6.5930/1

Hardware Architectures for Deep Learning

Overview of Deep Neural Network Components

February 7, 2024

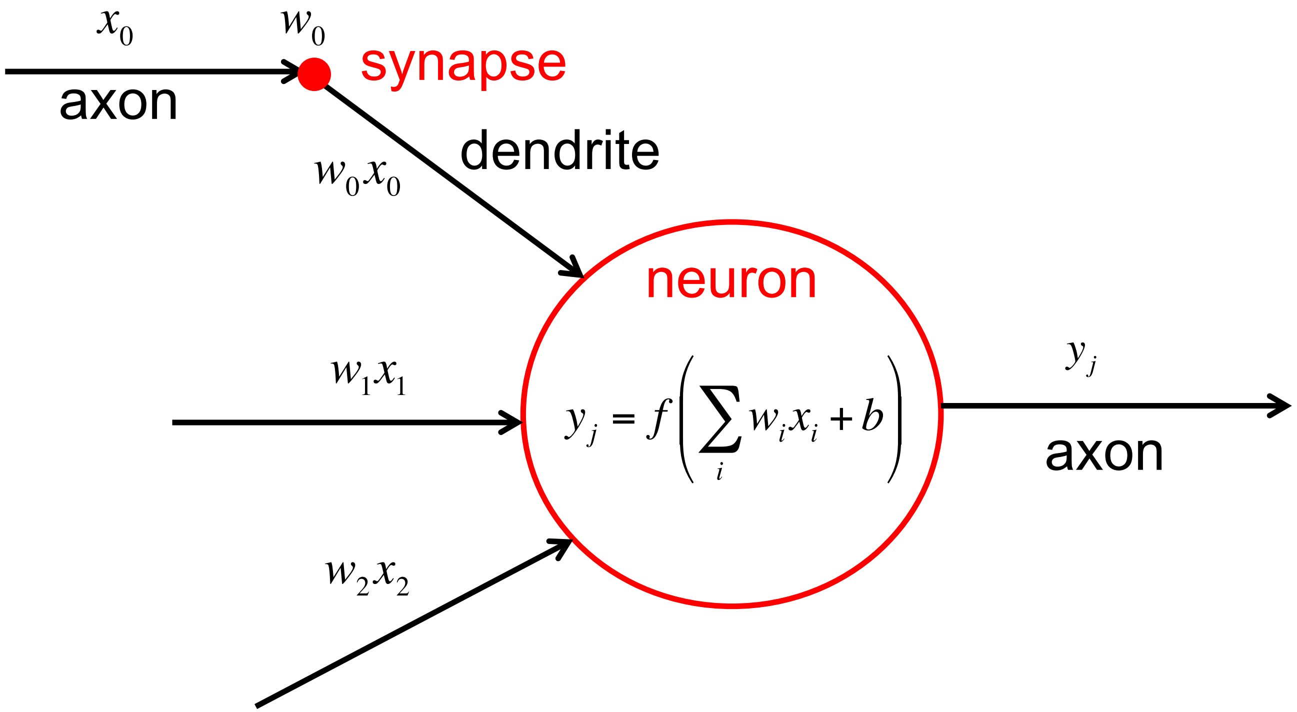
Joel Emer and Vivienne Sze

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

Goals of Today’s Lecture

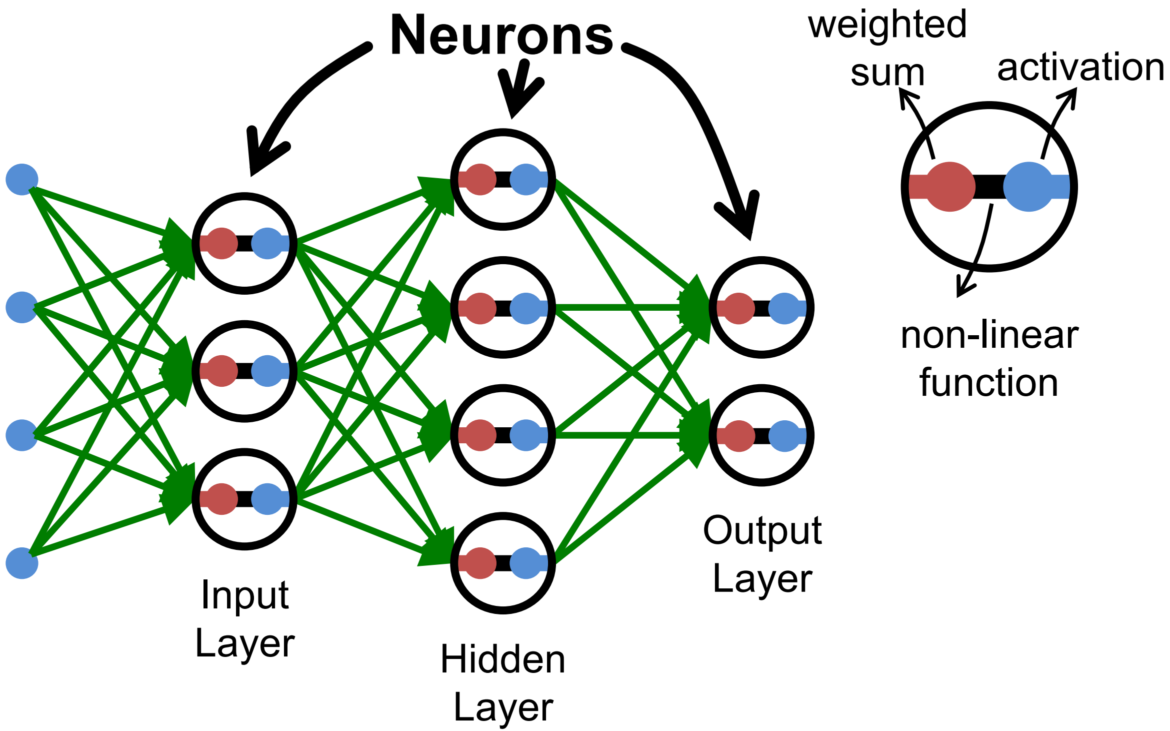
- Overview of the terminology use for Neural Networks
  - Research spans many fields
    - Many terms for the same thing
    - Same term for many different things
  - Define the terminology that we plan to use in this course
- Key building blocks in a Deep Neural Network
- Chapter 1 & 2 in book: https://doi.org/10.1007/978-3-031-01766-7
- For a more in-depth treatment, please see
  - MIT’s Machine Learning Courses (6.3900$_{[6.036]}$/ 6.7900$_{[6.867]}$)
  - MIT’s Computer Vision Course (6.8301$_{[6.819]}$/6.8300$_{[6.869]}$)
  - Class notes from Stanford’s CNN Course (cs231n)
  - www.deeplearningbook.org
  - https://d2l.ai/

Neural Networks: Weighted Sum

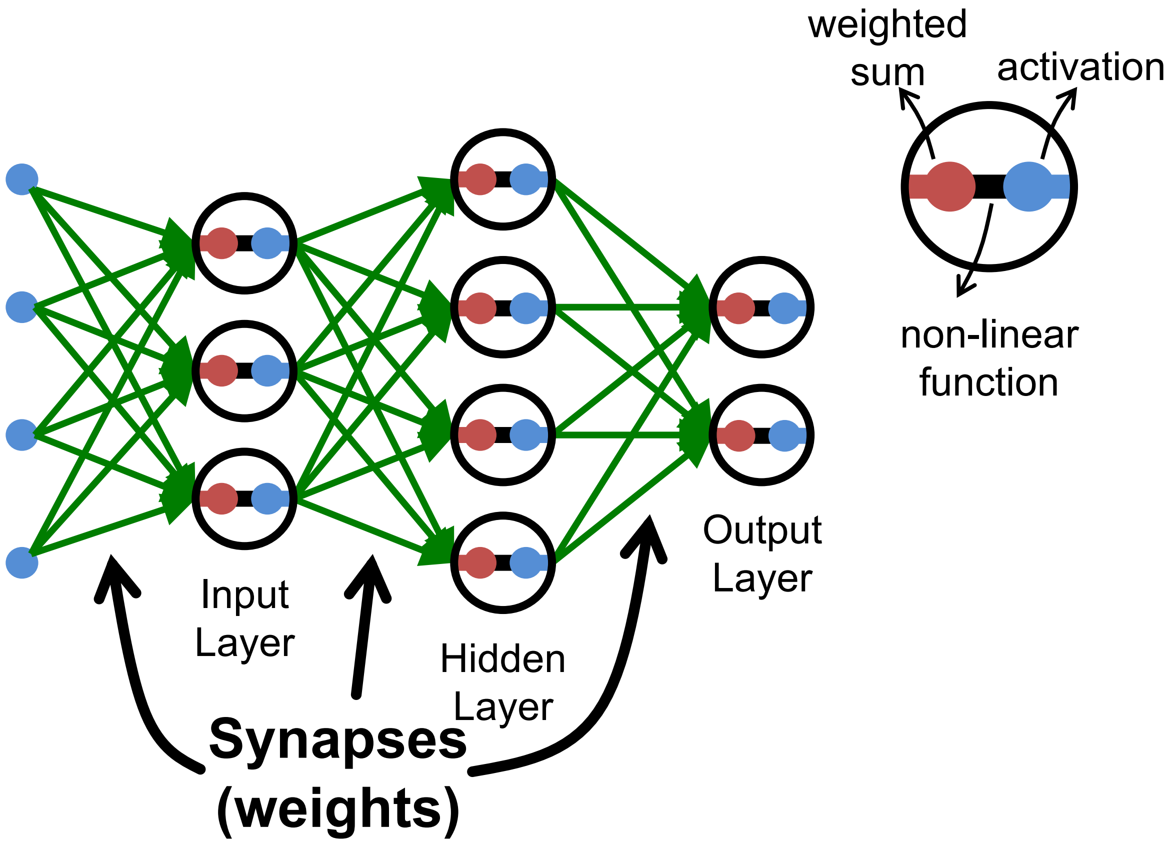


$$y_j = f\left(\sum_i w_i x_i + b\right)$$

DNN Terminology 101

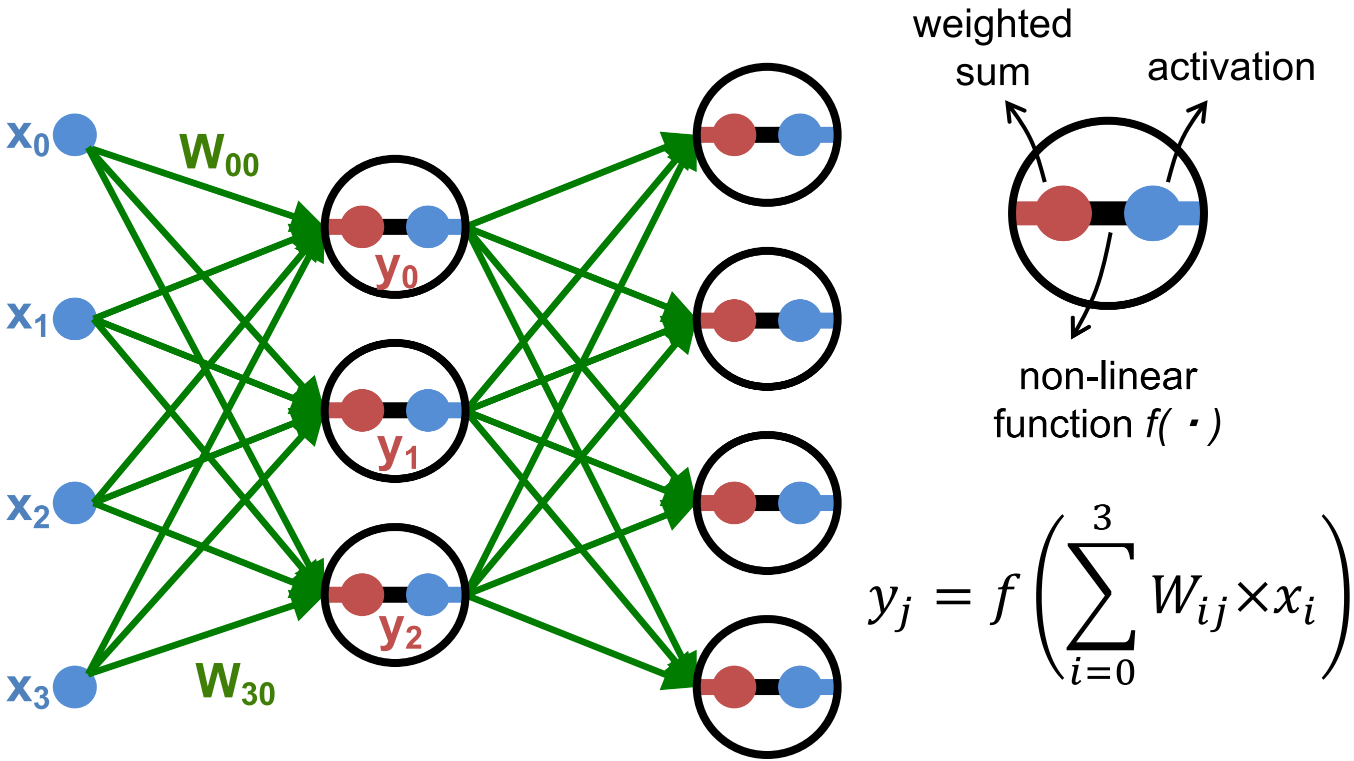


DNN Terminology 101



DNN Terminology 101

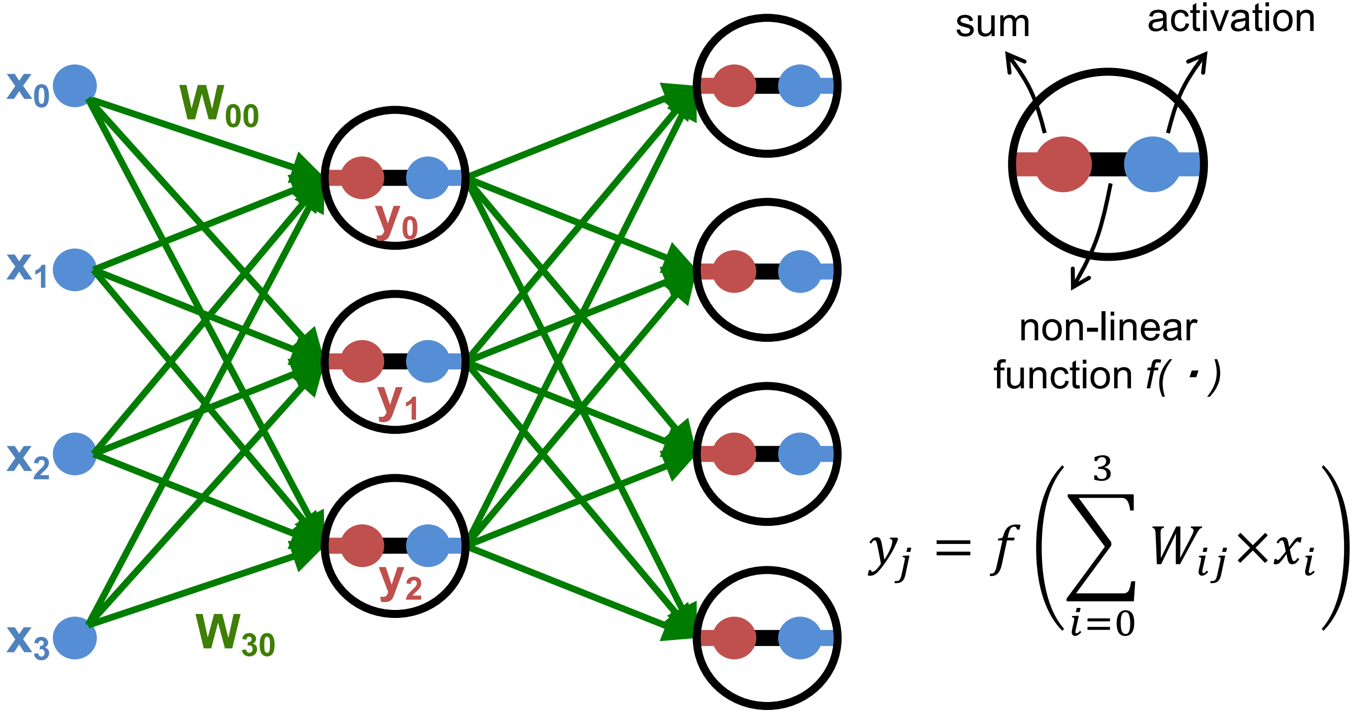
Each **synapse** has a **weight** for neuron **activation**



