# Background of Deep Neural Networks

## Slide Reference:

**Hardware Architectures for Deep Neural Networks**

**ISCA Tutorial**

**June 22, 2019**

Website:
http://eyeriss.mit.edu/tutorial.html

Massachusetts Institute of Technology

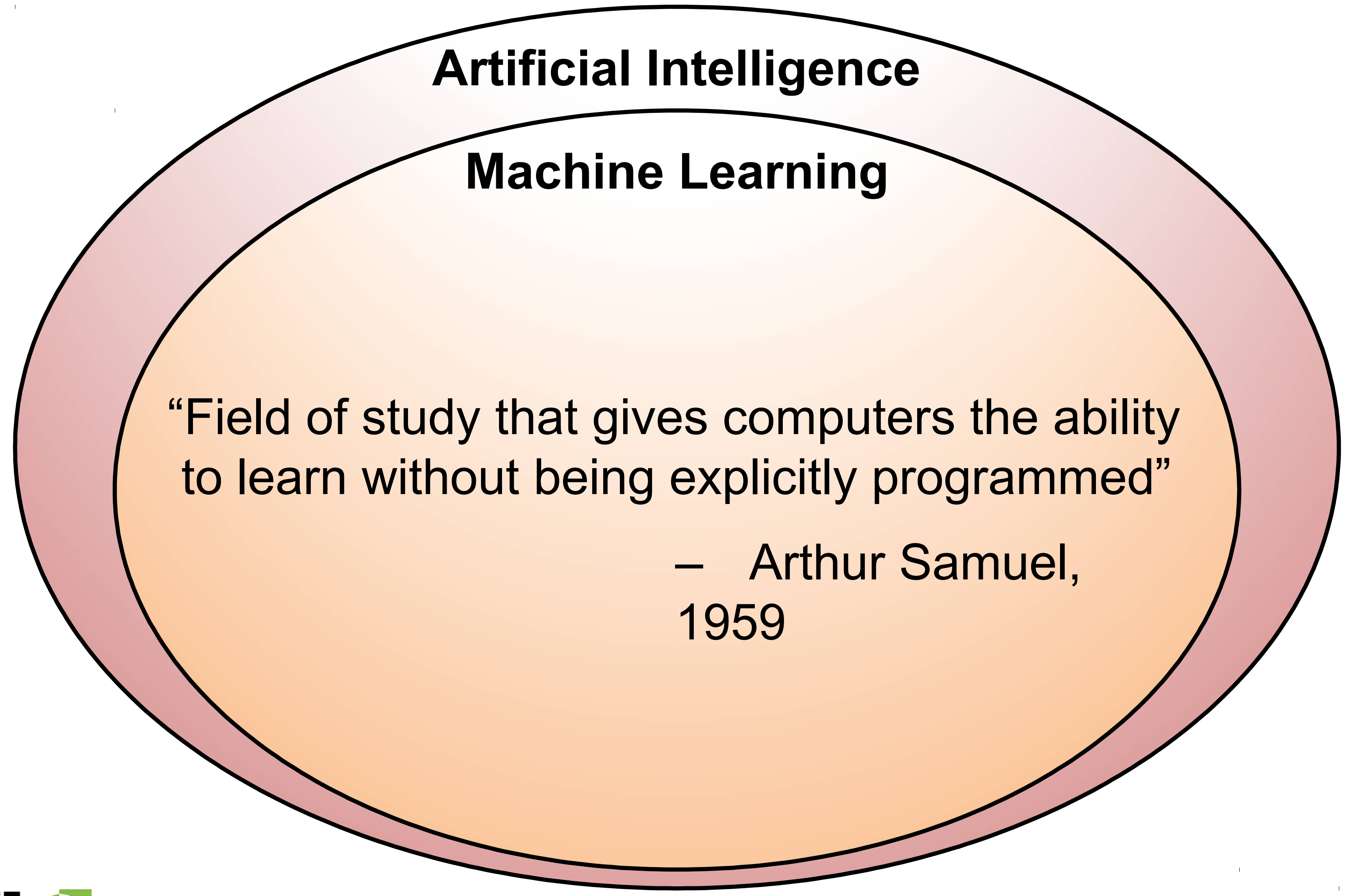
## Artificial Intelligence

**Artificial Intelligence**

"The science and engineering of creating intelligent machines"

- John McCarthy, 1956

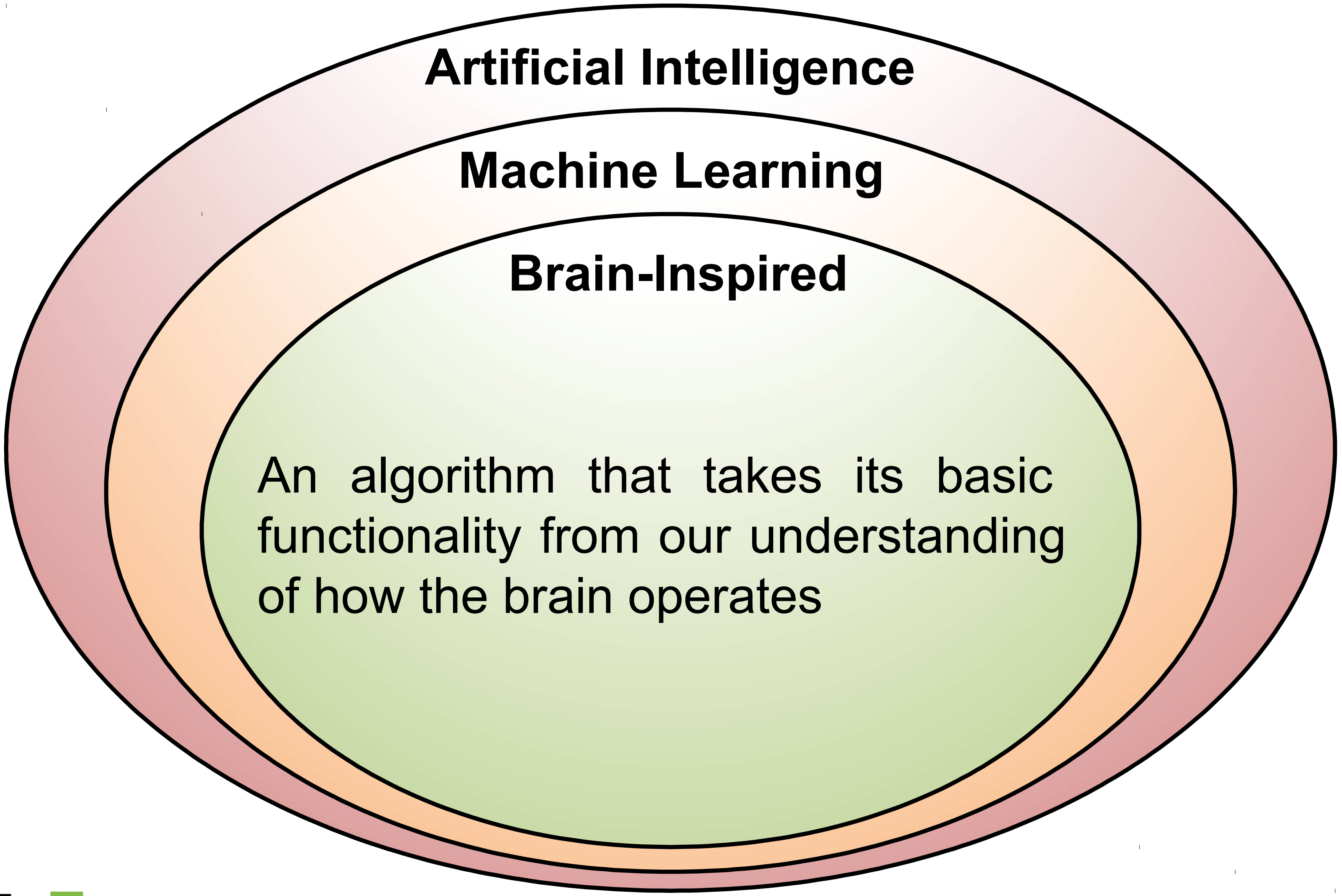
# AI and Machine Learning



"Field of study that gives computers the ability to learn without being explicitly programmed"

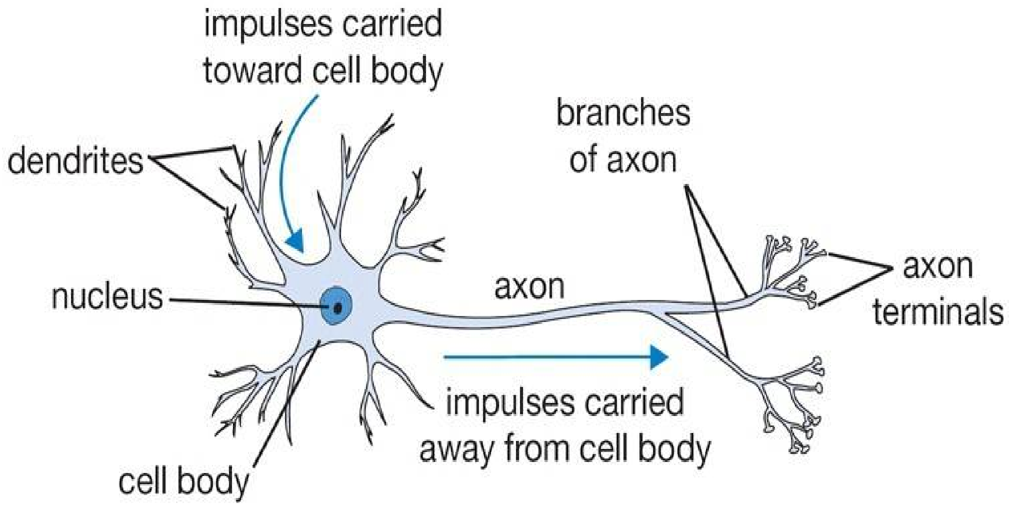
– Arthur Samuel, 1959

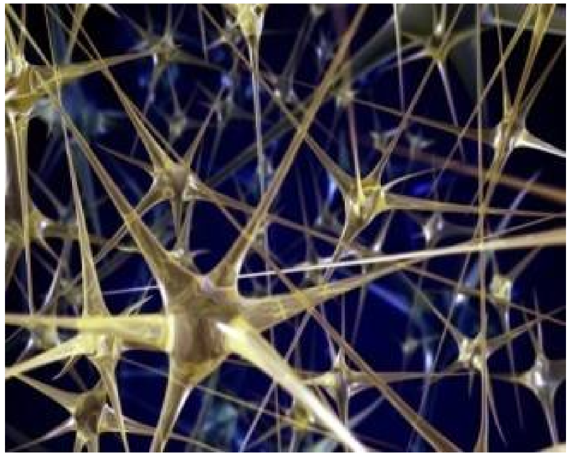
## Brain-Inspired Machine Learning



An algorithm that takes its basic functionality from our understanding of how the brain operates

# How Does the Brain Work?

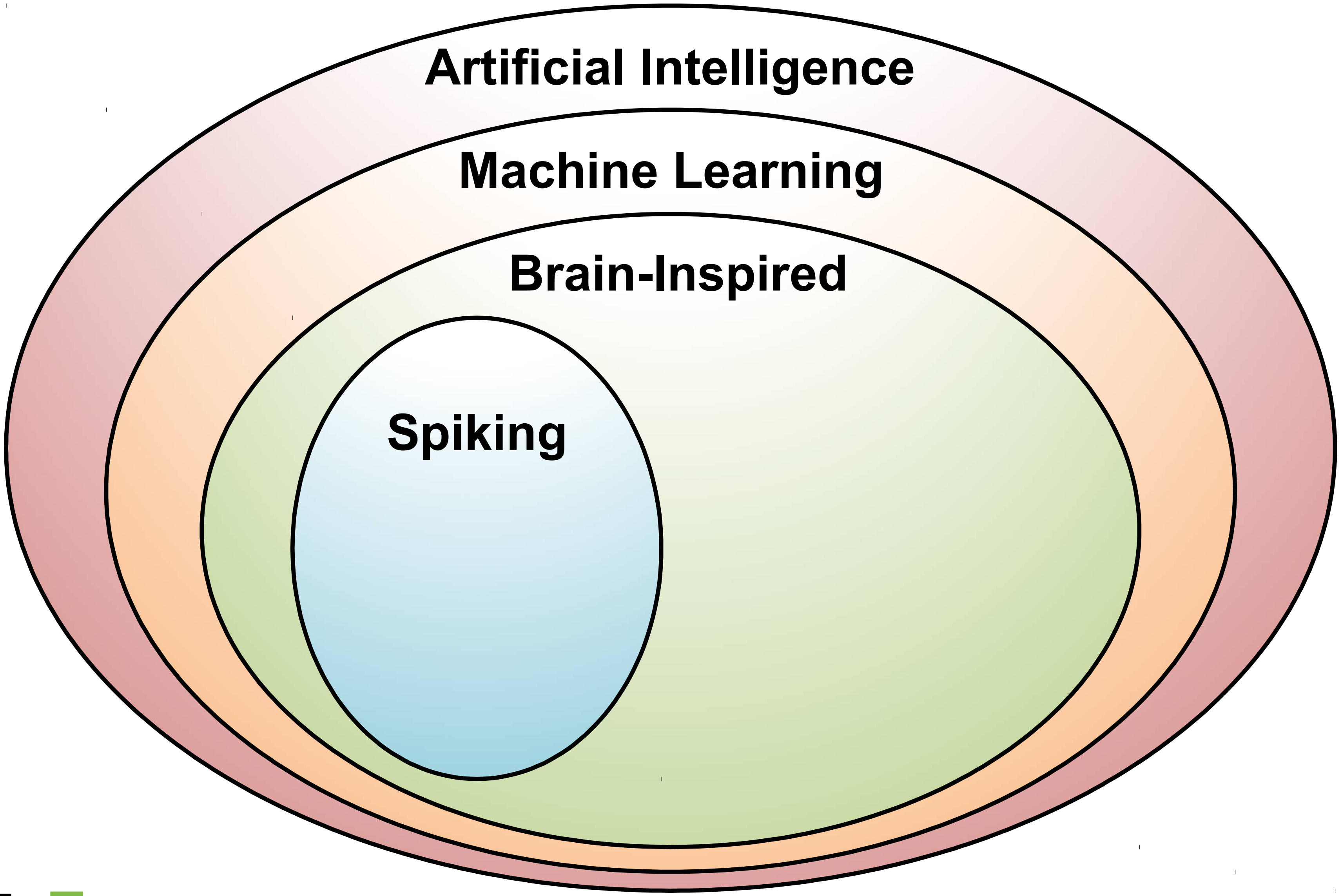

impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body



- The basic computational unit of the brain is a **neuron**
   86B neurons in the brain
- Neurons are connected with nearly **$10^{14}$ – $10^{15}$ synapses**
- Neurons receive input signal from **dendrites** and produce output signal along **axon**, which interact with the dendrites of other neurons via **synaptic weights**
- Synaptic weights – learnable & control influence strength

