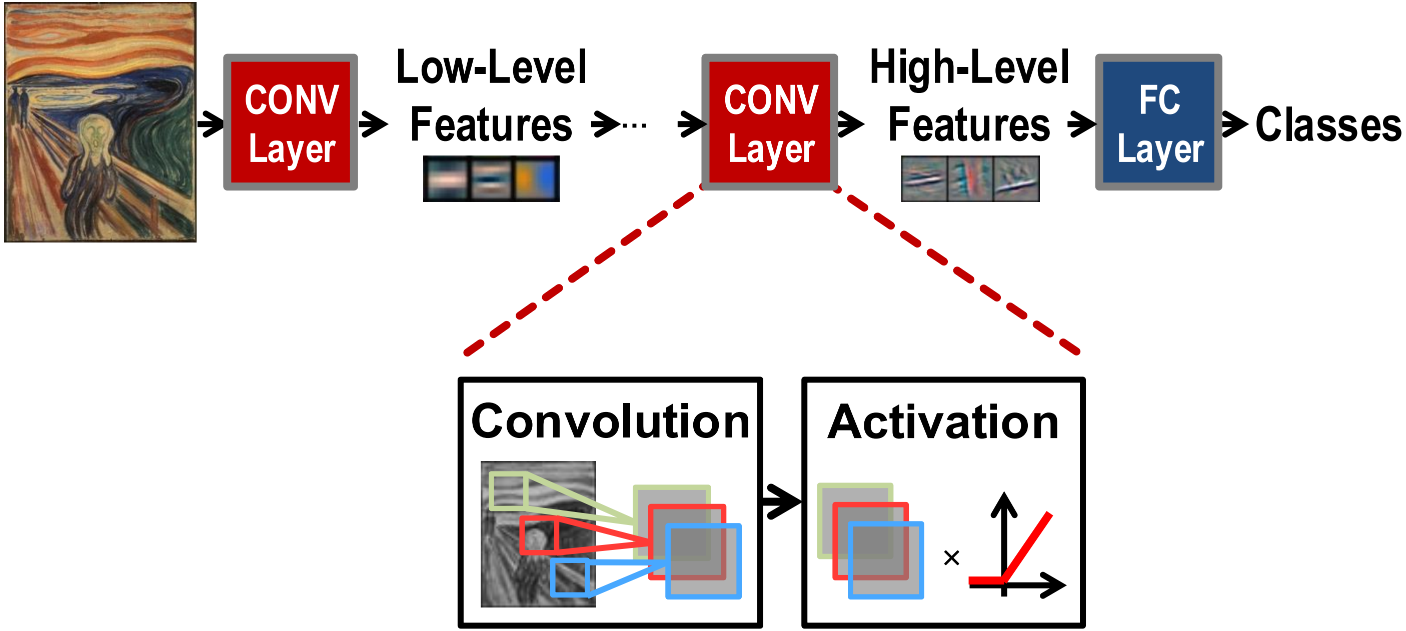


Depth of Network

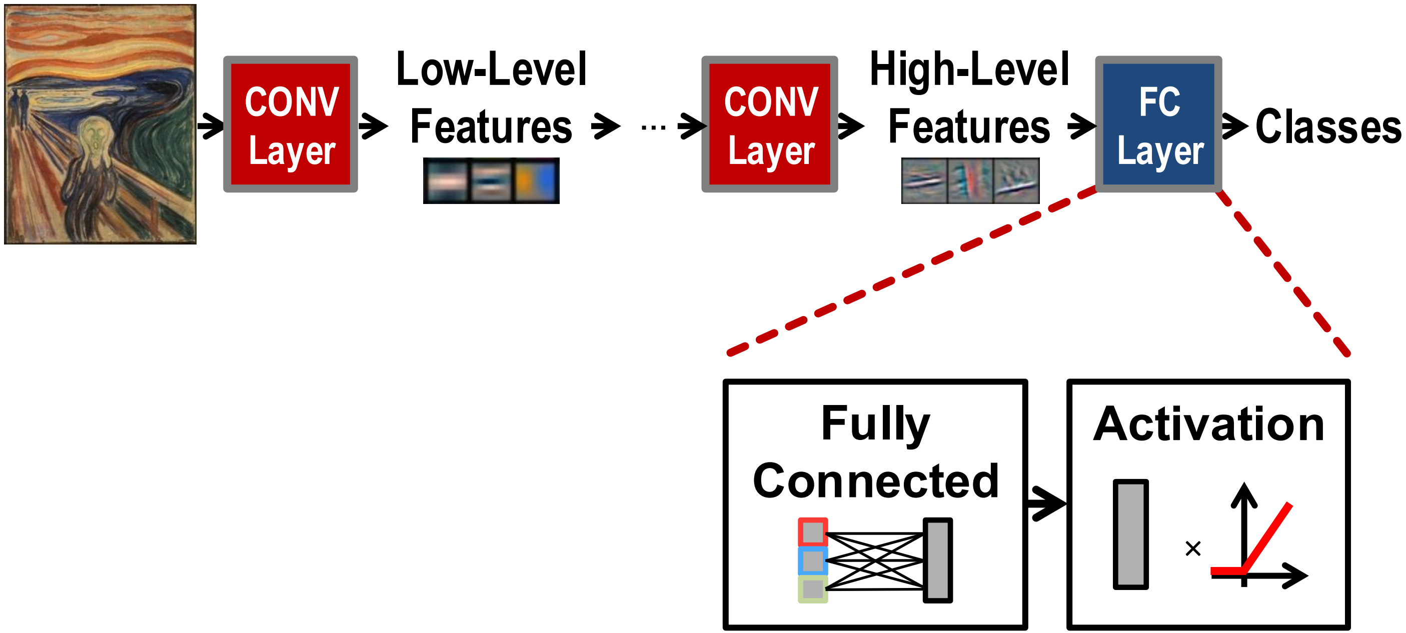


Modified Image Source: [**Lee**, *CACM* 2011]