$$y_j = f\left(\sum_{i=0}^{3} W_{ij} \times x_i\right)$$

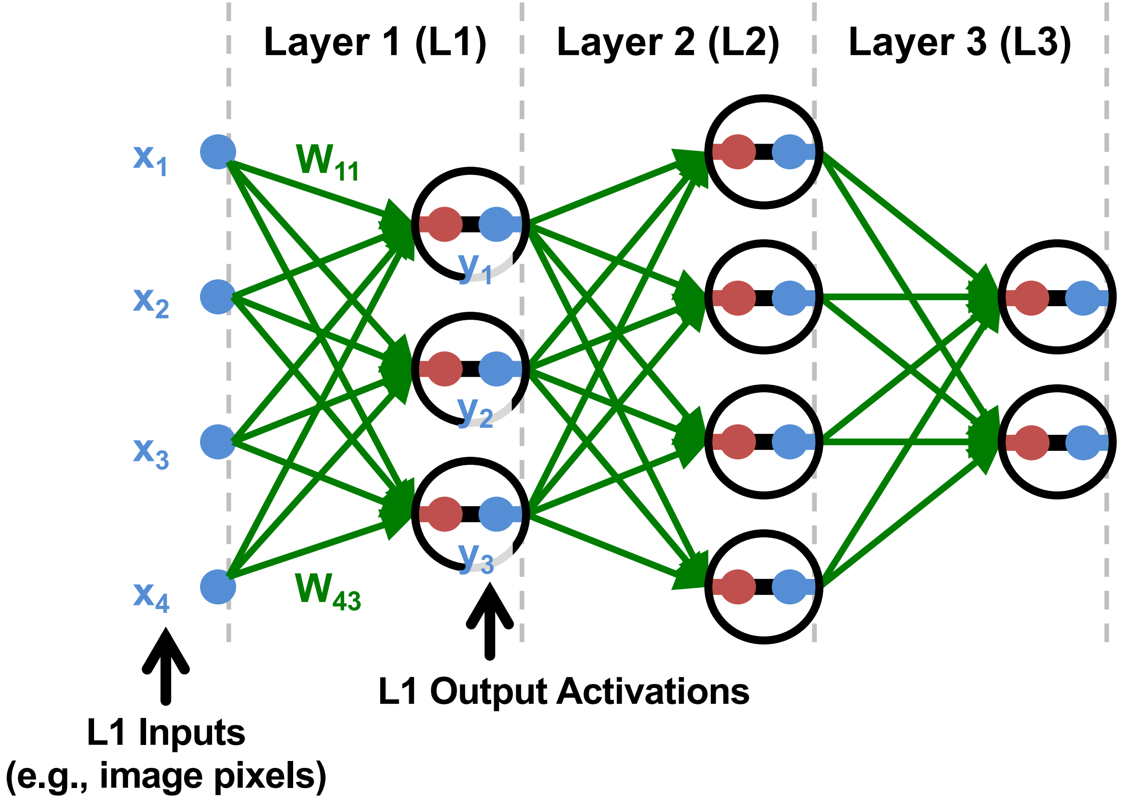
DNN Terminology 101

Weight Sharing: multiple synapses use the **same weight value**

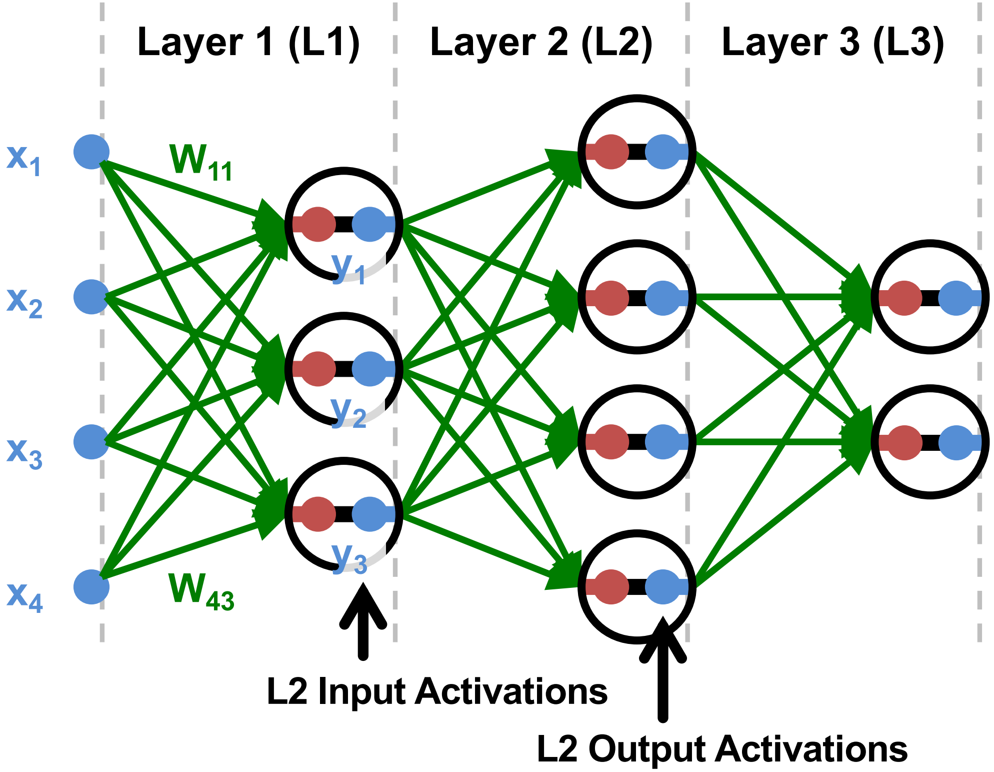


$$y_j = f\left(\sum_{i=0}^{3} W_{ij} \times x_i\right)$$

DNN Terminology 101

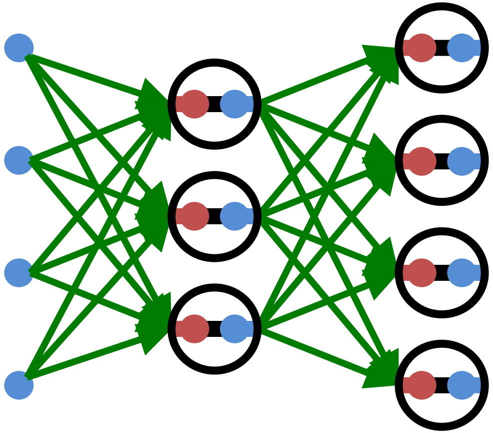


DNN Terminology 101

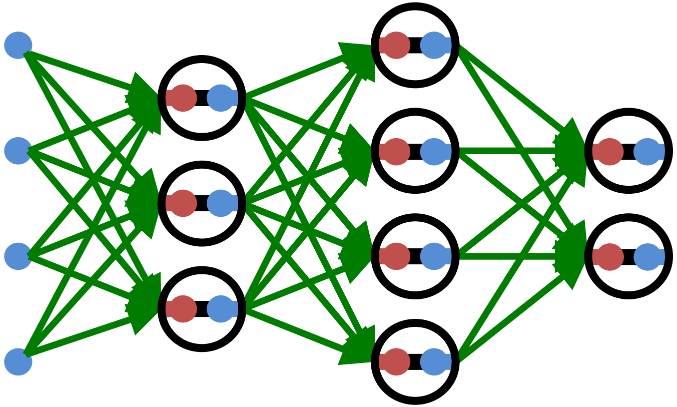


DNN Terminology 101

A **layer** can refer to a set activations or a set of weights.
In this class, we use **layer to refer to a set of weights**.



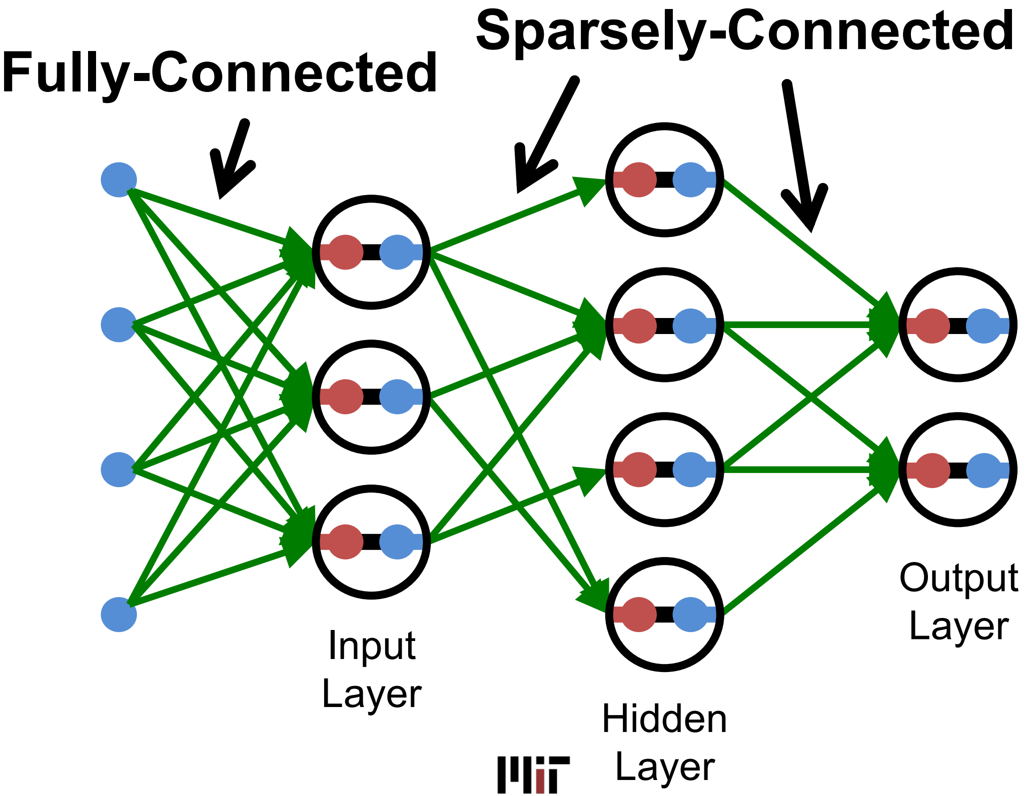
2-layer Neural Net
or
1-hidden-layer Neural Net



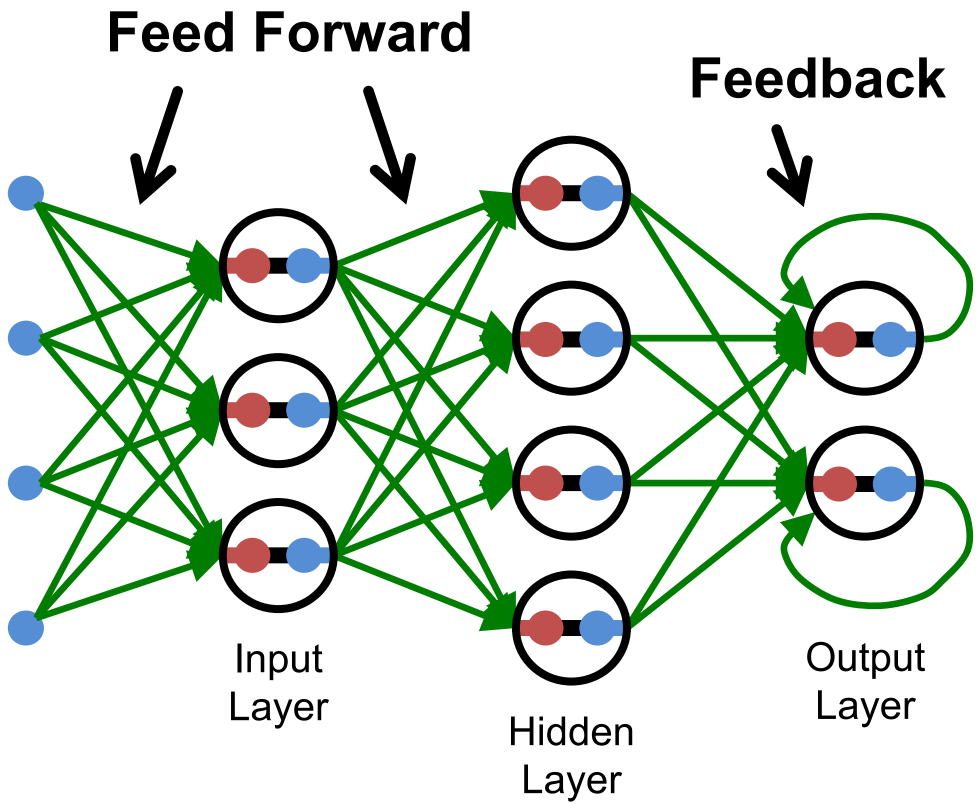
3-layer Neural Net
or
2-hidden-layer Neural Net