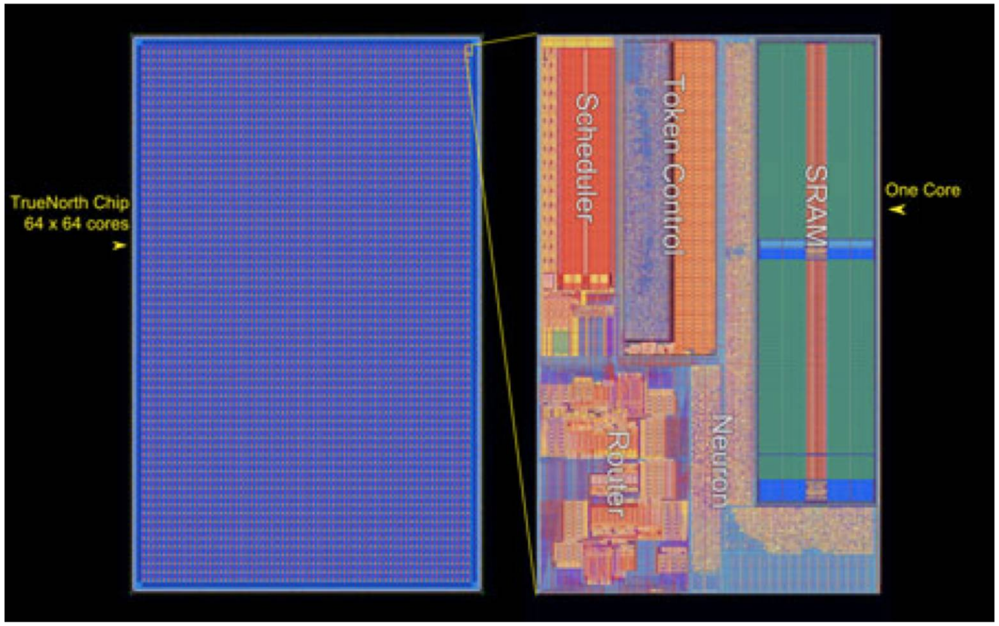
Image Source: Stanford

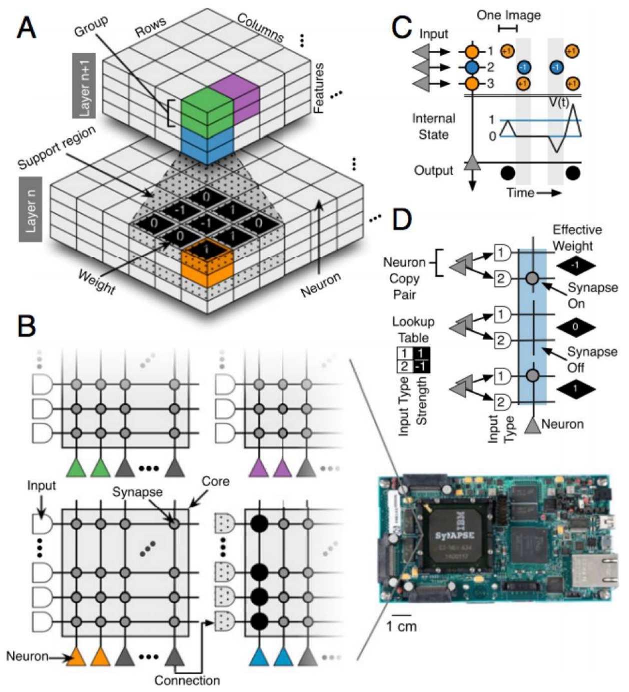
## Spiking-based Machine Learning



## Spiking Architecture

- Brain-inspired
- Integrate and fire: Once a neuron reaches a certain potential, it spikes and resets its potential
- Example: IBM TrueNorth

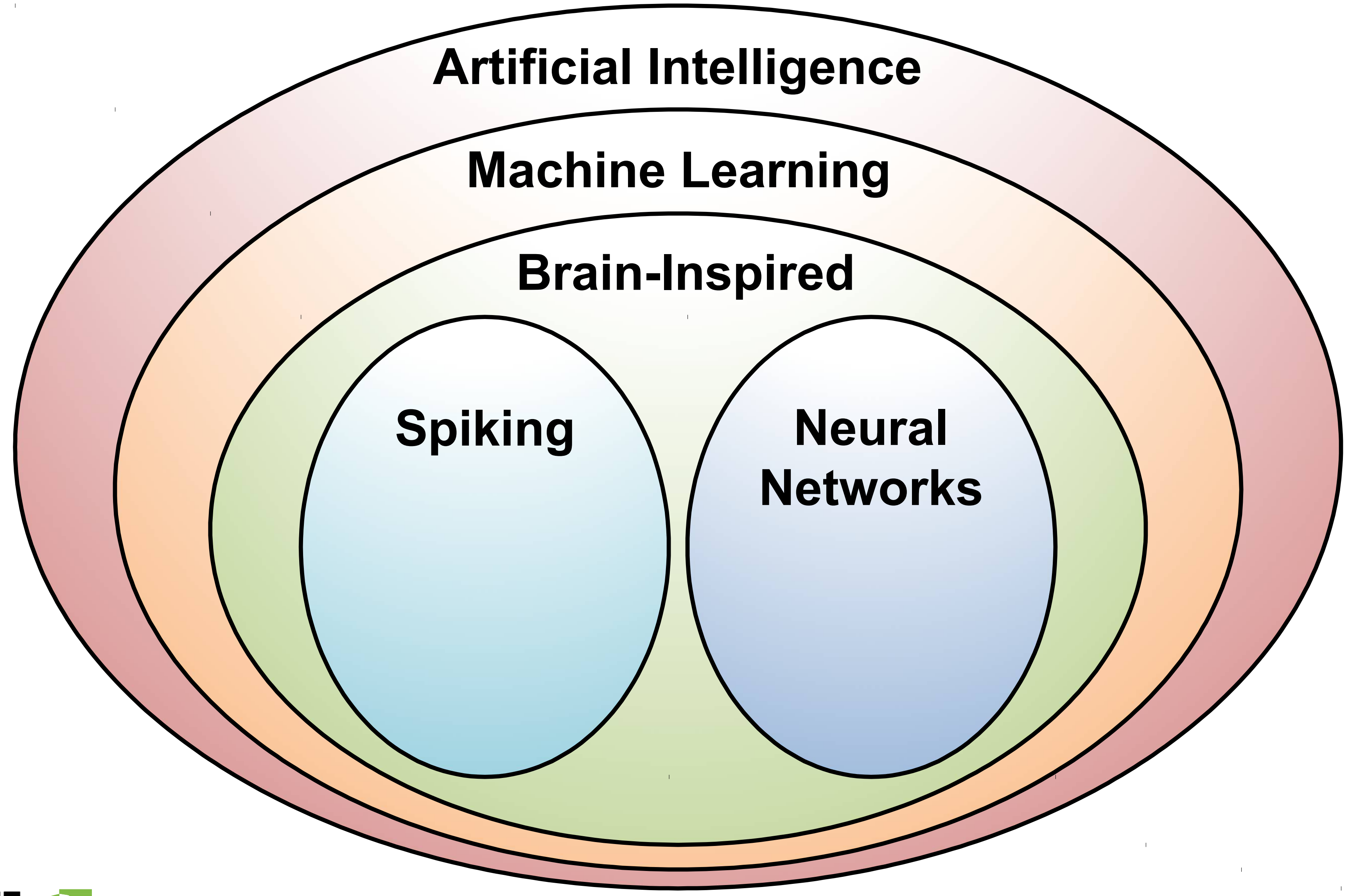




[Merolla et al., Science 2014; Esser et al., PNAS 2016]

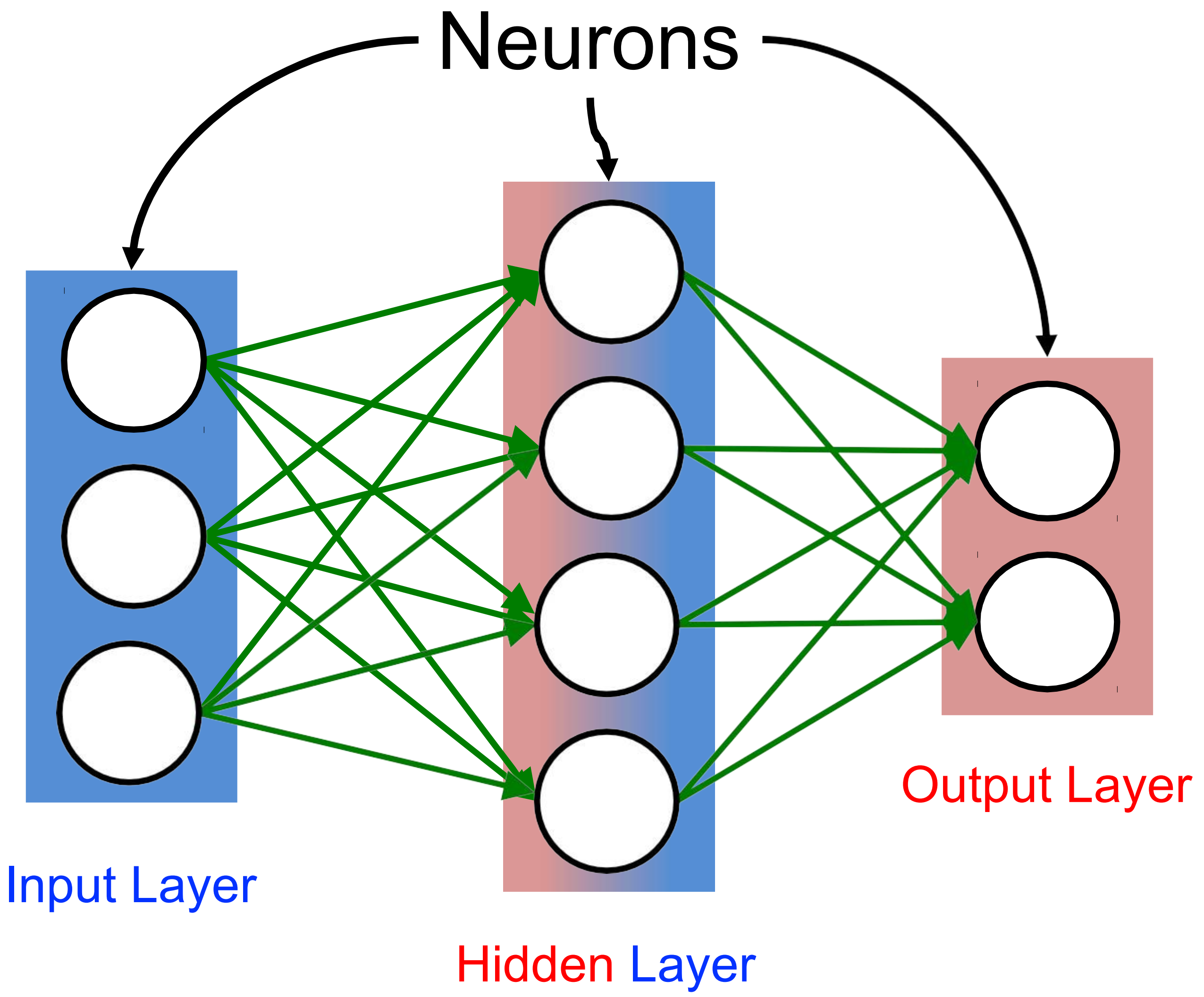
http://www.research.ibm.com/articles/brain-chip.shtml

