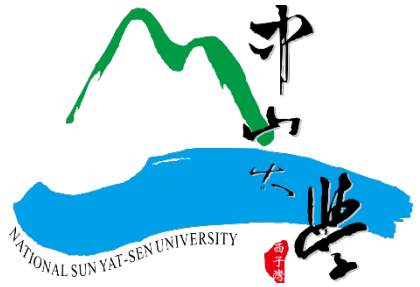
Convolutional Neural Network (CNN)



Outlines

- Artificial Neural Network (ANN)
- Convolutional Neural Network (CNN)
- Training

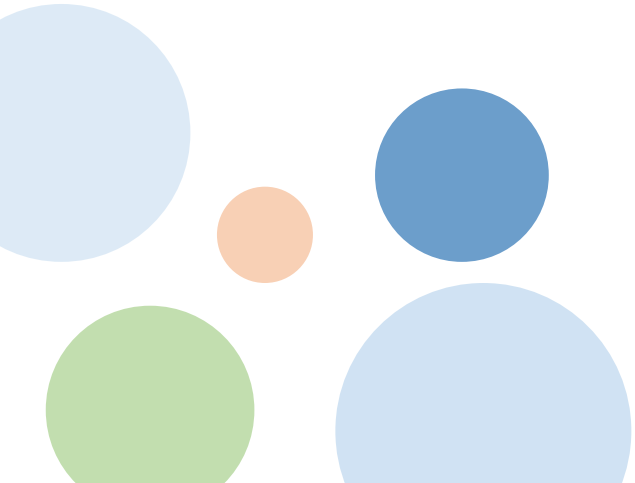
References and Credits

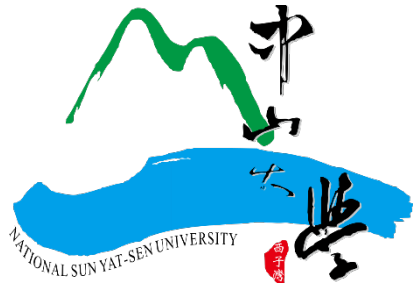
- Stanford CS231n, “Convolutional Neural Networks for Visual Recognition”
  - by Fei-Fei Li, Justin Johnson, and Serena Yeung
- MIT 6.S191, “Deep Learning”
  - by Alexander Amini, and Ava Soleimany
- UVA Deep Learning , Univ. of Amsterdam
  - by Efstratios Gavves
- CMSC 35264 Deep Learning, Univ. of Chicago
  - by Shubhendu Trivedi and Risi Kondor
- Deep Learning for Computer Vision
  - by 台大資工系 莊永裕 教授
- book: Deep Learning
  - by Ian Goodfellow, Yoshua Bengio, and Aaron Courville

References and Credits

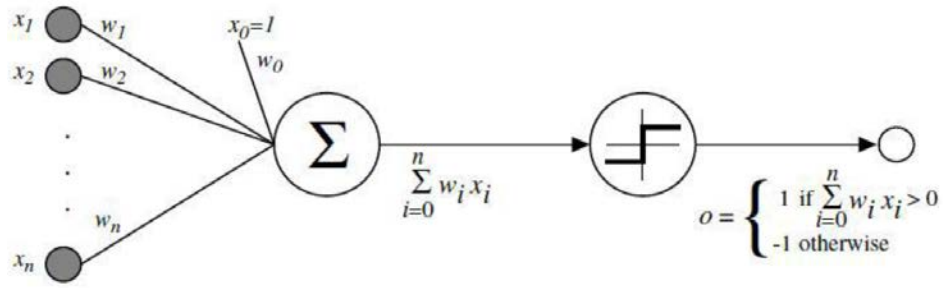
- Stanford CS231n, “Convolutional Neural Networks for Visual Recognition”
  - by Fei-Fei Li, Justin Johnson, and Serena Yeung
- MIT 6.S191, “Deep Learning”
  - by Alexander Amini, and Ava Soleimany
- UVA Deep Learning , Univ. of Amsterdam
  - by Efstratios Gavves
- CMSC 35264 Deep Learning, Univ. of Chicago
  - by Shubhendu Trivedi and Risi Kondor
- Deep Learning for Computer Vision
  - by 台大資工系 莊永裕 教授
- book: Deep Learning
  - by Ian Goodfellow, Yoshua Bengio, and Aaron Courville

Artificial Neural Networks (ANN)

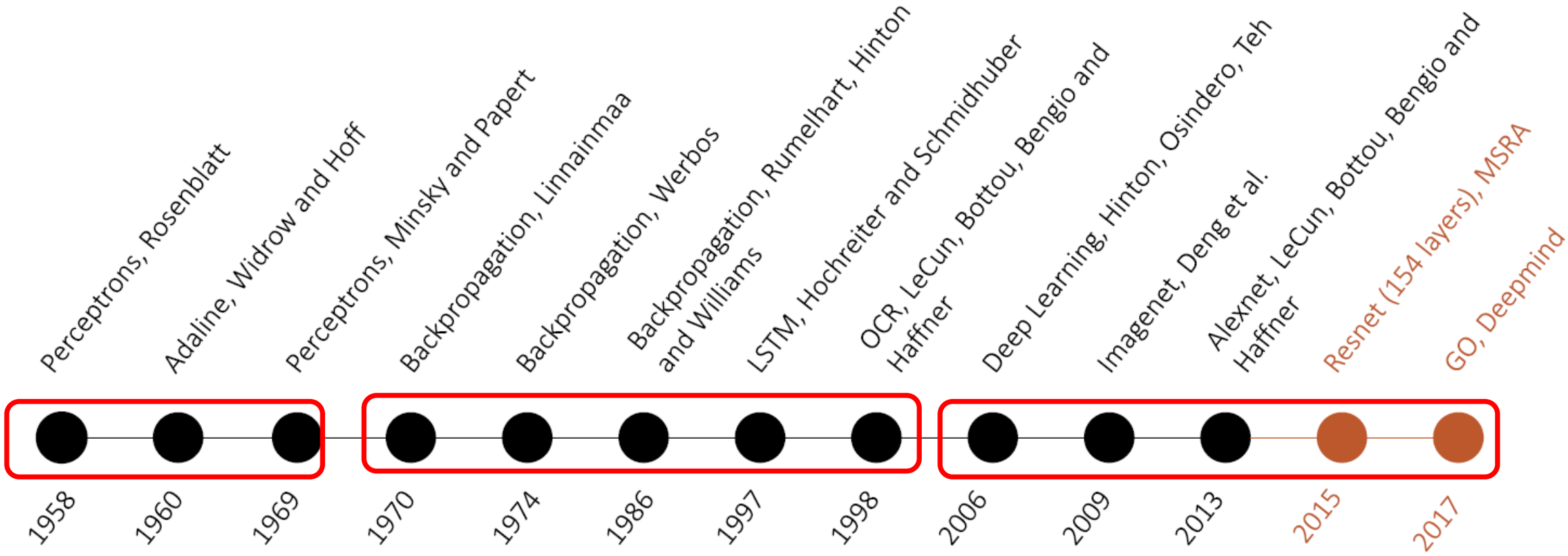




Evolution of AI

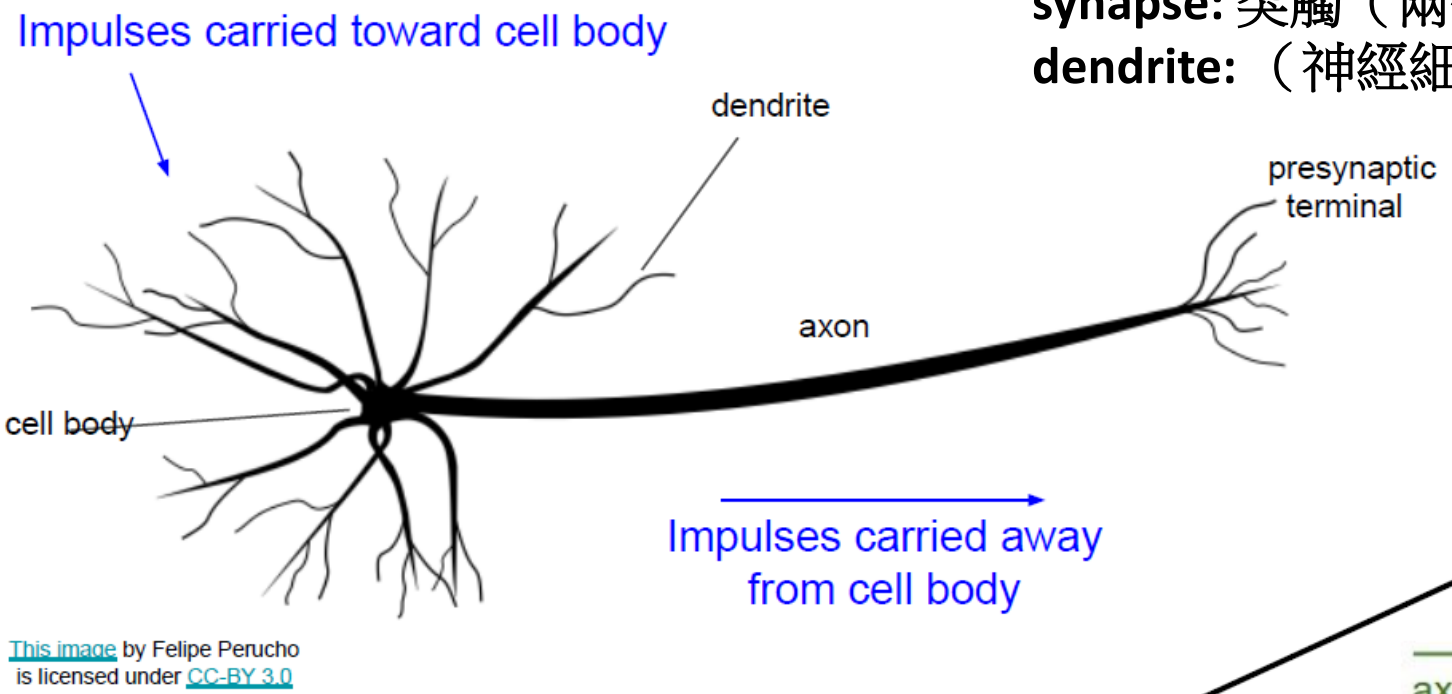


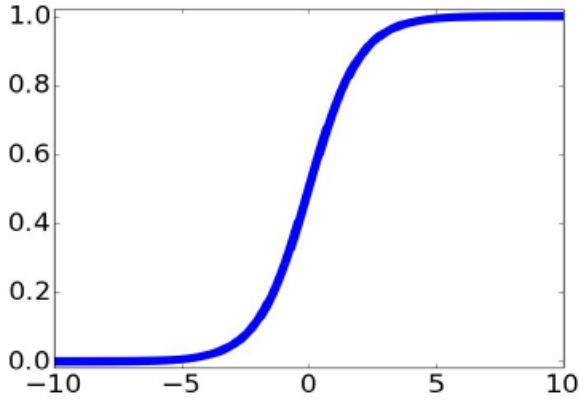
- first appeared in 1960s: perception for binary decision
- 1970~2000: backpropagation, recurrent nets with few layers
- 2005~: deep learning with many layers



Neuron Biology

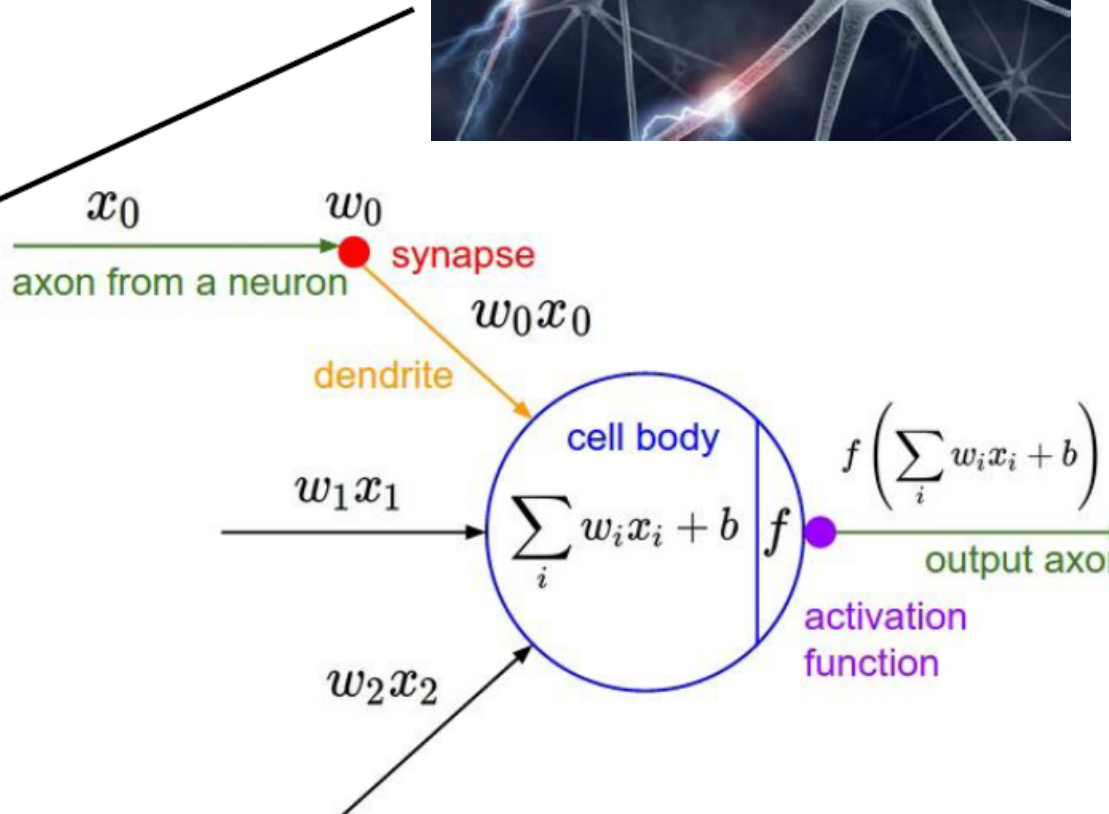
axon: 神經細胞之）軸索
synapse: 突觸（兩個神經原的相接處）
dendrite: （神經細胞的）樹狀突