Convolutional Neural Networks

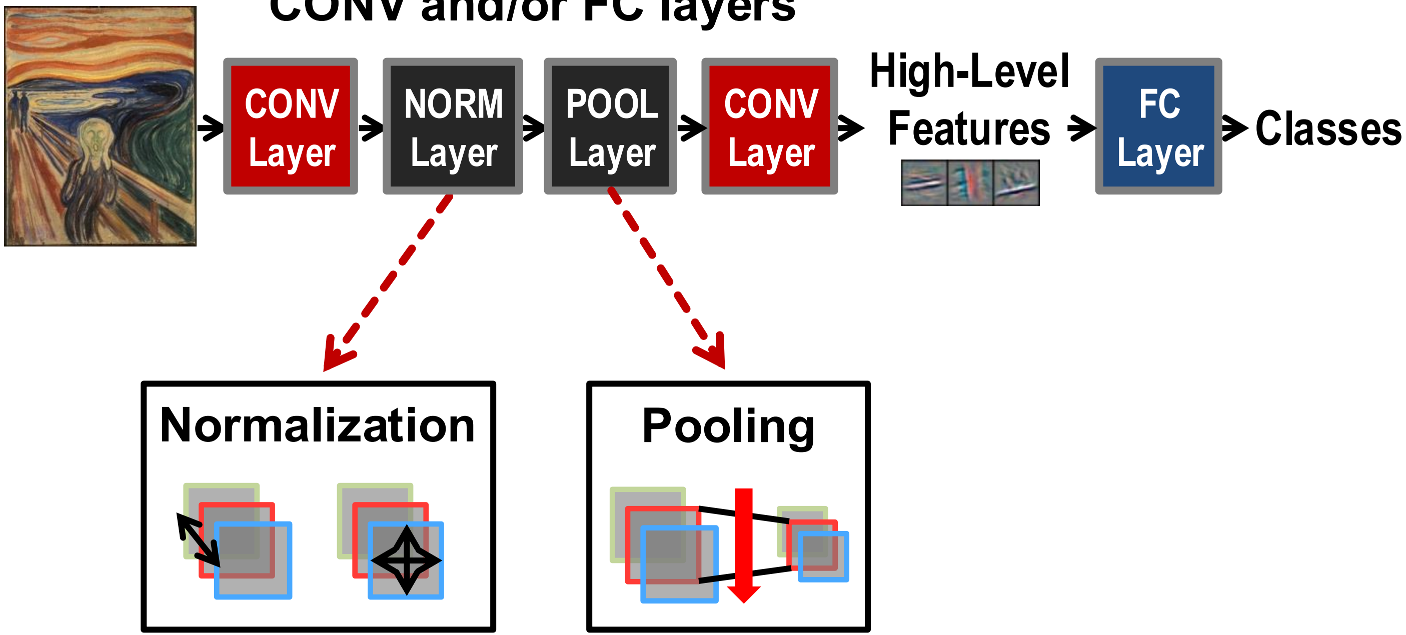


Convolutional Neural Networks

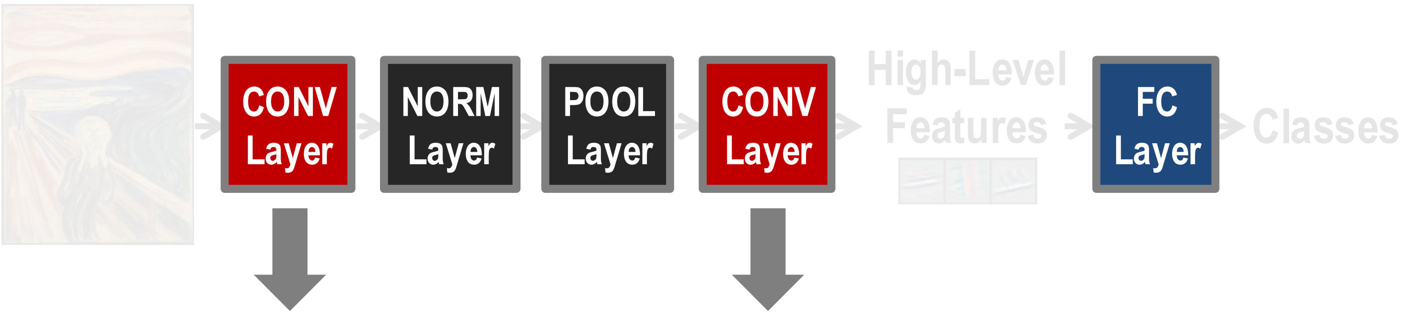


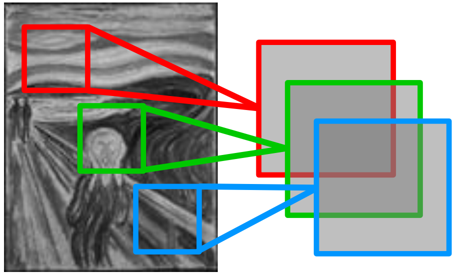
Convolutional Neural Networks

Optional layers in between CONV and/or FC layers



Convolutional Neural Networks



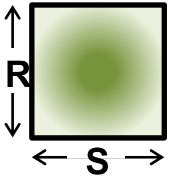


Convolutions account for more than 90% of overall computation, dominating **runtime** and **energy consumption**

Convolution (CONV) Layer

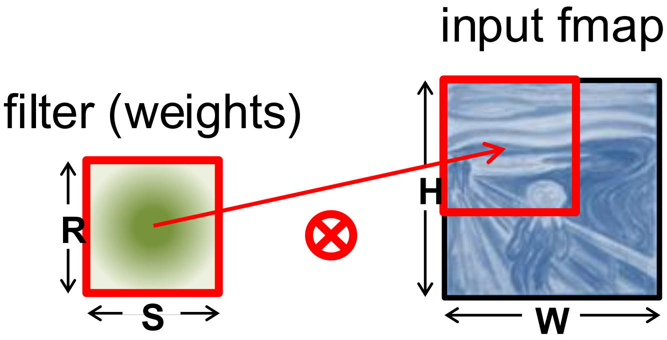
a plane of input activations
a.k.a. **input feature map (fmap)**

filter* (weights)



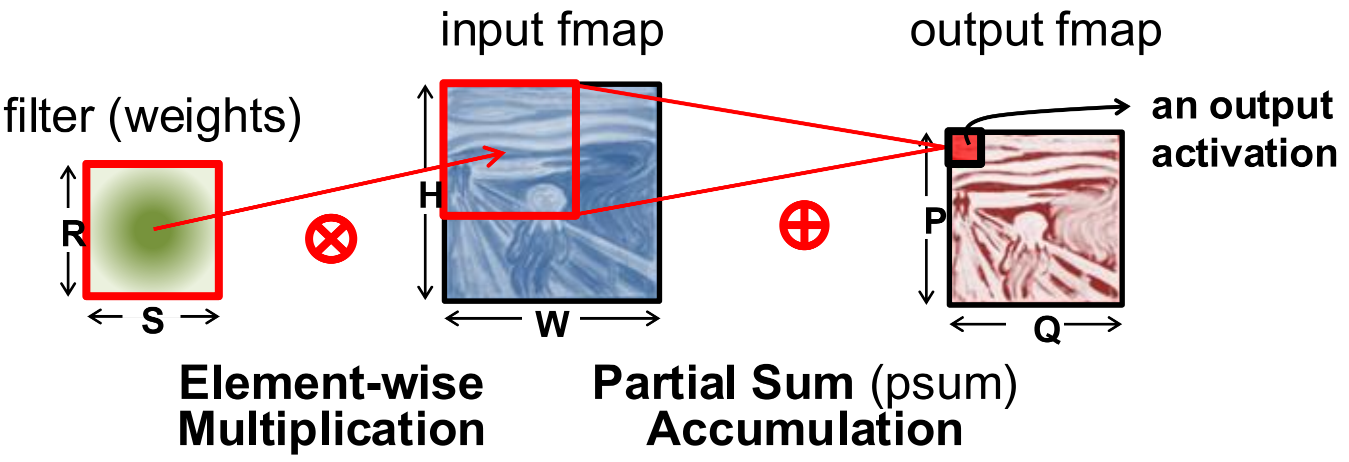
* also referred to as **kernel**

Convolution (CONV) Layer

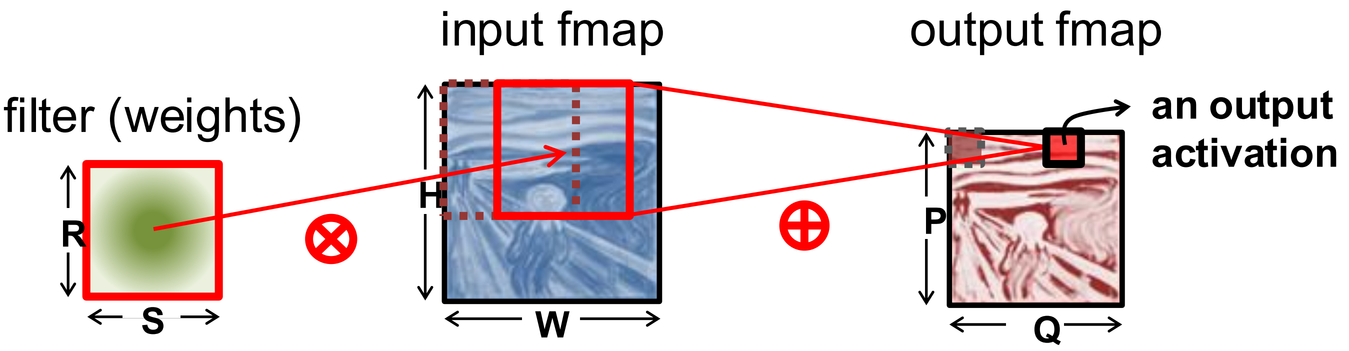


Element-wise Multiplication

Convolution (CONV) Layer



Convolution (CONV) Layer



Sliding Window Processing

2D Convolution Example

Convolution (Stride 1)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Filter support: 3x3
Also referred to as the **receptive field**
(each output requires 9 multiplications*)

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

*assume no optimization for zeros

2D Convolution Example

Convolution (Stride 1)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

7

2D Convolution Example

Convolution (Stride 1)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

7	8

2D Convolution Example

Convolution (Stride 1)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

7	8	8

2D Convolution Example

Convolution (Stride 1)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

7	8	8
5		

2D Convolution Example

Convolution (Stride 1)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

7	8	8
5	6	

2D Convolution Example

Convolution (Stride 1)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

7	8	8
5	6	7

2D Convolution Example

Convolution (Stride 1)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map (3x3)

7	8	8
5	6	7
6	5	7

Size of **Output Feature Map** = (Size of **Input Feature Map** – Size of **Filter** + **Stride**) / **Stride**

# of multiplications?

2D Convolution Example

Convolution (Stride 2)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

7

2D Convolution Example

Convolution (Stride 2)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

7	8

2D Convolution Example

Convolution (Stride 2)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

7	8
6	

2D Convolution Example

Convolution (Stride 2)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map

7	8
6	7

2D Convolution Example

Convolution (Stride 2)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map (2x2)

7	8
6	7

Size of **Output Feature Map** = (Size of **Input Feature Map** – Size of **Filter** + **Stride**) / **Stride**

# of multiplications?

2D Convolution Example

Convolution (Stride 3)

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (5x5)

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

Output Feature Map (1x1)

7

Size of **Output Feature Map** = (Size of **Input Feature Map** – Size of **Filter** + **Stride**) / **Stride**

# of multiplications?

Impact of Stride on Convolution

Stride > 1 is equivalent to **downsampling** the output feature map when Stride =1

Stride 1

Output Feature Map (3x3)

7	8	**8**
5	6	7
6	5	**7**

Stride 2

Output Feature Map (2x2)

7	8
6	7

Stride 3

Output Feature Map (1x1)

7

Zero Padding

- The size of the output shrinks relative to the input
- Use **zero padding** to control the size of the output
- Can set padding based on filter size such that the output size is equal to original the input size

0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

→

0	0	0	0	0	0	0
0	0	1	2	3	2	0
0	1	2	2	2	0	0
0	0	1	0	1	3	0
0	1	2	2	1	0	0
0	0	1	0	3	1	0
0	0	0	0	0	0	0

2D Convolution Example

Convolution (Stride 1) + zero padding

Filter (3x3)

0	1	0
1	1	1
0	1	0

Input Feature Map (7x7)

0	0	0	0	0	0	0
0	0	1	2	3	2	0
0	1	2	2	2	0	0
0	0	1	0	1	3	0
0	1	2	2	1	0	0
0	0	1	0	3	1	0
0	0	0	0	0	0	0