# Machine Learning with Neural Networks

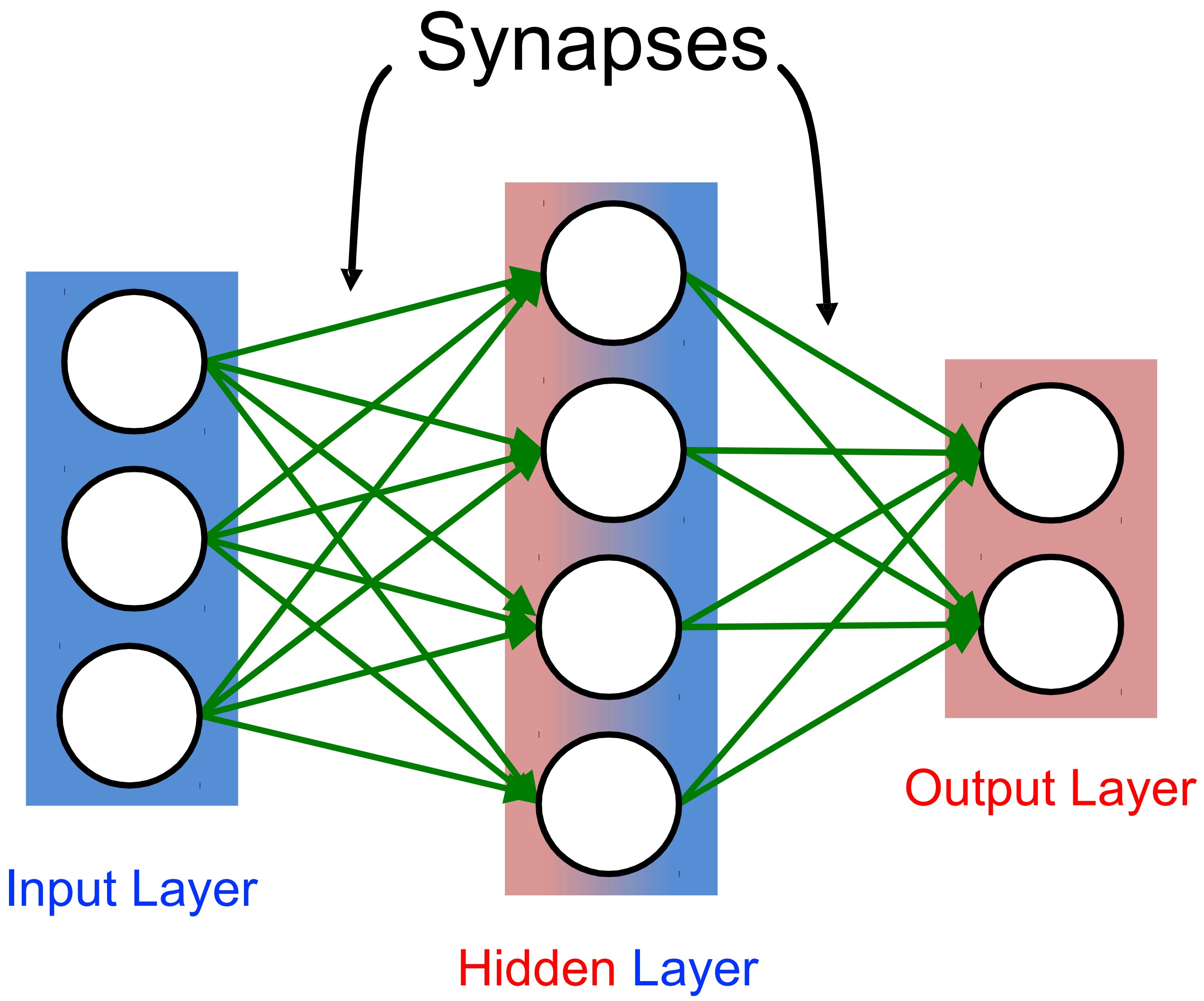


## Overview of Deep Neural Networks

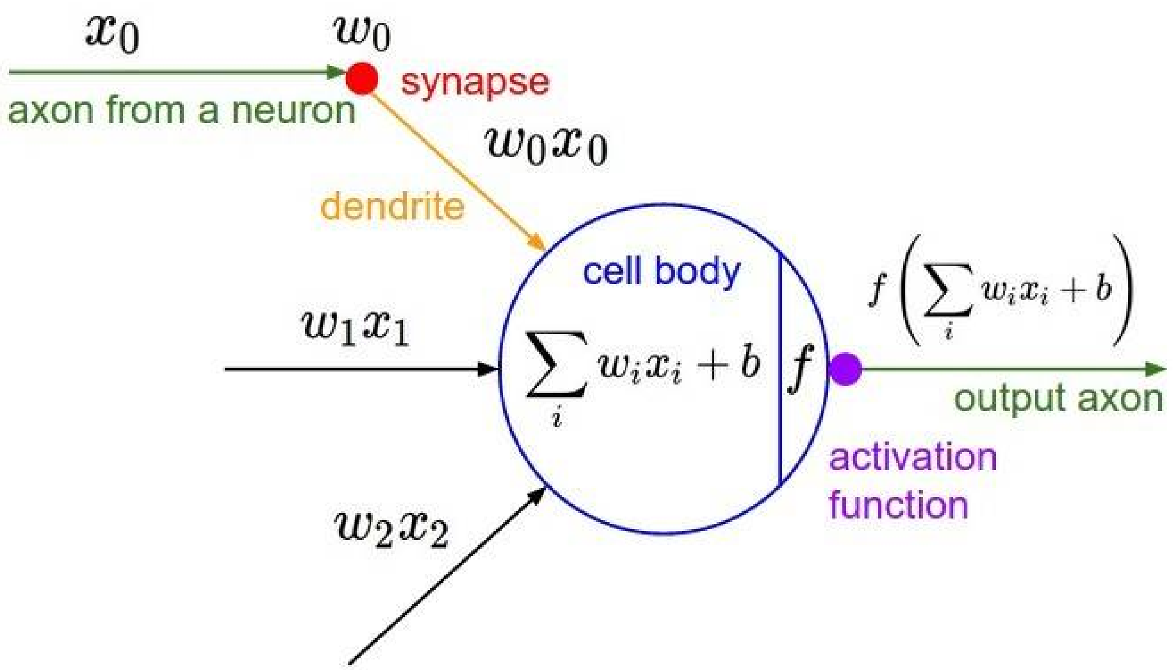
# DNN Terminology 101



# DNN Terminology 101



# Neural Networks: Weighted Sum

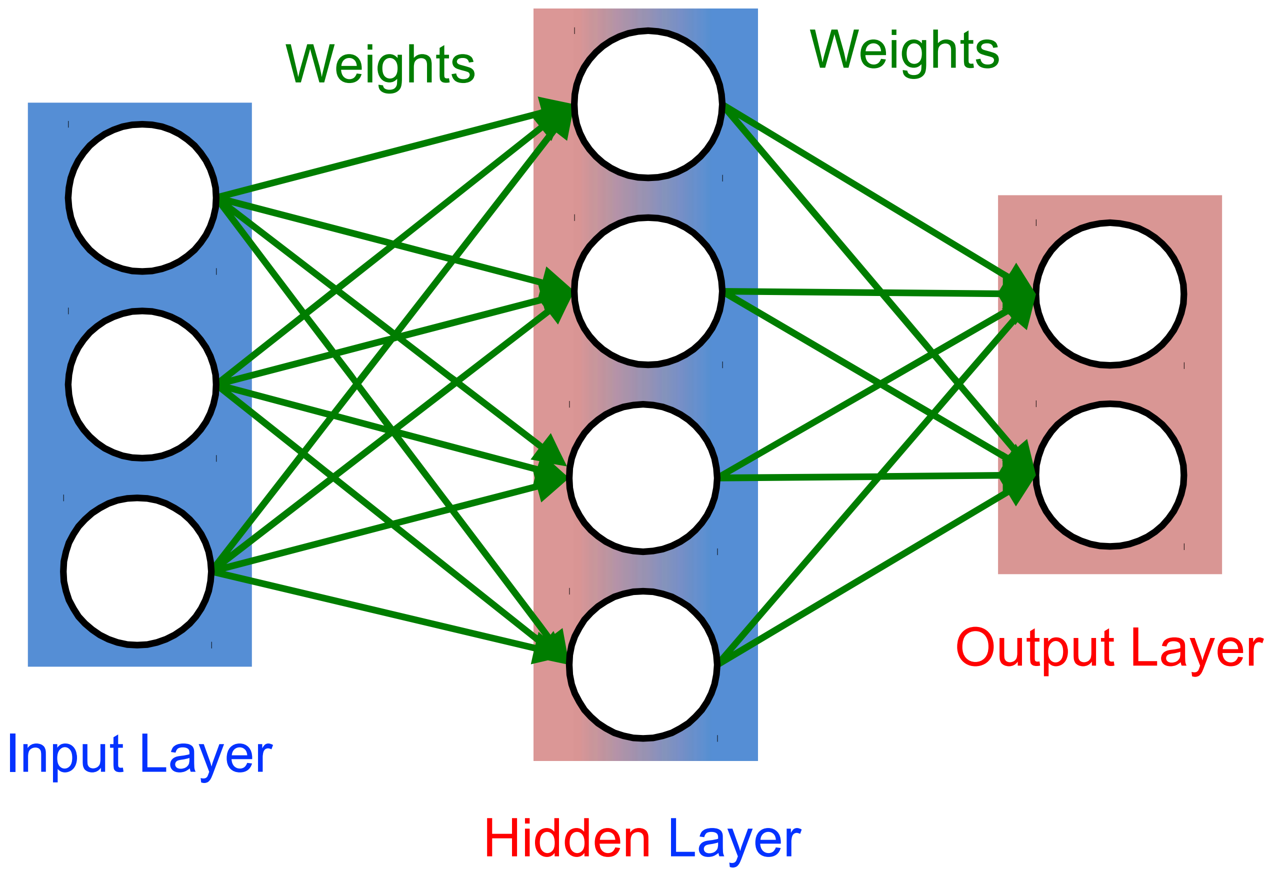


- A node, also called a neuron or Perceptron, is a computational unit that

has one or more weighted input connections, a transfer function that combines the inputs in some way, and an output connection

Image Source: Stanford

# Many Weighted Sums



# What is Deep Learning?

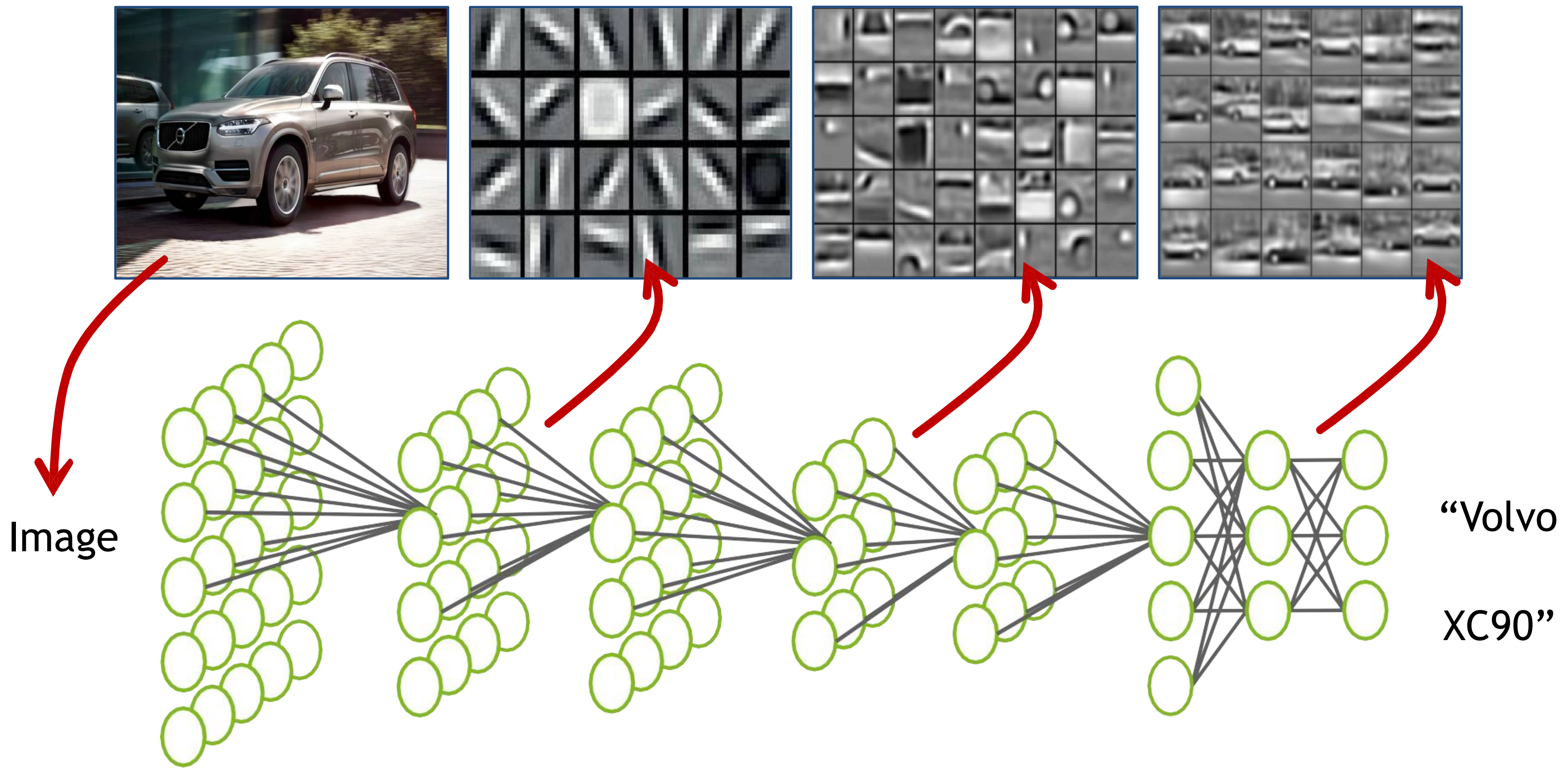
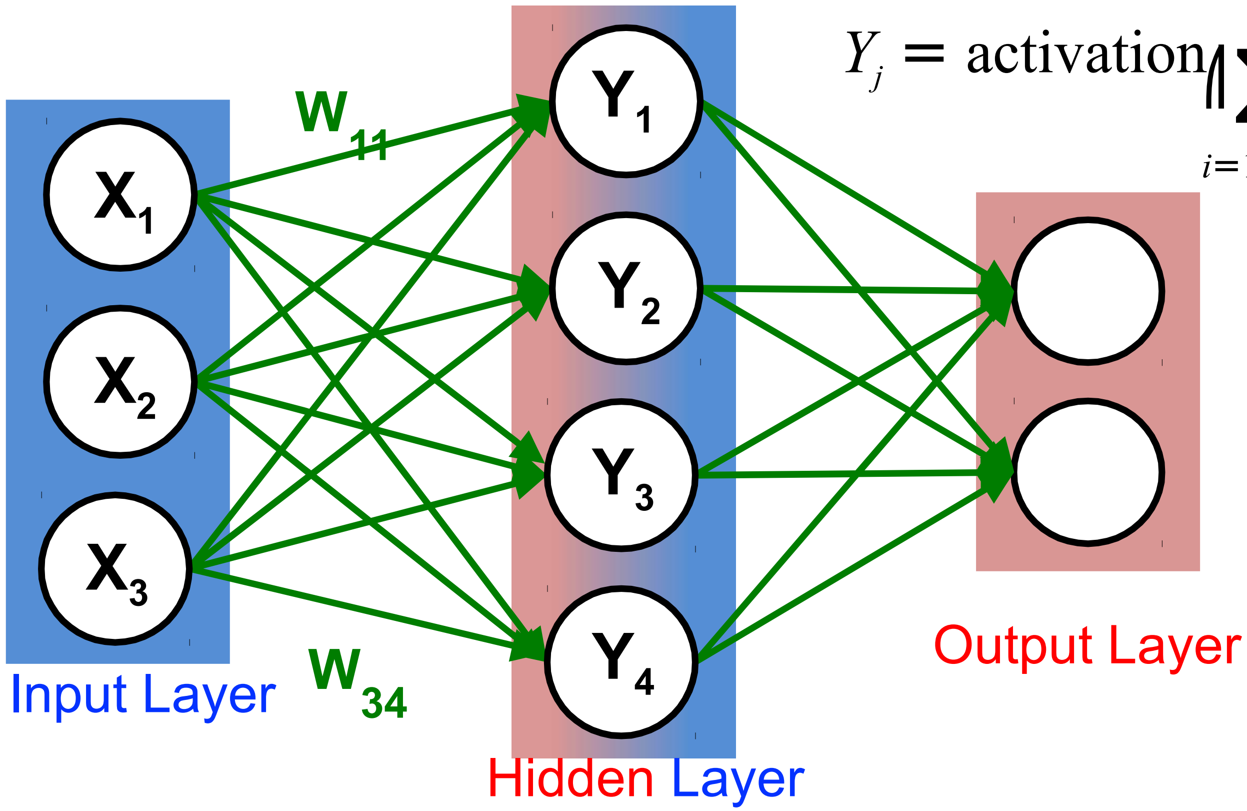


Image Source: [Lee et al., Comm. ACM 2011]

# DNN Terminology 101

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

$$Y_j = \text{activation}\left(\sum_{i=1}^{3} W_{ij} \times X_i\right)$$

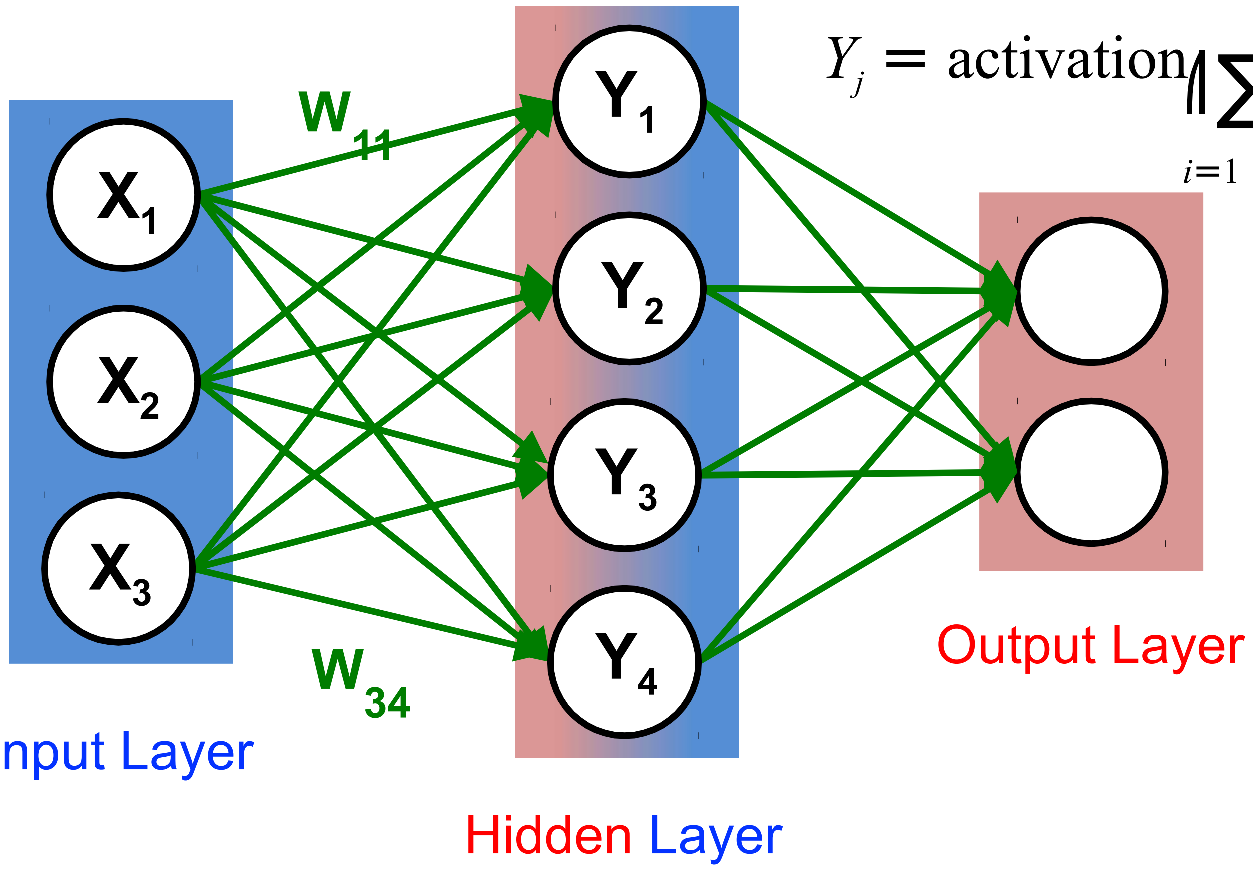


- An activation function in a neural network defines **how the weighted sum of the input is transformed into an output from a node**