sigmoid activation function

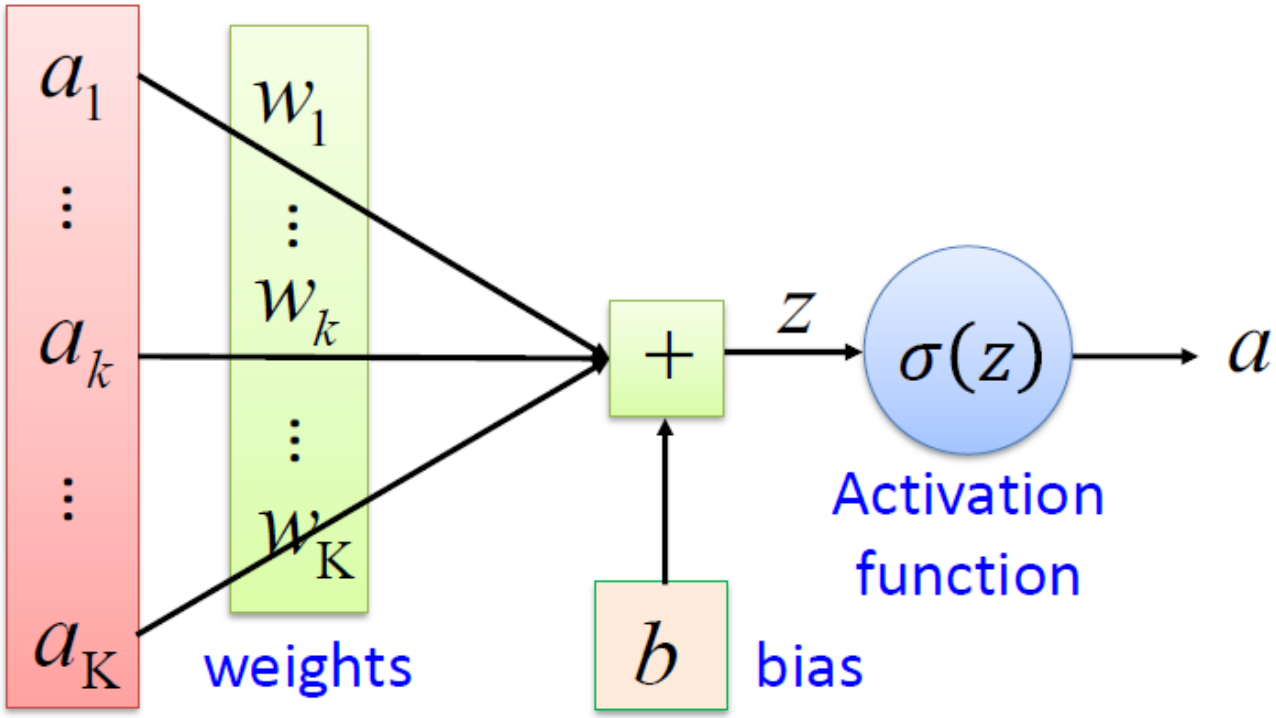
$$\frac{1}{1+e^{-x}}$$



Model for Artificial Neuron

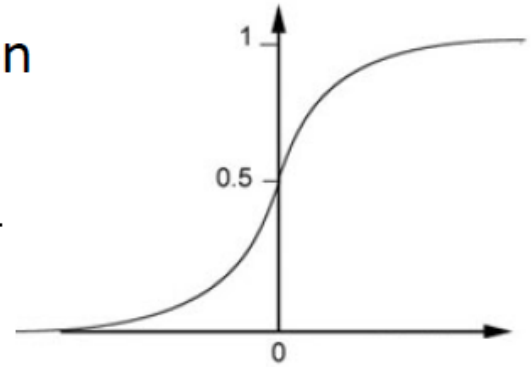
- weighted sum of inputs followed by non-linear activation function

$$z = a_1 w_1 + \cdots + a_k w_k + \cdots + a_K w_K + b$$



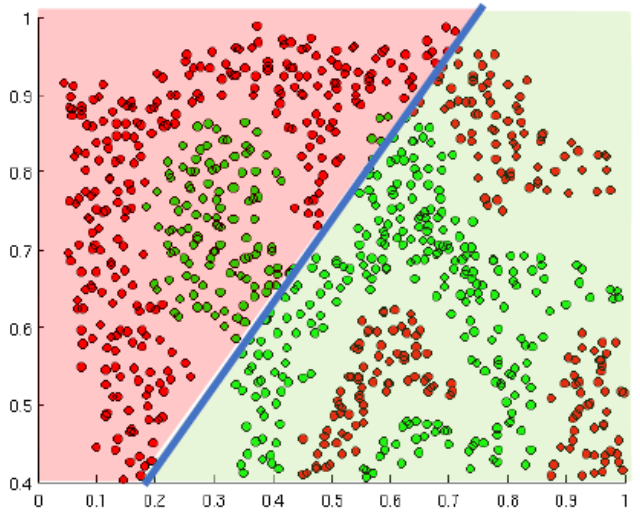
Sigmoid Function

$$\sigma(z) = \frac{1}{1+e^{-z}}$$



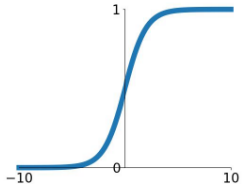
Non-linear Activation Functions

- introduce non-linearity into networks
  - for classification purpose
    - non-linearity approximates complex functions
- ANN usually uses tanh or sigmoid
- CNN usually uses ReLU (Rectified Linear Unit)
- GAN encoder usually uses leaky ReLU



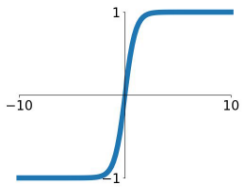
Sigmoid

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



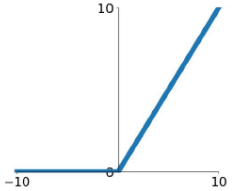
tanh

$$\tanh(x)$$



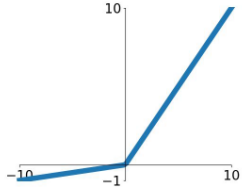
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

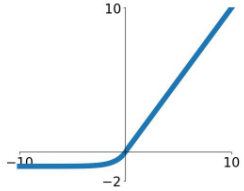


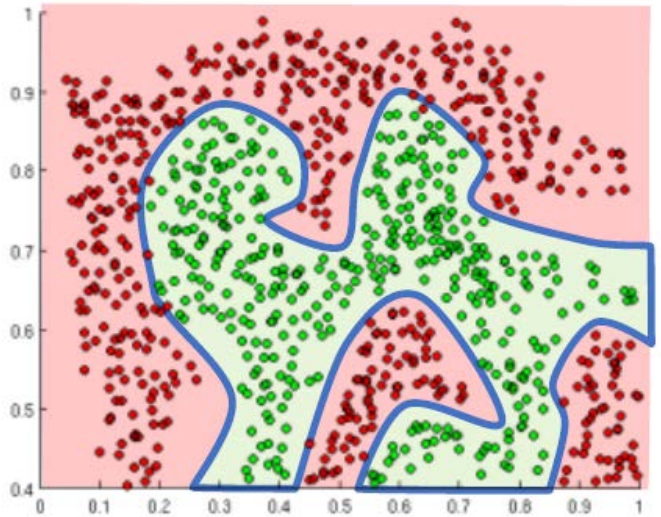
Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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

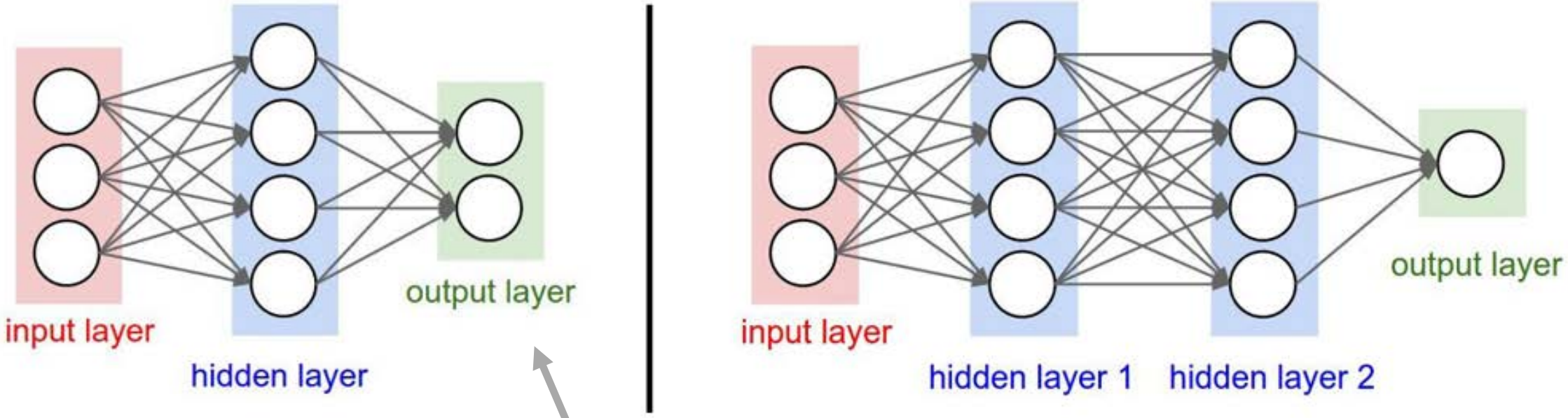




Artificial Neural Network (ANN)

- also called Multi-Layer Perception (MLP) , or
- Fully Connected (FC) layers in CNN models

total # of weights: 3x4+4x4+4x1=32
total # of operations (MAC): 3x4+4x4+4x1=32



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

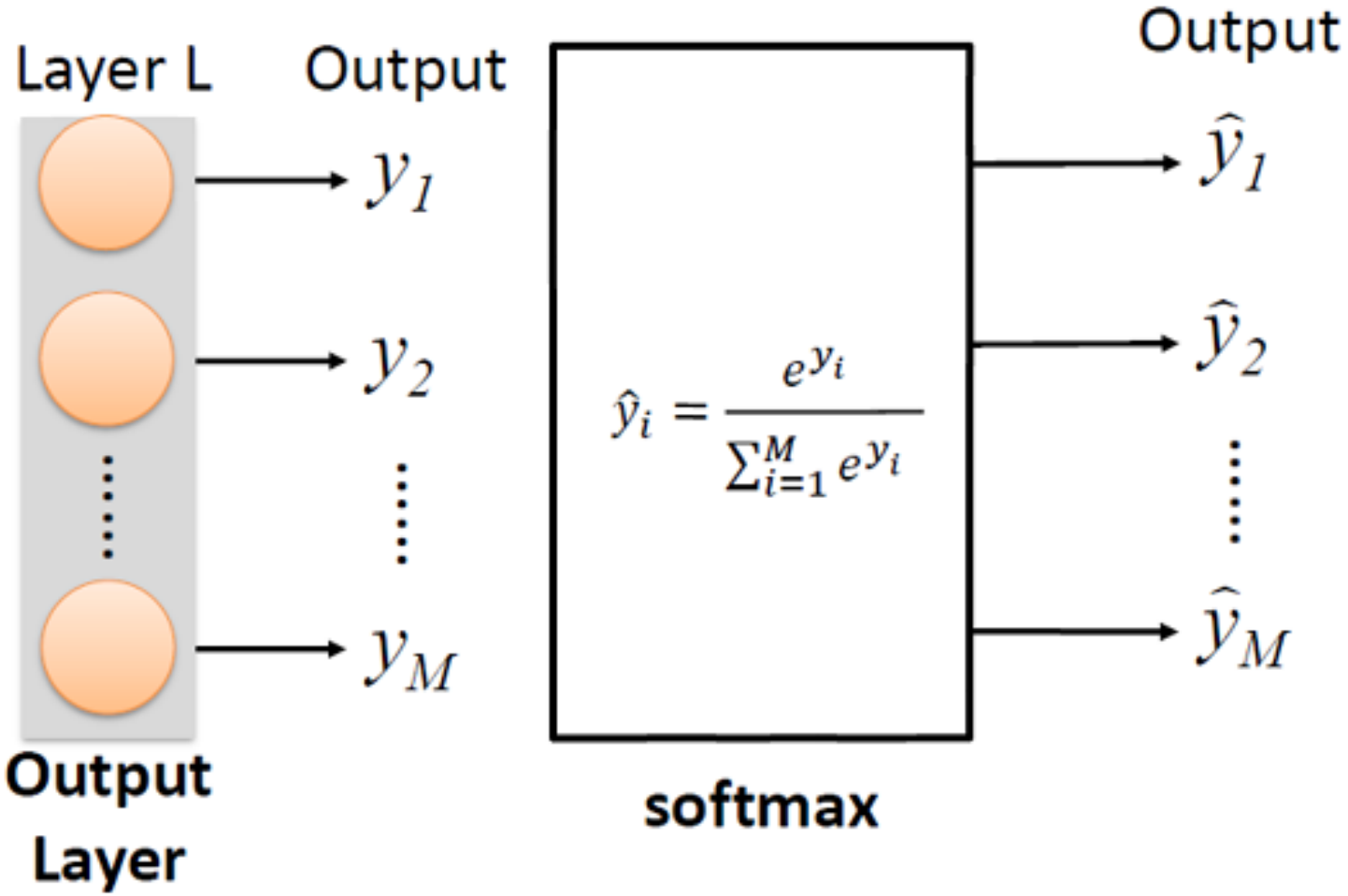
total # of weights: 3x4+4x2=20
total # of operations (MAC): 3x4+4x2=20