DNN Terminology 101

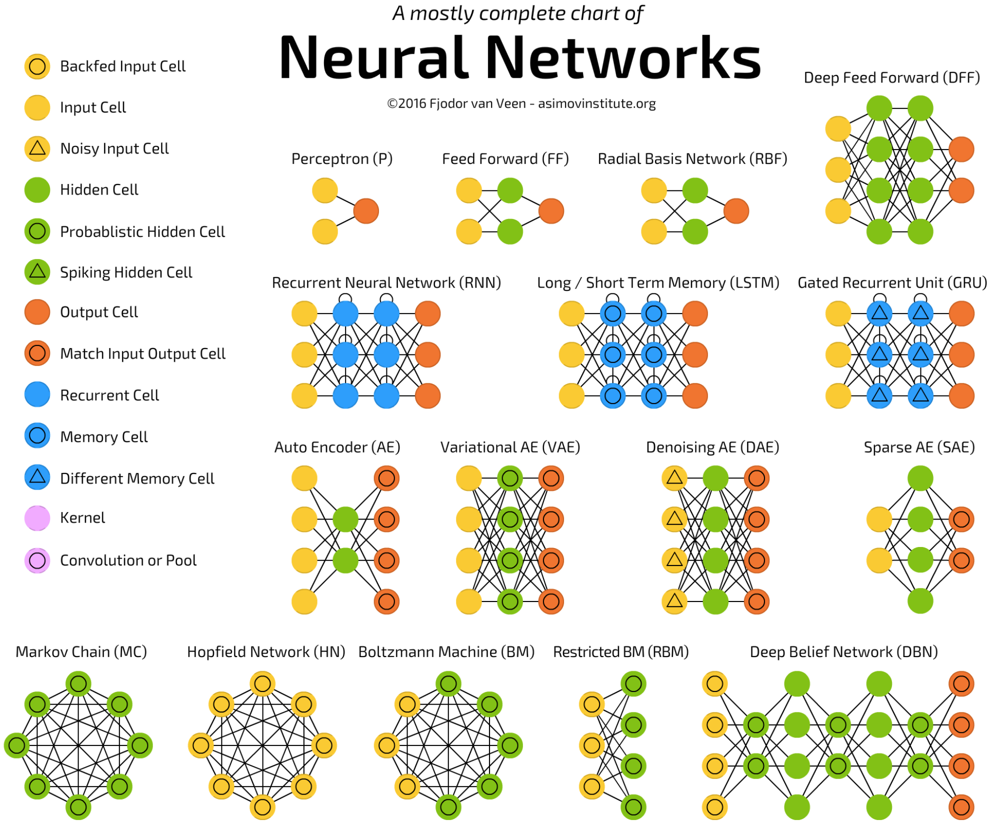
Fully-Connected: all i/p neurons connected to all o/p neurons



DNN Terminology 101



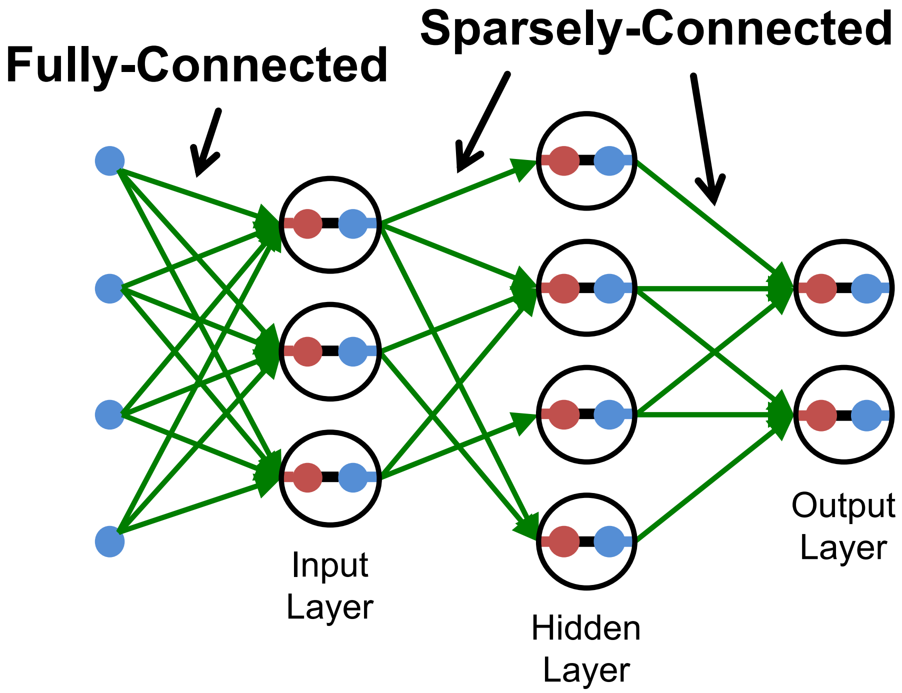
So Many Neural Networks!



http://www.asimovinstitute.org/neural-network-zoo/

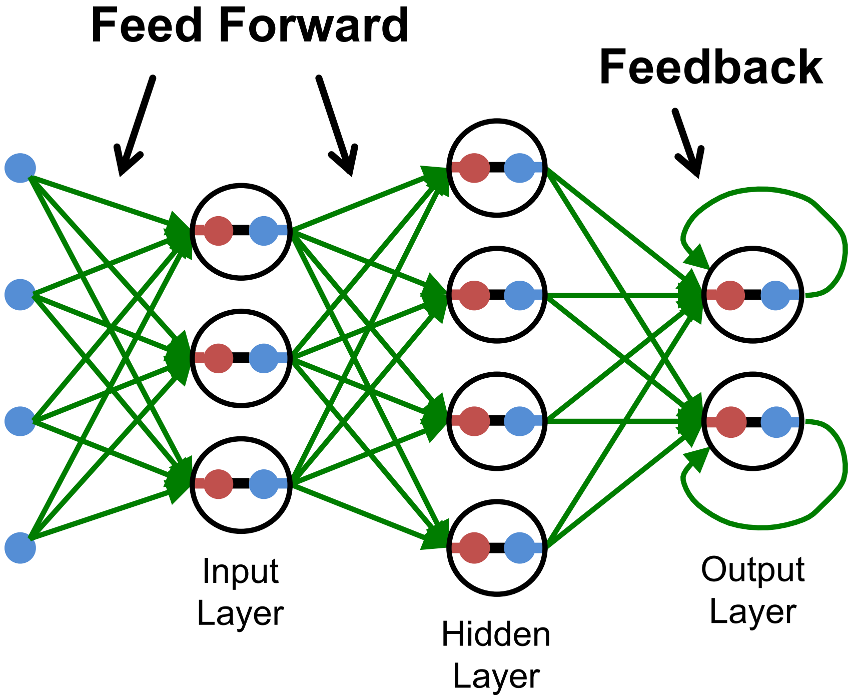
Popular Types of DNNs

- **Fully-Connected NN**
  - feed forward, a.k.a. multilayer perceptron (MLP)
- **Convolutional NN (CNN)**
  - feed forward, sparsely-connected w/ weight sharing



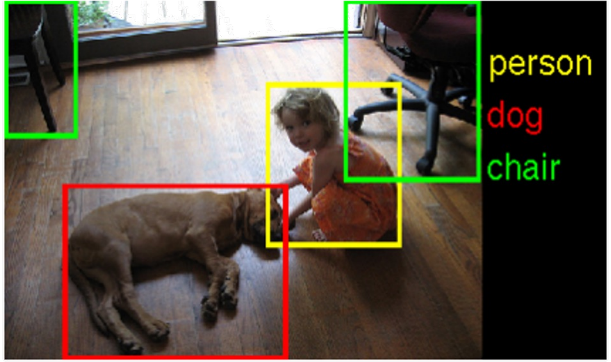
Popular Types of DNNs

- **Recurrent NN (RNN)**
  - feedback
- **Long Short-Term Memory (LSTM)**
  - feedback + storage
- **Encoders**
  - output smaller than input
- **Decoders**
  - output larger than input
- **Transformers**
  - “attention” mechanism

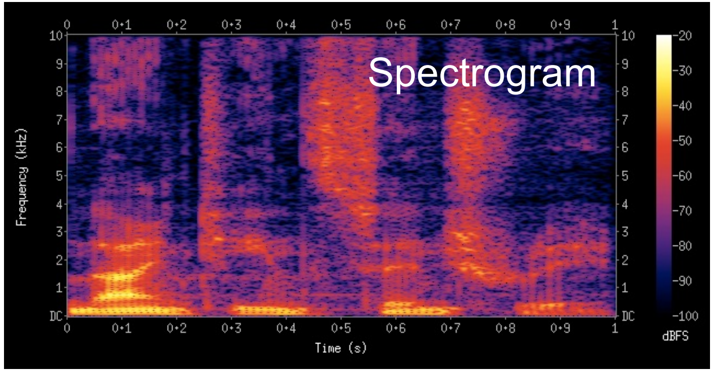


Applications of CNN

Computer Vision



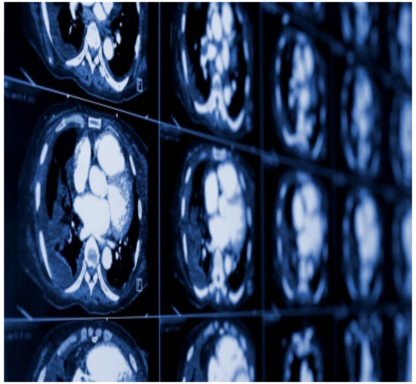
Speech Recognition



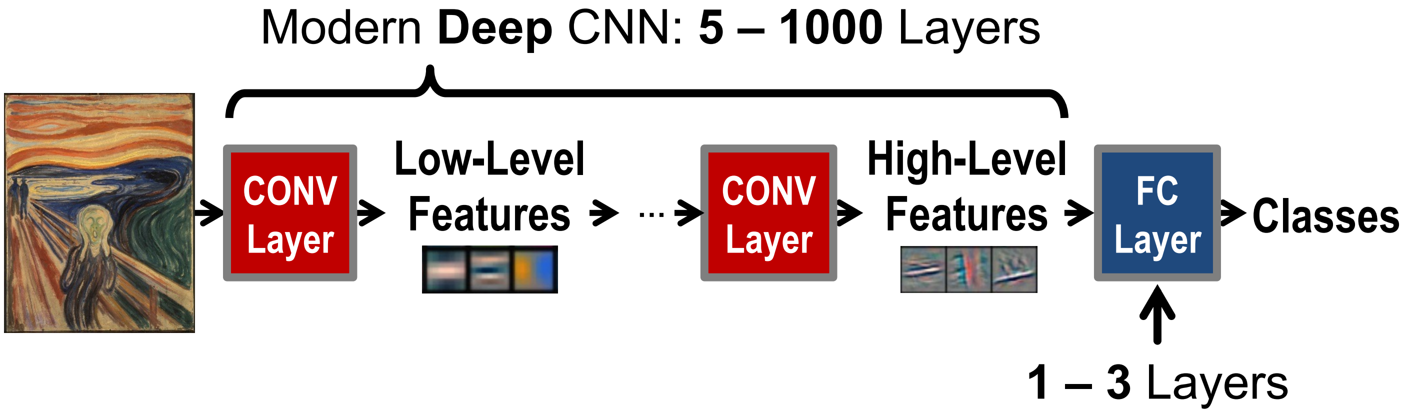
Game Play

nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
At last – a computer program that can beat a champion Go player PAGE 484
ALL SYSTEMS GO