Output Feature Map (5x5)

2	5	8	9	5
3	**7**	**8**	**8**	4
3	**5**	**6**	**7**	4
3	**6**	**5**	**7**	5
2	3	6	5	4

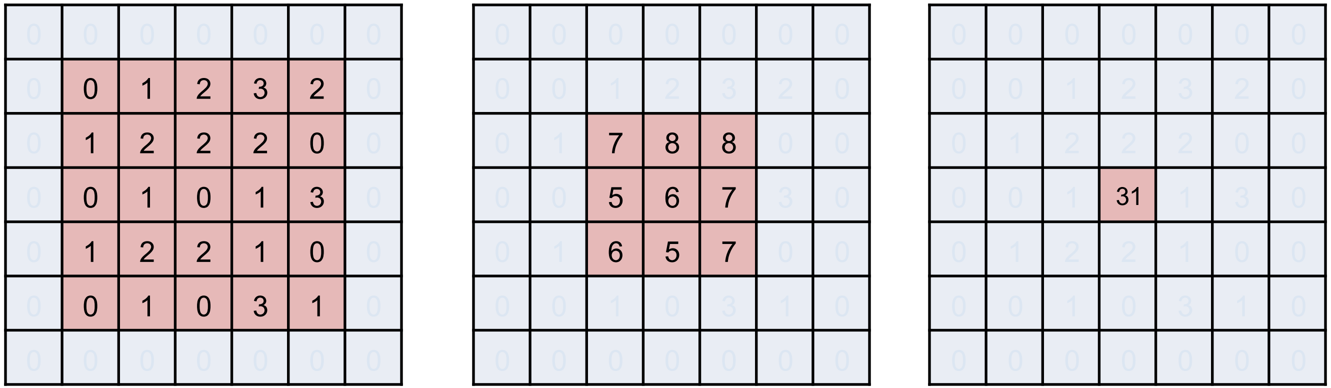
Zero Padding in PyTorch

- **padding** (*python:int or tuple, optional*) added to input. Default: 0
  - https://pytorch.org/docs/stable/nn.html#padding-layers
  - Ex: padding=1, pad 1 to the top, bottom, right, and left.
  - Ex. padding=[1,2], pad 1 to the top and bottom, pad 2 to the right and left
- Default: No zero padding
  - filter is RxS and input is HxW, and stride U
  - output is (H-R+U)/U x (W-S+U)/U
- Padding=[(R-1)/2, (S-1)/2]: zero padding so that output remains the same for U=1
  - filter is RxS and input is HxW, and stride U
  - output is ceil(H/U) x ceil(W/U)
- Padding is not always explicitly defined, but can be inferred from the size of the feature map
  - Deep networks use padding to prevent feature maps from shrinking
- Different frameworks can use different types of padding

Depth of Network: Convolution

As you go deeper into the network, more pixels contribute to each activation.

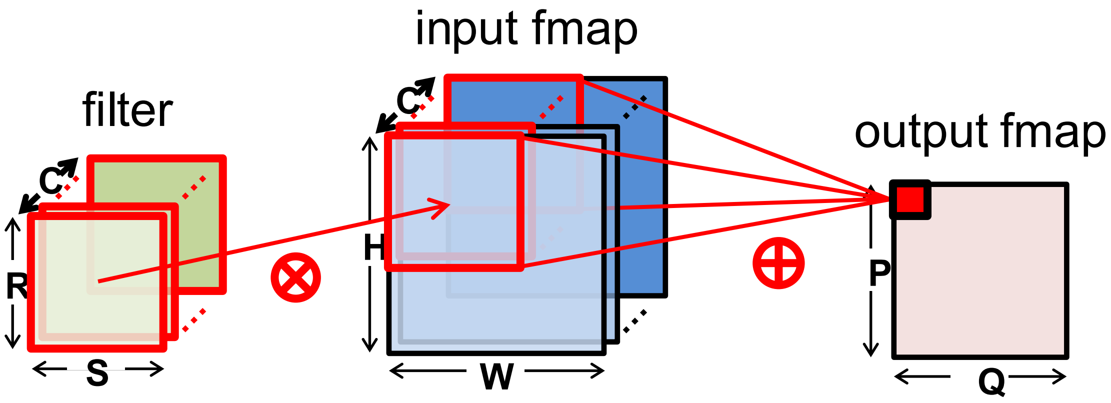
Example: 3x3 filter



Input to Layer 1 Layer 2 Layer 3

Feature maps of deep layers typically give higher level features

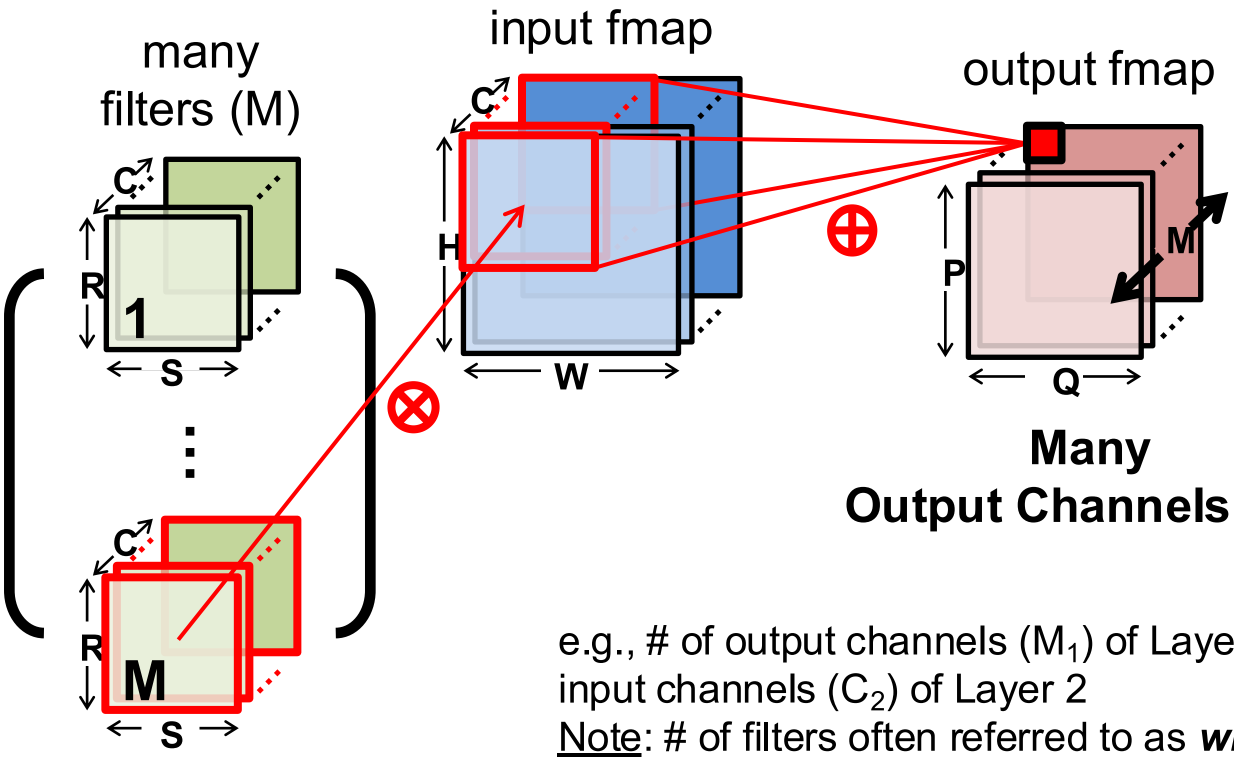
Convolution (CONV) Layer



Many Input Channels (C)