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

"Fully-connected" layers

Softmax

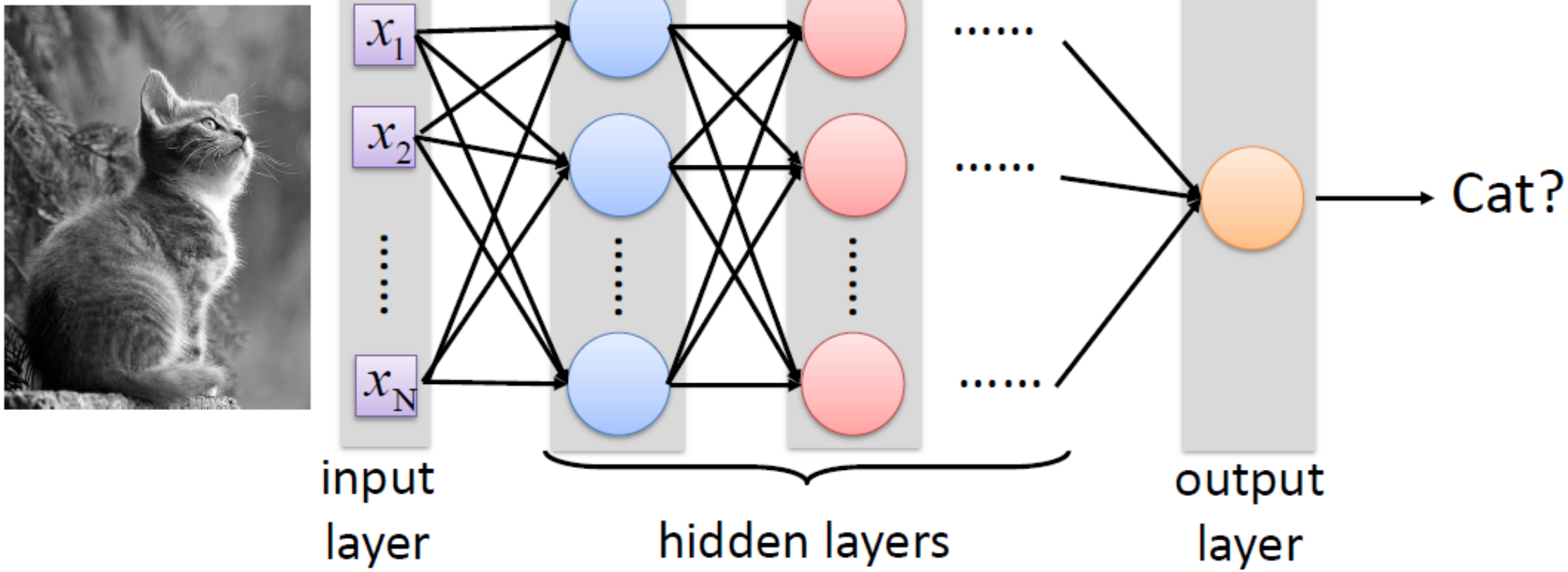
- append at the output layer to generate probability in multi-class classification



- $\hat{y}_i$ represents probability of the i-th class

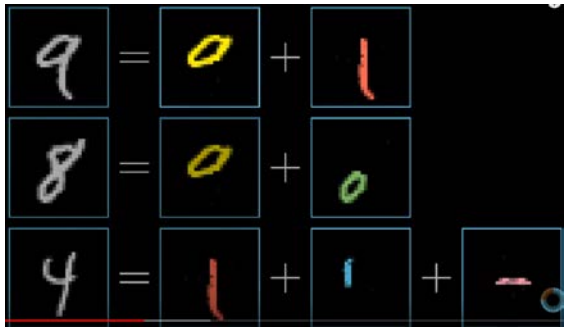
Image Classification with ANN

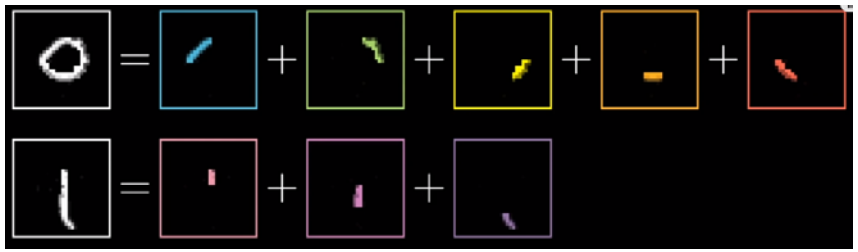
- deep means many hidden layers in ANN

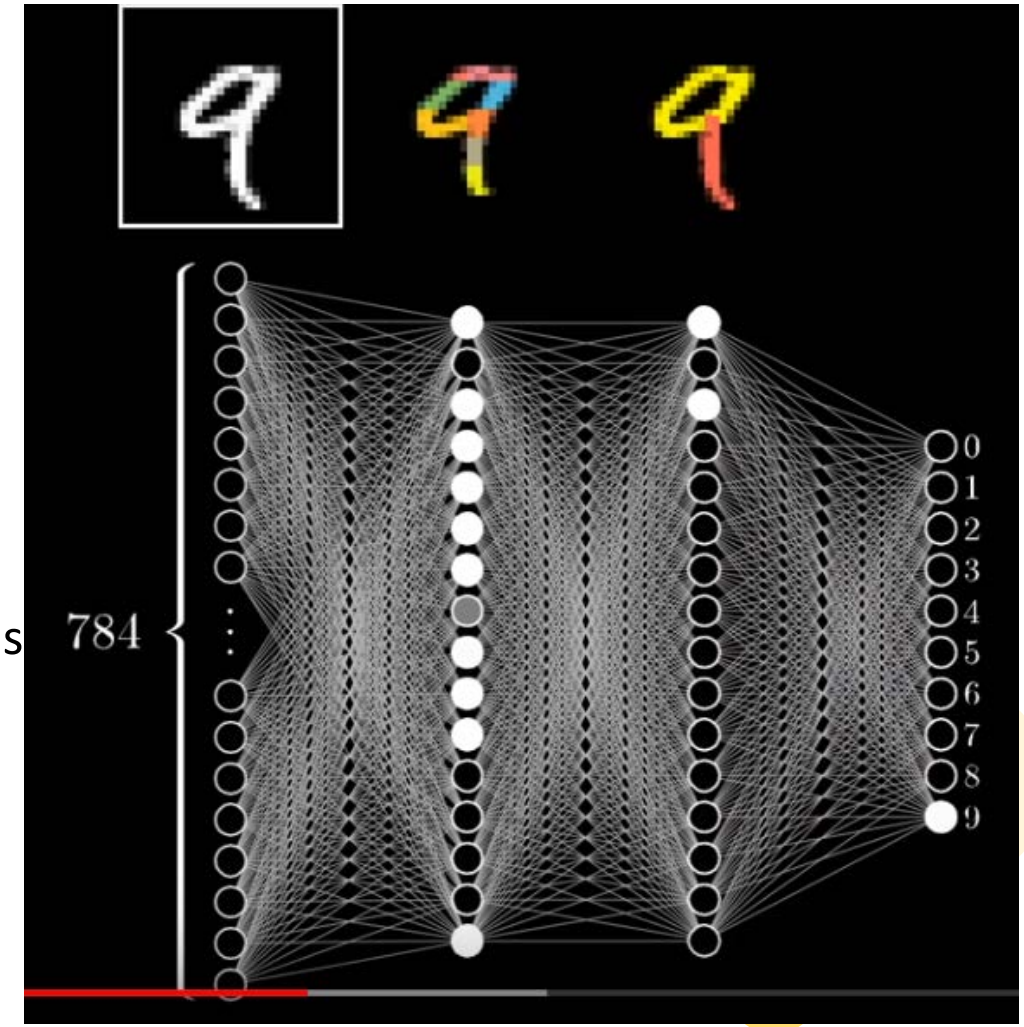


Digit Recognition Using ANN

- https://www.youtube.com/watch?time_continue=12&v=aircAruvnKk
- how ANN recognizes digits
  - 28x28=784 pixels (gray level)
- 5:50 ~ 7:00
  - digit decomposed into features
    - loops, edges, …
  - low-level features in 1st hidden layer
    - edges
  - mid-level features in 2nd hidden layer
    - loops constructed from edges of different orientations



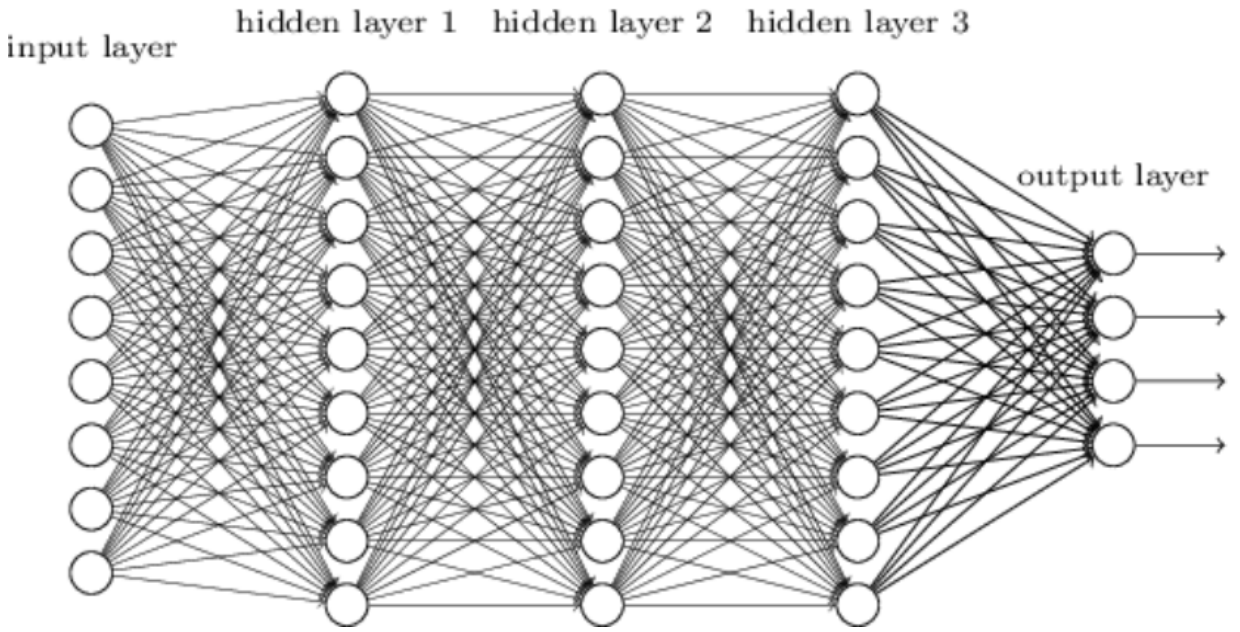




Fully-Connected ANN

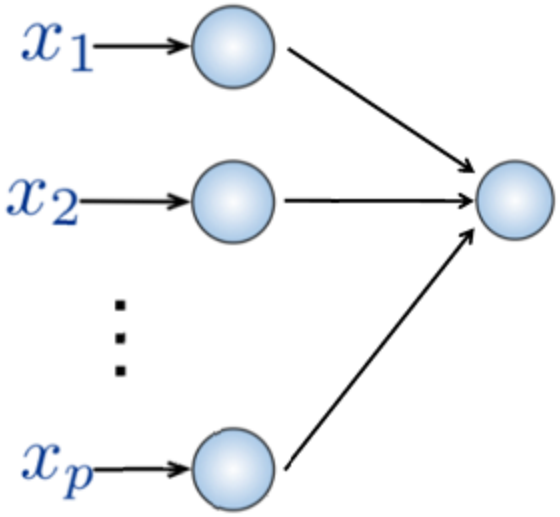
- Full connections between input and output neurons
- no spatial information among neighboring neurons
  - each neuron is treated independently
- many parameters

How can we use **spatial structure** in the input image to inform the architecture of the network?