Medical



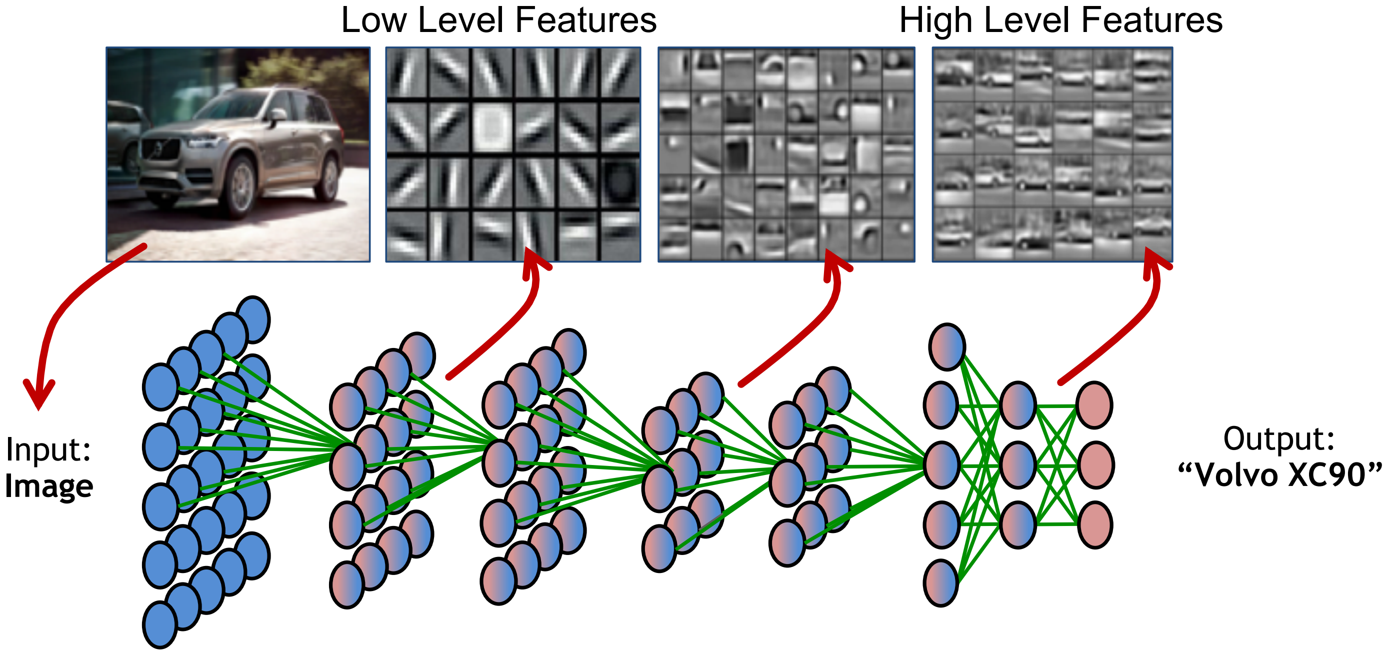
Convolutional Neural Networks



Modern **Deep** CNN: **5 – 1000** Layers

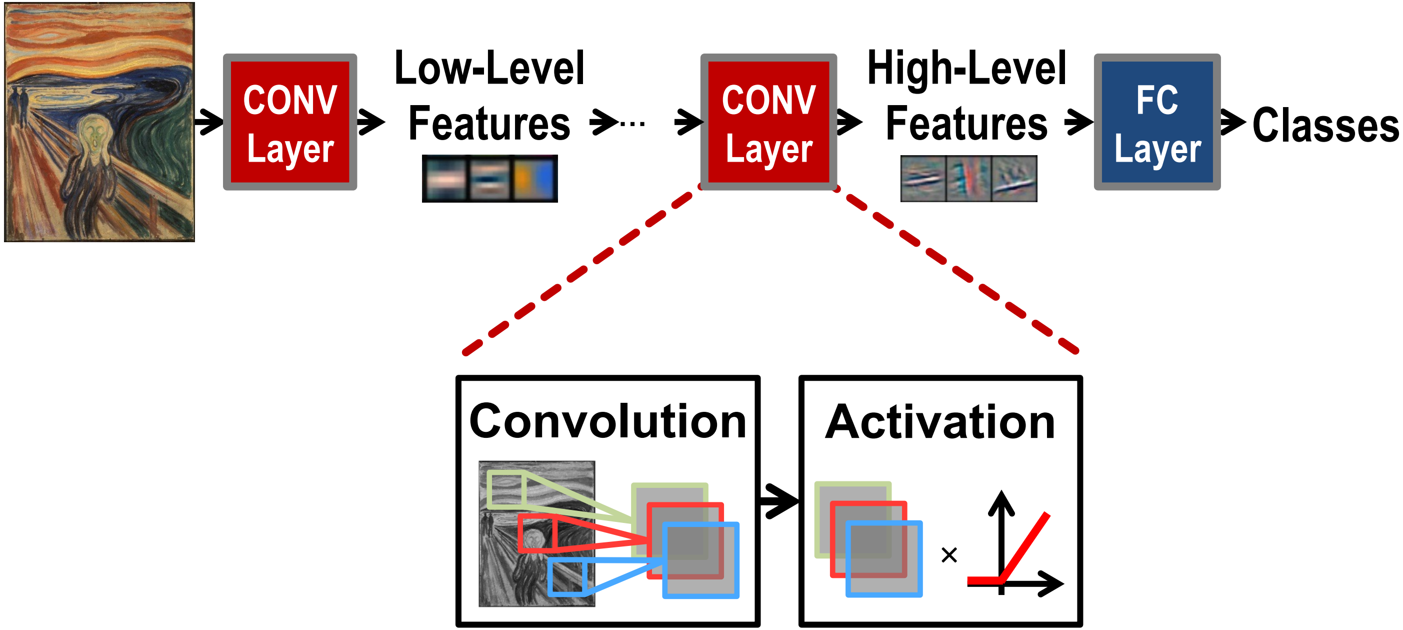
1 – 3 Layers

Depth of Network

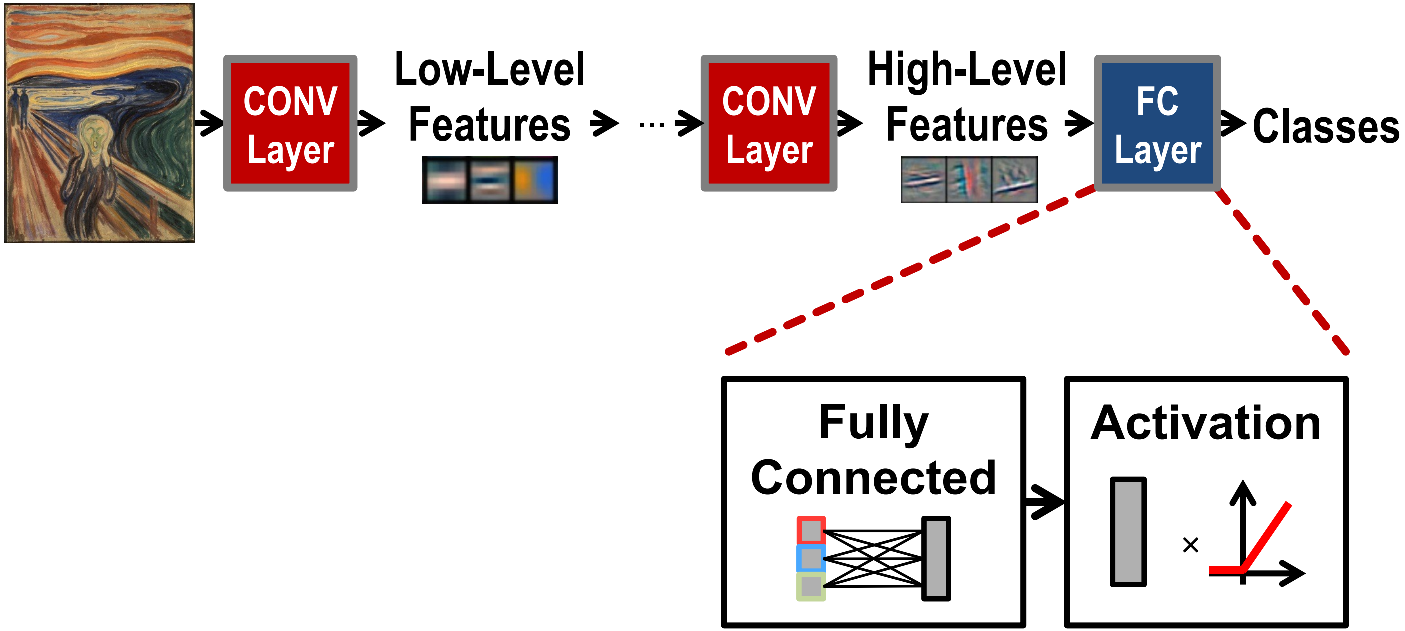


Modified Image Source: [**Lee**, *CACM* 2011]

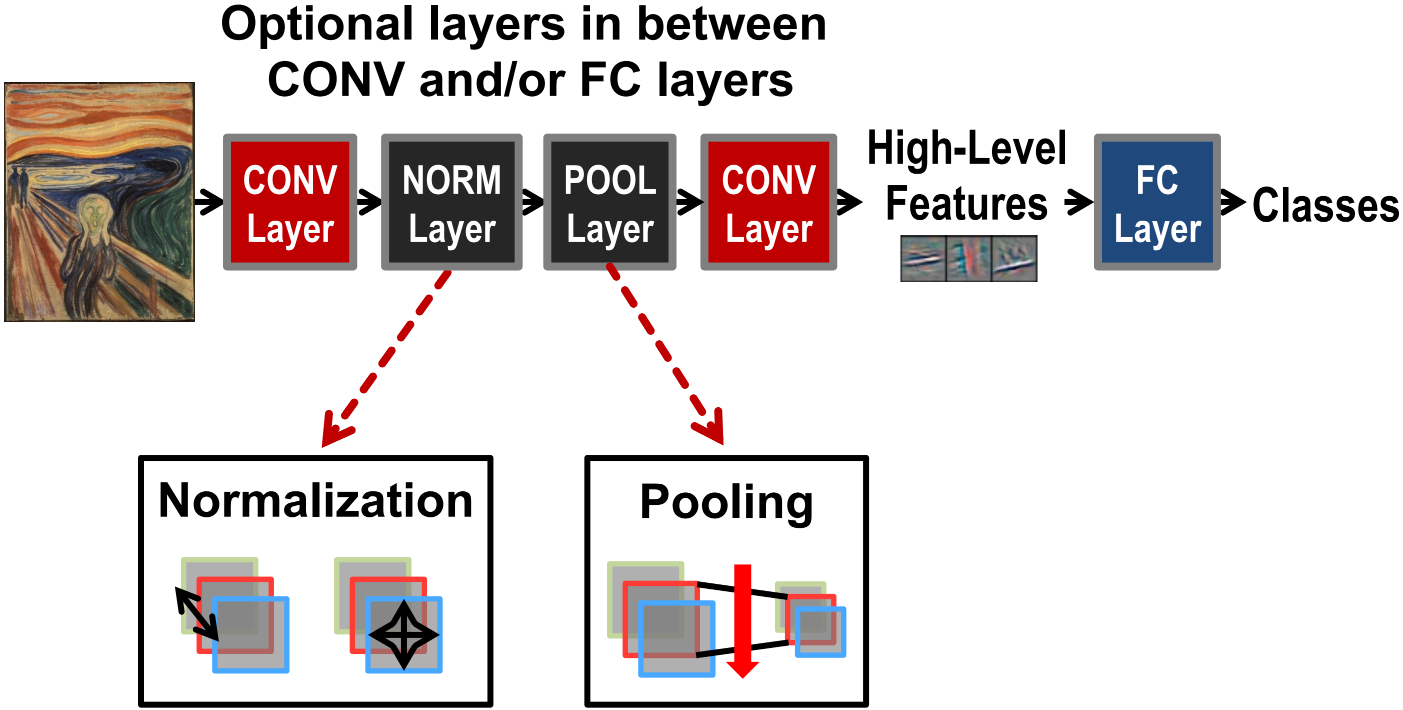
Convolutional Neural Networks



Convolutional Neural Networks

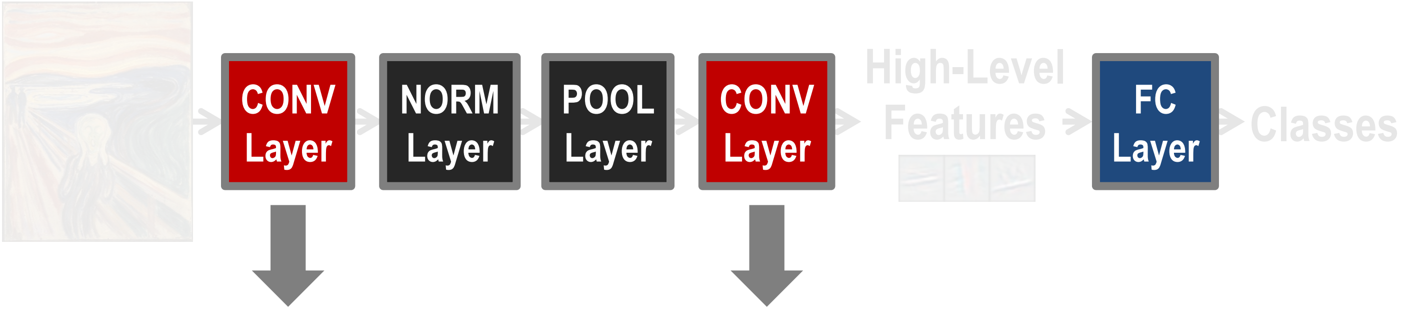


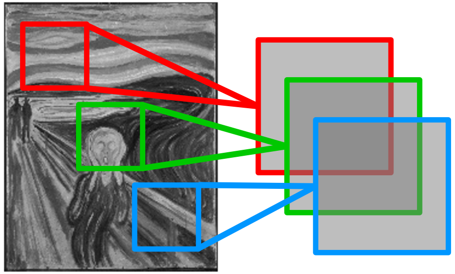
Convolutional Neural Networks



Optional layers in between CONV and/or FC layers

Convolutional Neural Networks



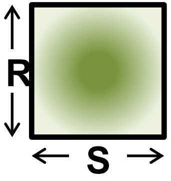


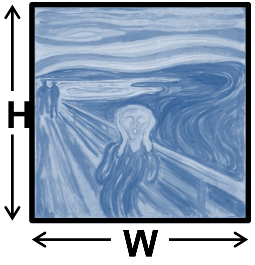
Convolutions account for more than 90% of overall computation, dominating **runtime** and **energy consumption**

Convolution (CONV) Layer

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

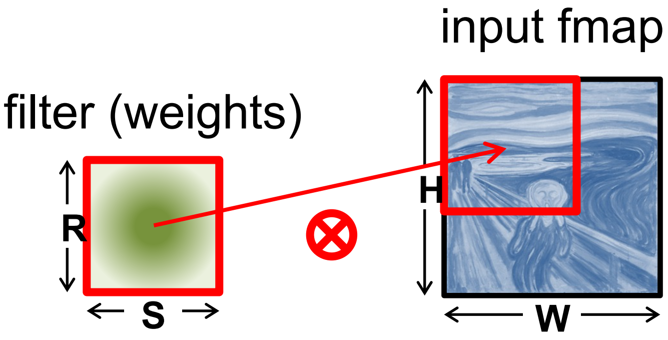
filter* (weights)





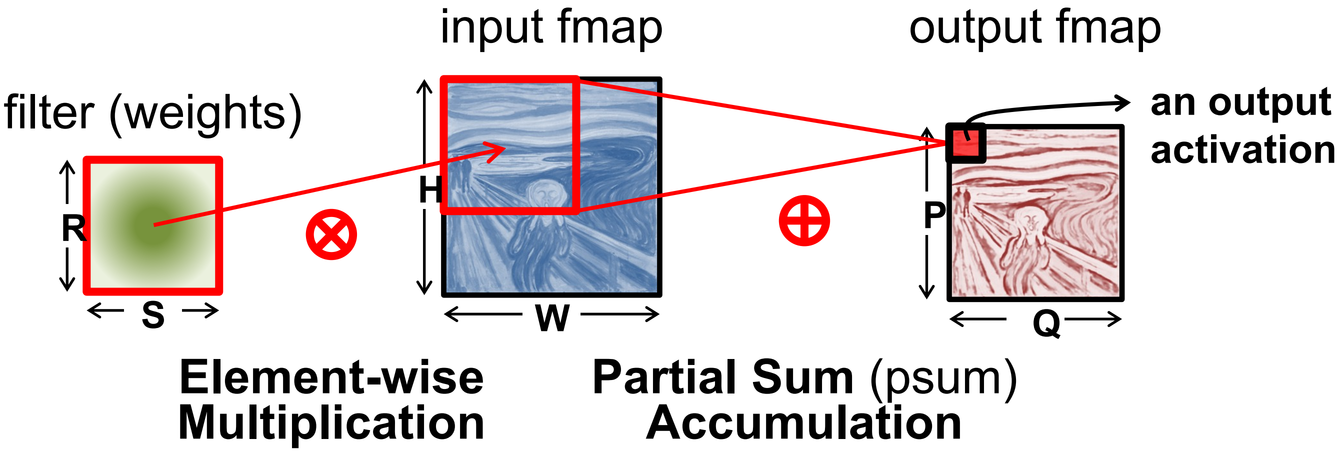
* also referred to as **kernel**

Convolution (CONV) Layer



Element-wise Multiplication

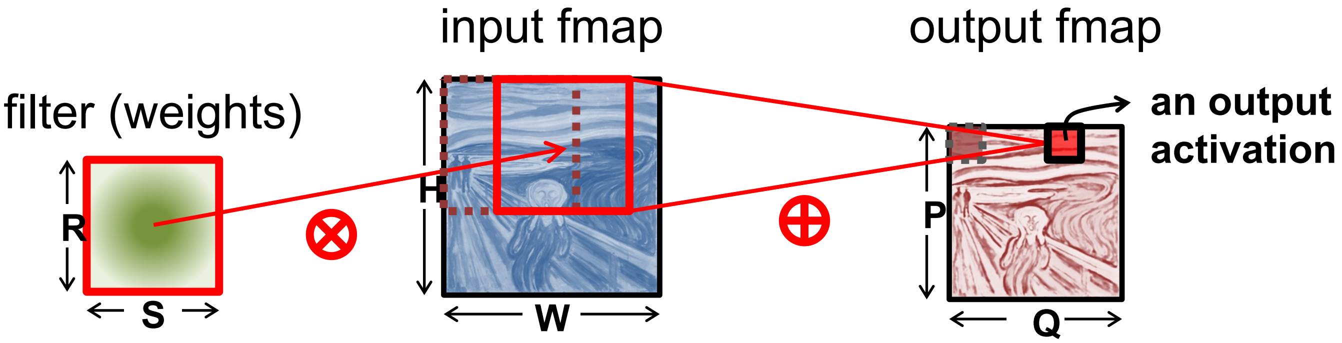
Convolution (CONV) Layer



Element-wise Multiplication

Partial Sum (psum) **Accumulation**

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)