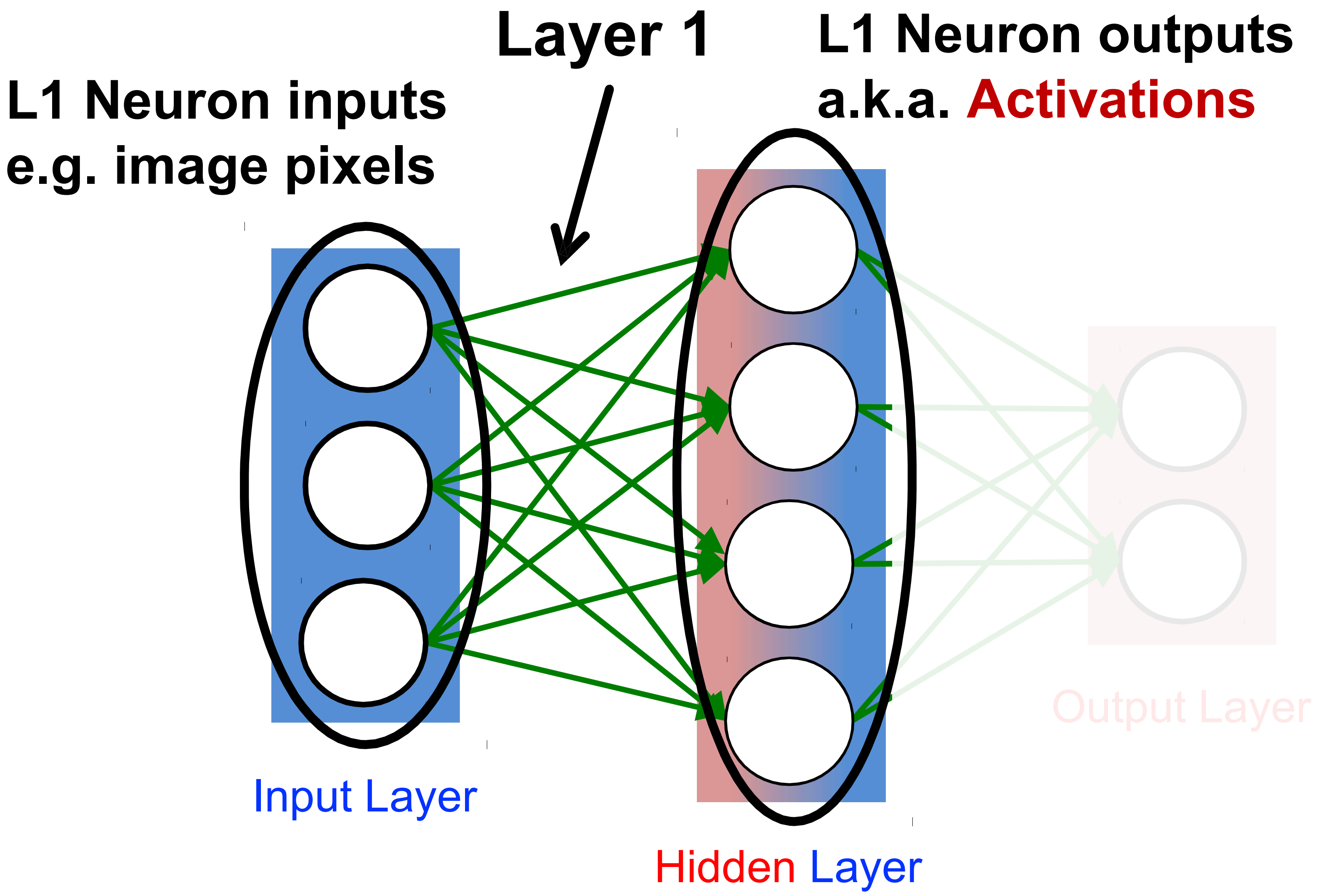
# DNN Terminology 101

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

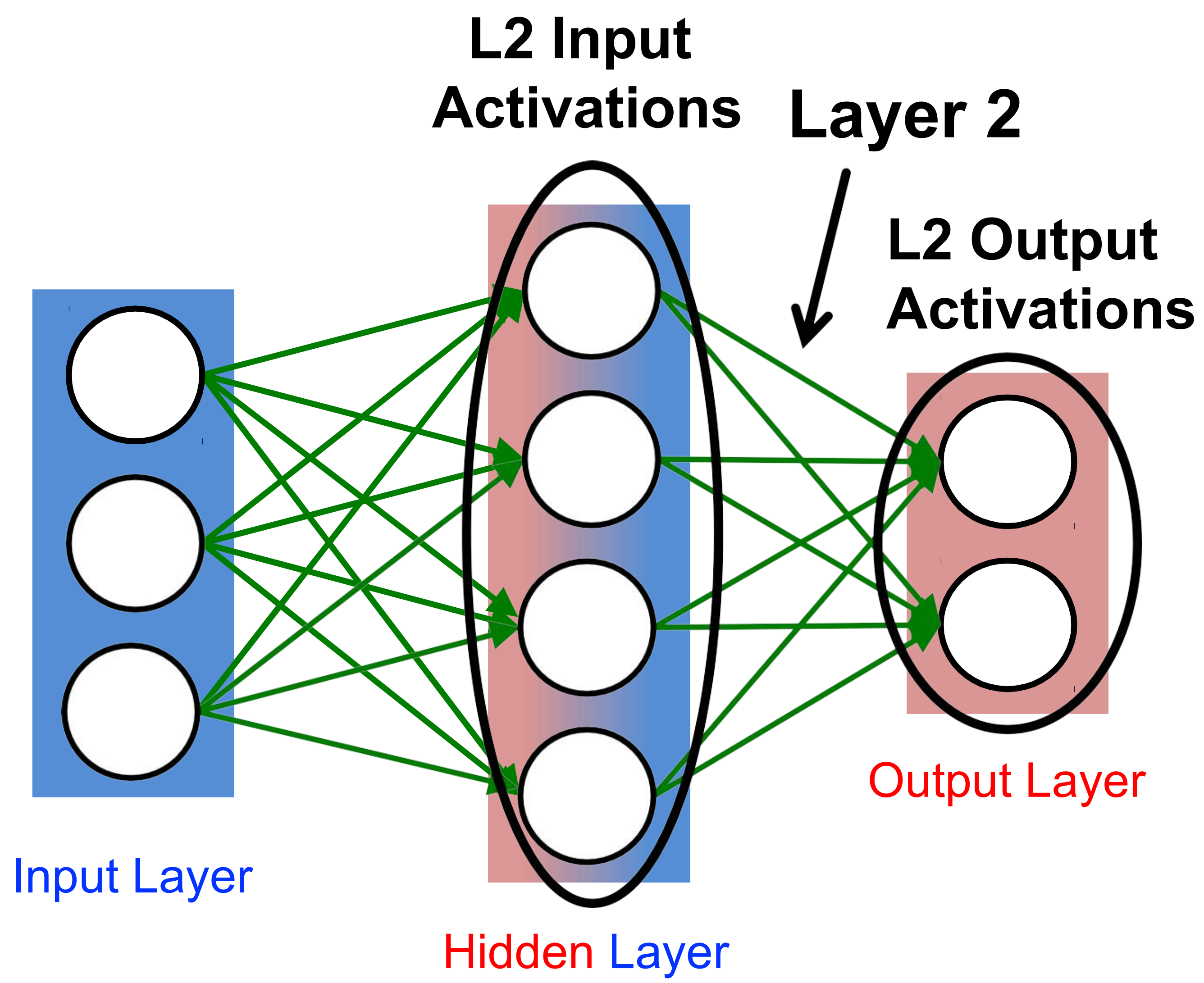
$$Y_j = \text{activation}\left(\sum_{i=1}^{3} W_{ij} \times X_i\right)$$



# DNN Terminology 101

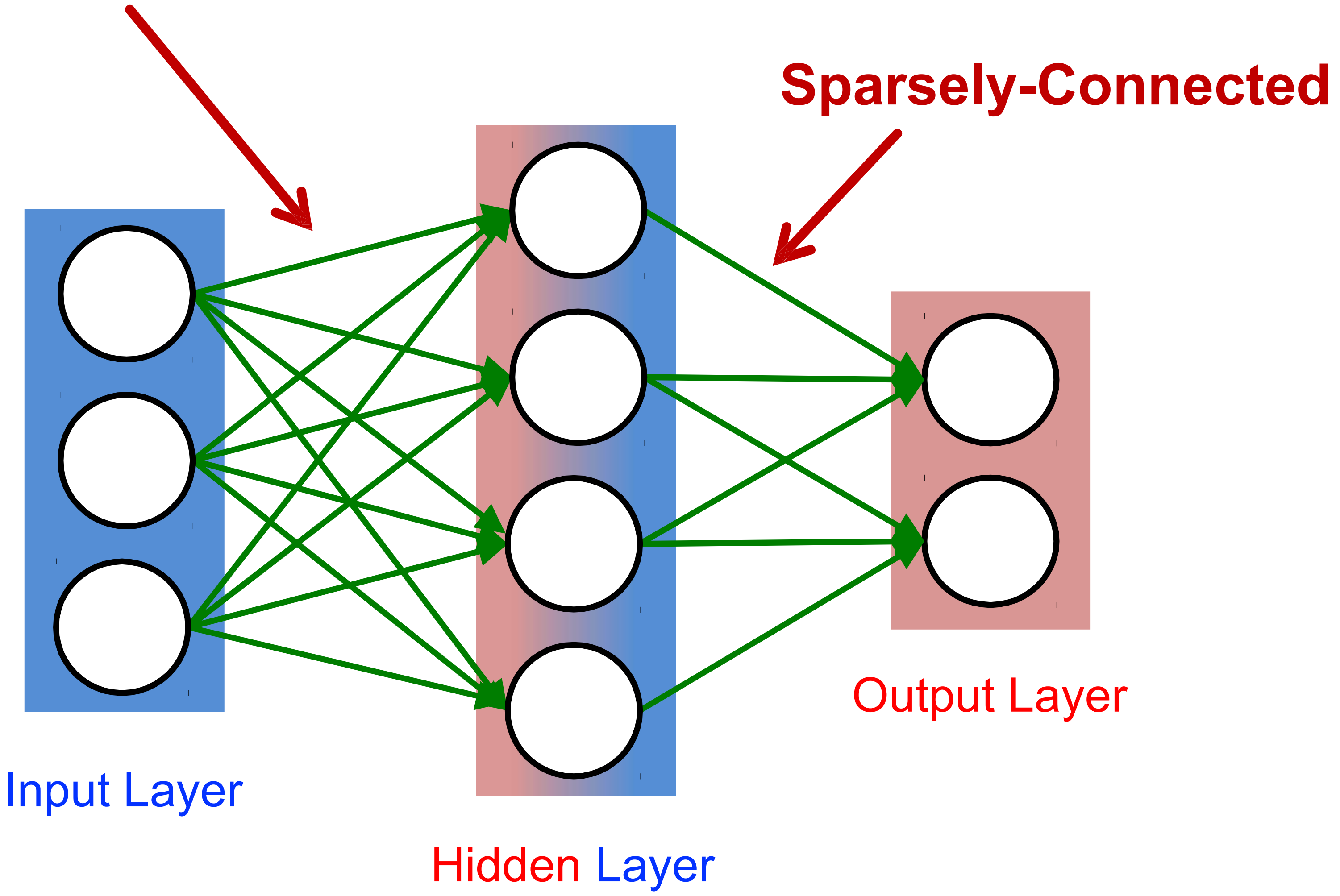


# DNN Terminology 101

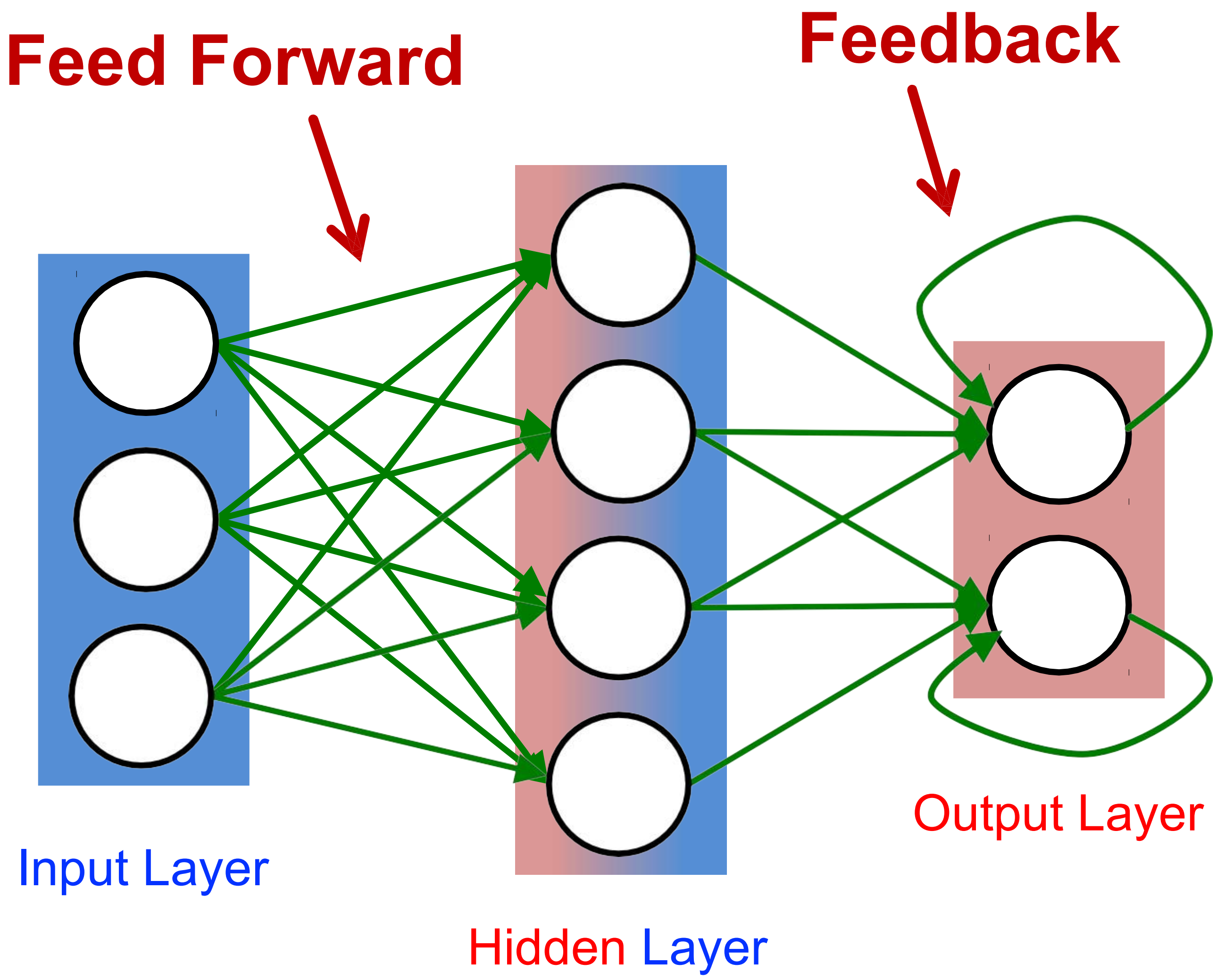


# DNN Terminology 101

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



# DNN Terminology 101



## Popular Types of DNNs

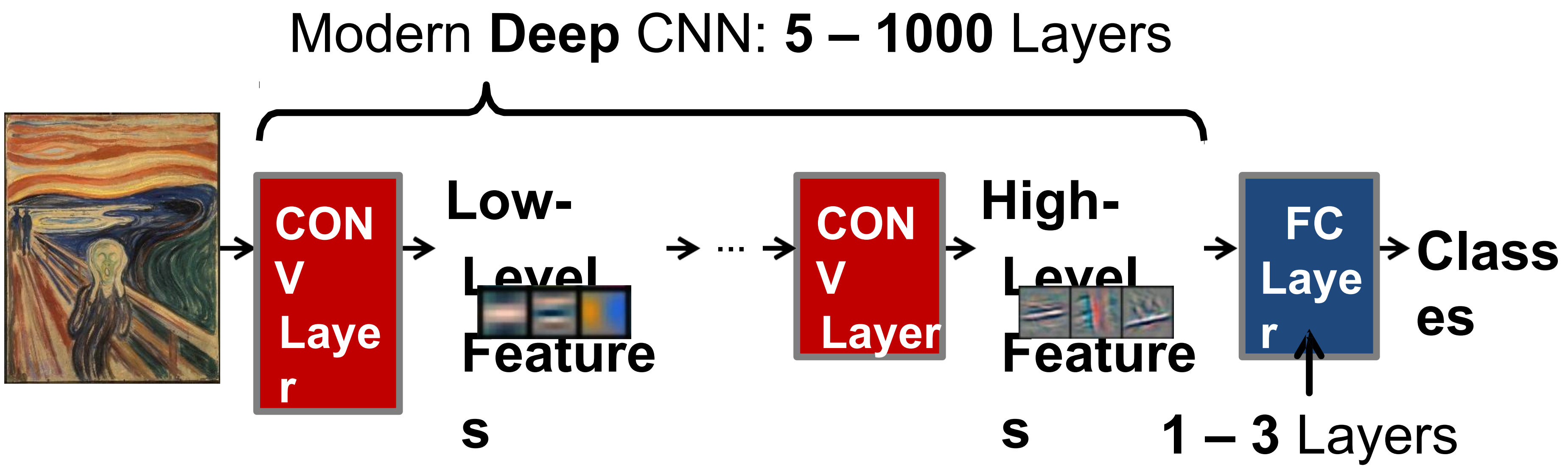
- **Fully-Connected NN**
  - feed forward, a.k.a. multilayer perceptron (MLP)
- **Convolutional NN (CNN)**
  - feed forward, sparsely-connected w/ weight sharing
- **Recurrent NN (RNN)**
  - feedback
- **Long Short-Term Memory (LSTM)**
  - feedback + storage