Input :

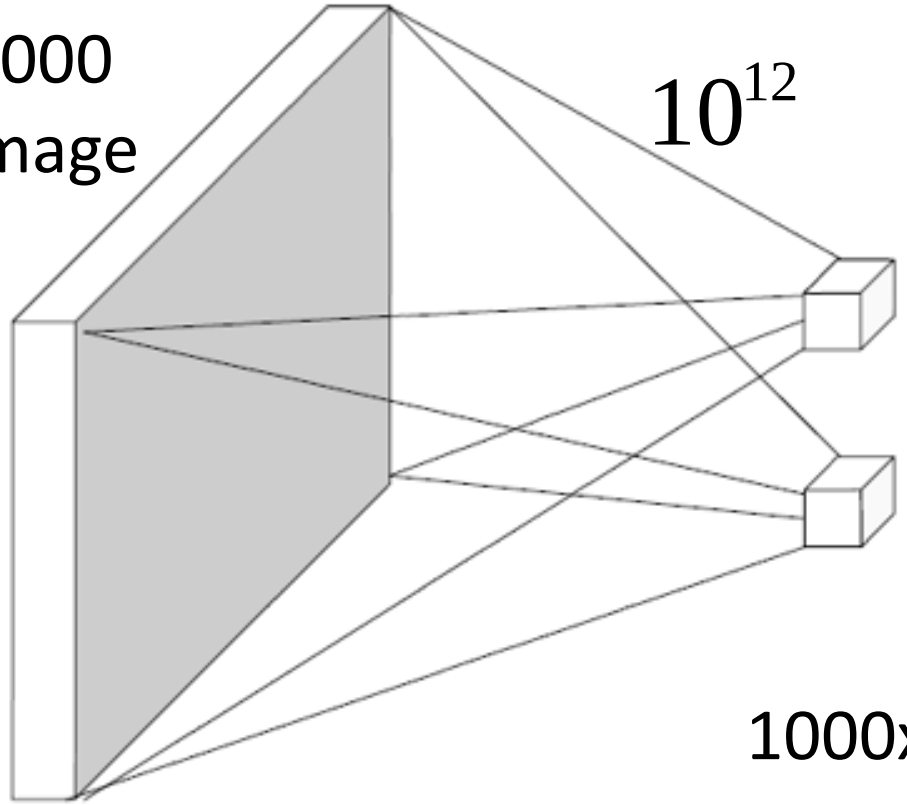
- 2D image
- Vector of pixel values



Problems with ANN

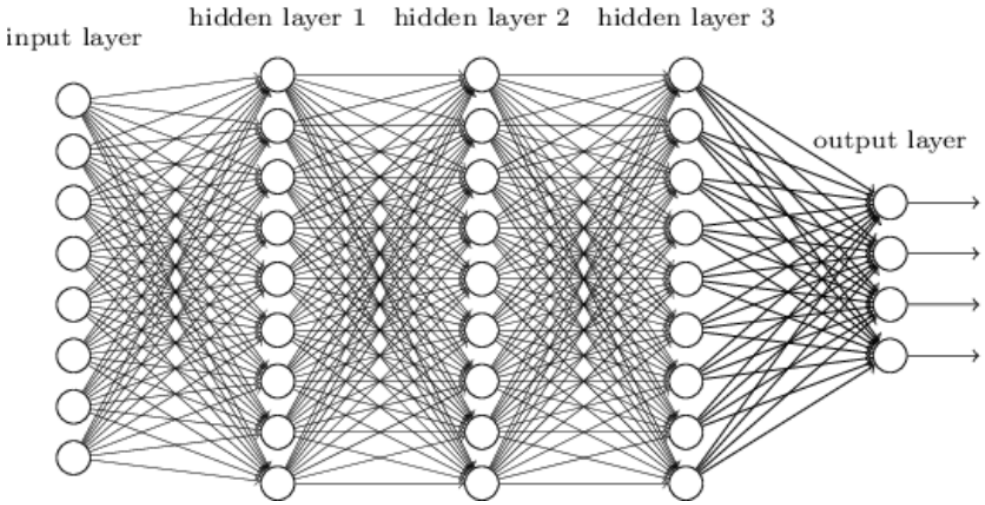
- Too many links when applying fully connected layers to images

1000x1000
input image



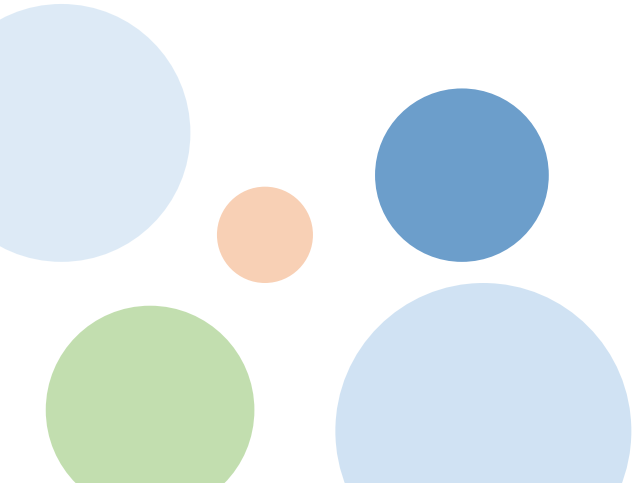
Fully connected
layer

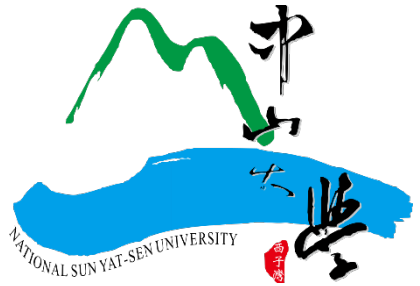
1000x1000
output image



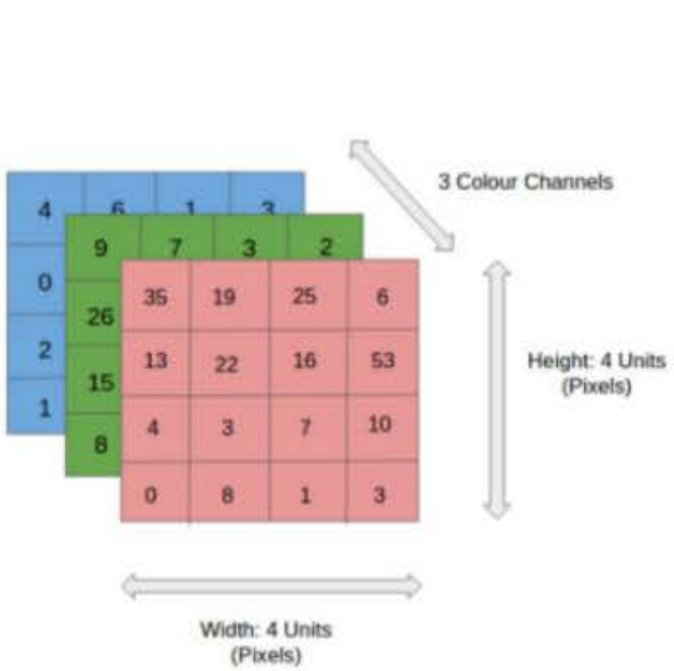
For a 1000x 1000 image,
there are 1M links for a
hidden node.
If there are 1M hidden nodes,
there are 10^{12} parameters

Convolutional Neural Networks (CNN)

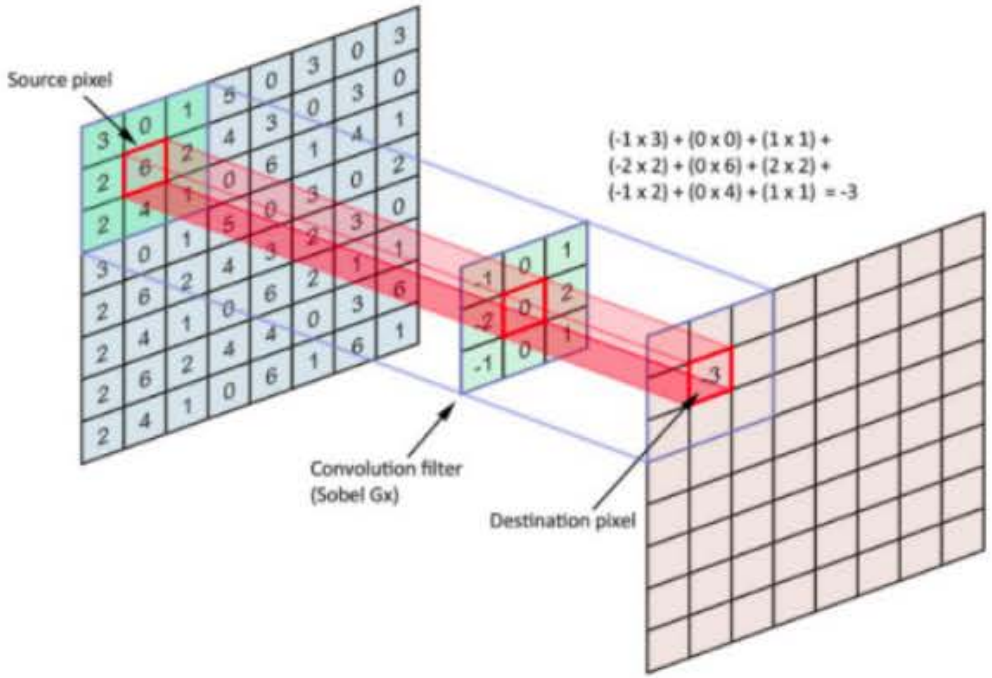




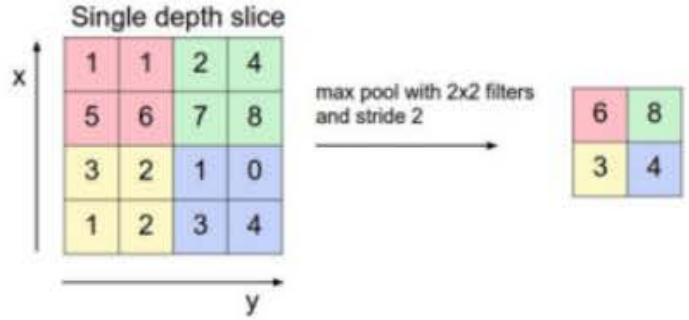
Convolutional and Pooling Layers



Source Image



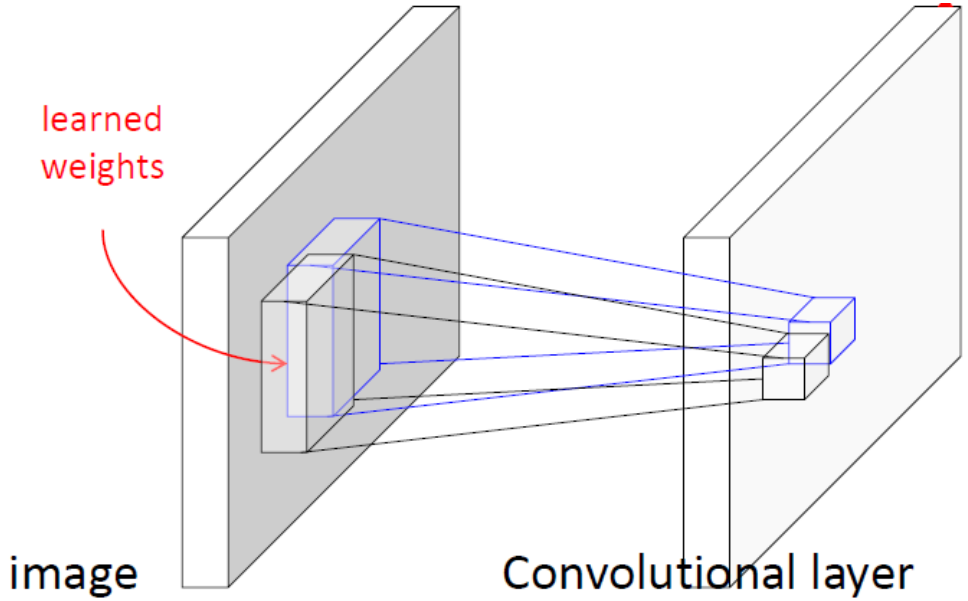
Convolution Layer

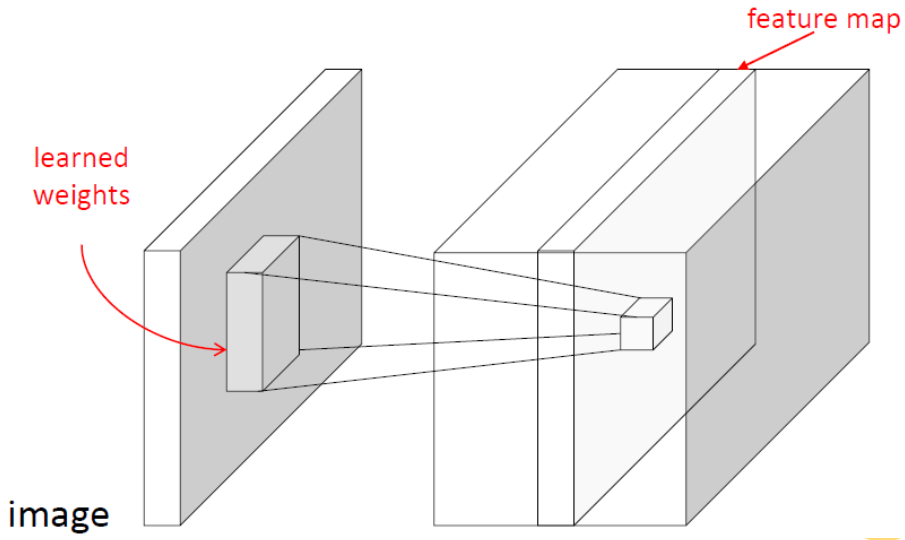


Pooling Layer

Convolutional Neural Networks (CNN)

- ANN consider each pixel in an image as an independent neuron
- But pixels in image have **spatial** dependencies
- CNN has local connectivity due to spatial dependencies in images
- shared convolution filter kernel across the same image
- M input feature maps (input channels) -> N output feature maps (output channels)
  - using 3D filter kernel for each output feature maps
    - 2D filter kernel for one input feature map + another dimension across input feature maps

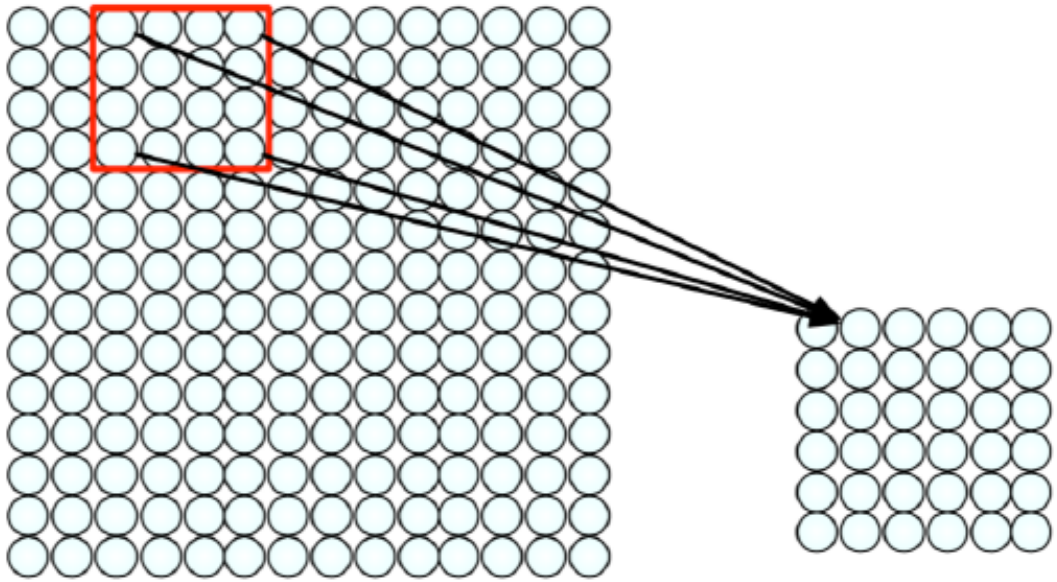




Convolution with Spatial Structure

Replace fully-connected ANN by convolution to reduce # of parameters and operations => Convolutional Neural Network (CNN)