e.g., For Layer 1, C=3 for the red, green, and blue components of an image

Convolution (CONV) Layer



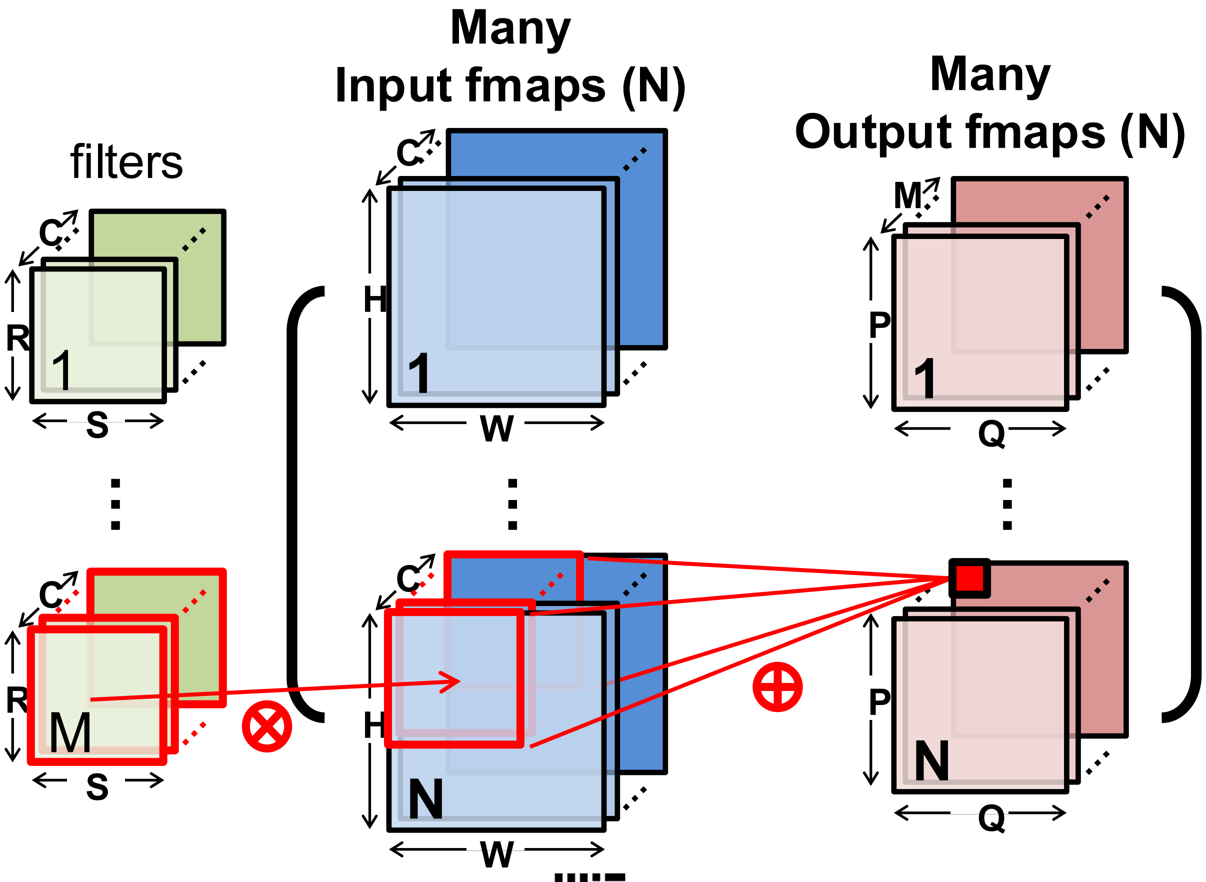
Many
Output Channels (M)*

e.g., # of output channels (M_1) of Layer 1 becomes # of input channels (C_2) of Layer 2

Note: # of filters often referred to as ***width of network***

*some works use K rather than M

Convolution (CONV) Layer



Batch Size (N)

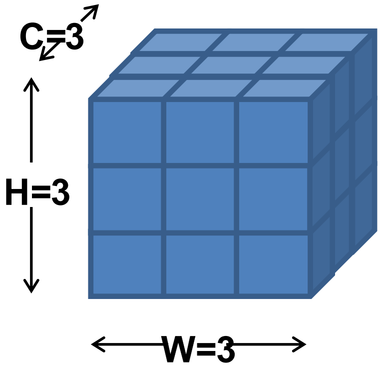
CNN Decoder Ring

- **N – Number of input fmaps/output fmaps (batch size)**
- **C – Number of channels in input fmaps (activations) & filters (weights)**
- **H – Height of input fmap (activations)**
- **W – Width of input fmap (activations)**
- **R – Height of filter (weights)**
- **S – Width of filter (weights)**
- **M – Number of channels in output fmaps (activations)**
- **P – Height of output fmap (activations)**
- **Q – Width of output fmap (activations)**
- **U – Stride of convolution**

These variables define the **rank** and **shape** of the various tensors (input fmap, filter, output fmap)

Input Feature Map (fmap) Tensor

Input fmap (activations)



I[C][H][W]

In this example, the input feature map has **three ranks*** named C, H and W

The **rank shapes are** C=3, H=3, and W=3

*technically also has fourth rank N, with shape of N=1

CONV Layer Tensor Computation

Output fmap (O)

Biases (B)

Input fmap (I)

Filter weights (W)

$$\underline{\mathbf{o}[n][m][p][q]} = \underline{\mathbf{b}[m]} + \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} \underline{\mathbf{i}[n][c][Up+r][Uq+s]} \times \underline{\mathbf{f}[m][c][r][s]},$$

$$0 \le n < N, 0 \le m < M, 0 \le p < P, 0 \le q < Q,$$
$$P = (H - R + U)/U, Q = (W - S + U)/U.$$

Shape Parameter	Description
N	batch size of 3-D fmaps
M	# of 3-D filters / # of ofmap channels
C	# of ifmap/filter channels
H/W	ifmap plane height/width
R/S	filter plane height/width (= H or W in FC)
P/Q	ofmap plane height/width (= 1 in FC)

Einstein Notation (Einsum)

Algebraic Notation

$$\mathbf{o}[n][m][p][q] = \mathbf{b}[m] + \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} \mathbf{i}[n][c][Up+r][Uq+s] \times \mathbf{f}[m][c][r][s],$$

Einsum Notation

$$O_{n,m,p,q} = B_m + I_{n,c,U\times p+r,U\times q+s} \times F_{m,c,r,s}$$

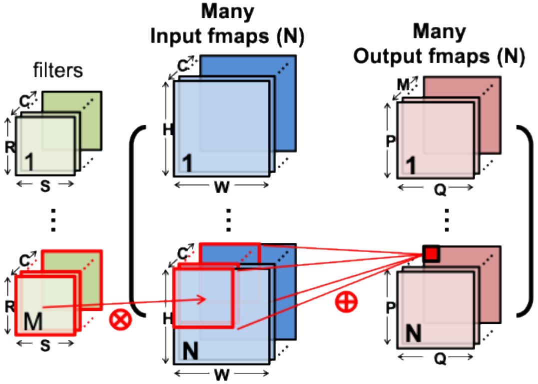
Einsum does not enforce any computational order

(function in Numpy, Pytorch and Tensorflow)

[**Einstein**, *Annalen der Physike* 1916], [**Kjolstad**, TACO, *OOPSLA* 2017], [**Parashar**, Timeloop, *ISPASS* 2019]

CONV Layer Implementation

Naïve 7-layer for-loop implementation:




```
for n in [0..N):
    for m in [0..M):
        for q in [0..Q):
            for p in [0..P):
```
for each output fmap value

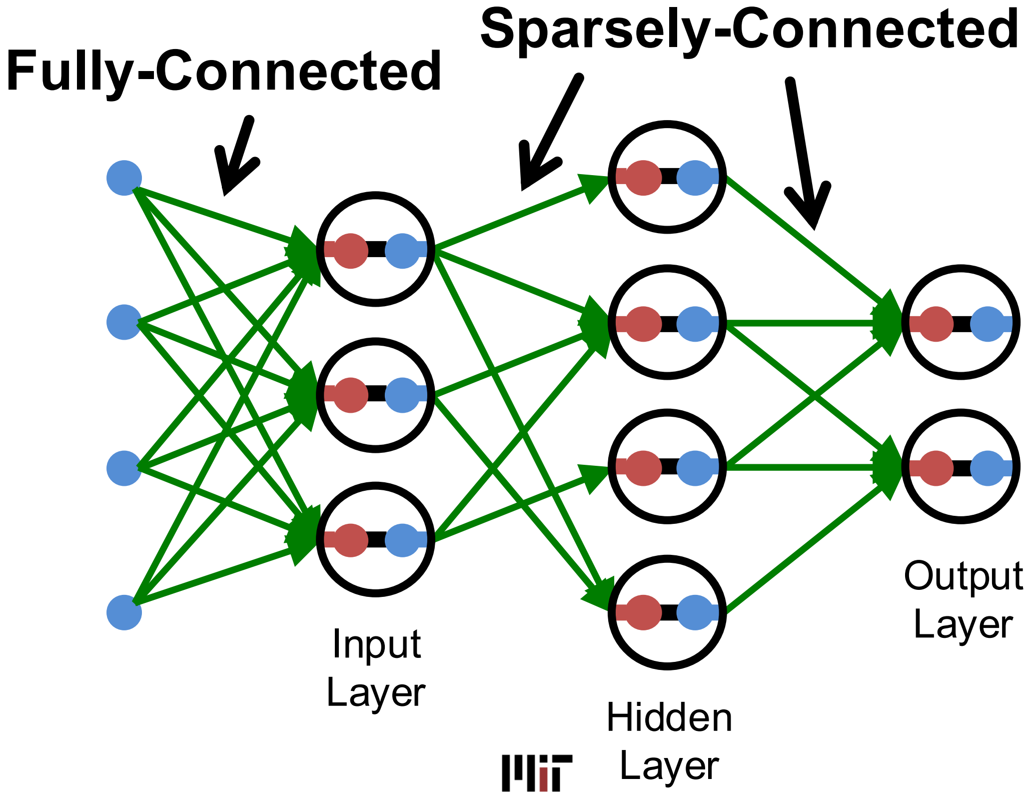
```
            O[n][m][p][q] = B[m];
            for c in [0..C):
                for r in [0..R):
                    for s in [0..S):
                        O[n][m][p][q] += I[n][c][Up+r][Uq+s]
                                       × F[m][c][r][s];

            O[n][m][p][q] = Activation(O[n][m][p][q]);
```
convolve a window and apply activation

Note that loop nest enforces an order → Einsum is more general!