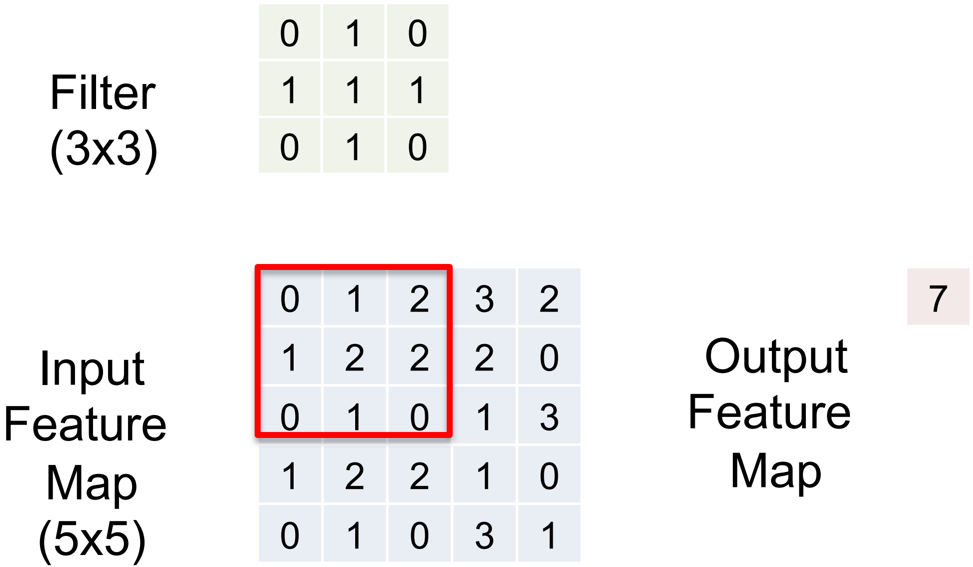
0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

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)

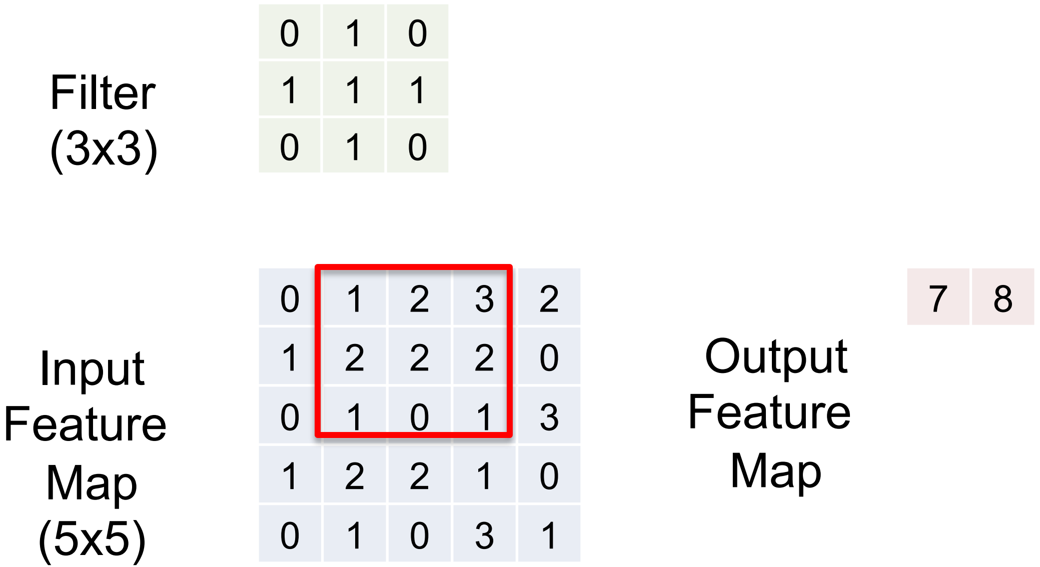
0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

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)

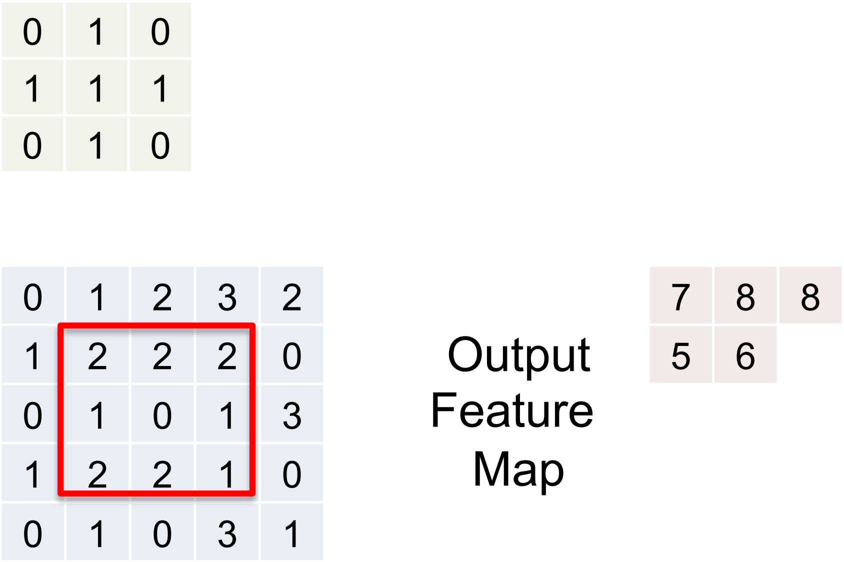
0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

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 Size of Size of
Output Feature Map = (**Input Feature Map** – **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)

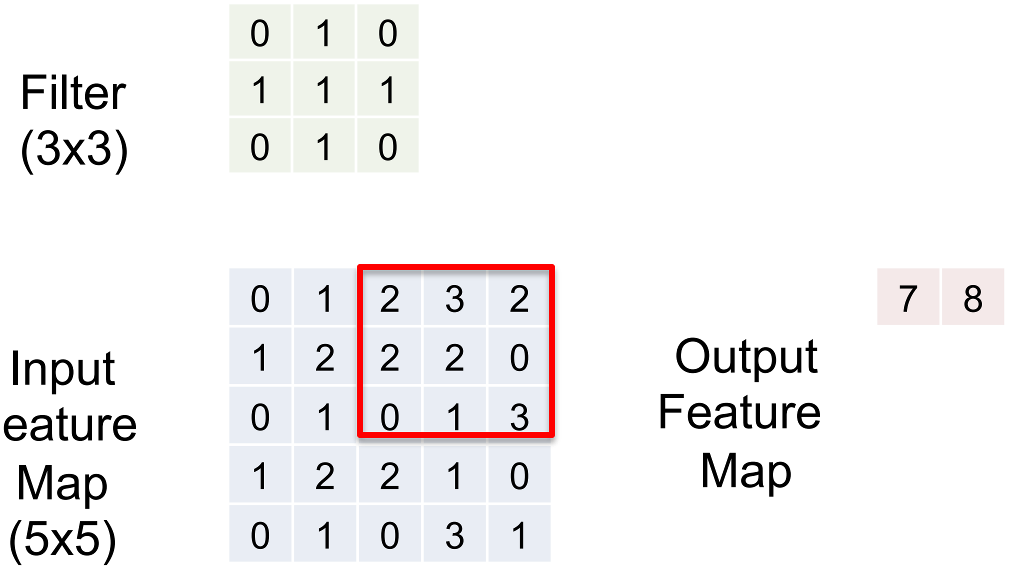
0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

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)

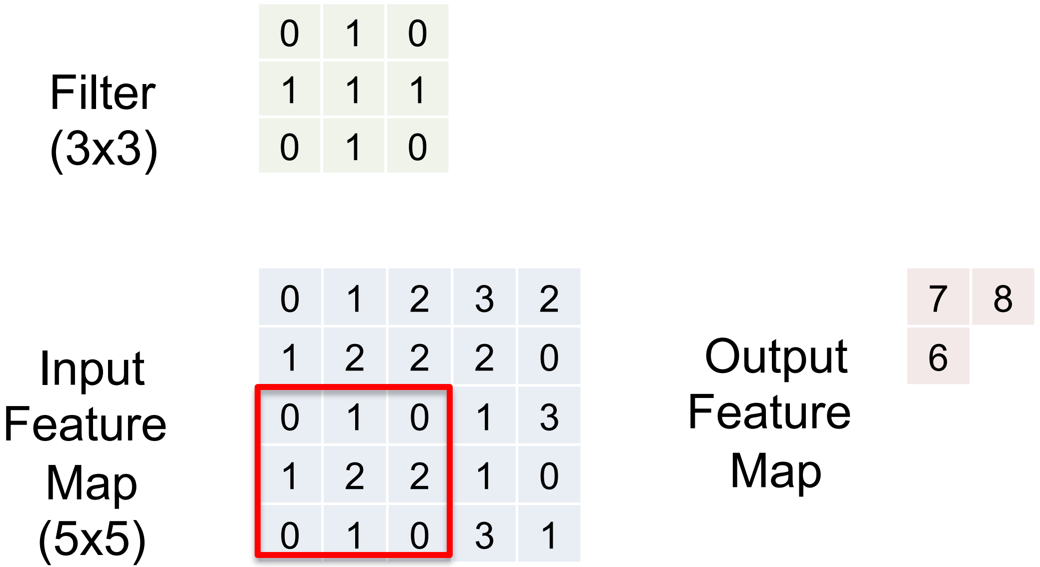
0	1	2	3	2
1	2	2	2	0
0	1	0	1	3
1	2	2	1	0
0	1	0	3	1

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

$$\overset{\text{Size of}}{\textbf{Output Feature Map}} = (\overset{\text{Size of}}{\textbf{Input Feature Map}} – \overset{\text{Size of}}{\textbf{Filter}} + \textbf{Stride}) / \textbf{Stride}$$

# 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

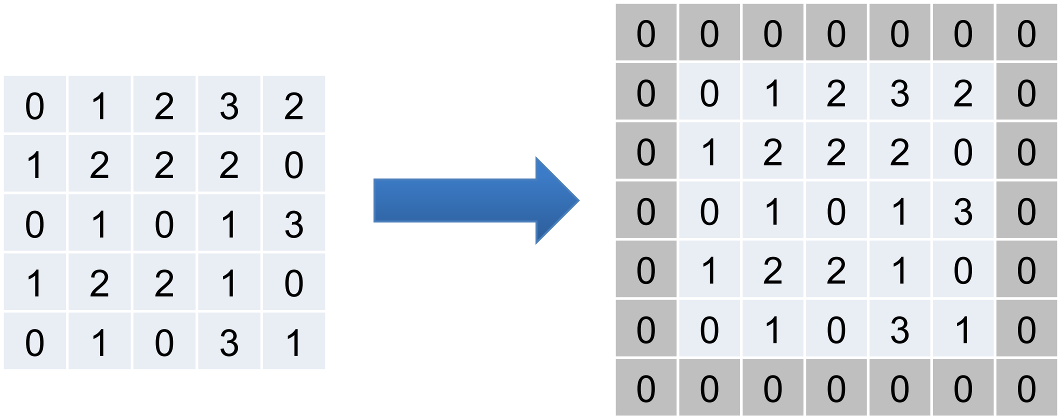
Stride 3

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

Signal Processing Perspective

Cross-Correlation rather than Convolution

Recall from **6.3000**$_{\textbf{[6.003]}}$ and **6.7010**$_{\textbf{[6.344]}}$, the filter needs to be flipped for a convolution.

$$\begin{aligned} y(n_1, n_2) &= x(n_1, n_2) * h(n_1, n_2) \\ &= \sum_{k_1} \sum_{k_2} x(k_1, k_2) \cdot \boxed{h(n_1 - k_1, n_2 - k_2)} \end{aligned}$$

For CNN, the filter is combined with an input window without reversing the filter.
Strictly speaking, this is a **cross-correlation**.

Size of Output after Filtering

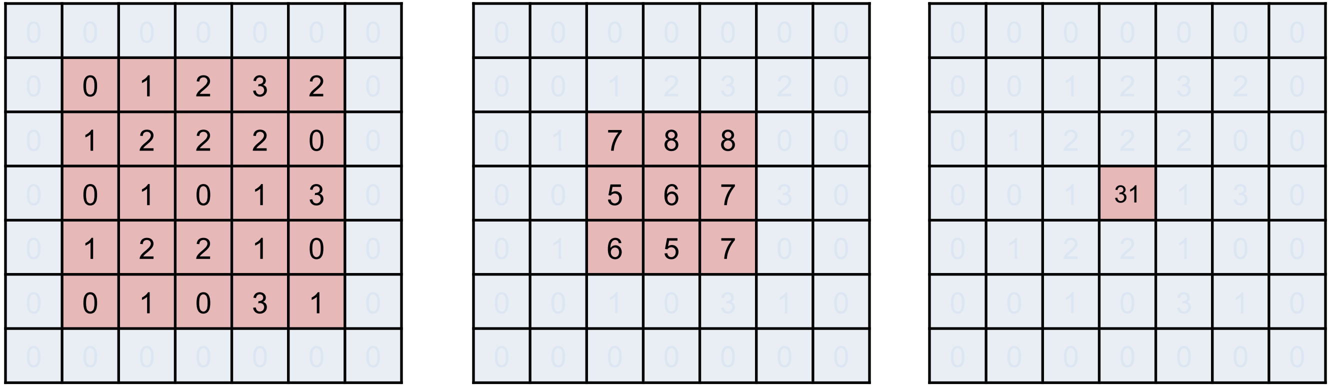
Recall from **6.3000**$_{\textbf{[6.003]}}$ and **6.7010**$_{\textbf{[6.344]}}$, if filter size is M and input is N, output is N+M-1.
No restriction on zero padding.

For CNN, the amount of zero padding can be varied to control the output size.
The output size is typically **equal or smaller** than the input size.