- example:
  - 2D filter of size 4x4 -> 16 different weights
  - Apply this same filter to 4x4 patches in the input image
  - Shift by 2 pixels for next patch (stride=2)
  - This "patchy" operation is called **convolution**



Assume one input feature map:

I) Apply a set of weights (a filter) to extract **local features** from an input feature map and generate an output feature map

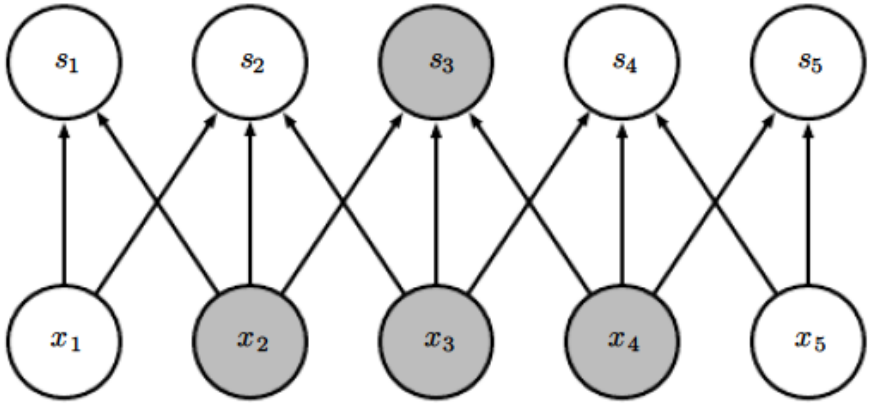
2) Use **multiple filters** to extract different features (generate **different output feature maps**)

3) **Spatially share** parameters of each filter across the same input feature map

Connection: Local vs. Full (in 1-D case)

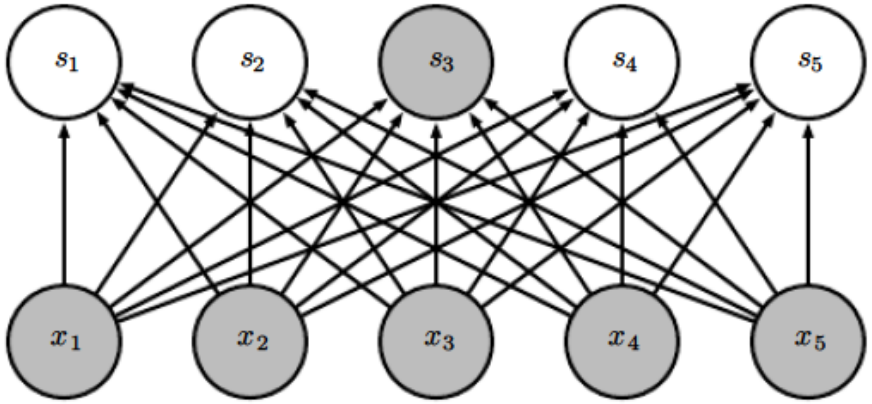
- **receptive field**: region of inputs that affect an output
- larger receptive filed => more input neurons considered in feature extraction
- output units affected by *x*3 input units affecting *s*3

Local Convolution



receptive field (3 local features)

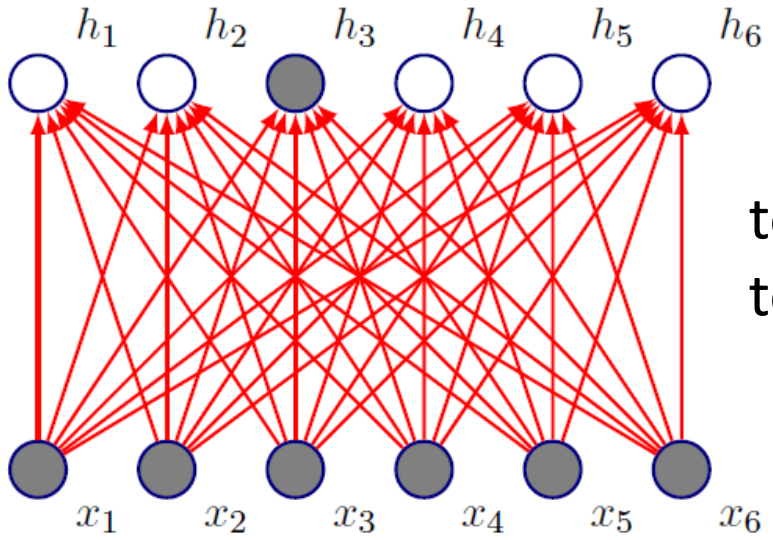
Fully Connected



receptive field (all 5 features)

Motivation for Convolution

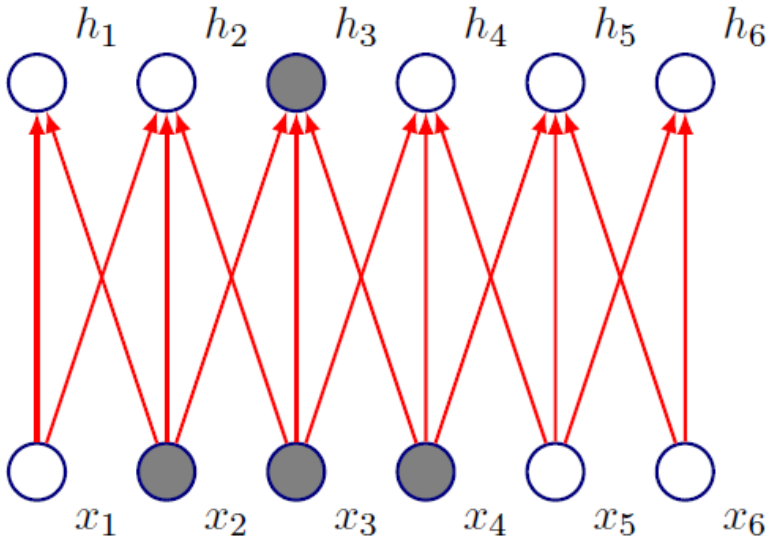
- Fully-connection in ANN
  - many edges (weights)
  - hard for training in case of many layers
  - receptive field of h_3: 6



total # of weights: 6x6=36

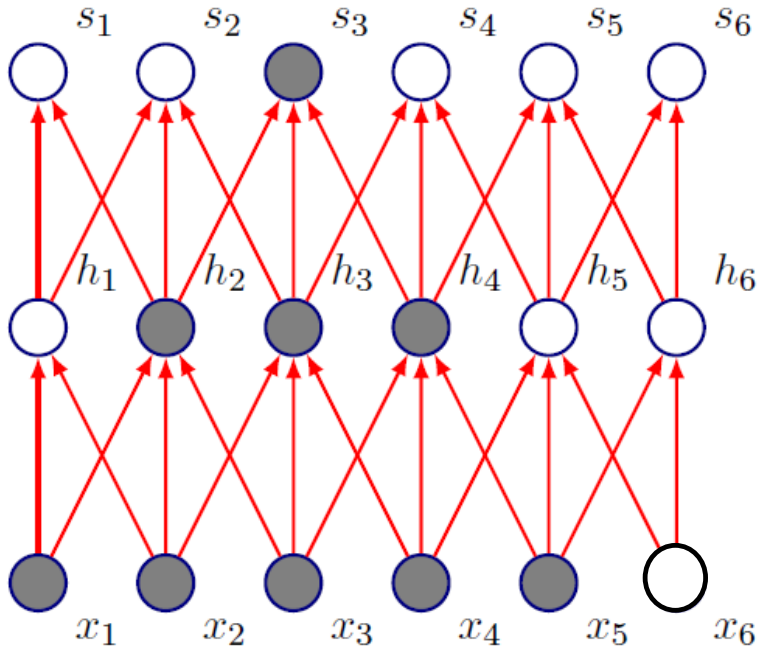
total # of operations: 6x6=36 MAC

- Convolution
  - sparse interaction
  - parameter sharing
  - enlarge receptive field via more layers, e.g.,
    - one layer => receptive field of h_3 : 3
    - two layers => receptive field of s_3 :5



total # of weights: 3

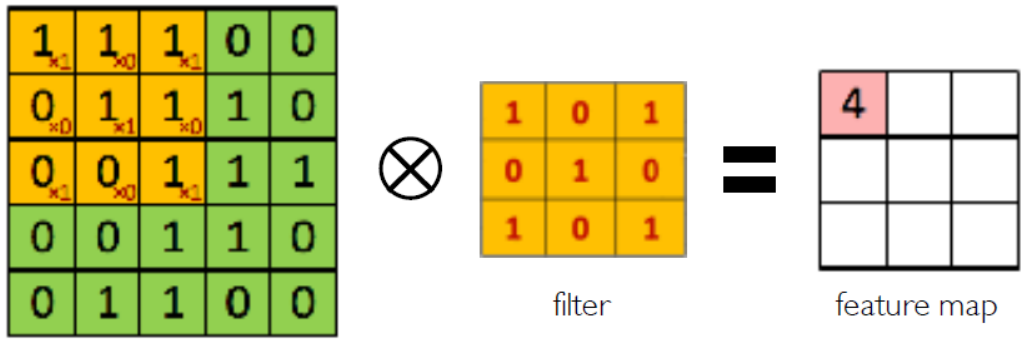
total # of ops: 3x6=18 MAC

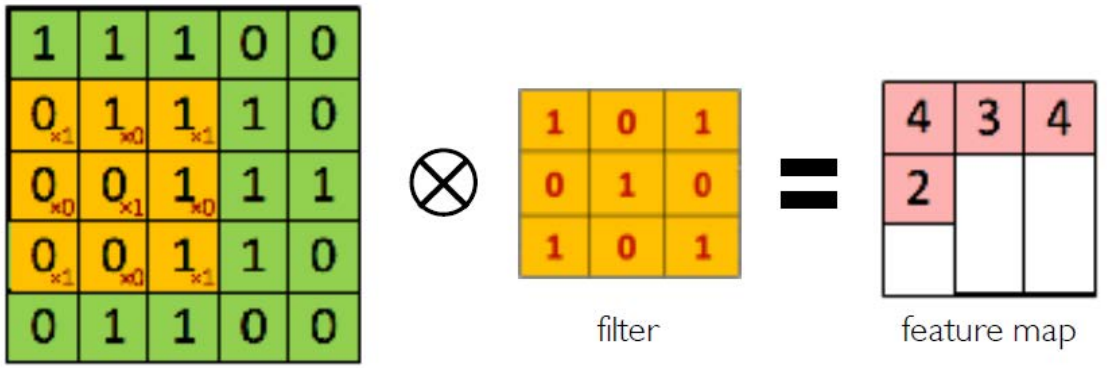


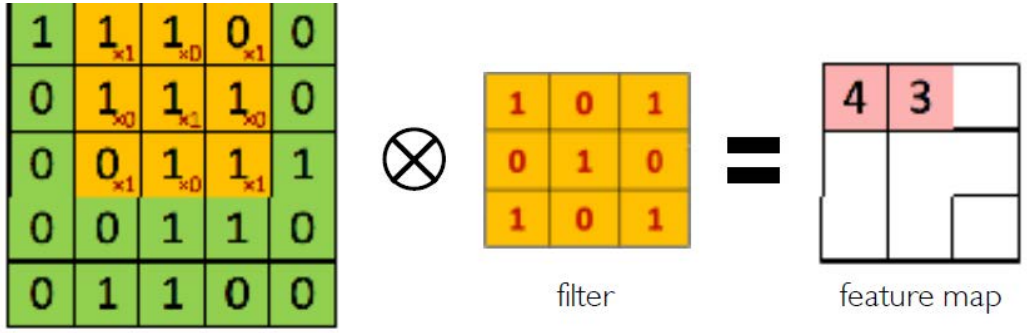
total # of weights: 3+3=6

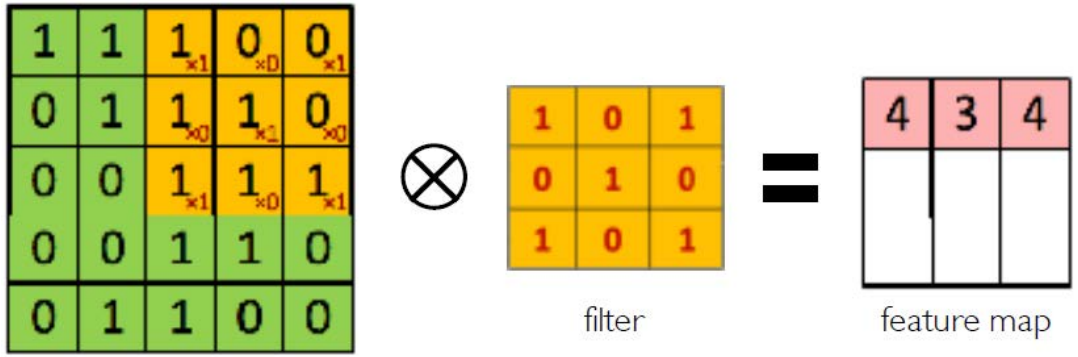
total # of ops: 3x6x2=36 MAC

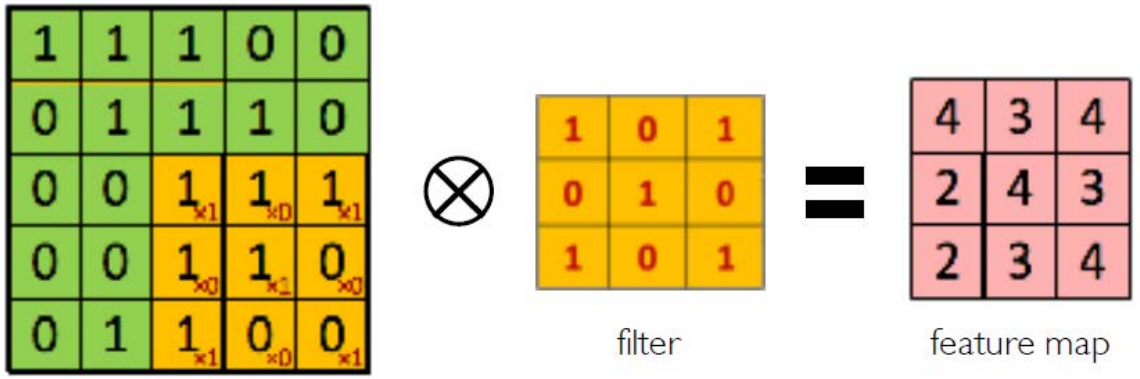
Convolution example (stride=1)