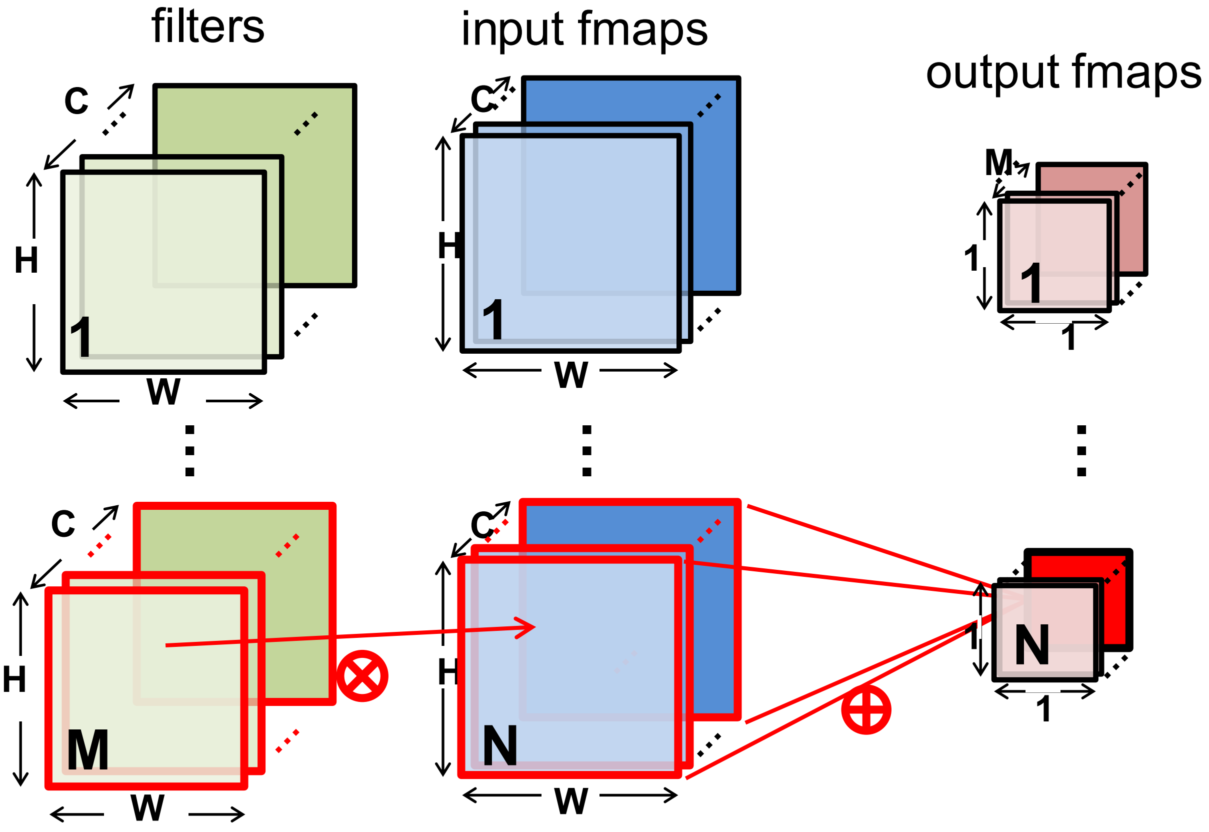
Fully Connected Layer

Fully-Connected (FC) Layer

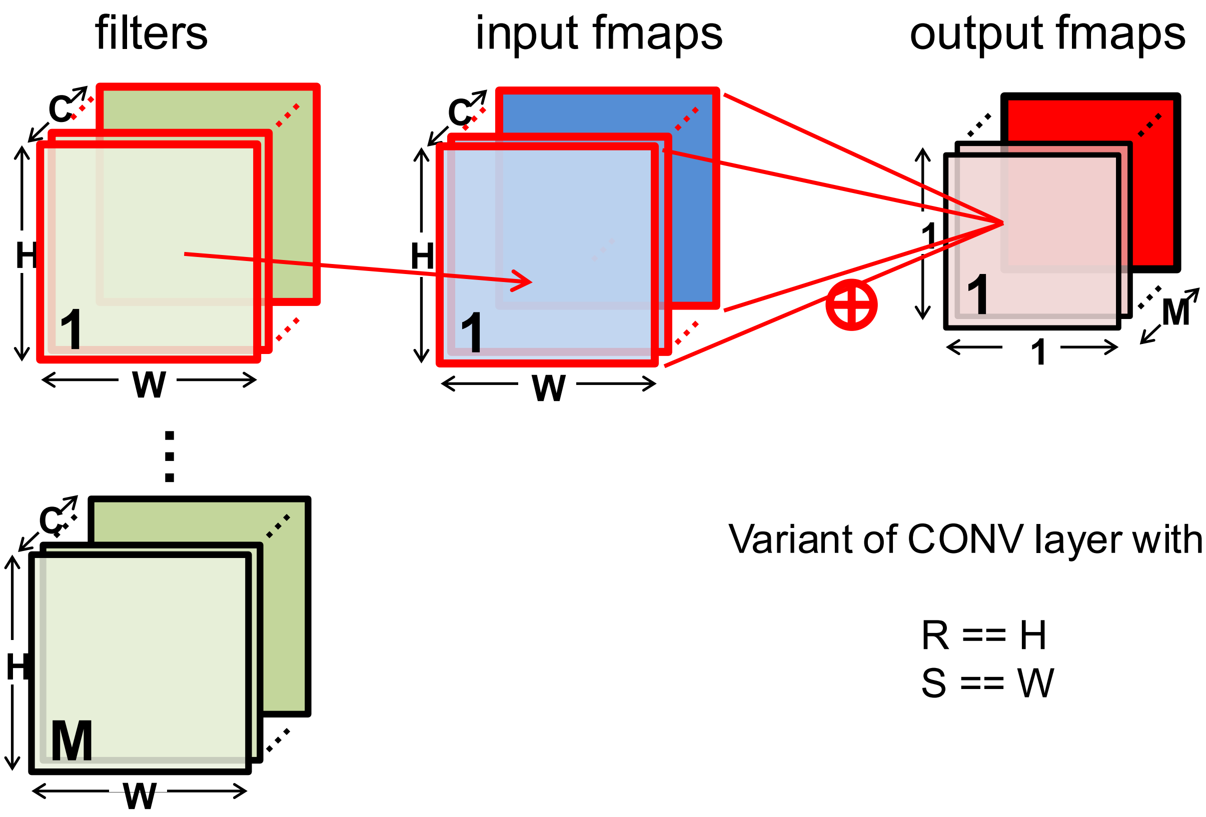
Fully-Connected: all i/p neurons connected to all o/p neurons



FC Layer – from CONV Layer POV



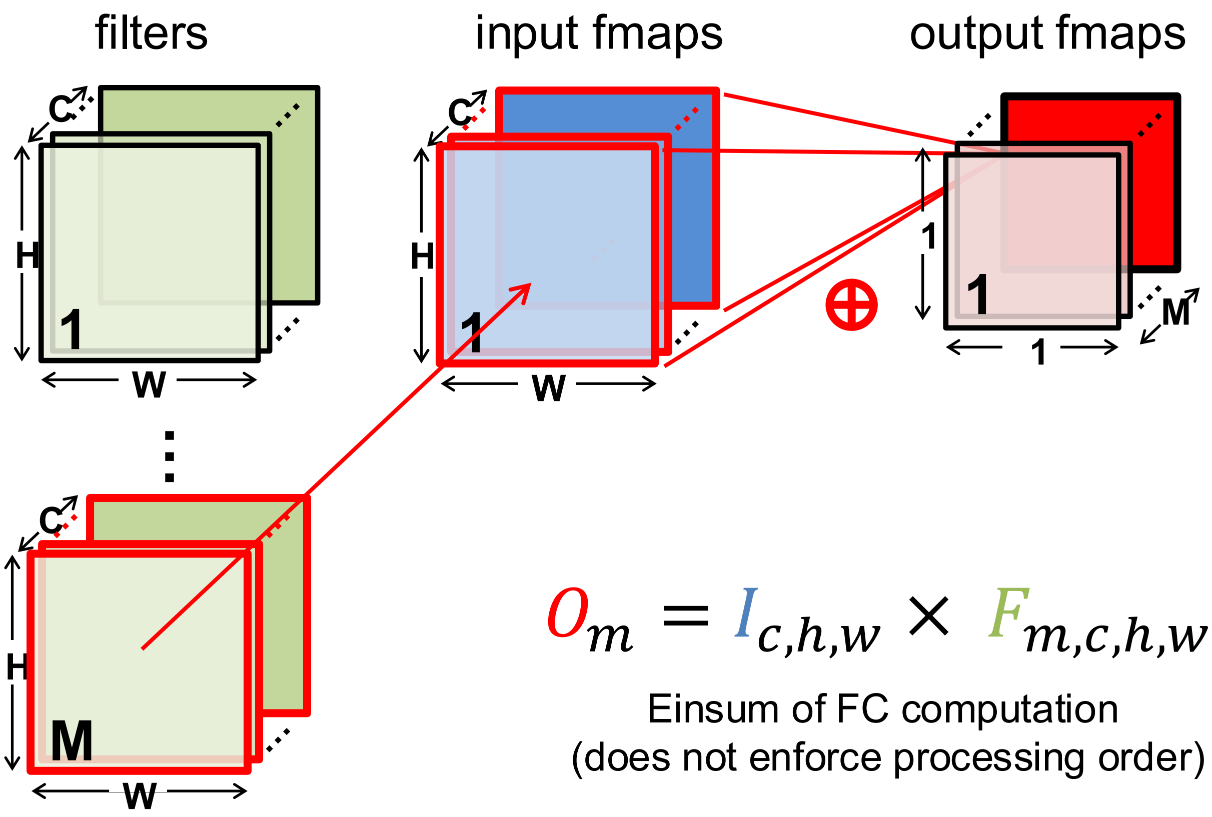
Fully Connected Computation



Variant of CONV layer with:

R == H
S == W

Fully Connected Computation



$$O_m = I_{c,h,w} \times F_{m,c,h,w}$$

Einsum of FC computation
(does not enforce processing order)

Fully Connected Computation

```
int i[C][H][W];      # Input activations
int f[M][C][H][W];   # Filter weights
int o[M];            # Output activations


for m in [0, M):
  o[m] = 0;
  for c in [0, C):
    for h in [0, H):
      for w in [0, W):
        o[m] += i[c][h][w]*f[m][c][h][w]
```

Should be bias, which we will ignore for simplicity

Loop nest of FC computation
(enforces some processing order)

Convert FC Compute to Matrix-Vector Multiply

```
int i[C][H][W];      # Input activations
int f[M][C][H][W];   # Filter weights
int o[M];            # Output activations


for m in [0, M):
  o[m] = 0;
  for c in [0, C):
    for h in [0, H):
      for w in [0, W):
        o[m] += i[c][h][w]*f[m][c][h][w]
```

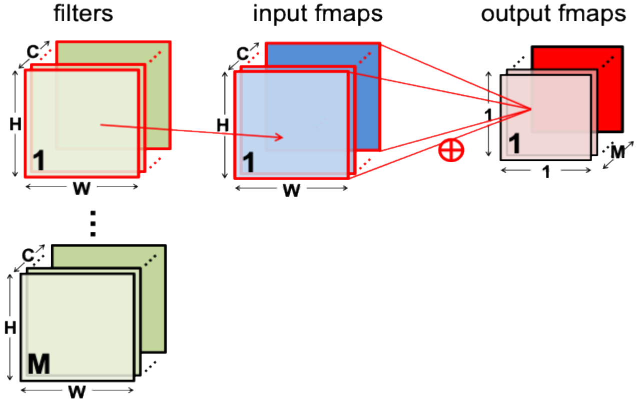
Flatten C, H, W ranks to CHW

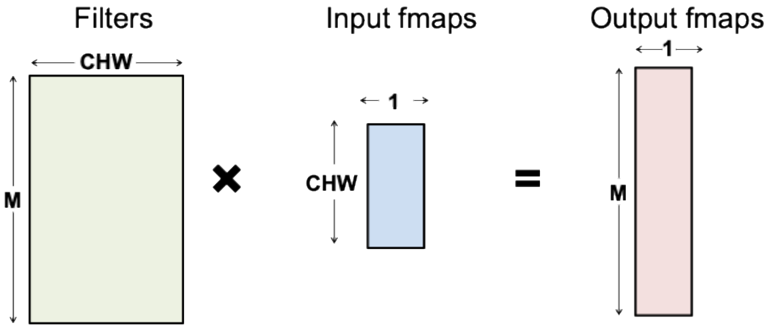
```
int i[CHW];          # Input activations
int f[M][CHW];       # Filter weights
int o[M];            # Output activations


for m in [0, M):
  o[m] = 0;
    for chw in [0, CHW):
        o[m] += i[chw]*f[CHW*m + chw]
```

Convert FC Compute to Matrix-Vector Multiply

```
int i[C][H][W];      # Input activations
int f[M][C][H][W];   # Filter weights
int o[M];            # Output activations


for m in [0, M):
  o[m] = 0;
  for c in [0, C):
    for h in [0, H):
      for w in [0, W):
        o[m] += i[c][h][w]*f[m][c][h][w]
```

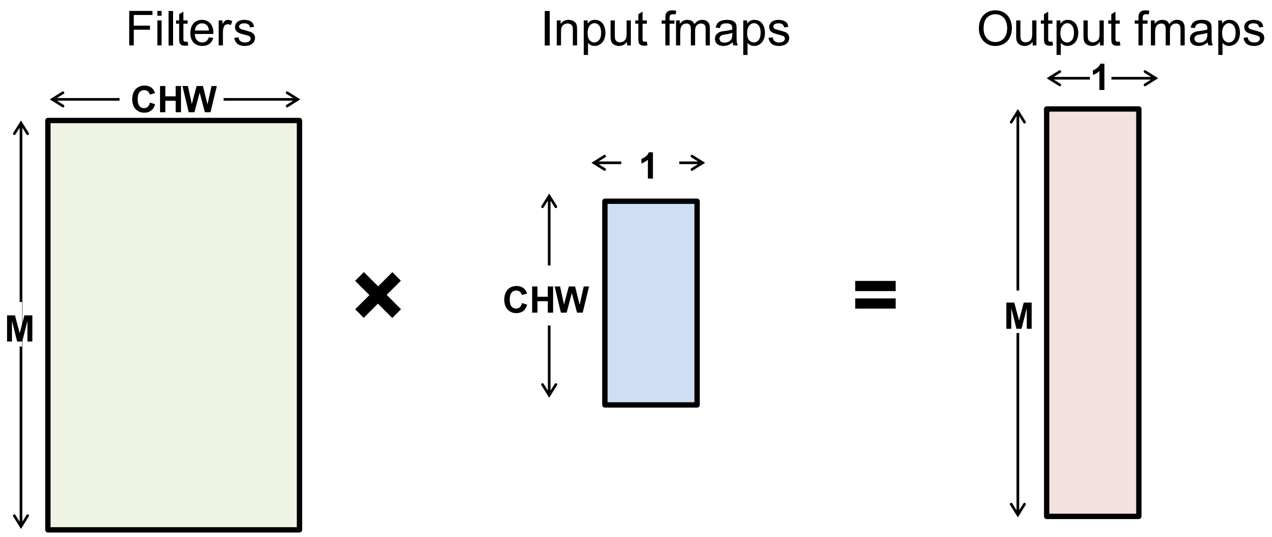
```
int i[CHW];          # Input activations
int f[M][CHW];       # Filter weights
int o[M];            # Output activations


for m in [0, M):
  o[m] = 0;
    for chw in [0, CHW):
      o[m] += i[chw]*f[m][chw]
```