# Inference vs. Training

- **Training:** Determine weights
  - **Supervised:**
    - Training set has inputs and outputs, i.e., labeled
  - **Unsupervised / Self-Supervised:**
    - Training set is unlabeled
  - **Semi-supervised:**
    - Training set is partially labeled
  - **Reinforcement:**
    - Output assessed via rewards and punishments

- **Inference:** Apply weights to determine output

# Deep Convolutional Neural Networks

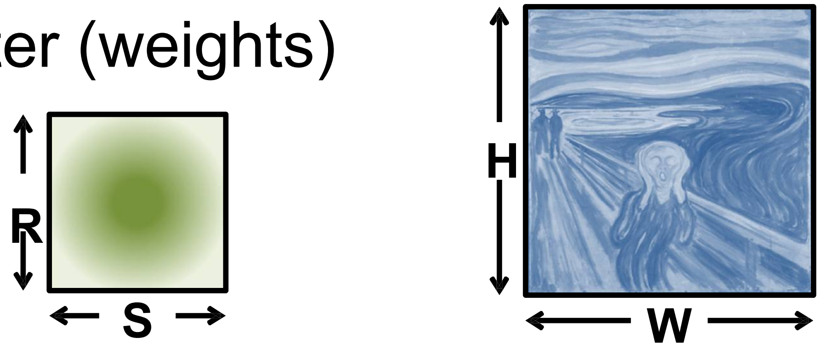


- The **features** are the elements of your input vectors.
- The number of features **is equal to the number of nodes in the input layer of the network**.
- Eg:to classify people as either men or women, the features would be things like height, weight, hair length etc.
- Fully connected layers are global (they can introduce any kind of dependence). This is also why convolutions work so well in domains like image analysis - due to their local nature.

# Convolution (CONV) Layer

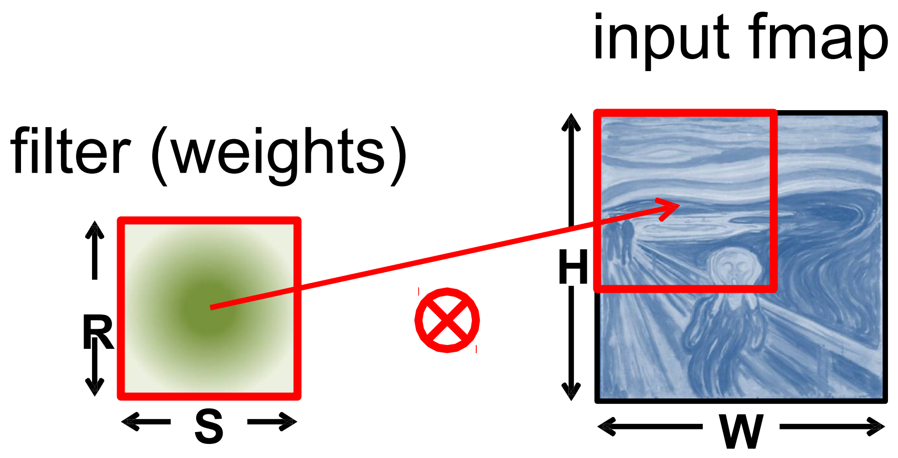
a plane of input activations
a.k.a. **input feature map (fmap)**

filter (weights)



- the input features can be concatenated into **one large matrix**.
- Each row of this matrix contains all input values necessary to compute one
- element of an output feature.
- This process means that each input element will be replicated multiple times.

# Convolution (CONV) Layer

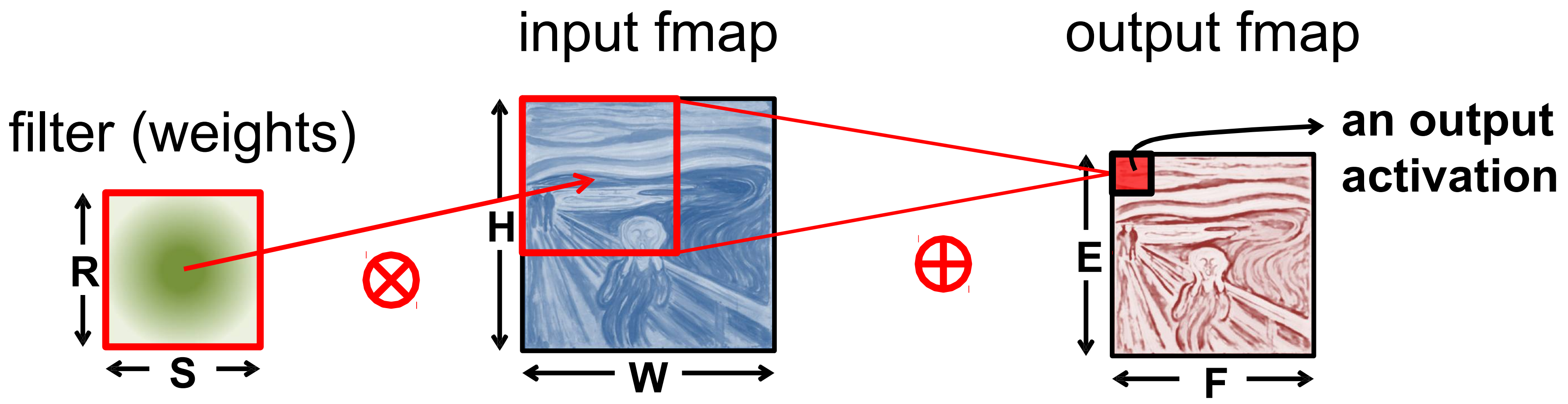


**Element-wise**

**Multiplication**

- The feature map is **the output of one filter applied to the previous layer**.