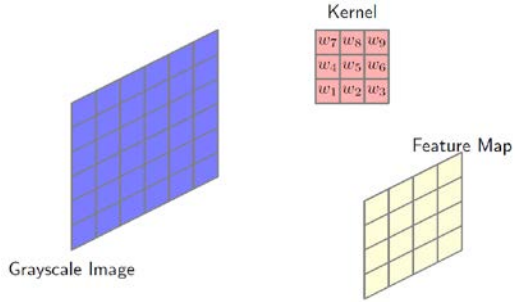




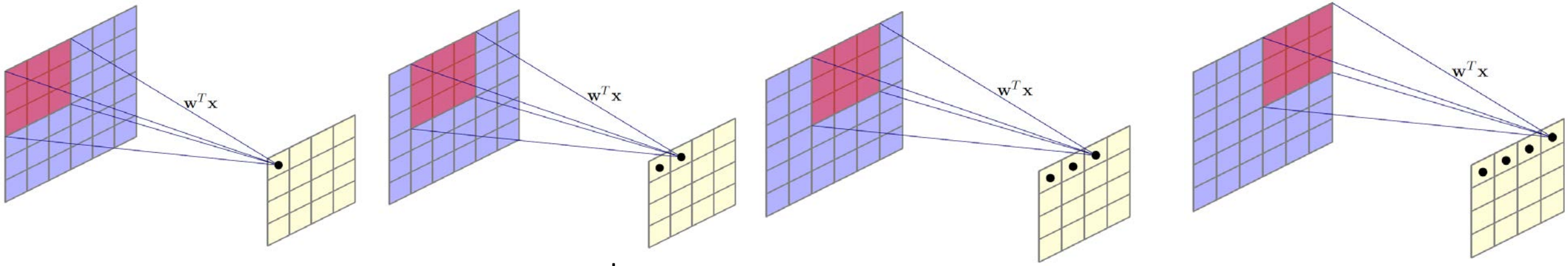




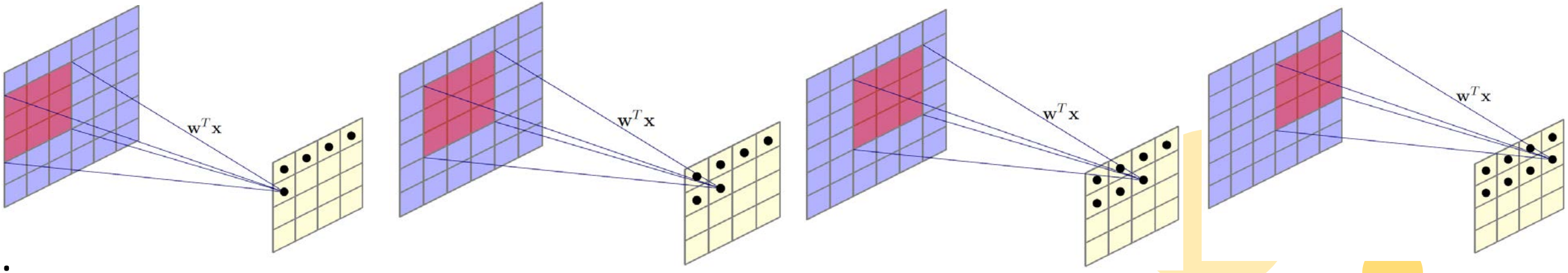
Convolution (in steps)



- convolution to generate the 1st row of an output feature map



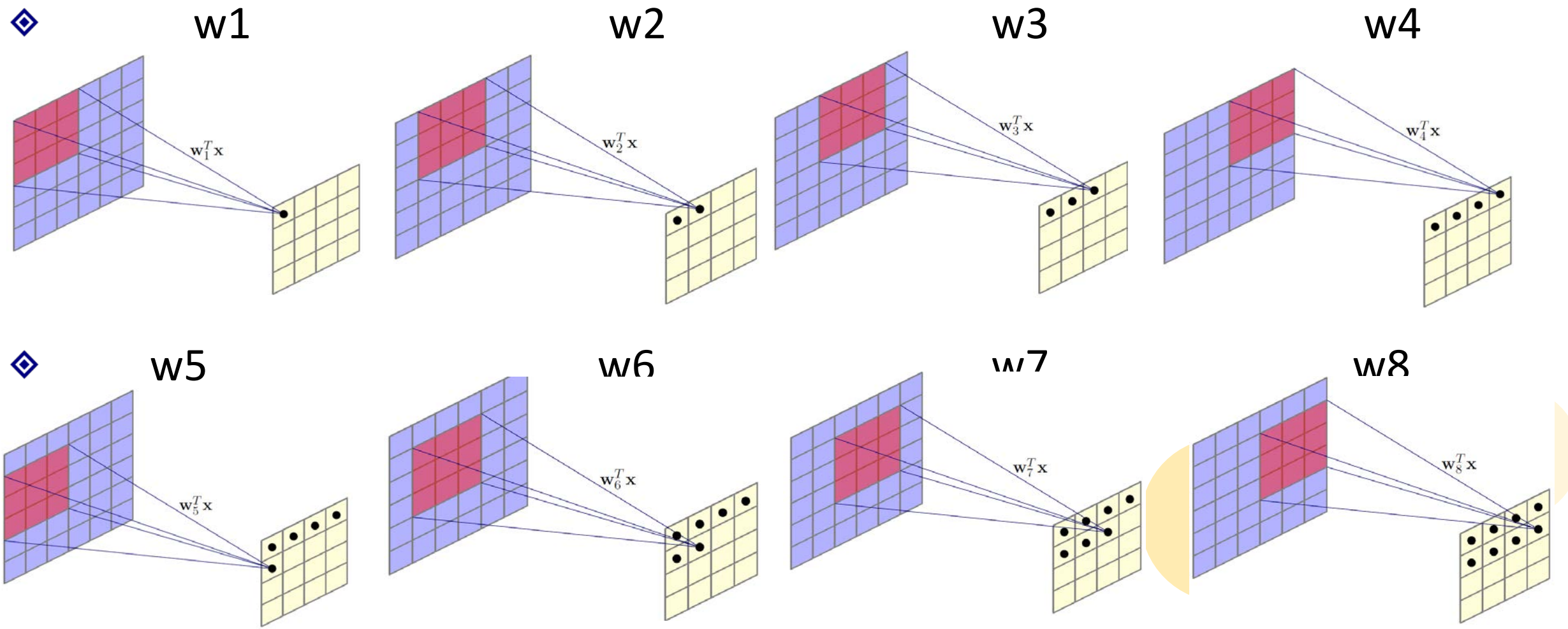
- convolution to generate the 2nd row of an output feature map



...

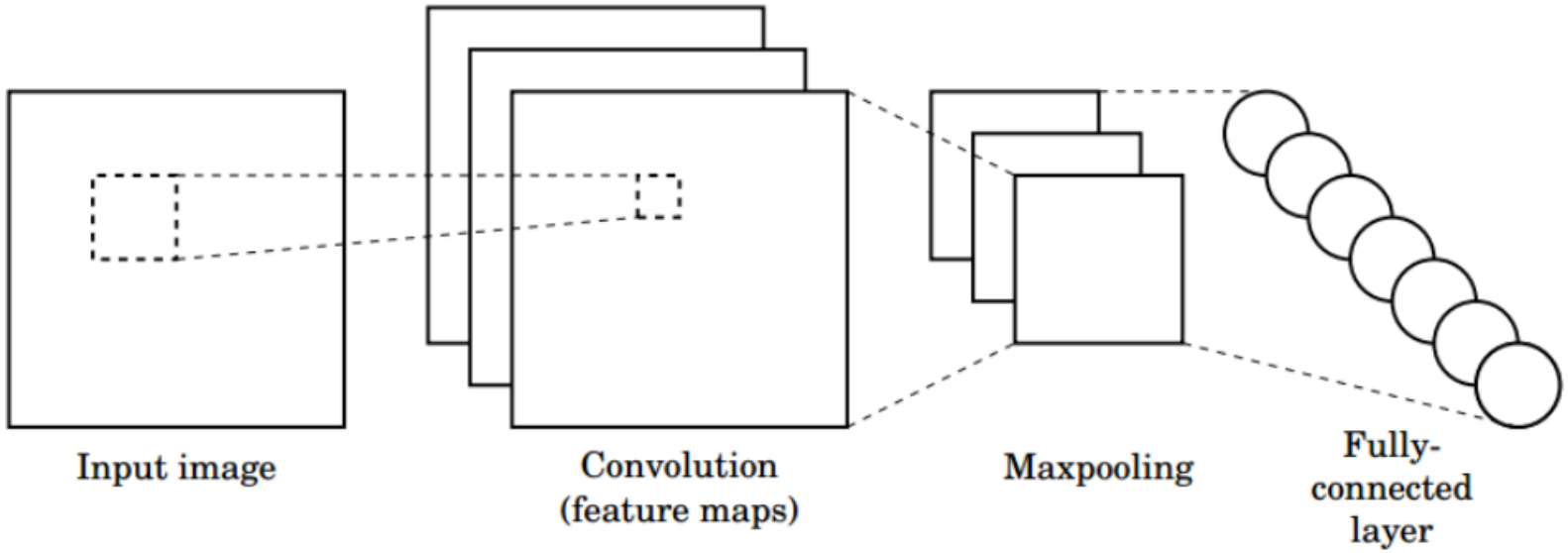
Locally Connected vs. Convolution

- convolution: **shared** filter weights for different output pixels in an feature map
- locally connected: **different** filter weights for different patches (output pixels)



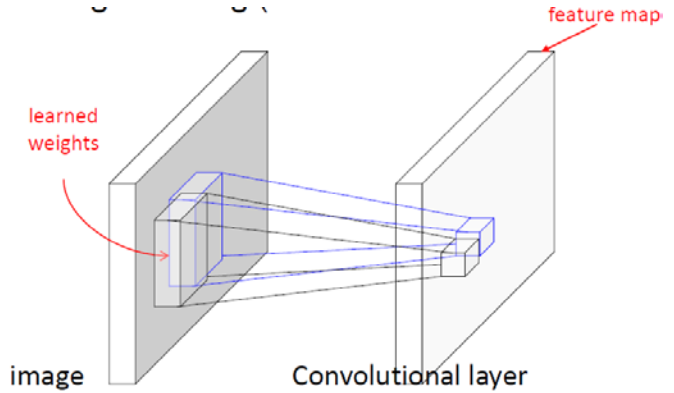
CNN for Classification

- CNN = Convolutional (Conv) layers + Fully Connected (FC) layers
- convolution
  - exploits **spatial structure** with **shared parameters** (filter weights) for the same feature map
  - used for feature extraction of various levels
- Pooling
  - exploit **local invariance**
  - used to reduce image size, and thus reduce computation complexity
- non-linear activation function (i.e. ReLU) after each layer
  - more layers of nonlinearity => more complicated model => more accurate classification
  - make model more powerful, especially in deep models with many layers (each layer has an activation function)

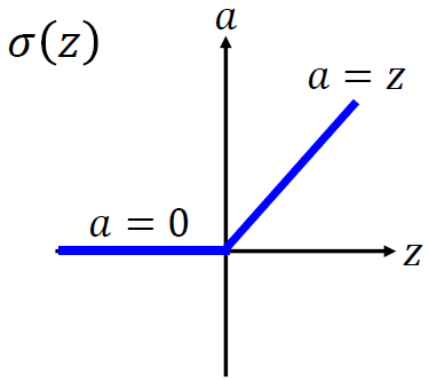


Building Blocks of CNN

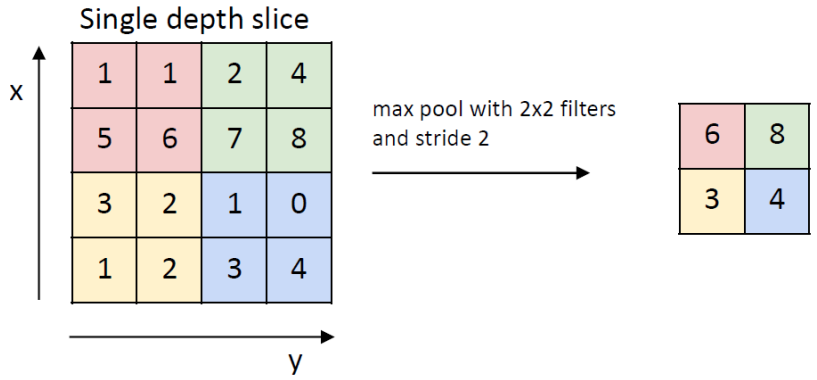
- Convolution (Conv) layers
  - Convolutions with 3D filter kernel