Depth of Network: Convolution

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

Example: 3x3 filter



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

Input to Layer 1

0	0	0	0	0	0	0
0	0	1	2	3	2	0
0	1	7	8	8	0	0
0	0	5	6	7	3	0
0	1	6	5	7	0	0
0	0	1	0	3	1	0
0	0	0	0	0	0	0

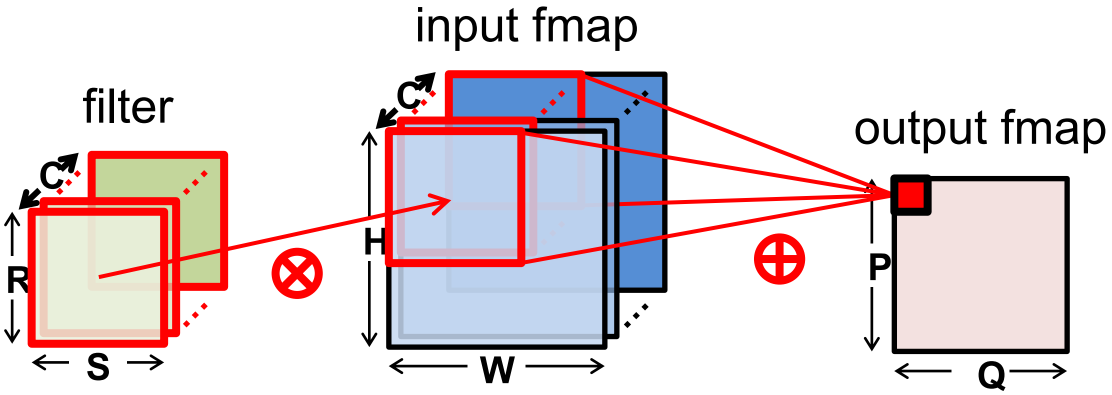
Layer 2

0	0	0	0	0	0	0
0	0	1	2	3	2	0
0	1	2	2	2	0	0
0	0	1	31	1	3	0
0	1	2	2	1	0	0
0	0	1	0	3	1	0
0	0	0	0	0	0	0

Layer 3

Feature maps of deep layers typically give higher level features

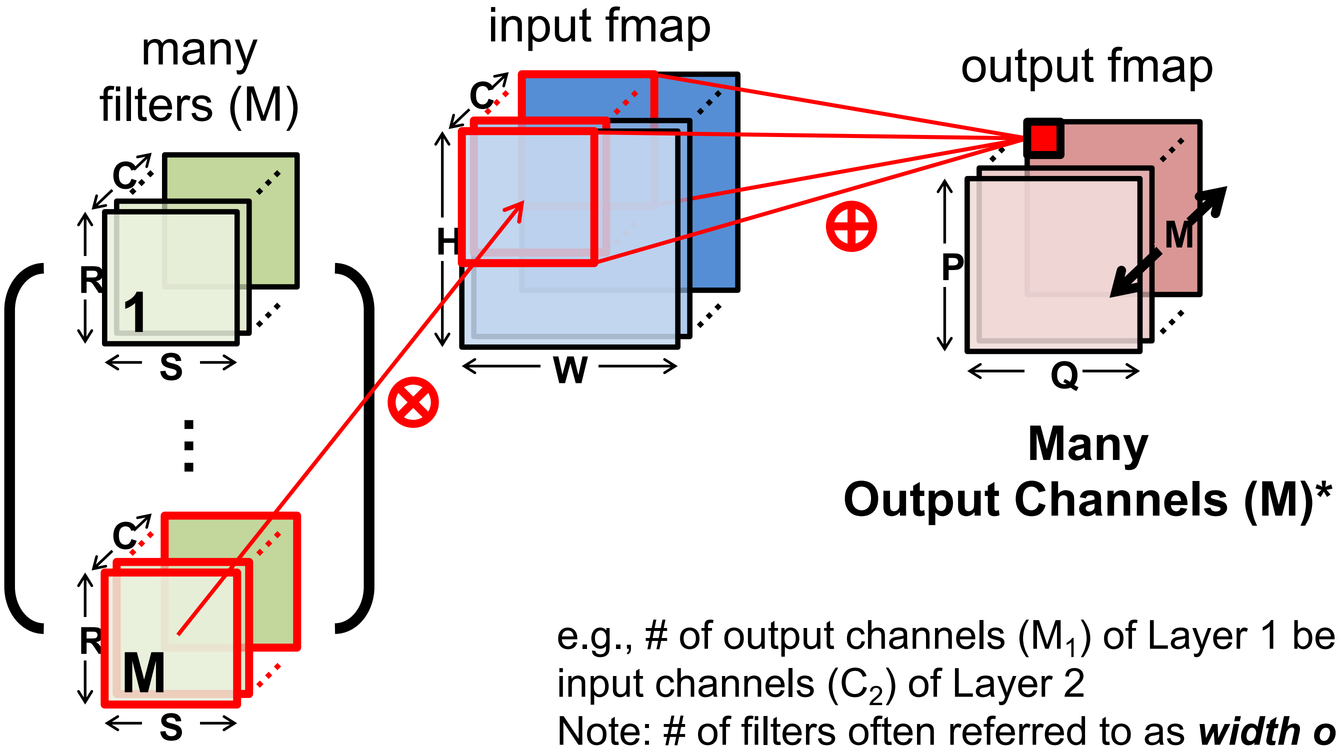
Convolution (CONV) Layer



Many Input Channels (C)

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

Convolution (CONV) Layer

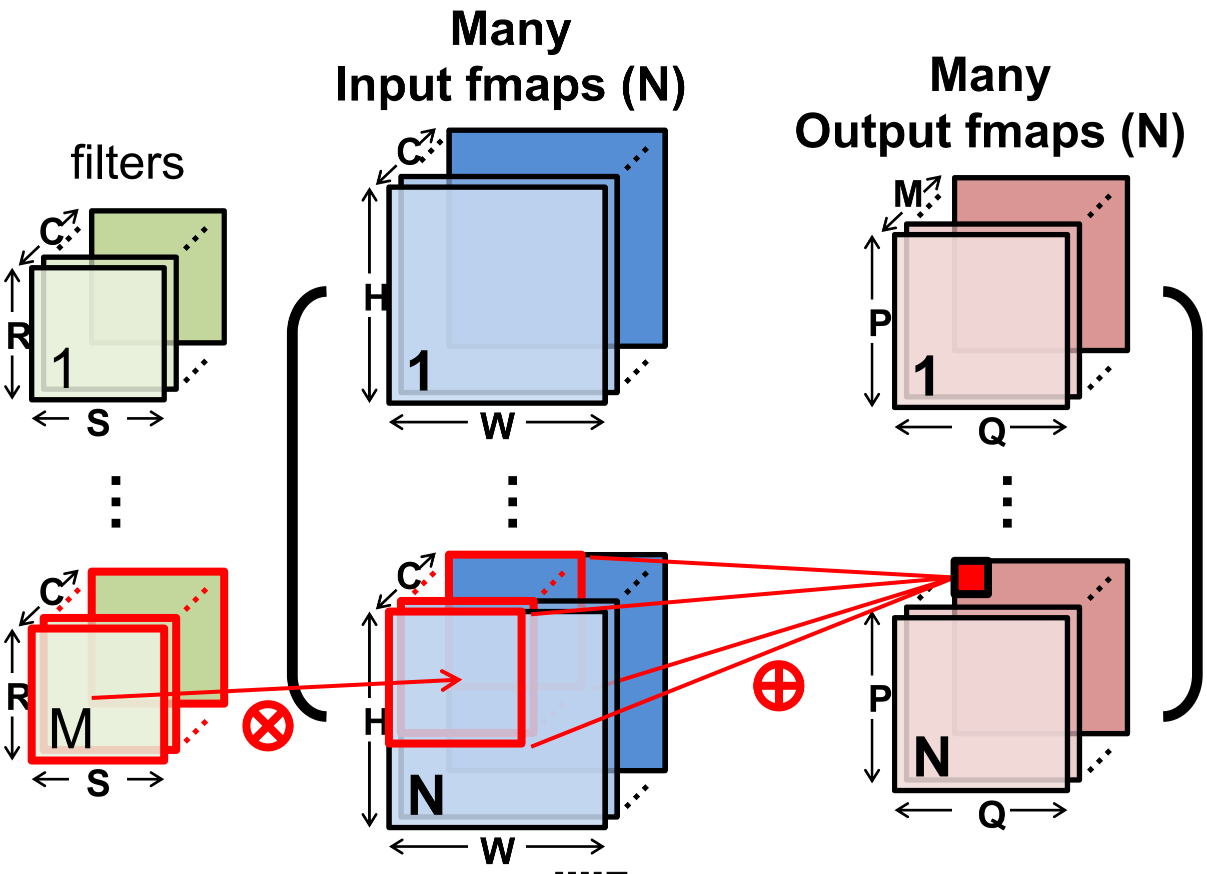


Many
Output Channels (M)*

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



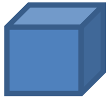
Batch Size (N)

CNN Decoder Ring

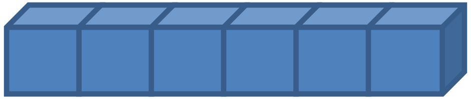
- N – Number of input fmaps/output fmaps (batch size)
- C – Number of channels in input fmaps (activations) & filters (weights)
- H – Height of input fmap (activations)
- W – Width of input fmap (activations)
- R – Height of filter (weights)
- S – Width of filter (weights)
- M – Number of channels in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

Tensors

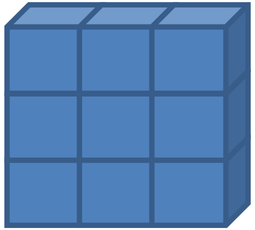
Rank-0: Scalar



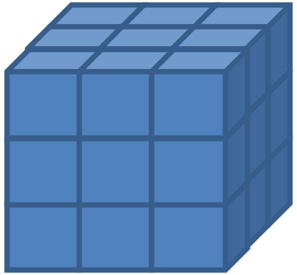
Rank-1: Vector



Rank-2: Matrix

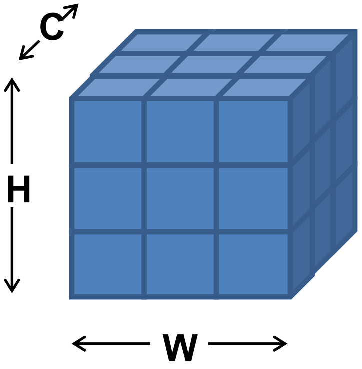


Rank-3: Cube



Input Feature Map (fmap)

Input fmap (activations)



I[C][H][W]

CONV Layer Tensor Computation

Output fmap (O)

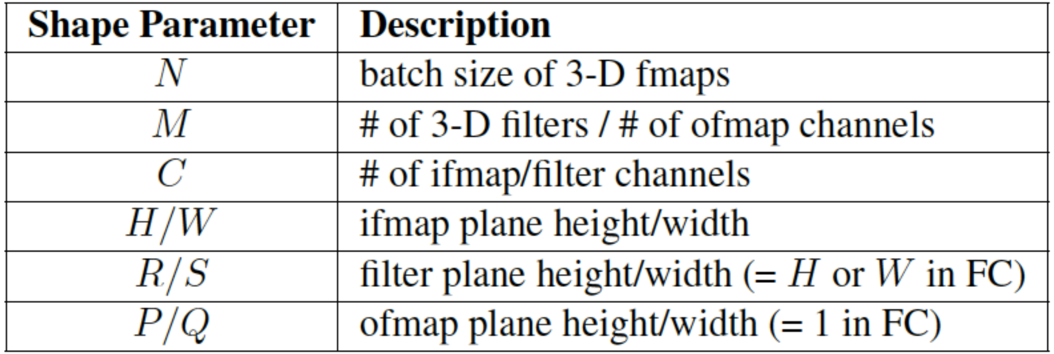
Biases (B)

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

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

Einsum does not enforce any computational order

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

CONV Layer Implementation

Naïve 7-layer for-loop implementation:

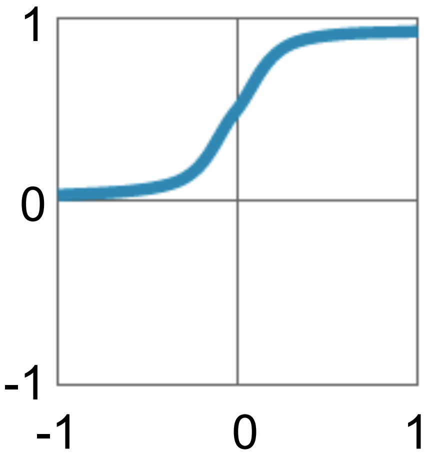
```
for n in [0..N):
    for m in [0..M):
        for q in [0..Q):
            for p in [0..P):
                O[n][m][p][q] = B[m];
                for c in [0..C):
                    for r in [0..R):
                        for s in [0..S):
                            O[n][m][p][q] += I[n][c][Up+r][Uq+s]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

                O[n][m][p][q] = Activation(O[n][m][p][q]);
```

for each output fmap value

convolve a window and apply activation

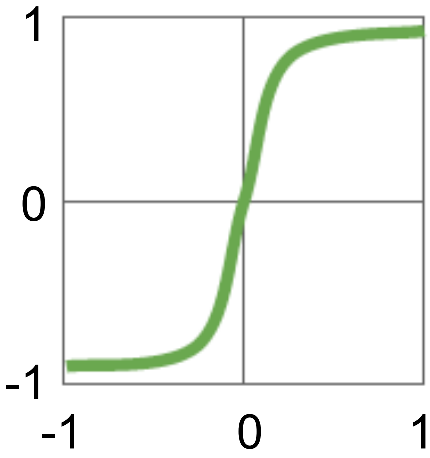
Traditional Activation Functions

Sigmoid



$y=1/(1+e^{-x})$

Hyperbolic Tangent



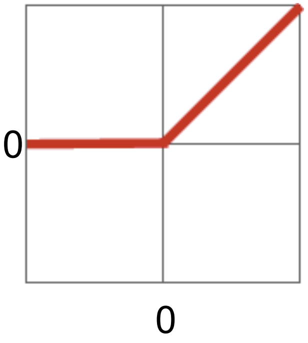
$y=(e^{x}-e^{-x})/(e^{x}+e^{-x})$

Note: Also referred to as the non-linearity

Image Source: Caffe Tutorial

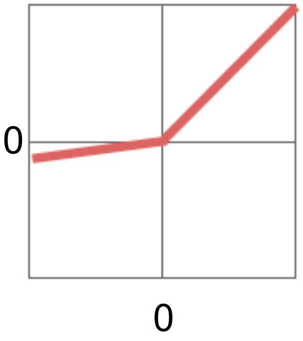
Modern Activation Functions

Rectified Linear Unit (ReLU)



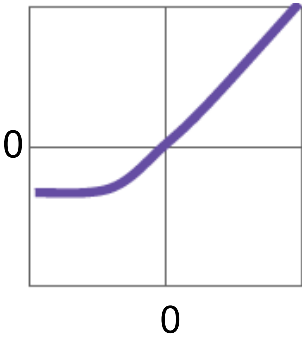
$y=\max(0,x)$

Leaky ReLU



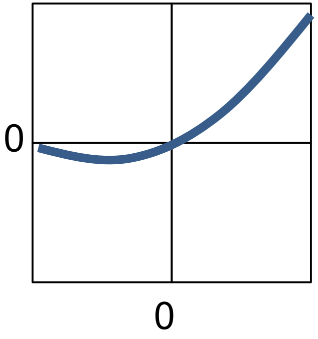
$y=\max(\alpha x,x)$

Exponential LU (ELU)



$$y=\begin{cases} x, & x\geq 0 \\ \alpha(e^{x}-1), & x<0 \end{cases}$$

Swish



$y=x*\text{sigmoid}(\alpha x)$

Variants: e.g., **ReLU6** (clipped max value to 6) and **h-swish** (replace sigmoid with piecewise linear function)