# Convolution (CONV) Layer

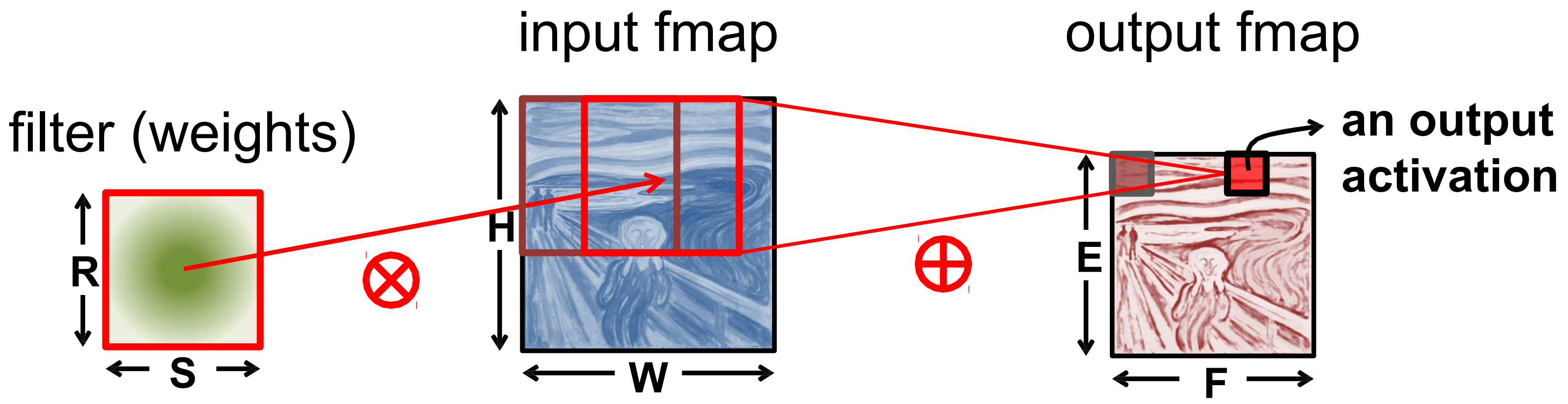


**Element-wise**

**Multiplication**

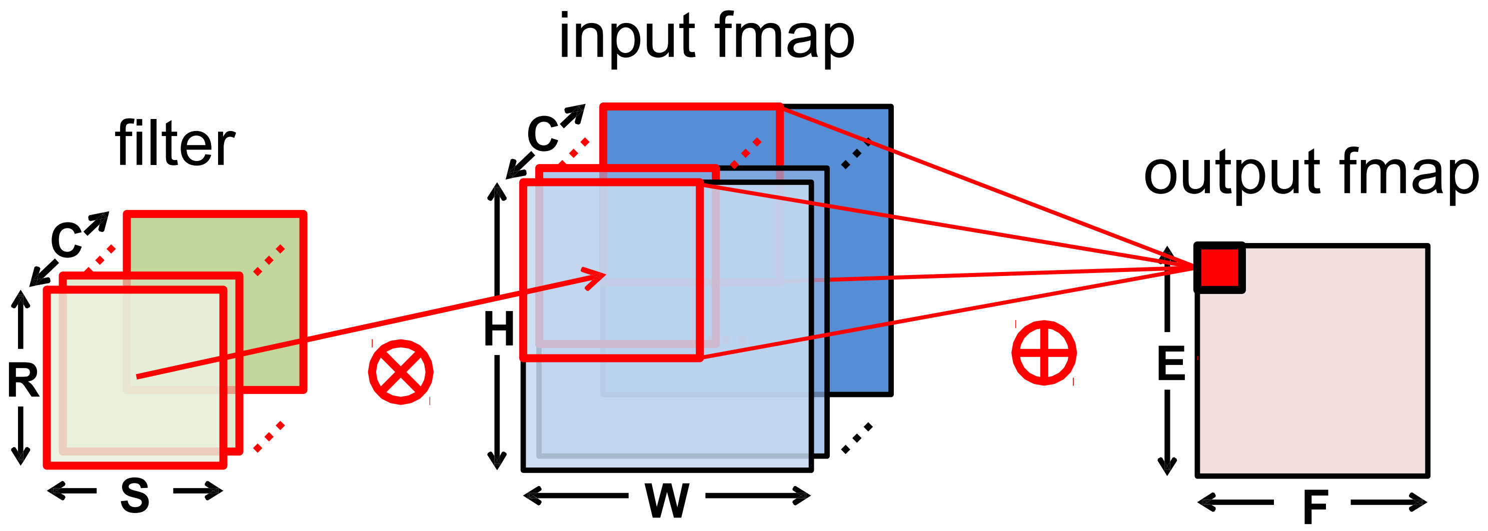
**Partial Sum** (psum) **Accumulation**

# Convolution (CONV) Layer



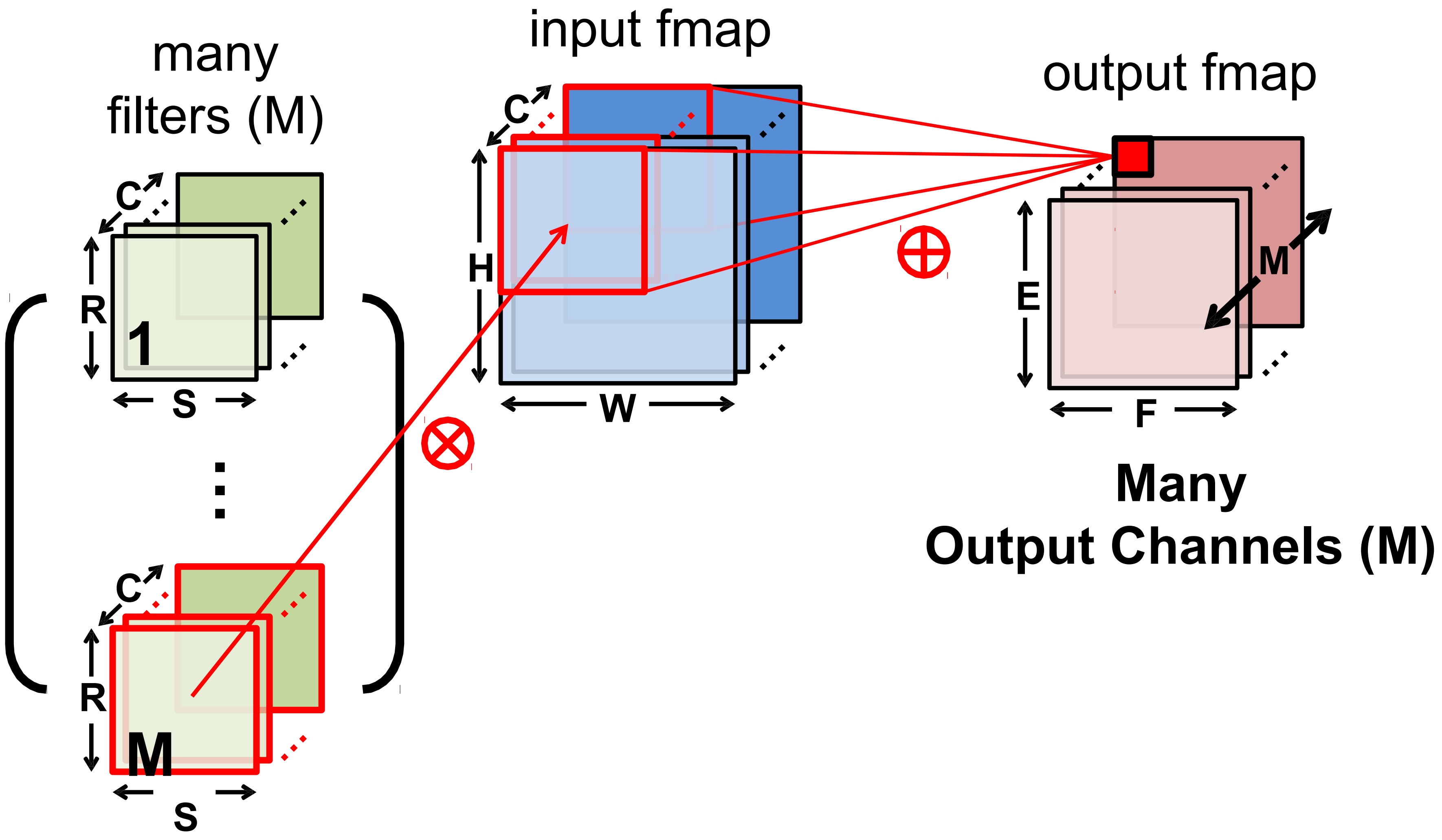
**Sliding Window Processing**

# Convolution (CONV) Layer



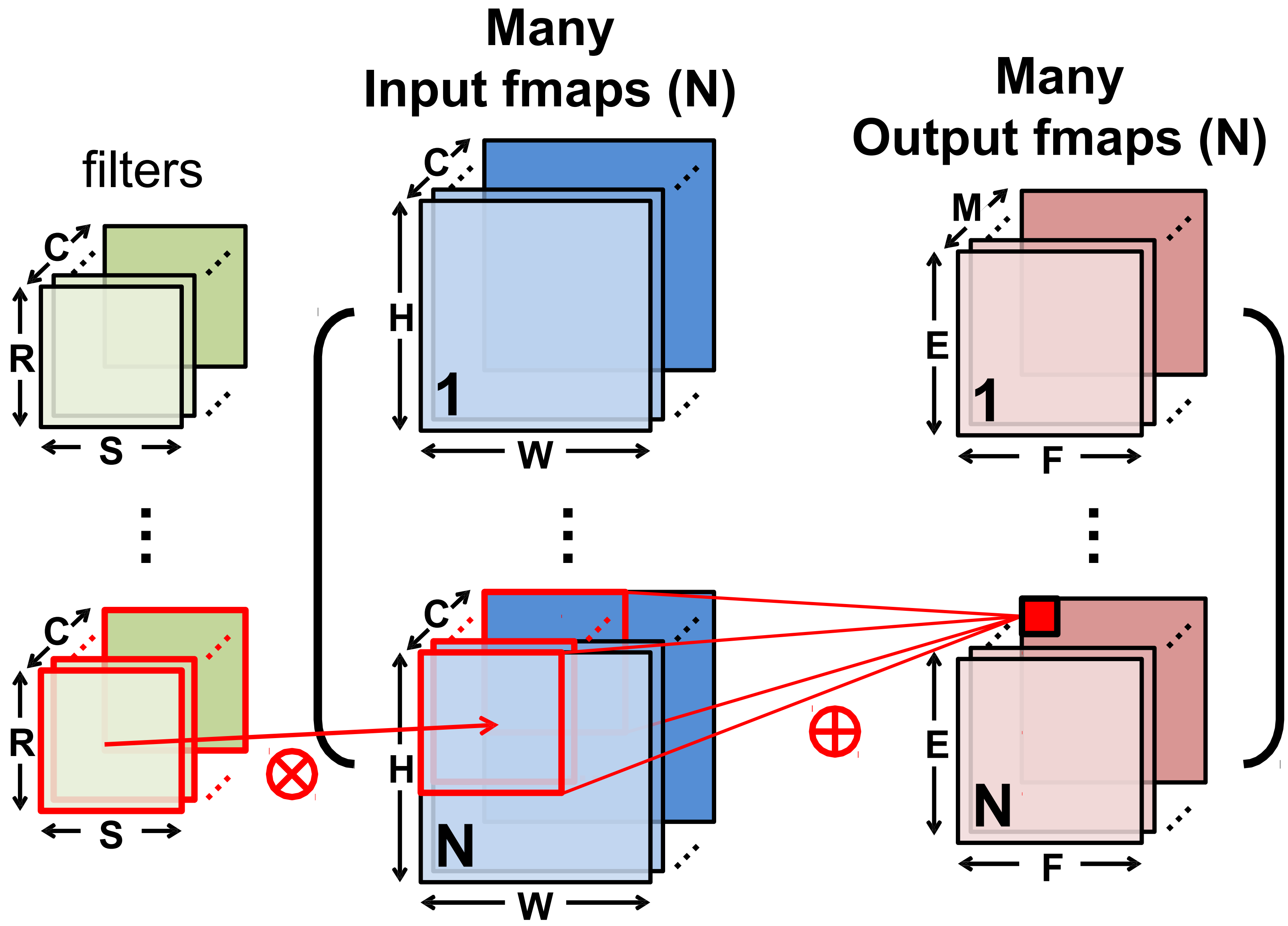
**Many Input Channels (C)**

# Convolution (CONV) Layer



**Many Output Channels (M)**

# Convolution (CONV) Layer



# CNN Decoder Ring

- N – Number of input fmaps/output fmaps (batch size)
- C – Number of 2-D input fmaps /filters (channels)
- H – Height of input fmap (activations)
- W – Width of input fmap (activations)
- R – Height of 2-D filter (weights)
- S – Width of 2-D filter (weights)
- M – Number of 2-D output fmaps (channels)
- E – Height of output fmap (activations)
- F – Width of output fmap (activations)

# CONV Layer Tensor Computation

Output fmaps (O)

Biases (B)

Input fmaps (I)

Filter weights (W)

$$\underline{\mathbf{O}[n][m][x][y]} = \text{Activation}(\underline{\mathbf{B}[m]} + \sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{k=0}^{C-1}\underline{\mathbf{I}[n][k][Ux+i][Uy+j]} \times \underline{\mathbf{W}[m][k][i][j]}),$$

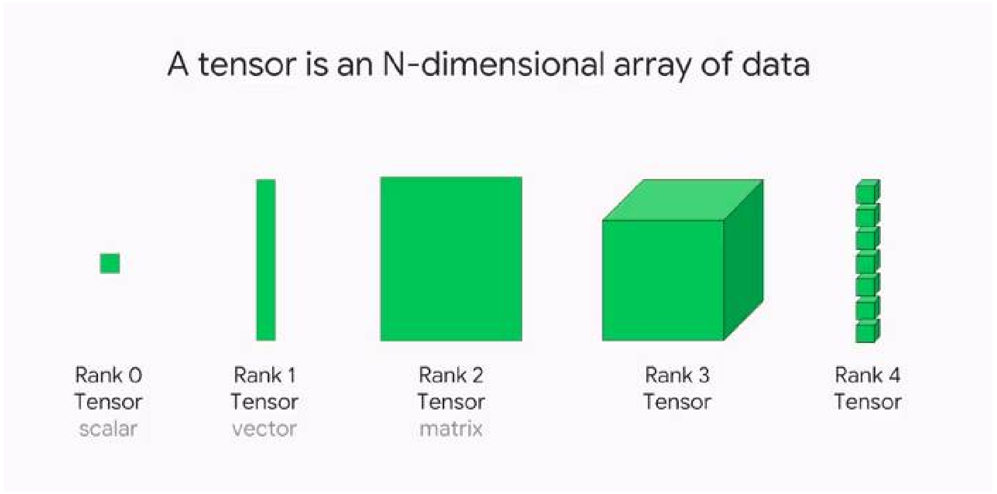
$$0 \le n < N, 0 \le m < M, 0 \le y < E, 0 \le x < F,$$

$$E = (H - R + U)/U, F = (W - S + U)/U.$$

| Shape Parameter | Description |
|---|---|
| $N$ | fmap batch size |
| $M$ | # of filters / # of output fmap channels |
| $C$ | # of input fmap/filter channels |
| $H/W$ | input fmap height/width |
| $R/S$ | filter height/width |
| $E/F$ | output fmap height/width |
| $U$ | convolution stride |

## What is Tensor?

- It is a container of Data, which helps to store different dimensions of Data in Neural Networks.
- Google’s Machine Learning Library TensorFlow was named after them.



**Vector Data:** 2D tensors of shape (samples, features)
**Time-Series Data or Sequence Data:**
3D tensors of shape (samples, timesteps,features)
**Images:**
4D tensors of shape (samples, height, width, channels)
**Videos:**

# CONV Layer Tensor Computation

Output fmaps (O)

Biases (B)

Input fmaps (I)

Filter weights (W)

$$\underline{\mathbf{O}[n][m][x][y]} = \text{Activation}(\underline{\mathbf{B}[m]} + \sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{k=0}^{C-1} \underline{\mathbf{I}[n][k][Ux+i][Uy+j]} \times \underline{\mathbf{W}[m][k][i][j]}),$$

$$0 \le n < N, 0 \le m < M, 0 \le y < E, 0 \le x < F,$$

$$E = (H - R + U)/U, F = (W - S + U)/U.$$

| Shape Parameter | Description |
|---|---|
| $N$ | fmap batch size |
| $M$ | # of filters / # of output fmap channels |
| $C$ | # of input fmap/filter channels |
| $H/W$ | input fmap height/width |
| $R/S$ | filter height/width |
| $E/F$ | output fmap height/width |
| $U$ | convolution stride |

# CONV Layer Implementation

**Naïve 7-layer for-loop implementation:**

```
for (n=0; n<N; n++) {
    for (m=0; m<M; m++) {
        for (x=0; x<F; x++) {
            for (y=0; y<E; y++)
            {
                O[n][m][x][y] = B[m];
                for (i=0; i<R; i++) {
                    for (j=0; j<S; j++) {
                        for (k=0; k<C; k++) {
                            O[n][m][x][y] += I[n][k][Ux+i][Uy+j] × W[m][k][i]
                            [j];
                        }
                    }
                }
                O[n][m][x][y] = Activation(O[n][m][x]
                [y]);
            }
        }
    }
}
```

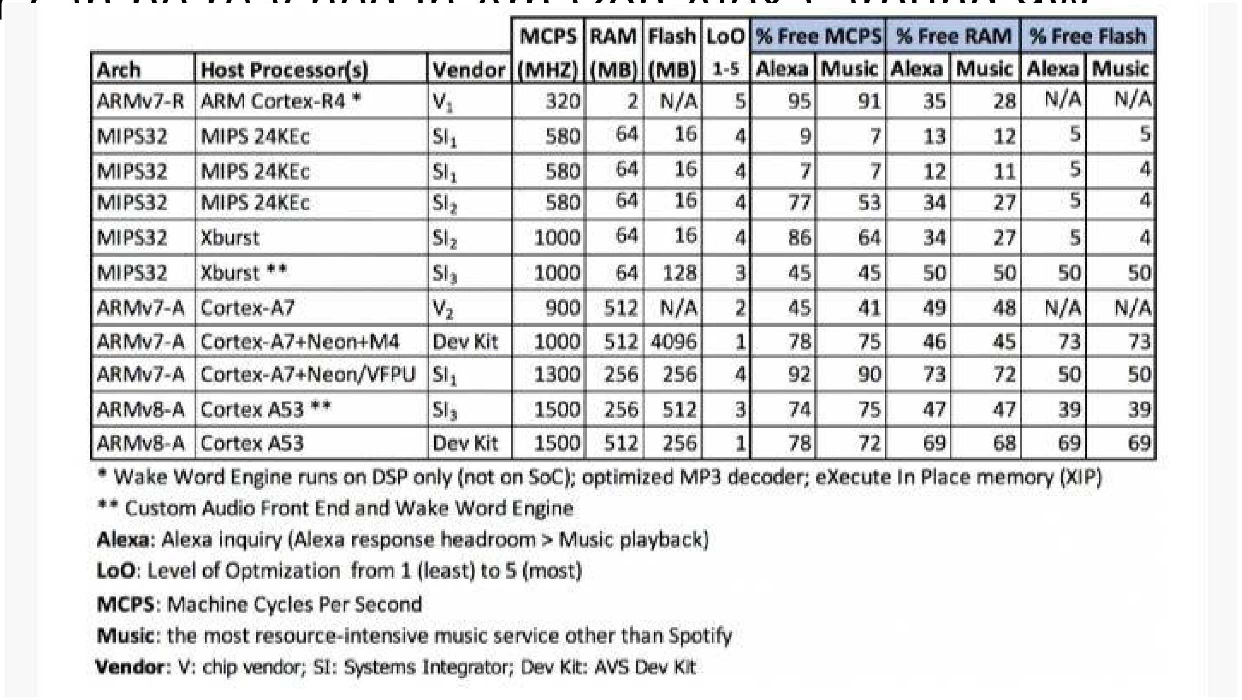
for each output fmap value

convolve a window and apply activation

## Embedded Scenario Facts

The Amazon Echo hardware complement includes a Texas Instruments DM3725 ARM Cortex-A8 processor, **256MB of LPDDR1 RAM** and 4GB of storage space.

Maximum power can be reached in Amazon Alexa around 9W



| | | | MCPS | RAM | Flash | LoO | % Free MCPS | | % Free RAM | | % Free Flash | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Arch** | **Host Processor(s)** | **Vendor** | **(MHZ)** | **(MB)** | **(MB)** | **1-5** | **Alexa** | **Music** | **Alexa** | **Music** | **Alexa** | **Music** |
| ARMv7-R | ARM Cortex-R4 * | $V_1$ | 320 | 2 | N/A | 5 | 95 | 91 | 35 | 28 | N/A | N/A |
| MIPS32 | MIPS 24KEc | $SI_1$ | 580 | 64 | 16 | 4 | 9 | 7 | 13 | 12 | 5 | 5 |
| MIPS32 | MIPS 24KEc | $SI_1$ | 580 | 64 | 16 | 4 | 7 | 7 | 12 | 11 | 5 | 4 |
| MIPS32 | MIPS 24KEc | $SI_2$ | 580 | 64 | 16 | 4 | 77 | 53 | 34 | 27 | 5 | 4 |
| MIPS32 | Xburst | $SI_2$ | 1000 | 64 | 16 | 4 | 86 | 64 | 34 | 27 | 5 | 4 |
| MIPS32 | Xburst ** | $SI_3$ | 1000 | 64 | 128 | 3 | 45 | 45 | 50 | 50 | 50 | 50 |
| ARMv7-A | Cortex-A7 | $V_2$ | 900 | 512 | N/A | 2 | 45 | 41 | 49 | 48 | N/A | N/A |
| ARMv7-A | Cortex-A7+Neon+M4 | Dev Kit | 1000 | 512 | 4096 | 1 | 78 | 75 | 46 | 45 | 73 | 73 |
| ARMv7-A | Cortex-A7+Neon/VFPU | $SI_1$ | 1300 | 256 | 256 | 4 | 92 | 90 | 73 | 72 | 50 | 50 |
| ARMv8-A | Cortex A53 ** | $SI_3$ | 1500 | 256 | 512 | 3 | 74 | 75 | 47 | 47 | 39 | 39 |
| ARMv8-A | Cortex A53 | Dev Kit | 1500 | 512 | 256 | 1 | 78 | 72 | 69 | 68 | 69 | 69 |

* Wake Word Engine runs on DSP only (not on SoC); optimized MP3 decoder; eXecute In Place memory (XIP)

** Custom Audio Front End and Wake Word Engine

**Alexa**: Alexa inquiry (Alexa response headroom > Music playback)

**LoO**: Level of Optmization from 1 (least) to 5 (most)

**MCPS**: Machine Cycles Per Second

**Music**: the most resource-intensive music service other than Spotify

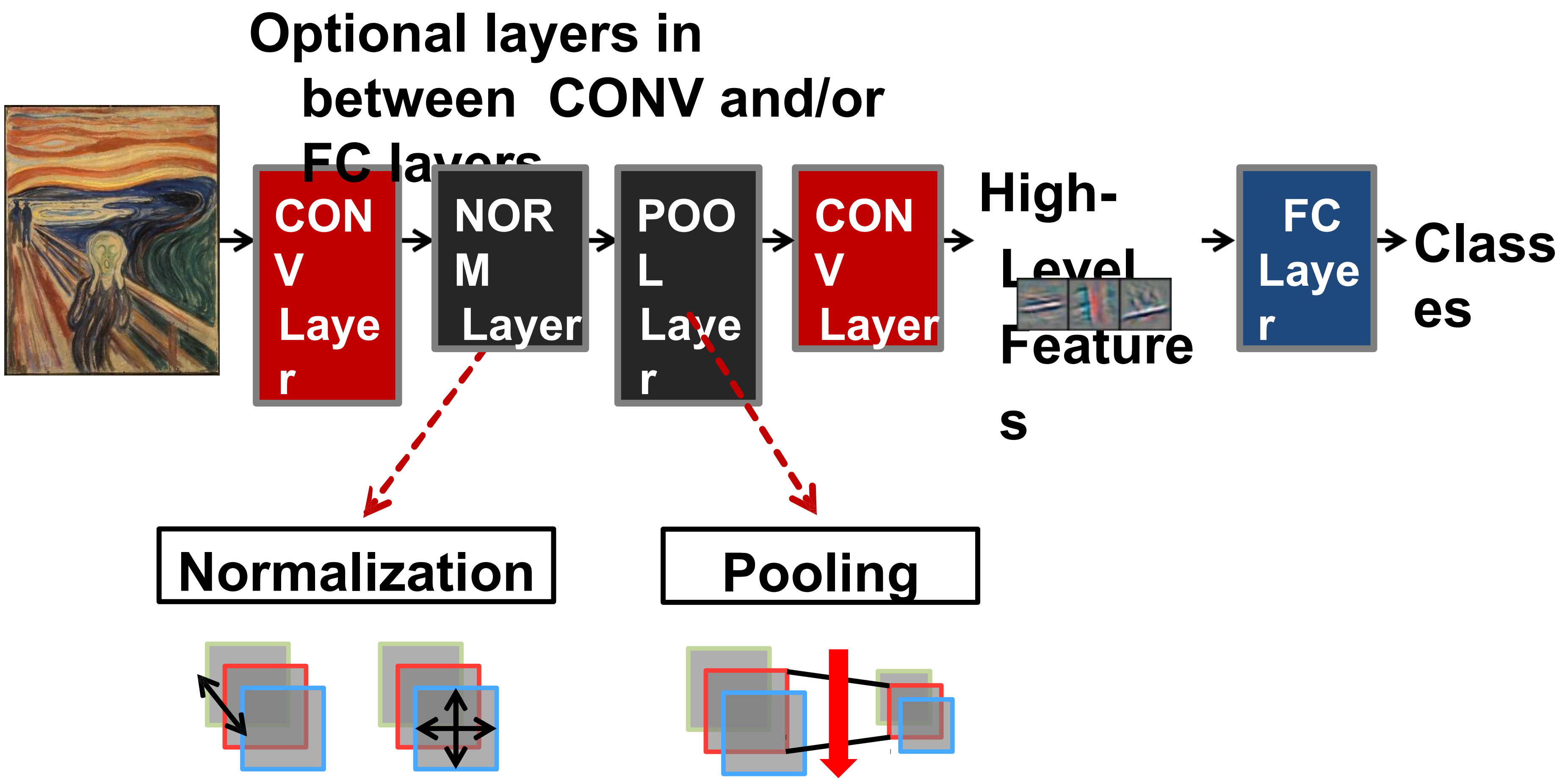
**Vendor**: V: chip vendor; SI: Systems Integrator; Dev Kit: AVS Dev Kit

https://developer.amazon.com/blogs/alexa/post/2a32d792-d471-4136-8262-7972/cpu-memory-and-storage-for-alexa-built-in-devices