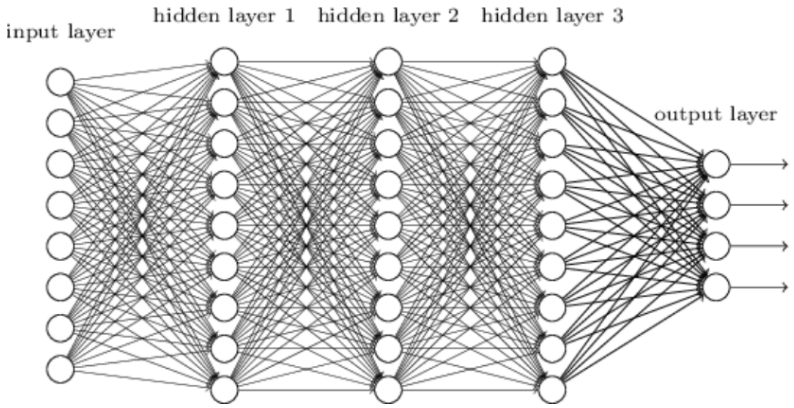
- Activation
  - ReLU



- Pooling layers
  - size reduction of feature maps

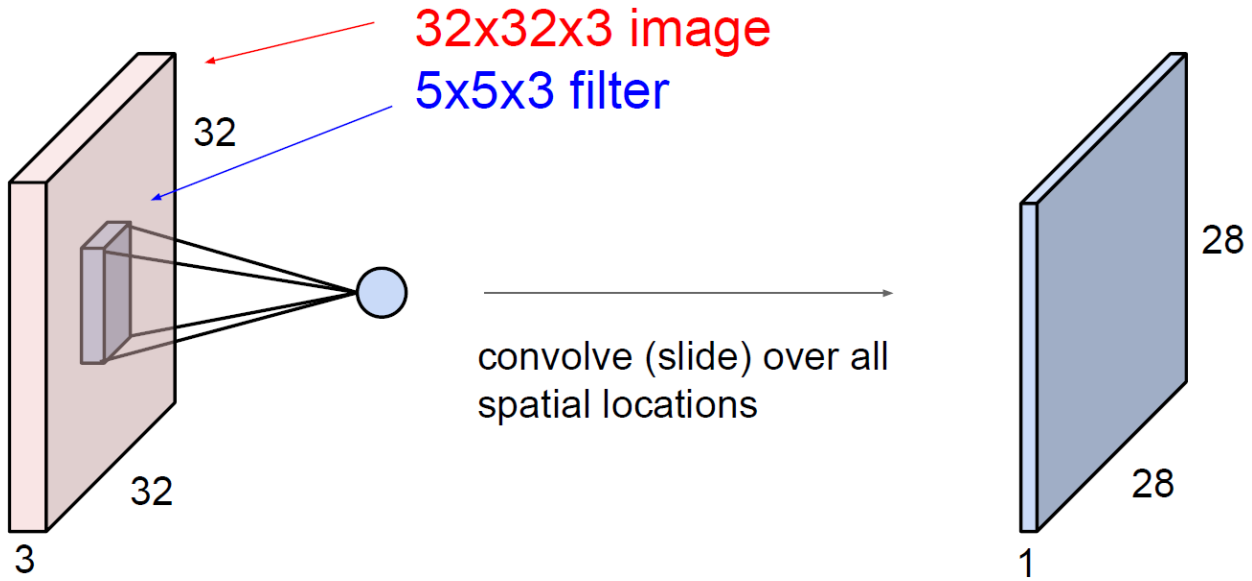


- Fully Connected (FC) layers
  - feature extraction
  - classification



Convolutional Layer (1 output channel)

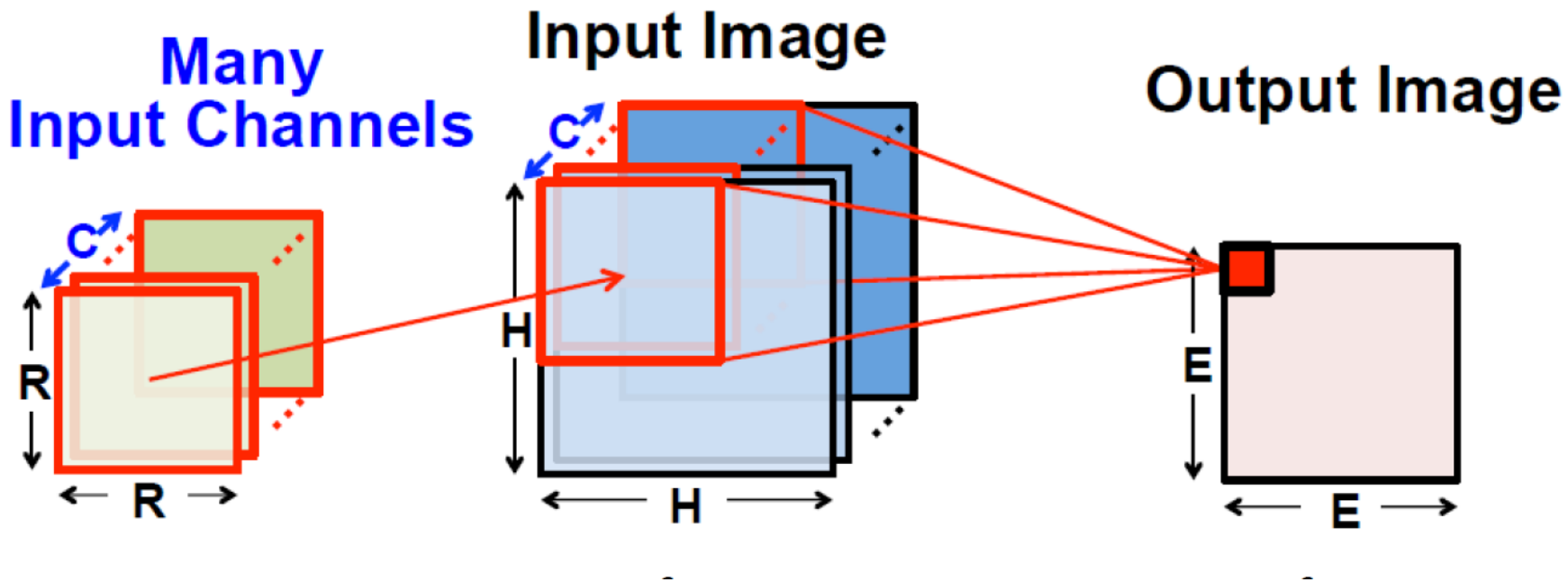
- each filter is used for 1 input channel (input feature maps)
  - e.g., 5x5 filter
- CNN convolution = sum of the convolutions for all input channels
  - e.g. CNN convolution with 5x5x**3** filters for **3** input channels
- each output pixel is connected to a small region (called **receptive field**) of input pixels
  - e.g. 5x5 receptive field for each output pixel
- output channel size is slightly reduced if no padding is used
  - (e.g., from 32x32 to 28x28 via 5x5 filtering)
  - could use **padding** for input feature map to maintain size of input and output feature maps



if the original 32x32 input feature map is padded to 36x36 with two surrounding borders of zeros
=> output feature map size is 32x32 (instead of 28x28 without padding)

Convolution with One Output Feature Map

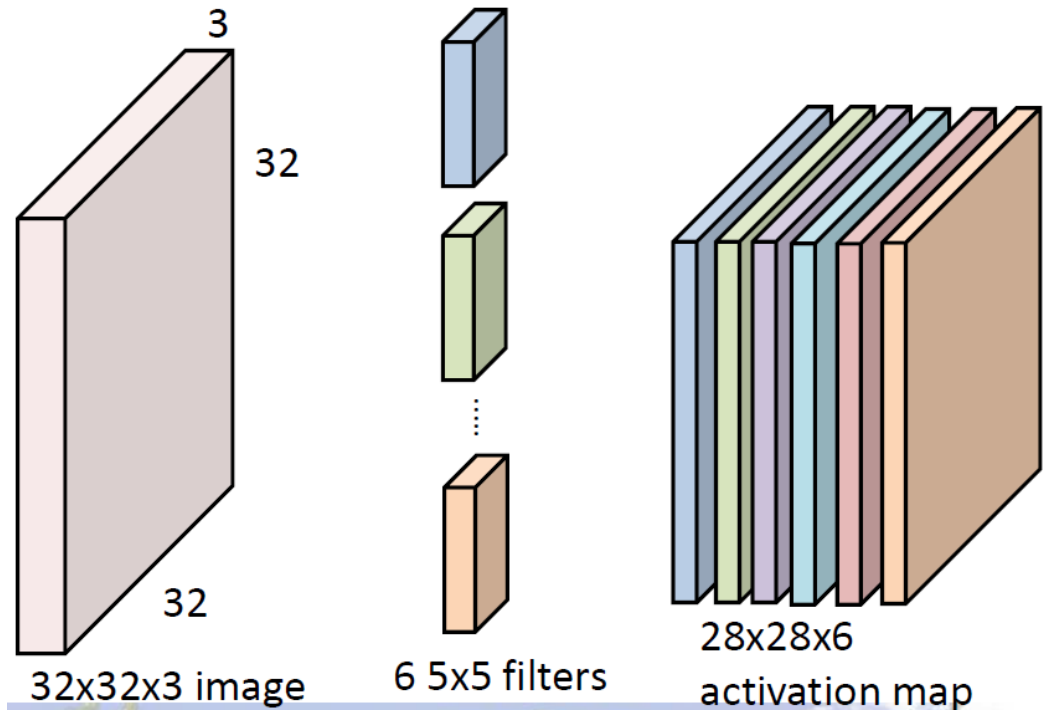
- a receptive field in input feature maps gives an output neuron
- each output feature map has its own set of filter kernel weights
- e.g., **C** input feature maps (input channels) of size **HxH**, **C** filter kernels of size **RxR**, one output feature map (output channel) of size **ExE**



a 3D filter of size RxRxC for an output feature map

Convolutional Layer (6 output channels)

- each filter is used for 1 pair of input and output
- e.g., **6** 5x5x**3** filters for **3** input channels and **6** output channels
- a Conv layer generates output pixels (neurons) in 3D grid

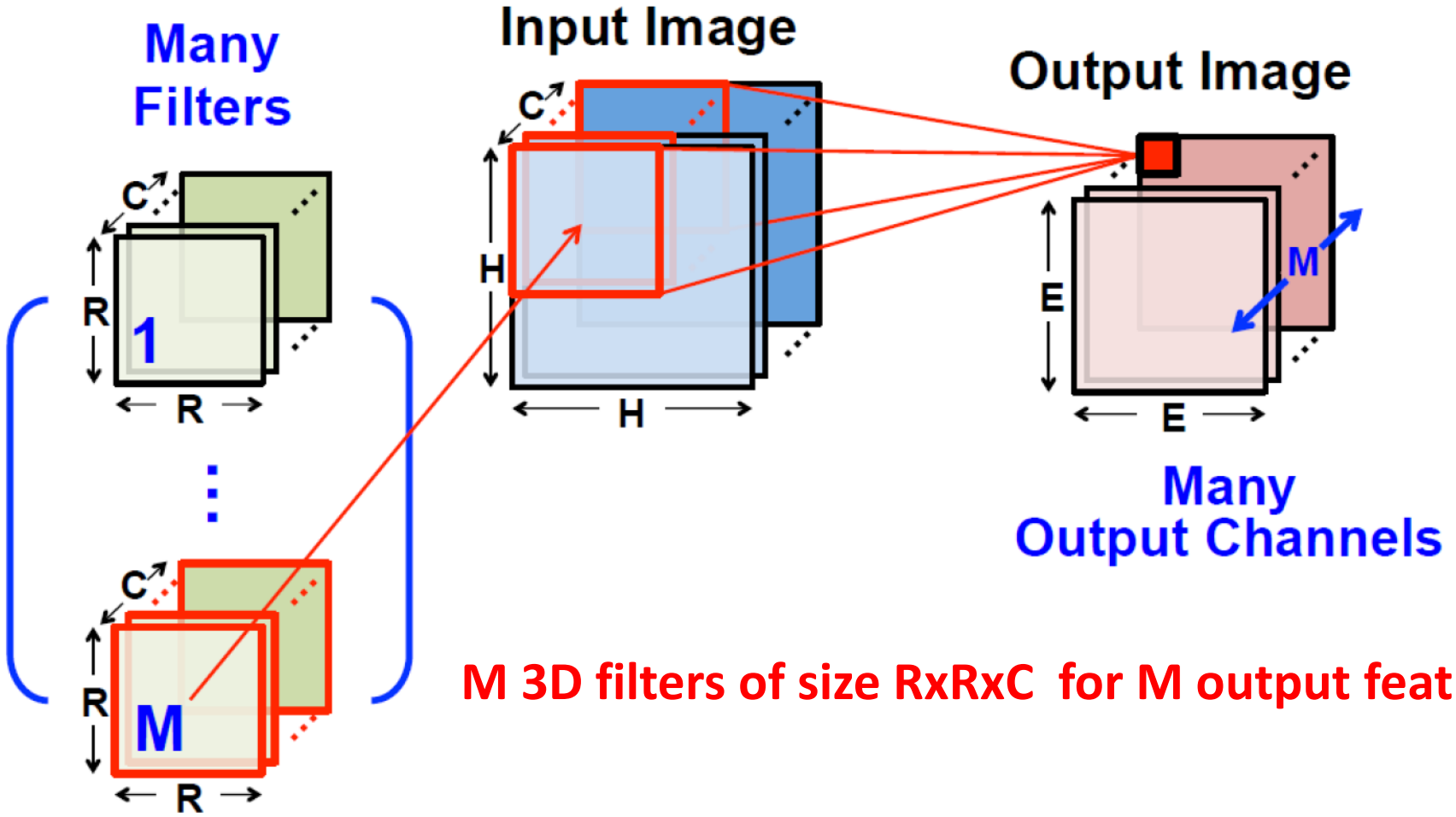



```
import keras
from keras.models import Sequential
from keras.layers import Conv2D

model = Sequential ()

model.add (Conv2D (filters=6
                   kernel_size=(5,5)
                   stride=1
                   padding='valid'
                   activation='relu'
                   input_shape=(32,32,3) )
```

```
model = keras.Sequential ([
keras.layers.Conv2D(6, (5, 5), strides=1, padding='valid', activation='relu', input_shape=(32,32, 3)  ])
```