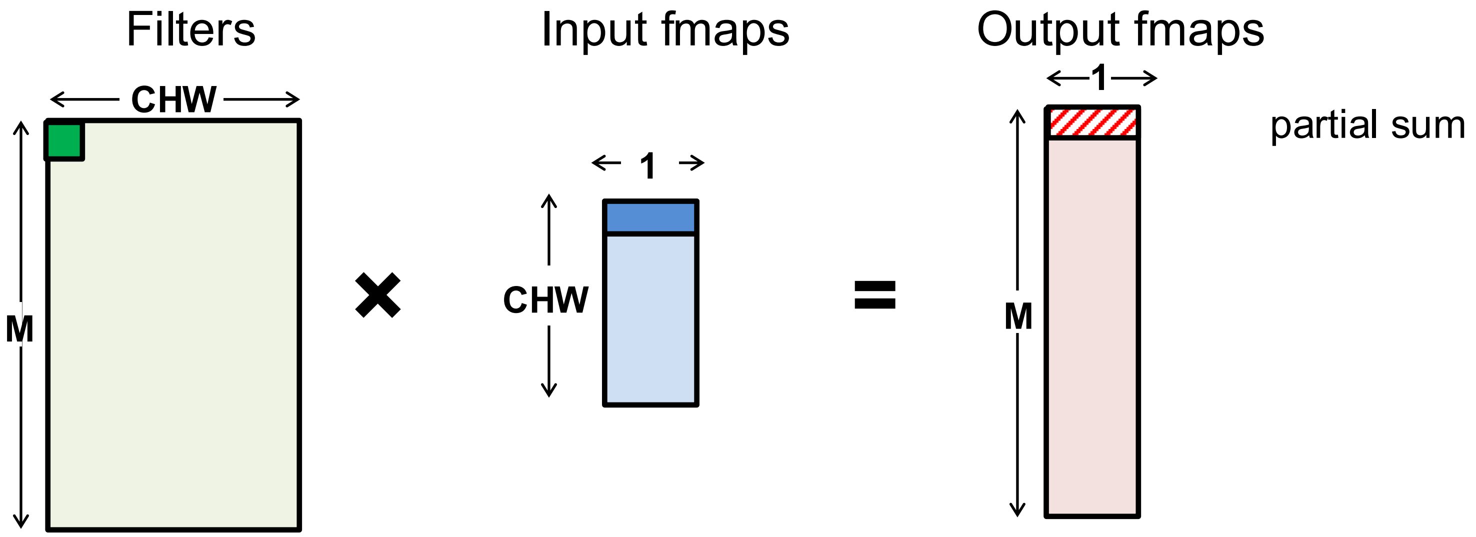
FC Compute as Matrix-Vector Multiply

Multiply all inputs in all channels by a weight and sum



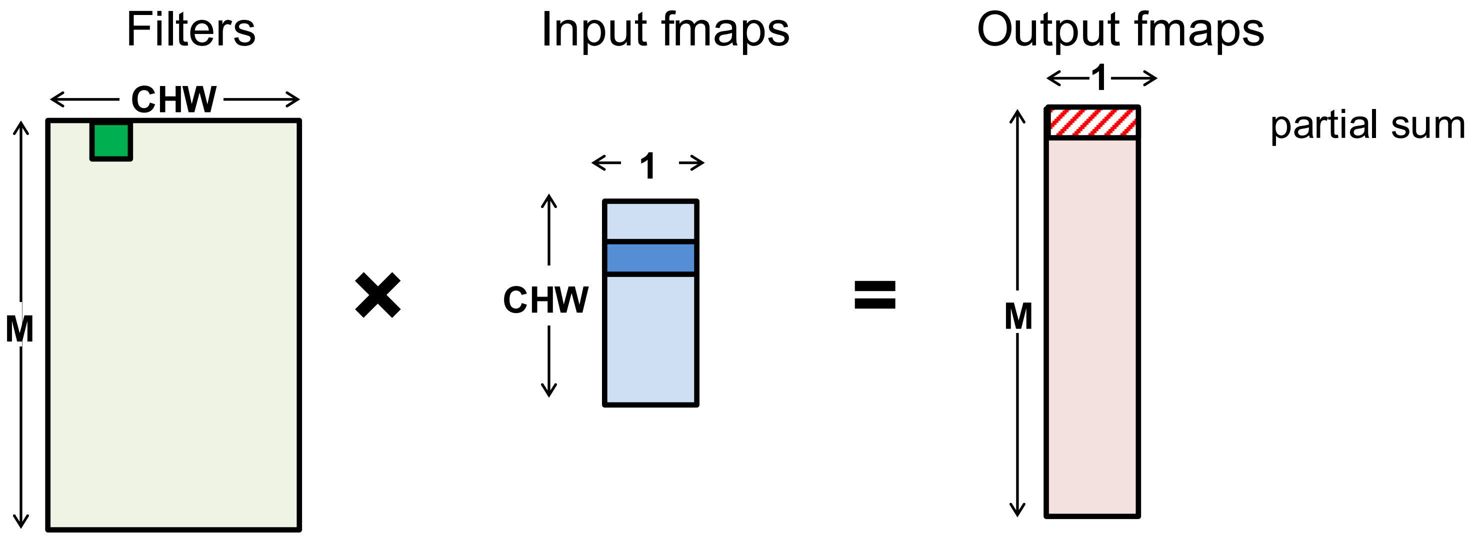
FC Compute as Matrix-Vector Multiply

Multiply all inputs in all channels by a weight and sum
(increment chw)



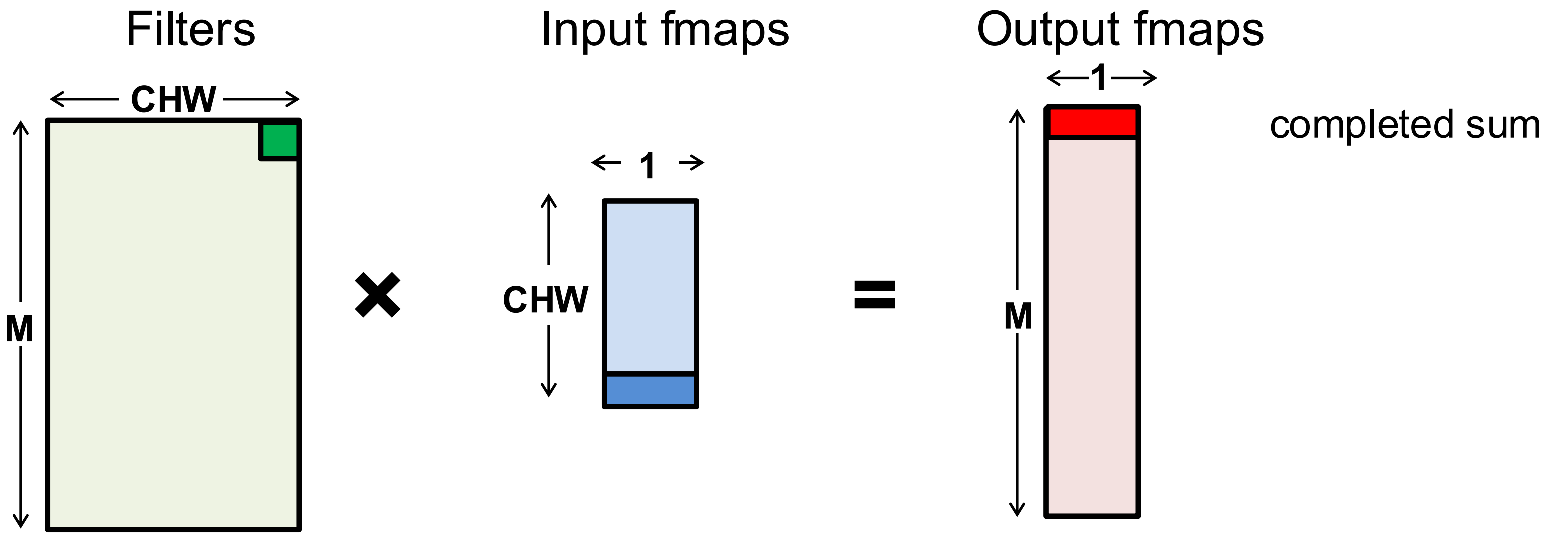
FC Compute as Matrix-Vector Multiply

Multiply all inputs in all channels by a weight and sum (increment chw)



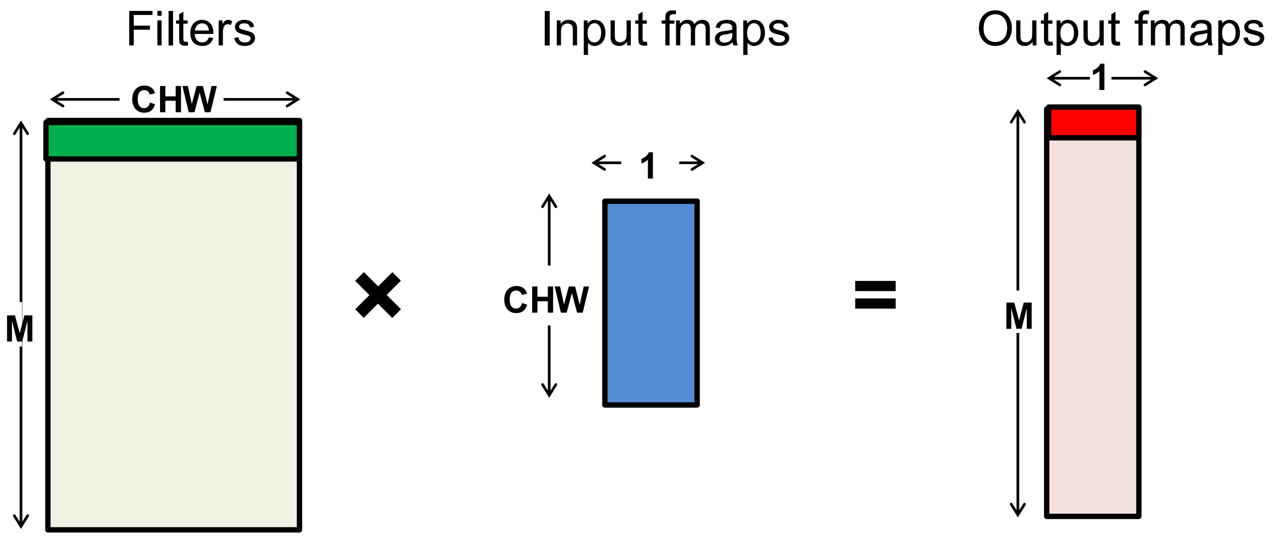
FC Compute as Matrix-Vector Multiply

Multiply all inputs in all channels by a weight and sum (increment chw)



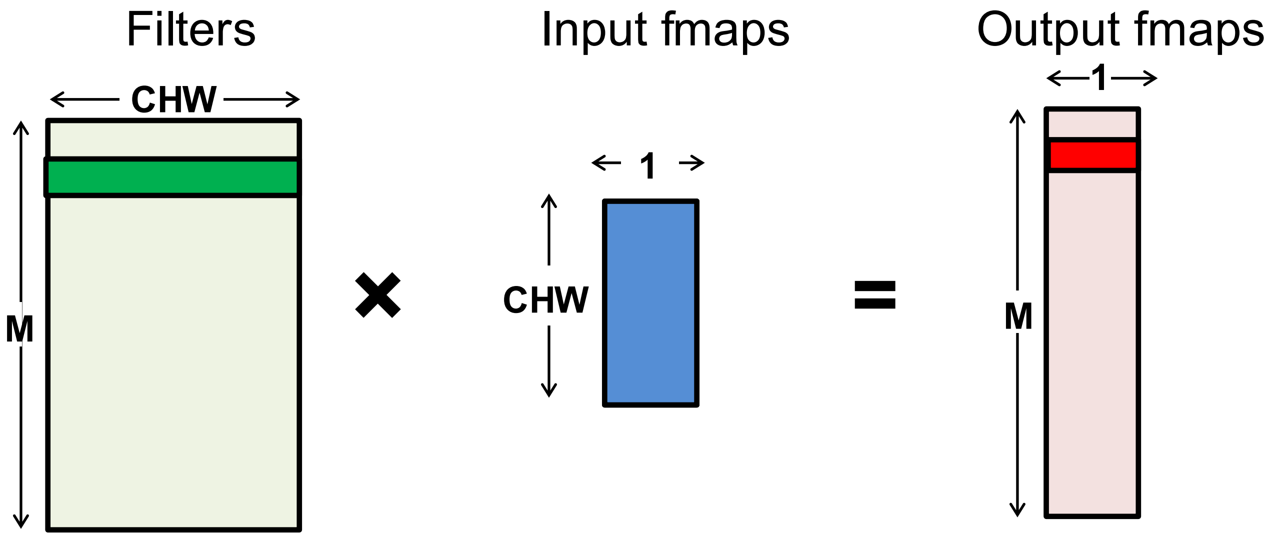
FC Compute as Matrix-Vector Multiply

Multiply all inputs in all channels by a weight and sum



FC Compute as Matrix-Vector Multiply

Multiply all inputs in all channels by a weight and sum (increment m)