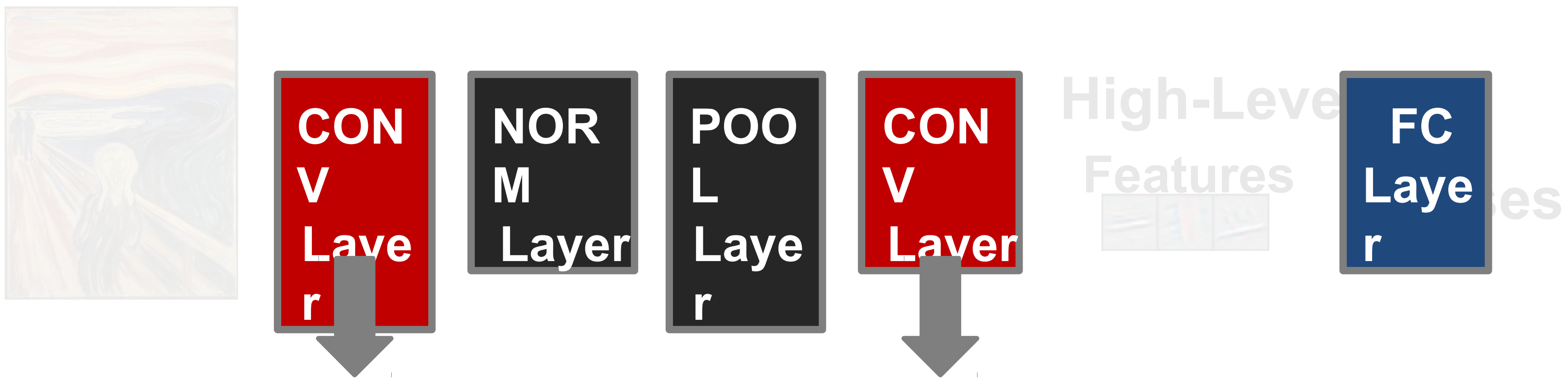
# Try The Following

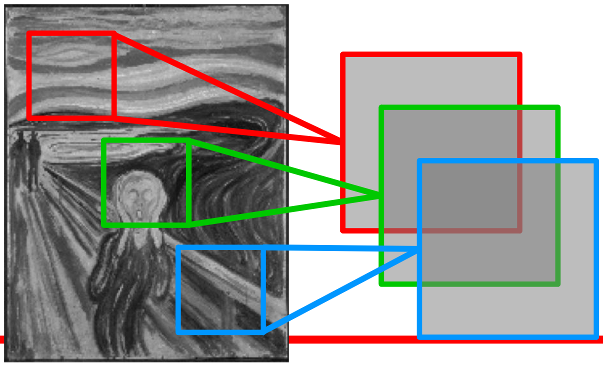
- TensorFlow is software framework to train and classify machine learning tasks. Tensorflow supports different APIs for Machine Learning based tasks.
- It supports wide range of classification tasks: texts, images, videos, etc.
- Open the tutorial: [https://www.tensorflow.org/tutorials/images/classification](https://www.tensorflow.org/tutorials/images/classification)
- There are three options available:
  - Run in google Colab
  - View source on Github
  - Download the notebook
- To check without any local setup choose first option: Run in google colab, you need to login with your google account.
- The notebook explains how an artificial neural network trains on a

# Deep Convolutional Neural Networks



# Deep Convolutional Neural Networks

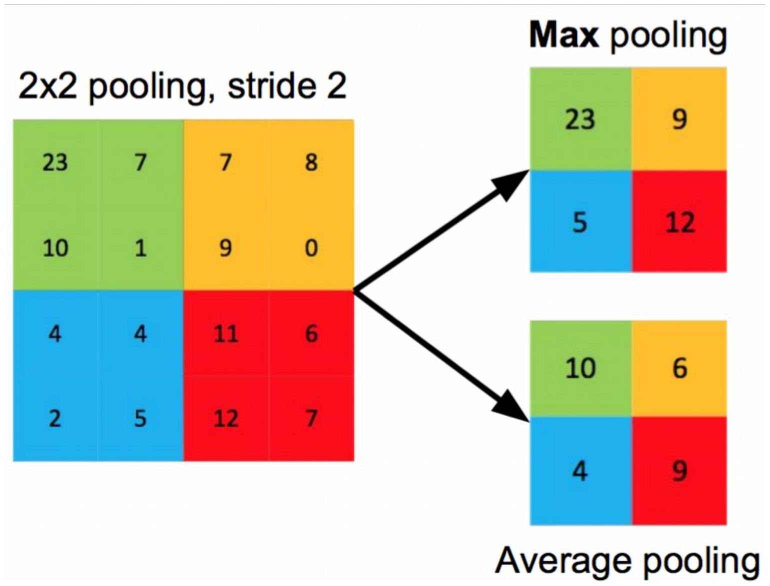




**Convolutions** account for more than 90% of overall computation, dominating **runtime** and **energy consumption**

# Pooling (POOL) Layer

- Pooling layers are **used to reduce the dimensions of the feature maps**
- Reduce resolution of each channel independently
- Overlapping or non-overlapping  depending on stride



Increases translation-invariance and noise-resilience

Image Source: Caffe Tutorial

## Pooling

- A limitation of the feature map output of convolutional layers is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a different feature map. This can happen with re-cropping, rotation, shifting, and other minor changes to the input image.

- A common approach to addressing this problem from signal processing is called down sampling. This is where a lower resolution version of an input signal is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task.
- Down sampling can be achieved with convolutional layers by changing the stride of the convolution across the image.

- A more robust and common approach is to use a pooling layer.

- **Translation invariance** means that the system produces exactly the same response, regardless of how its input is shifted.

# POOL Layer Implementation

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

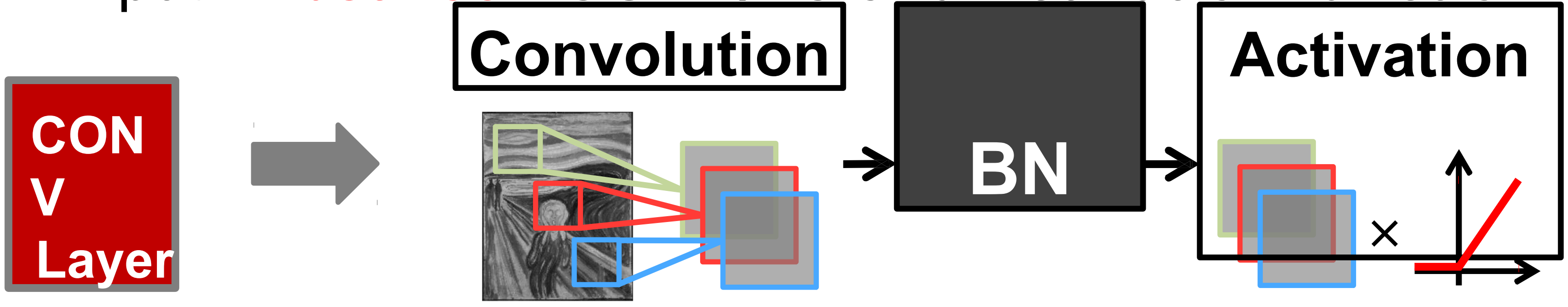
```
for (n=0; n<N; n++) {
    for (m=0; m<M; m++) {
        for (x=0; x<F; x++) {
            for (y=0; y<E; y++)
            {
                max = -Inf;
                for (i=0; i<R; i++) {
                    for (j=0; j<S; j++) {
                        if (I[n][m][Ux+i][Uy+j] >
                        max) {
                            max = I[n][m][Ux+i]
                            [Uy+j];
                        }
                    }
                }
                O[n][m][x][y] =
                max;
            }
        }
    }
}
```

for each pooled value

find the max
with in a window

# 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 et al., ICML 2015]

# BN Layer Implementation

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

data mean

learned scale factor

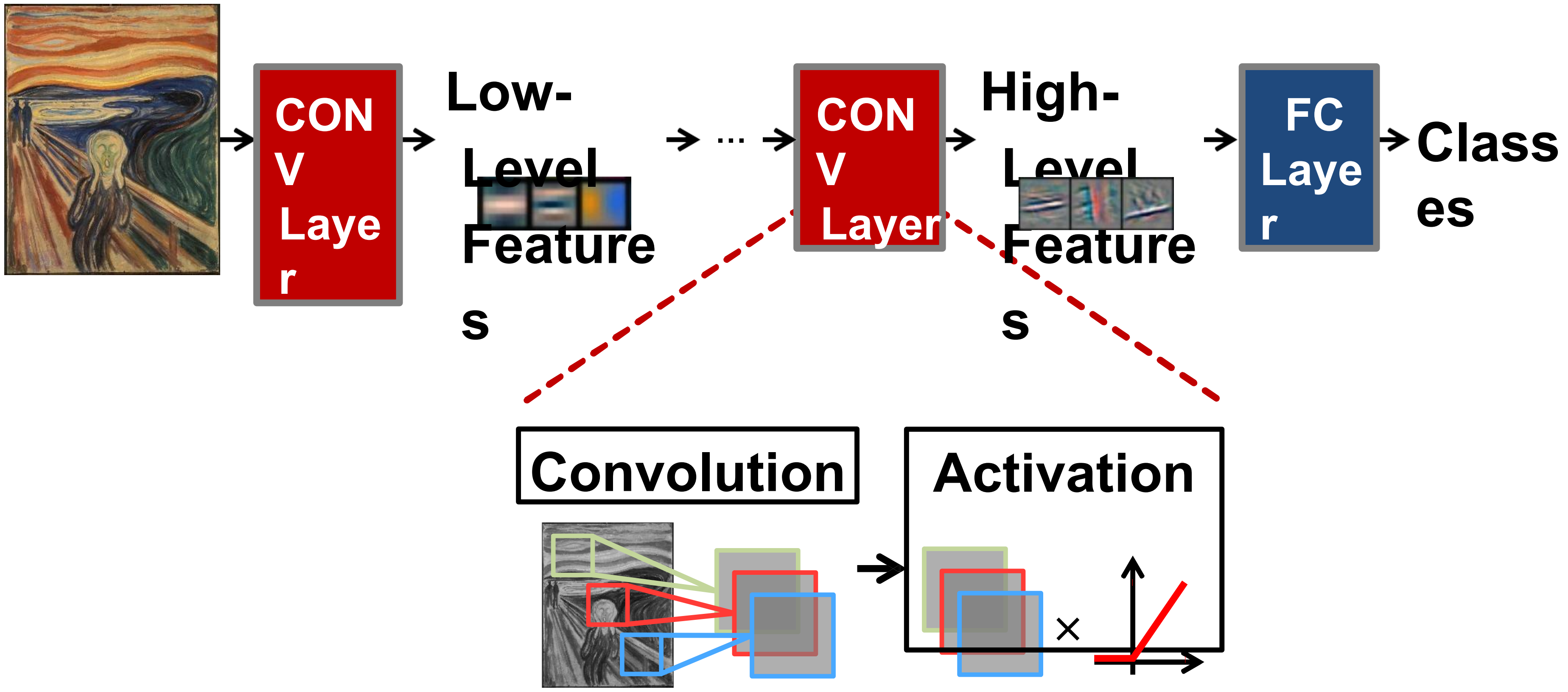
$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta$$

learned shift factor

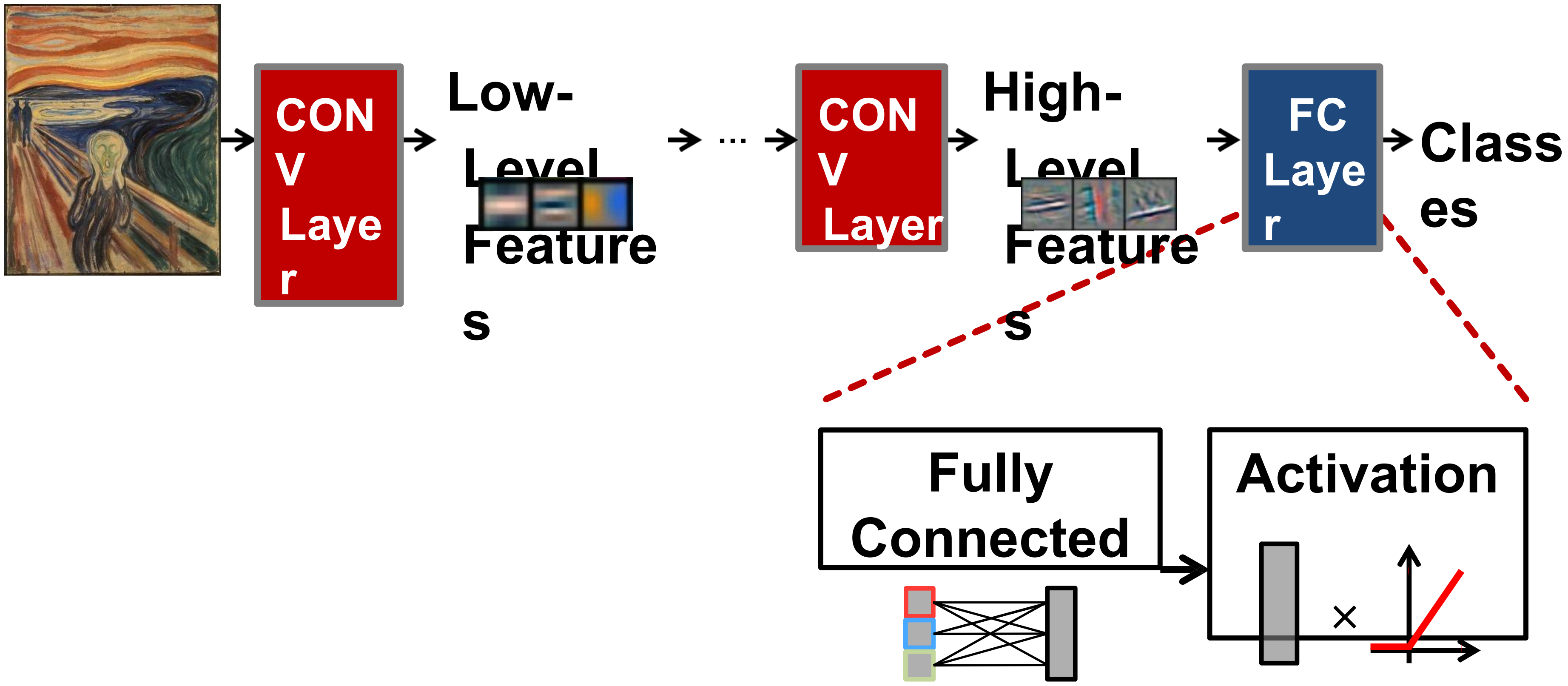
data std. dev.