Image Source: Caffe Tutorial

Comparison of Activations

Sigmoid/Hyperbolic Tangent

- Difficult to train due to vanishing gradient problem
  - ***Small gradient*** at high and low activation values

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha \frac{\partial L}{\partial w_{ij}}$$

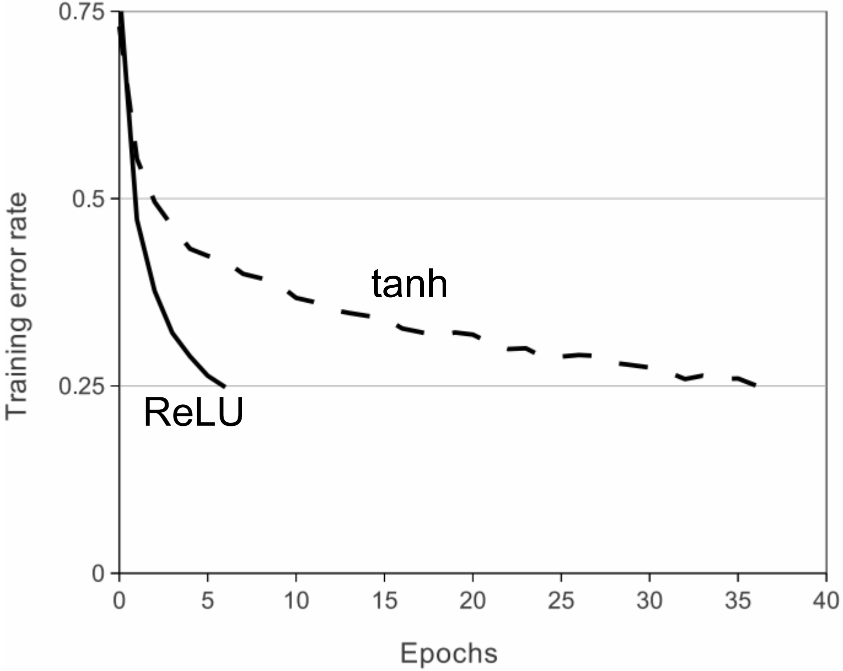
- Not easy to implement
  - Typically use a look up table (LUT)

ReLU

- Gradient does not vanish at high activation values → faster training
- Easy to implement
- Leads to sparsity in activations, which has additional implementation benefits

Training Speed: tanh vs. ReLU

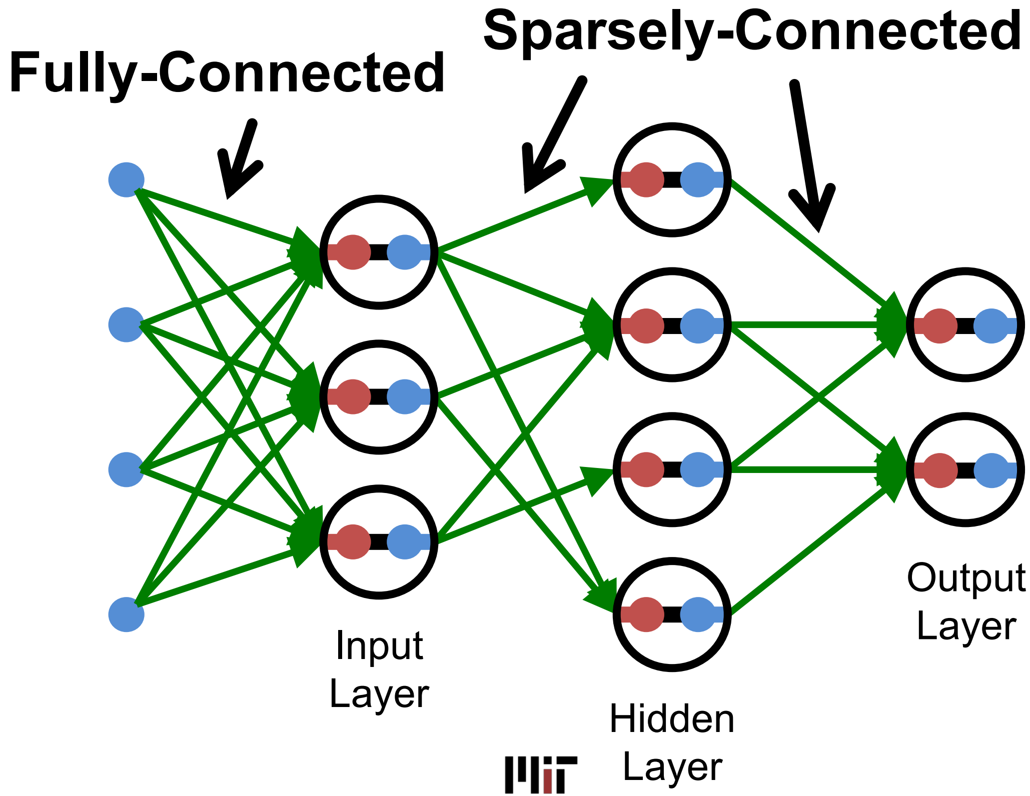
ReLU reaches a 25% training error rate on CIFAR-10 six times faster than tanh



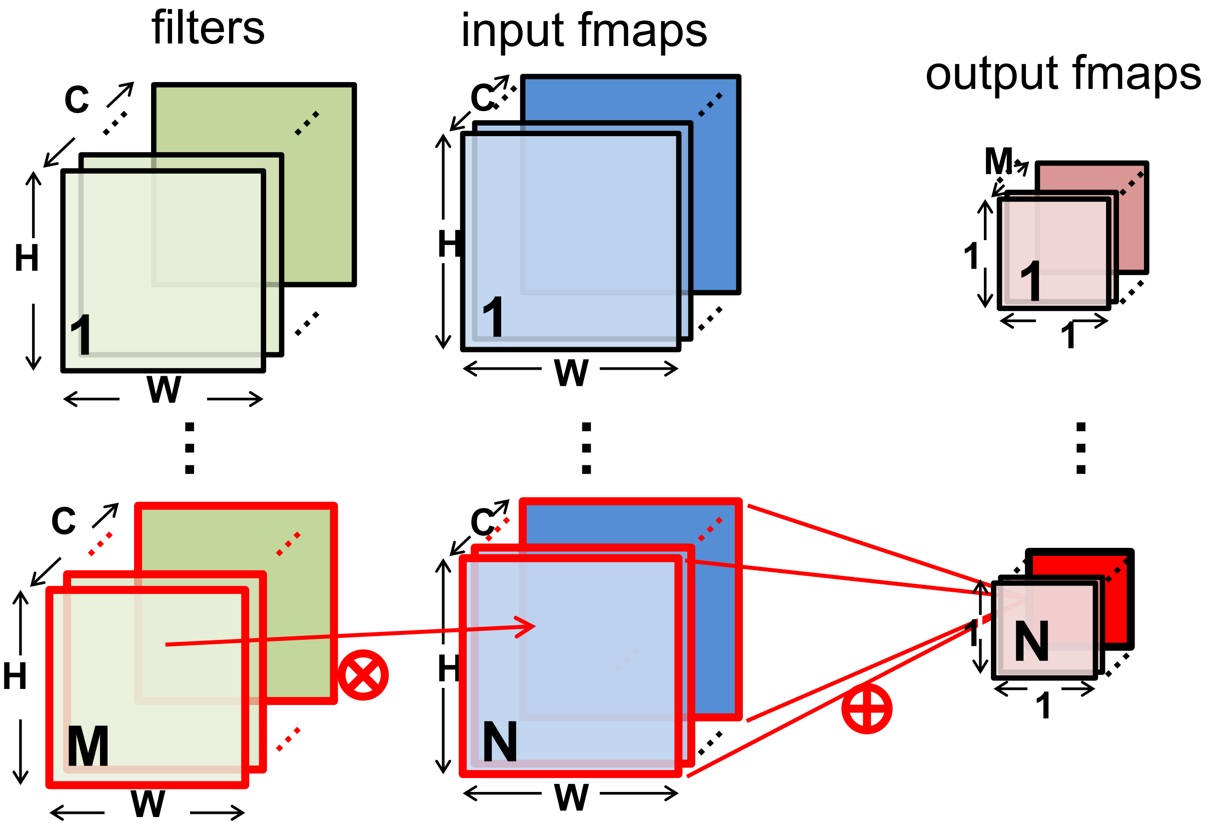
[**Krizhevsky**, *NeurIPS* 2012]

Fully-Connected (FC) Layer

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

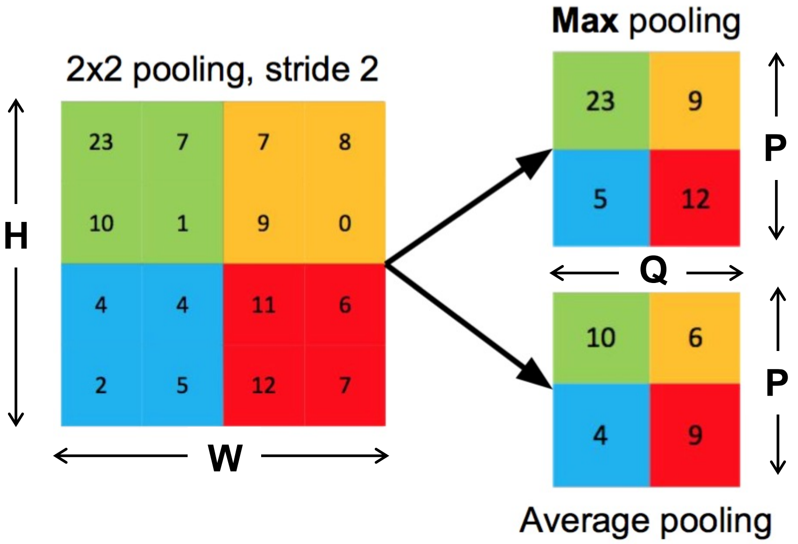


FC Layer – from CONV Layer POV



Pooling (POOL) Layer

- **Reduce** resolution of each channel independently
  - Specifically, for shape parameters: P ≤ H, Q ≤ W, M = C
- Overlapping or non-overlapping → depending on stride



2x2 pooling, stride 2

23	7	7	8
10	1	9	0
4	4	11	6
2	5	12	7

Max pooling

23	9
5	12

Average pooling

10	6
4	9

Increases translation-invariance and noise-resilience

Used in *encoder* DNN models

Image Source: Caffe Tutorial

POOL Layer Implementation

Naïve 6-layer for-loop max-pooling implementation:

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

                max = -Inf
                for r in [0..R):
                    for s in [0..S):
                        if I[n][m][Up+r][Uq+s] > max:
                            max = I[n][m][Up+r][Uq+s];