Einsum for Flattened FC

Original | Flattened

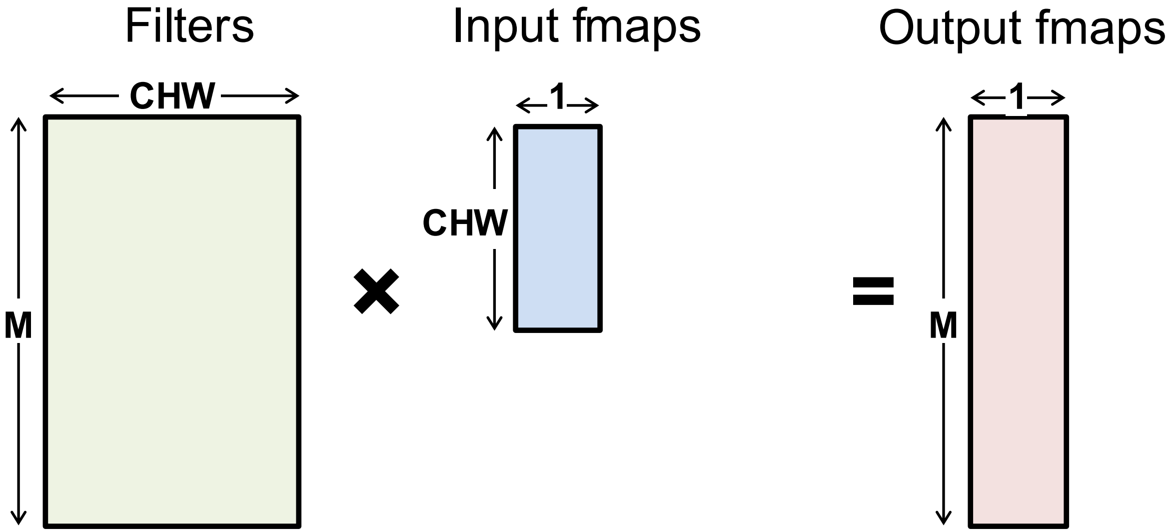
$$I_{c,h,w} \rightarrow I_{H\times W\times c+W\times h+w} \rightarrow I_{chw}$$

$$F_{m,c,h,w} \rightarrow F_{m,H\times W\times c+W\times h+w} \rightarrow F_{m,chw}$$

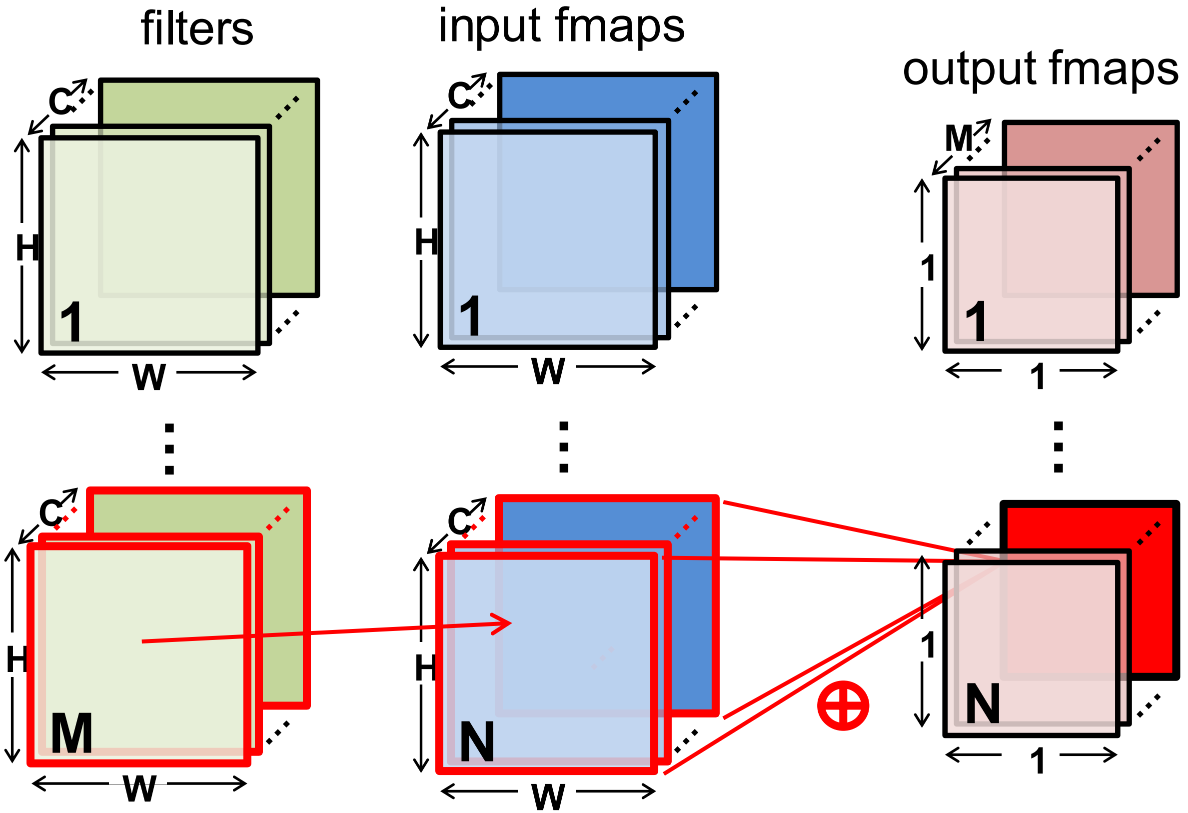
$$O_m = I_{c,h,w} \times F_{m,c,h,w} \rightarrow O_m = I_{chw} \times F_{m,chw}$$

Einsum for FC as Matrix Vector

$$O_m = I_{chw} \times F_{m,chw}$$

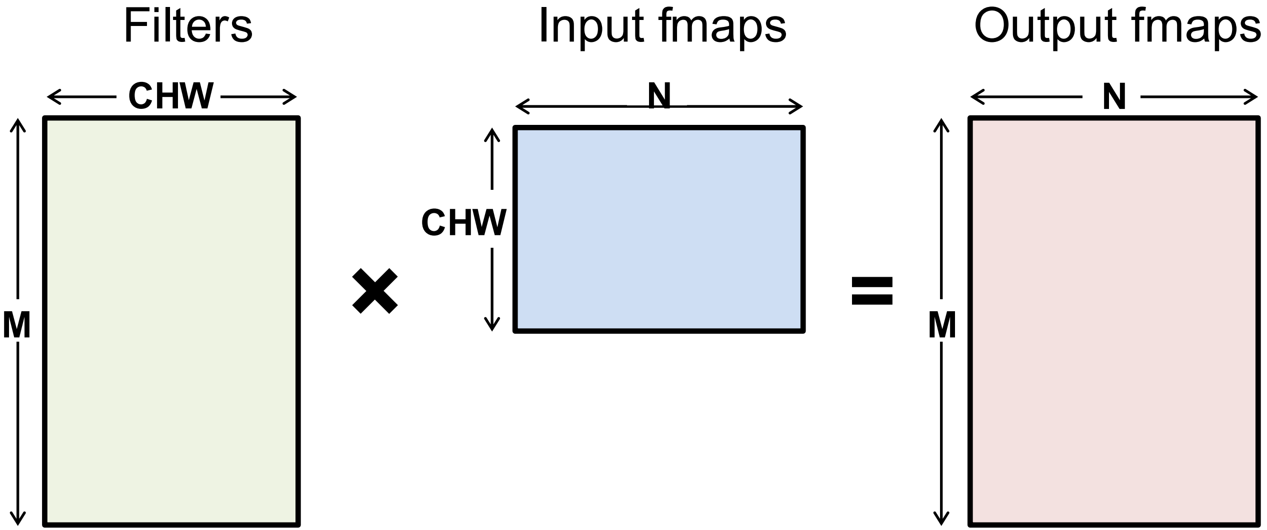


FC Layer – Batch (N)



FC Compute → Matrix-Matrix Multiply

$$O_{\boldsymbol{n},m} = I_{\boldsymbol{n},chw} \times F_{m,chw}$$



After flattening, having a batch size of N turns the **matrix-vector** multiply into a **matrix-matrix** multiply

FC Compute → Matrix-Matrix Multiply

$$O_{n,m} = I_{n,chw} \times F_{m,chw}$$

reduction on rank **chw**

Typical matrix multiplication notation

$$C_{m,n} = A_{m,k} \times B_{k,n}$$

reduction on rank **k**

Note: for Einsum, the order of ranks does not matter