small const. to avoid numerical problems

# Deep Convolutional Neural Networks

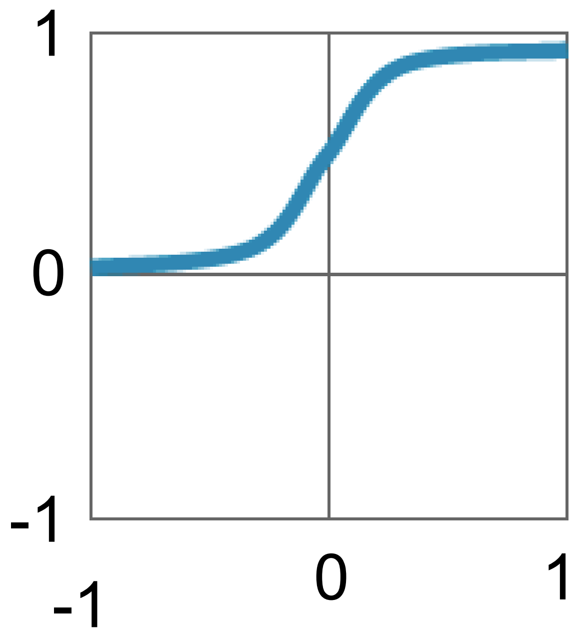


# Deep Convolutional Neural Networks



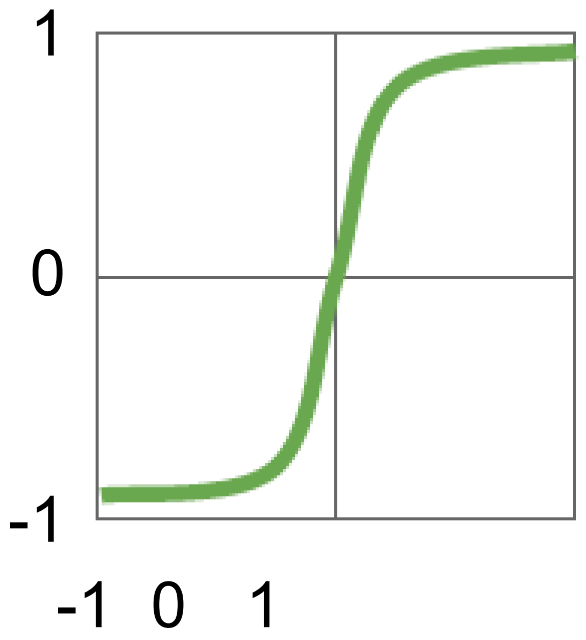
# Traditional Activation Functions

## Sigmoid



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

## Hyperbolic Tangent

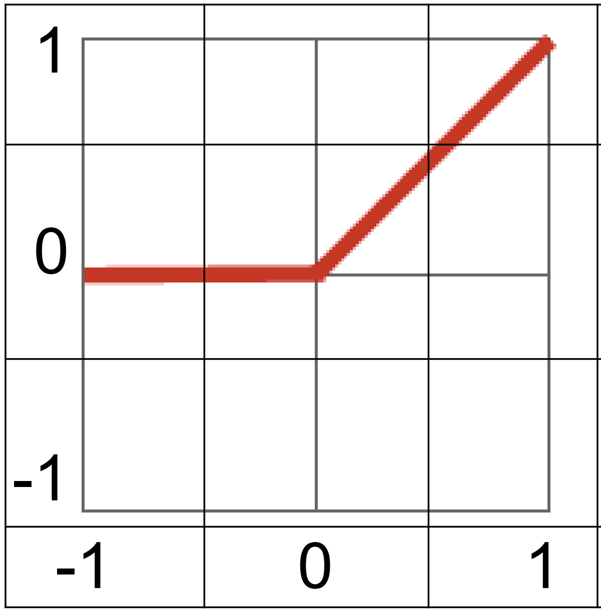


$y=(e^{x}-e^{-x})/(e^{x}+e^{-x})$

Image Source: Caffe Tutorial

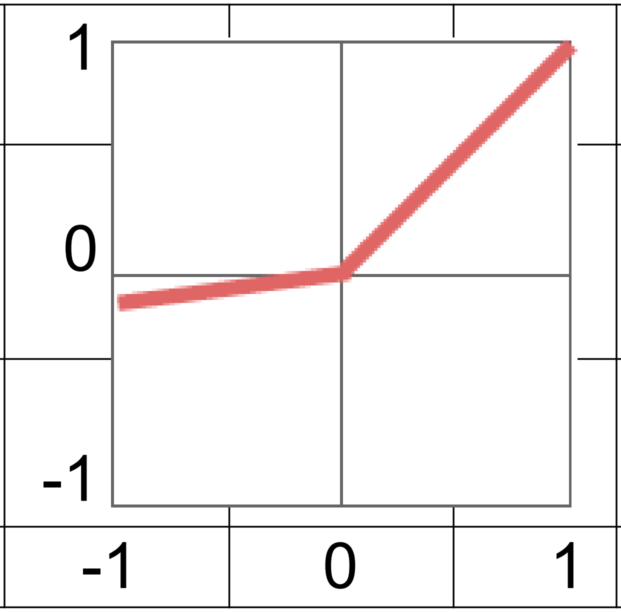
# Modern Activation Functions

**Rectified Linear Unit (ReLU)**



$y=\max(0,x)$

**Leaky ReLU**

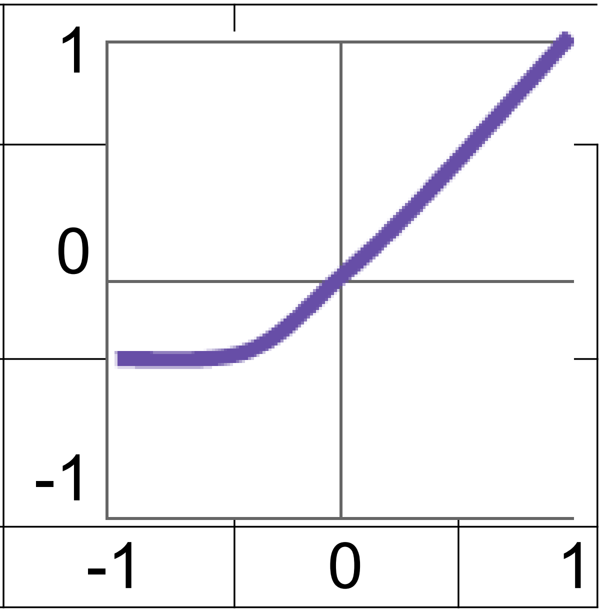


$y=\max(\alpha x,x)$

y=

α = small const. (e.g. 0.1)

**Exponential LU**



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

Image Source: Caffe Tutorial

## Activation Functions

- The choice of activation function in the hidden layer will control how well the network model learns the training dataset. The choice of activation function in the output layer will define the type of predictions the model can make.

- Sometimes the activation function is called a “*transfer function*.”
- If the output range of the activation function is limited, then it may be called a “*squashing function*.”
- Many activation functions are nonlinear and may be referred to as the “*nonlinearity*” in the layer or the network design.

- The activation function is used within or after the internal processing of each node in the network, although networks are designed to use the same activation function for all nodes in a layer.

## Activation Functions

- A network may have three types of layers: input layers that take raw input from the domain, **hidden layers** that take input from another layer and pass output to another layer, and **output layers** that make a prediction.
- All hidden layers typically use the same activation function.
- The output layer will typically use a different activation function from the hidden layers and is dependent upon the type of prediction required by the model.
- Activation functions are also typically differentiable, meaning the first-order derivative can be calculated for a given input value. This is required given that neural networks are typically trained using the backpropagation of error algorithm that requires the derivative of prediction error in order to update the weights of the model.

## Activation for Hidden Layers

A hidden layer in a neural network is a layer that receives input from another layer (such as another hidden layer or an input layer) and provides output to another layer (such as another hidden layer or an output layer).
A neural network may have zero or more hidden layers.

Typically, a differentiable nonlinear activation function is used in the hidden layers of a neural network. This allows the model to learn more complex functions than a network trained using a linear activation function.:

- **ReLU Hidden Layer Activation Function**
- It is common because it is both simple to implement and effective at overcoming the limitations of other previously popular activation functions, such as Sigmoid and Tanh.
- it is less susceptible to vanishing gradients that prevent deep models from being trained, although it can suffer from other problems like saturated or "*dead*" units

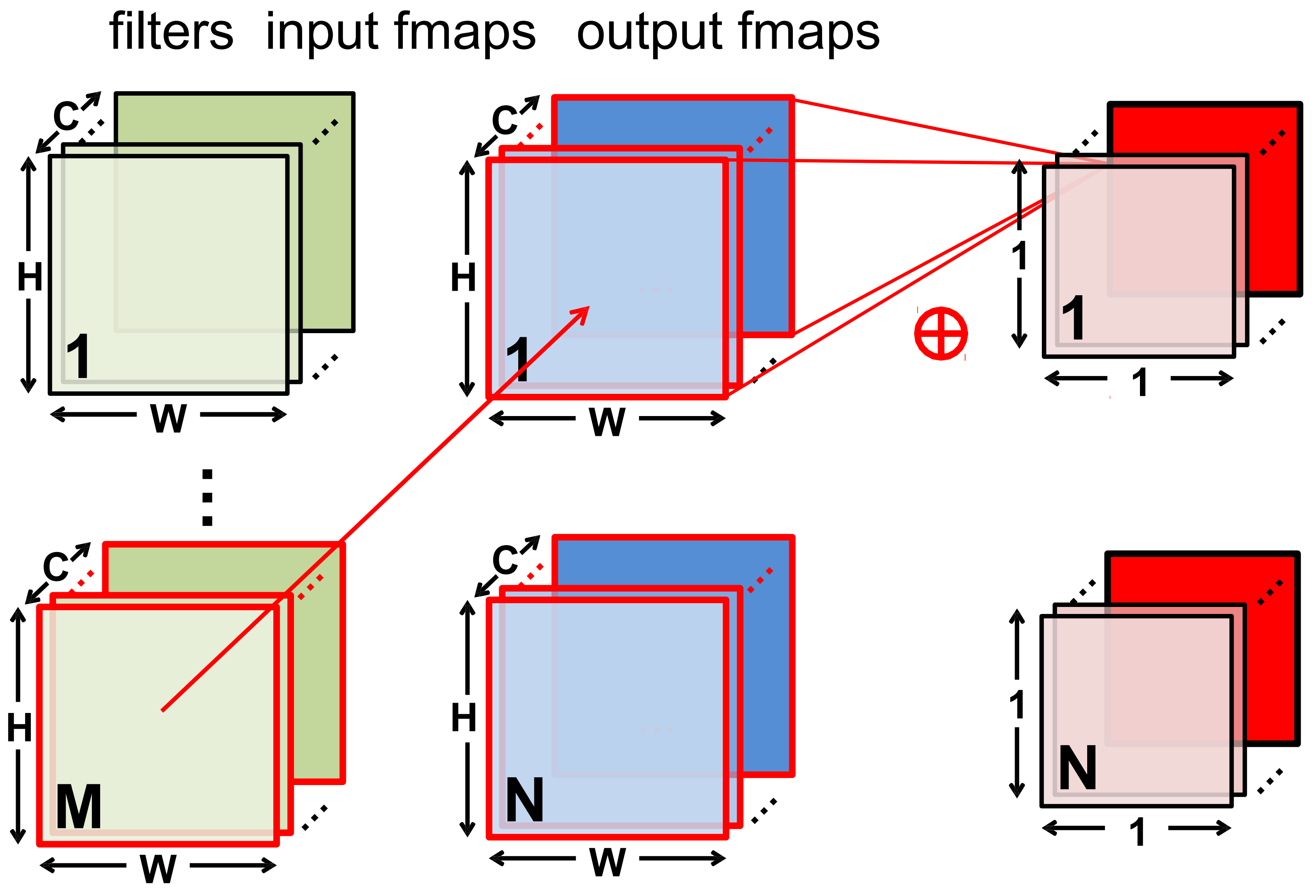
## Output Activation Functions

**Linear Output Activation Function**
The linear activation function is also called "*identity*" (multiplied by 1.0) or "*no activation*."
This is because the linear activation function does not change the weighted sum of the input in any way and instead returns the value directly.

**Sigmoid Output Activation Function**
- Target labels used to train a model with a sigmoid activation function in the output layer will have the values 0 or 1.

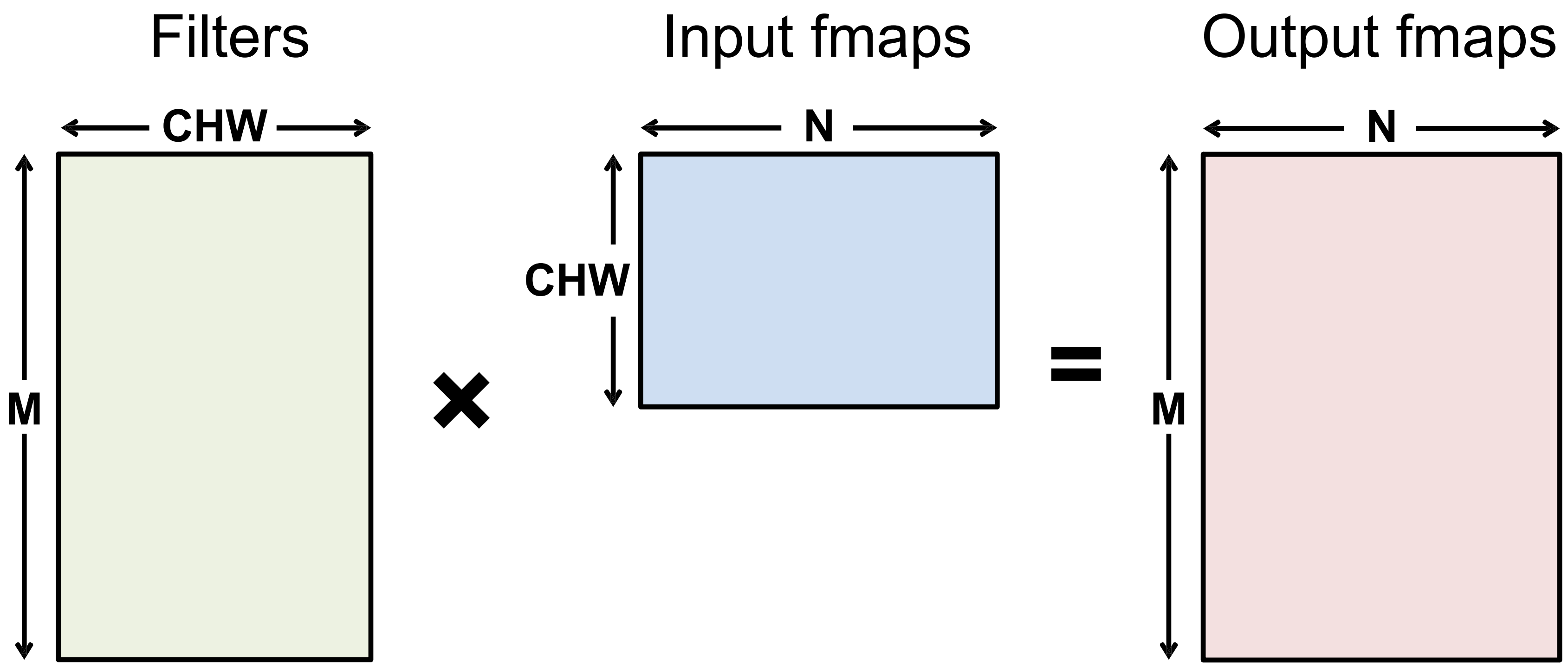
**Softmax Output Activation Function**
The softmax function outputs a vector of values that sum to 1.0 that can be interpreted as probabilities of class membership.
- As such, the input to the function is a vector of real values and the output is a vector of the same length with values that sum to 1.0 like probabilities.

# FC Layer – from CONV Layer POV



# Fully-Connected (FC) Layer

- Height and width of output fmaps are 1 (E = F = 1)
- Filters as large as input fmaps (R = H, S = W)
- Implementation: **Matrix Multiplication**



- Deep architectures often advantageous when dealing with complex learning problems.
- The stacking of multiple linear and non-linear processing units in a layer-wise fashion provides the ability to learn complex representations at different levels of abstraction.
- The availability of big data and advancements in hardware are also the main reasons for the recent success of deep CNNs.