                O[n][m][p][q] = max
```

for each pooled value

find the max in a window

Pooling Einsums

Average Pooling

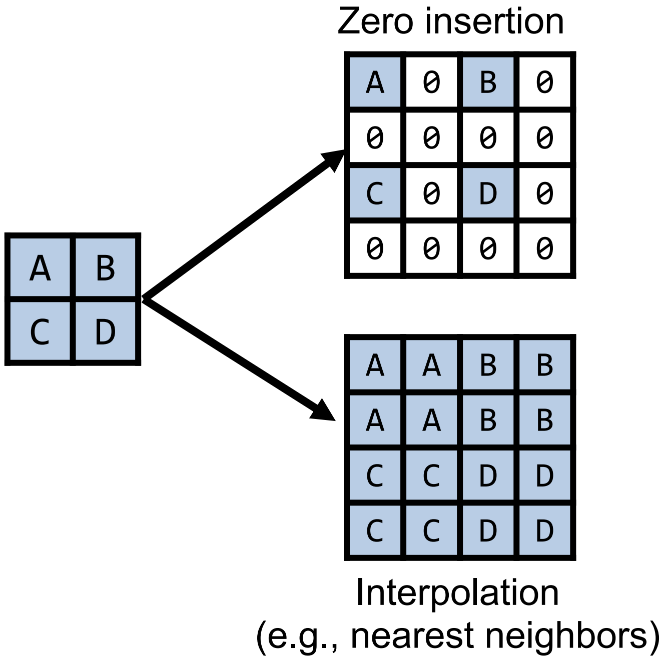
$$O_{n,m,p,q} = \frac{I_{n,m,Up+r,Uq+s}}{U^2}$$

Maximum Pooling

$$O_{n,m,p,q} = Max\left(I_{n,m,Up+r,Uq+s}\right)$$

Upsampling Layer

- **Increase** resolution of each channel independently
  - Specifically, for shape parameters: P ≥ H, Q ≥ W, M = C



Used in *decoder* DNN models

Upsampling Einsums

Zero insertion

$$O_{m,n,U\times h,U\times w} = I_{m,n,h,w}$$

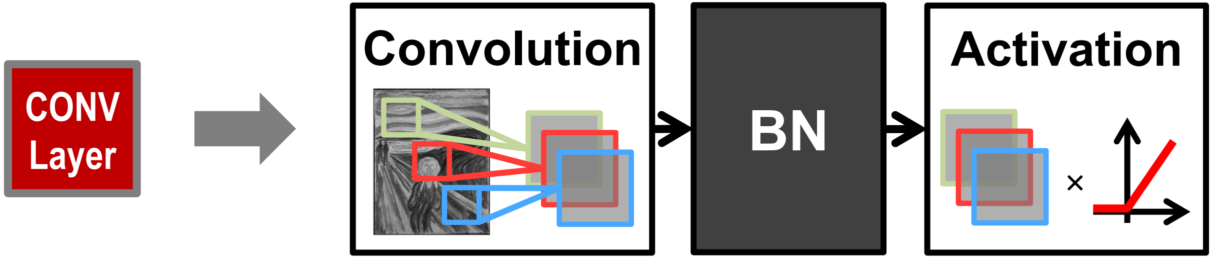
Interpolation

$$O_{m,n,h+r,w+s} = I_{m,n,h,w}$$

where r & s vary over a range of [0,U)

Normalization (NORM) Layer

- **Batch Normalization (BN)**
  - Normalize activations towards mean=0 and std. dev.=1 based on the statistics of the training dataset
  - put **in between** **CONV/FC** and **Activation function**

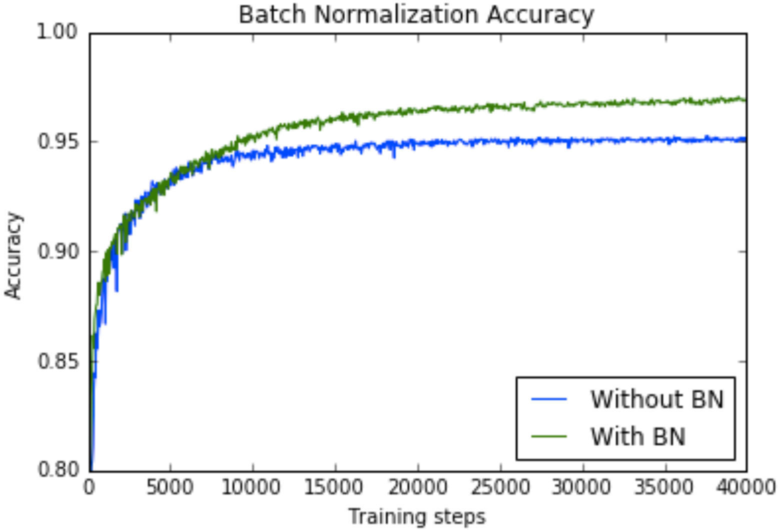


Believed to be key to getting high accuracy and faster training on very deep neural networks.

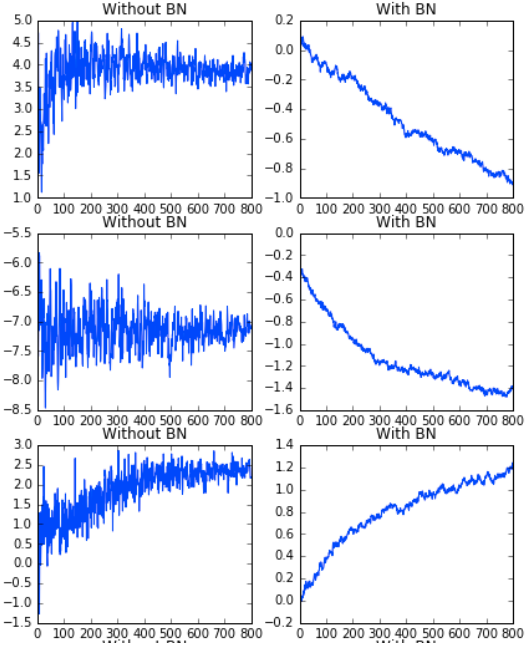
[Ioffe, *ICML* 2015]

Impact of Batch Normalization

Faster Training



Less Noisy Activations



time

time

Image Source: r2rt.com

BN Layer Implementation

The normalized value is further scaled and shifted, the parameters of which are learned from training

data mean

learned scale factor

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta$$

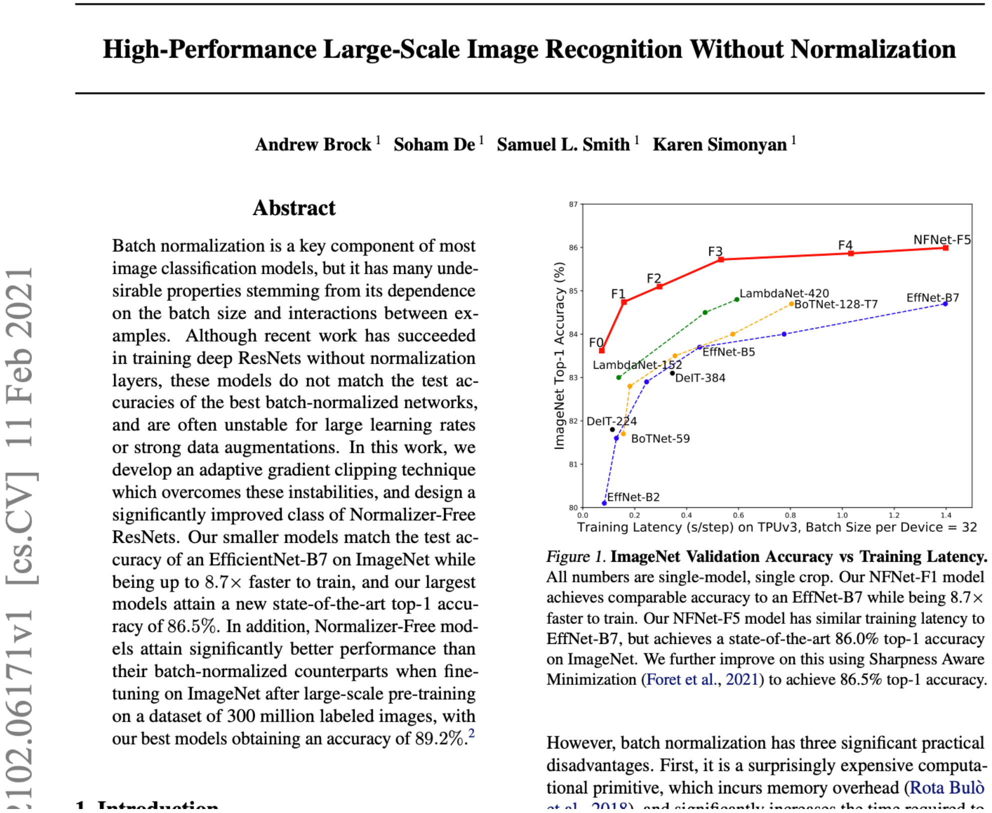
learned shift factor

data std. dev.

small const. to avoid numerical problems

For inference, computation can be folded into the weights of the CONV or FC

Normalization-Free Nets: No Need for Batch Norm!



High-Performance Large-Scale Image Recognition Without Normalization

Andrew Brock [1] **Soham De** [1] **Samuel L. Smith** [1] **Karen Simonyan** [1]

Abstract

Batch normalization is a key component of most image classification models, but it has many undesirable properties stemming from its dependence on the batch size and interactions between examples. Although recent work has succeeded in training deep ResNets without normalization layers, these models do not match the test accuracies of the best batch-normalized networks, and are often unstable for large learning rates or strong data augmentations. In this work, we develop an adaptive gradient clipping technique which overcomes these instabilities, and design a significantly improved class of Normalizer-Free ResNets. Our smaller models match the test accuracy of an EfficientNet-B7 on ImageNet while being up to 8.7× faster to train, and our largest models attain a new state-of-the-art top-1 accuracy of 86.5%. In addition, Normalizer-Free models attain significantly better performance than their batch-normalized counterparts when finetuning on ImageNet after large-scale pre-training on a dataset of 300 million labeled images, with our best models obtaining an accuracy of 89.2%.[2]

1. Introduction

Figure 1. **ImageNet Validation Accuracy vs Training Latency.** All numbers are single-model, single crop. Our NFNet-F1 model achieves comparable accuracy to an EffNet-B7 while being 8.7× faster to train. Our NFNet-F5 model has similar training latency to EffNet-B7, but achieves a state-of-the-art 86.0% top-1 accuracy on ImageNet. We further improve on this using Sharpness Aware Minimization (Foret et al., 2021) to achieve 86.5% top-1 accuracy.

However, batch normalization has three significant practical disadvantages. First, it is a surprisingly expensive computational primitive, which incurs memory overhead (Rota Bulò et al., 2018), and significantly increases the time required to

State-of-the-art accuracy without batch normalization!

[Brock, *ICML* 2021]

Relevant Components for Class

- Typical operations that we will use:
  - Convolution (CONV)
  - Fully-Connected (FC)
  - Max Pooling
  - ReLU

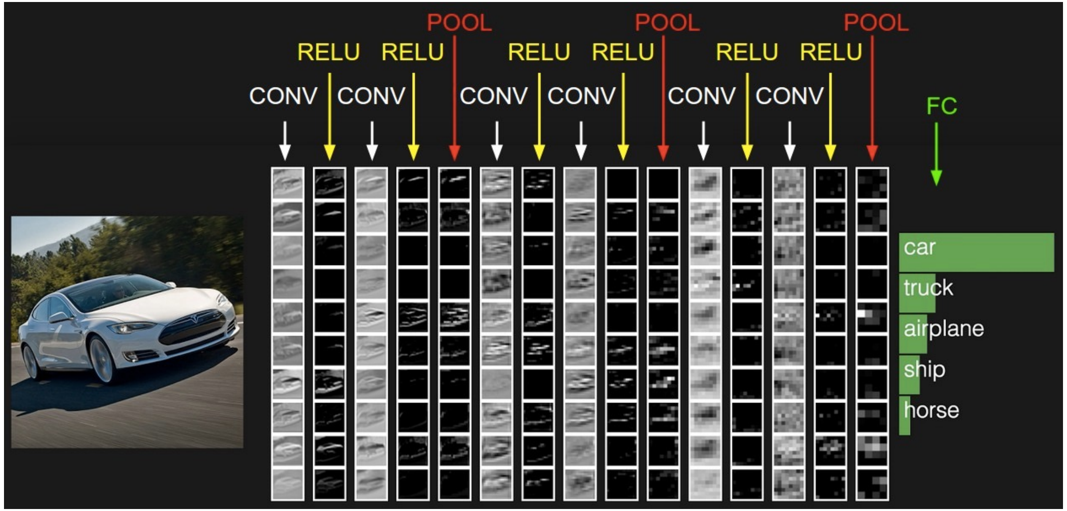
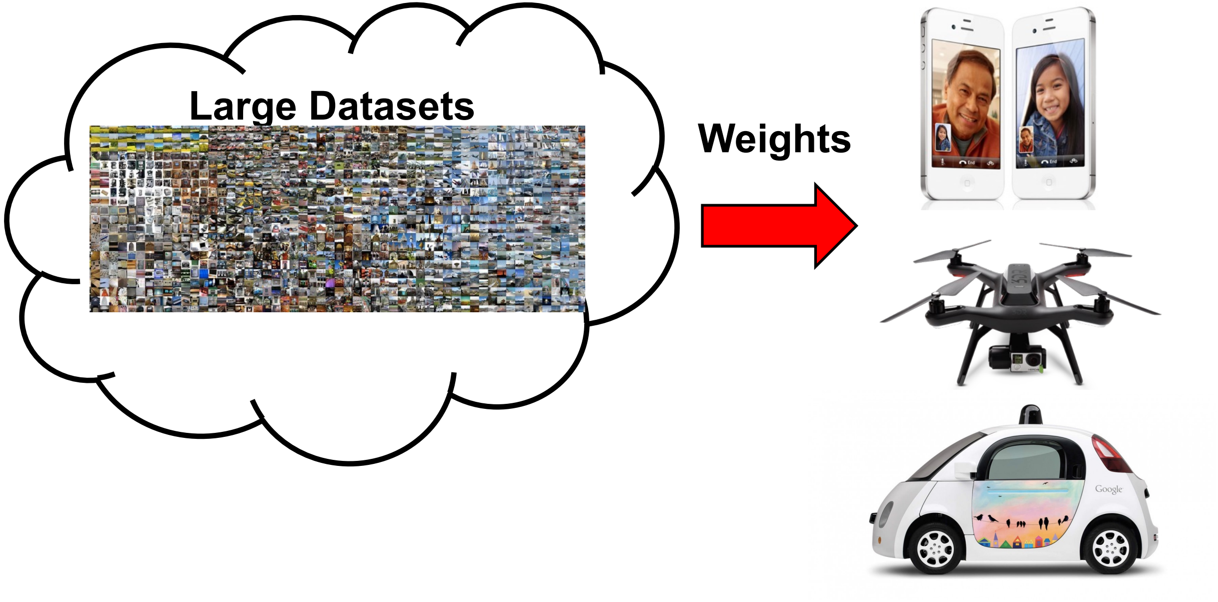


Image Source: Stanford

Training versus Inference

Training
(determine weights)

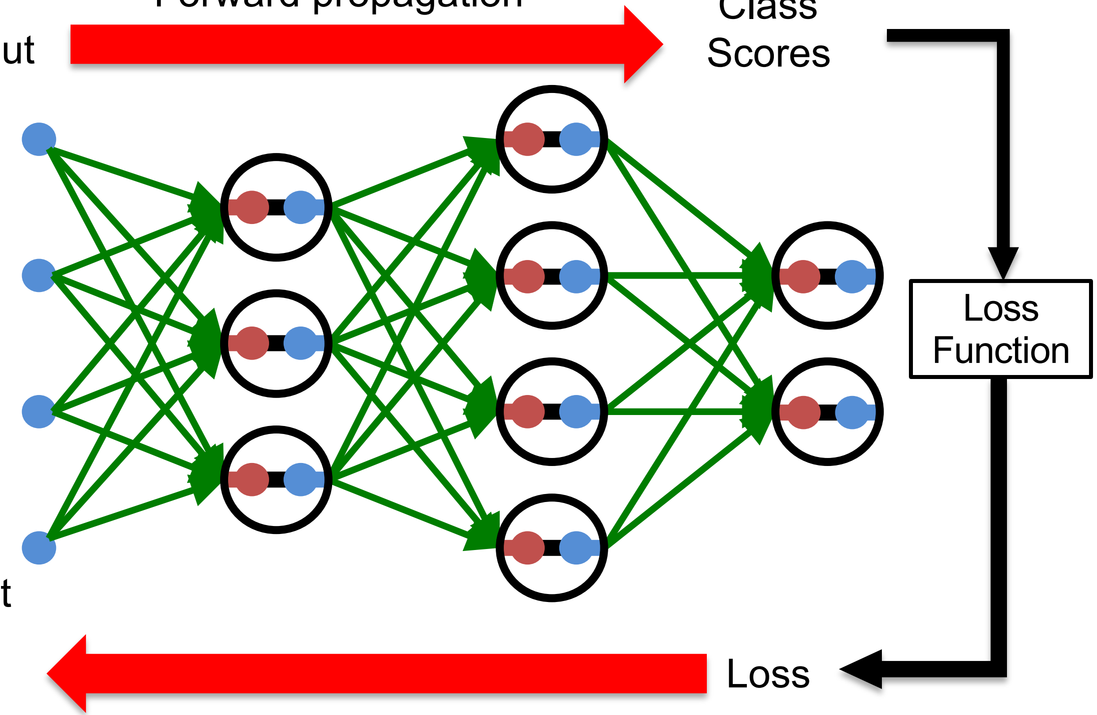
Inference
(use weights)



Large Datasets

Weights

Training DNN



Forward propagation*

Input

Class Scores

Loss Function

Loss

Gradient

$$\frac{\partial L}{\partial w_{ij}}$$

Back propagation

* inference also uses forward propagation

Summary

- Terminology for Deep Neural Networks (DNN)
  - synapse → **weights**, neuron output → **activations**
  - **filter** = set of weights; **feature map** = set of activations
- Different **layers** in a DNN
  - Convolution (**CONV**), Pooling (**POOL**), Activation (**RELU**), Normalization (**NORM**), Fully Connected (**FC**)
  - Configuration Options: filter shapes (R,S,C,M), zero padding, avg/max pooling, activation function, etc.
- Training with forward and backward propagation

References

- Textbook: Chapter 1 & 2
  - https://doi.org/10.1007/978-3-031-01766-7
- Stanford cs231n
  - http://cs231n.github.io/convolutional-networks/
- http://www.deeplearningbook.org/
  - Chapter 9 http://www.deeplearningbook.org/contents/convnets.html
- Other Works Cited in Lecture
  - Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift," ICML 2015.