Convolution with Multiple Output Feature Maps

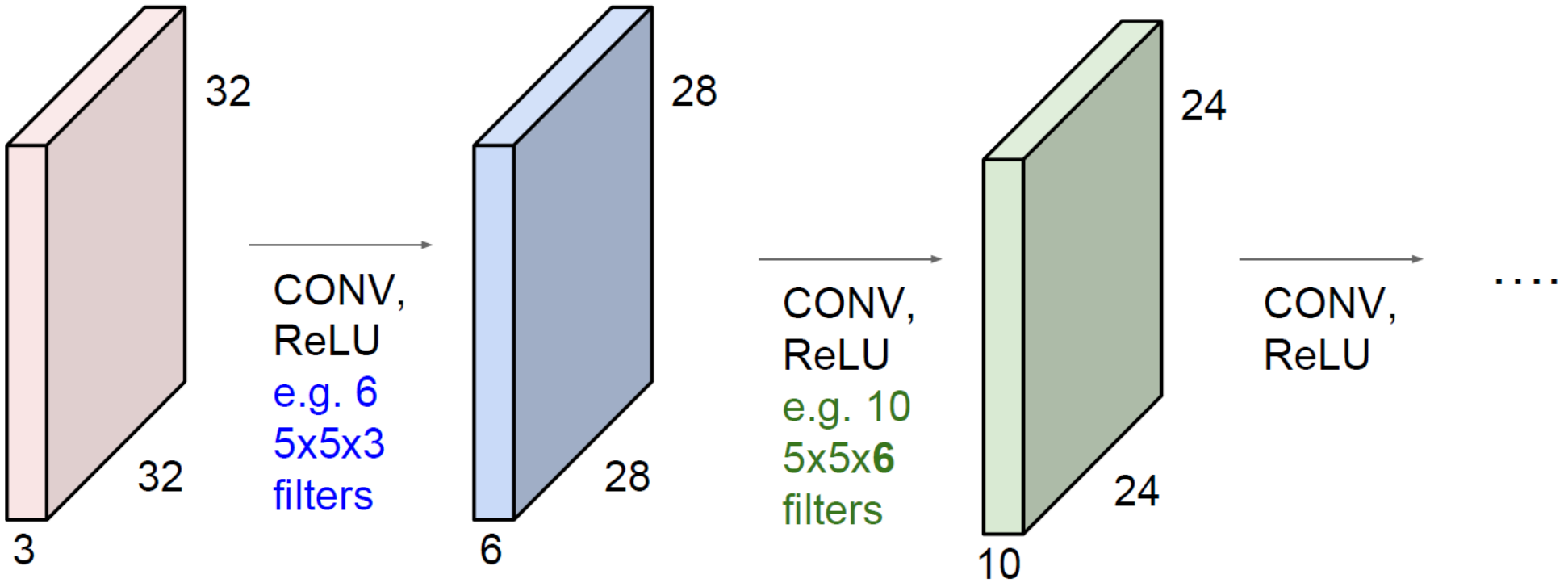
- **C** input features => **M** output feature maps
- need **CxM** filter kernels, **C** filter kernels for each output channel
- Quiz: what are the total numbers of parameters and MAC operations?



M 3D filters of size RxRxC for M output feature maps

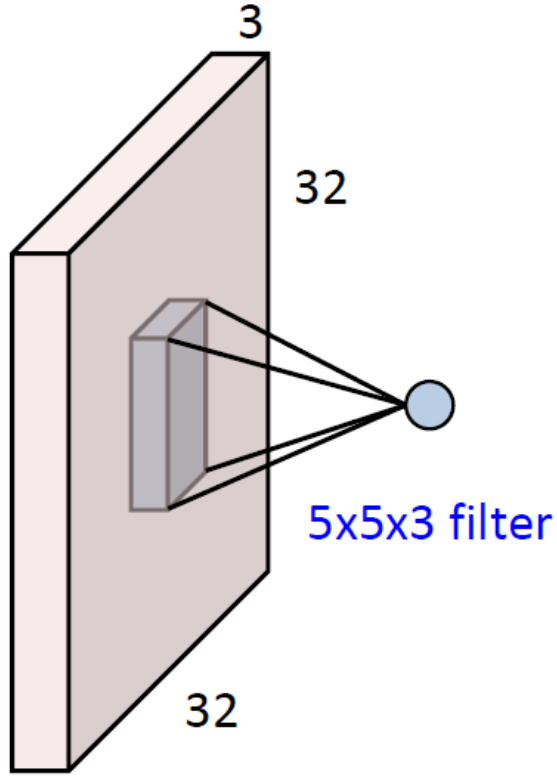
Convolution+ ReLU

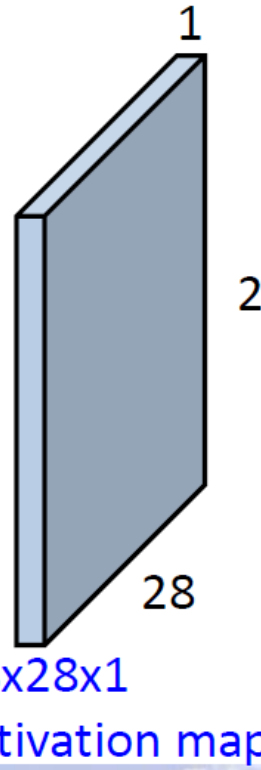
- output neurons after convolutions are connected to non-linear activation function (usually ReLU in CNN)
- What is the receptive field in the two Conv layers below? (Ans.: 9x9)
- a Conv layer = Convolution + activation

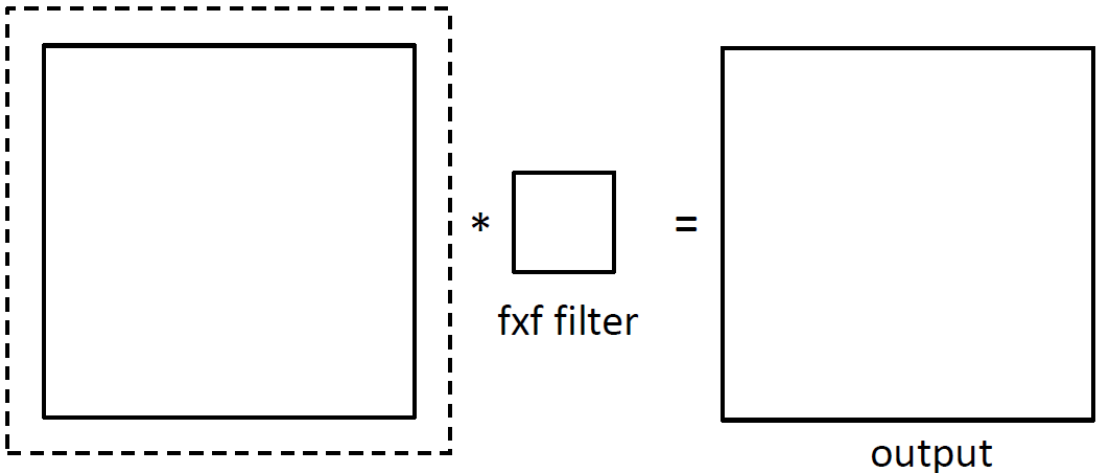


Padding

- input feature map size: NxN = 32x32
- Convolution with fxf=5x5 filter and stride s=1
- output feature map size: 28x28
- what is the size of output feature map with fxf filter?
- how to keep output feature map size fixed?
  - padding of input feature map with two borders
- what size of padding is required for fxf filter?







size of output feature map **=(N-f)/s +1**

NxN: input feature map size

fxf filter kernel, Stride=s

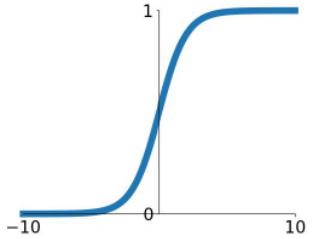
zero padding with **(f-1)/2** to preserve size

Activation Functions

- without activation functions, the output is just linear combination of inputs!
  - human brain is much complicate than a “linear” operator
- activation functions create non-linearity

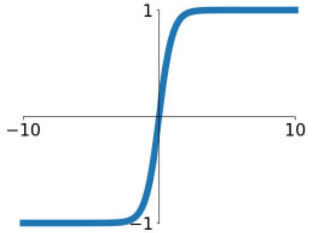
Sigmoid

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



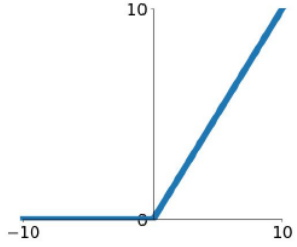
tanh

$$\tanh(x)$$



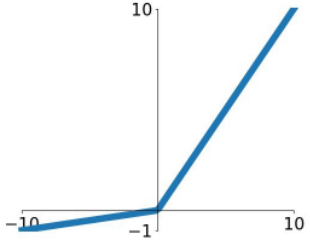
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

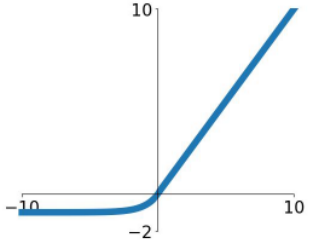


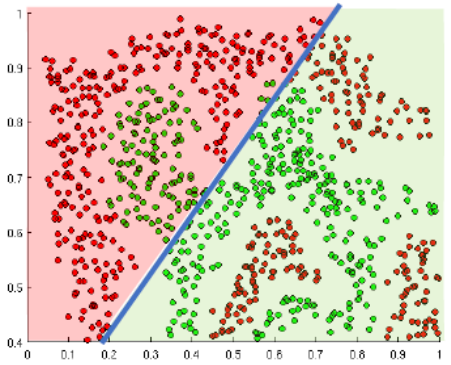
Maxout

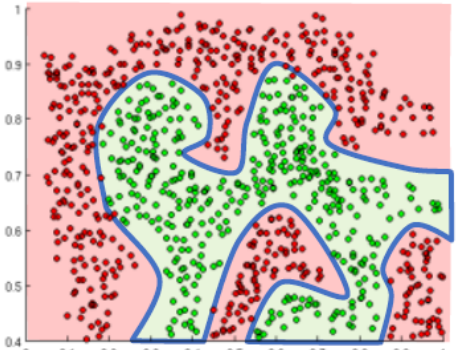
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



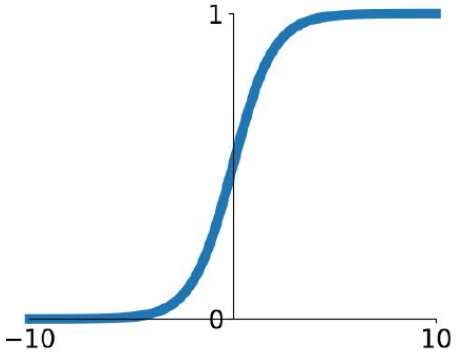




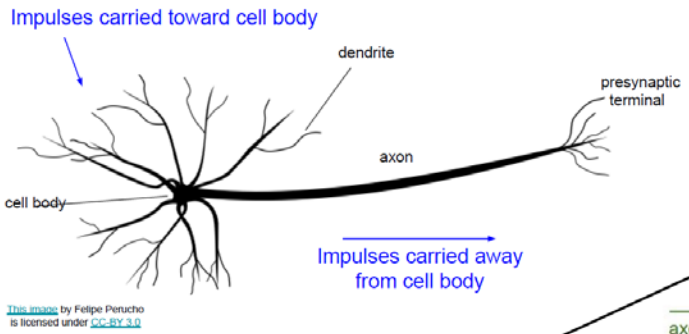
sigmoid

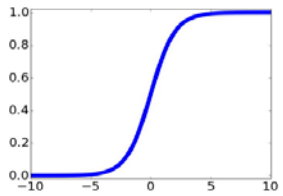
$$\sigma(x) = 1/(1+e^{-x})$$

- output range in [0,1]
  - not zero-centered
- historically popular
  - nice interpretation as a saturating “firing rate” of a neuron
- problems
  - saturated neurons “kill” gradients during training
  - output are not zero-centered
  - exponential function is compute-expensive



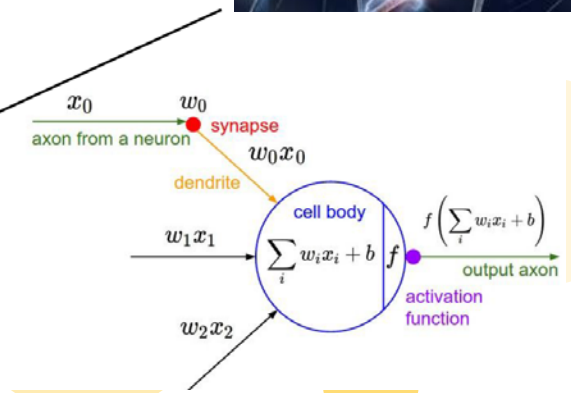
Sigmoid





sigmoid activation function

$$\frac{1}{1+e^{-x}}$$

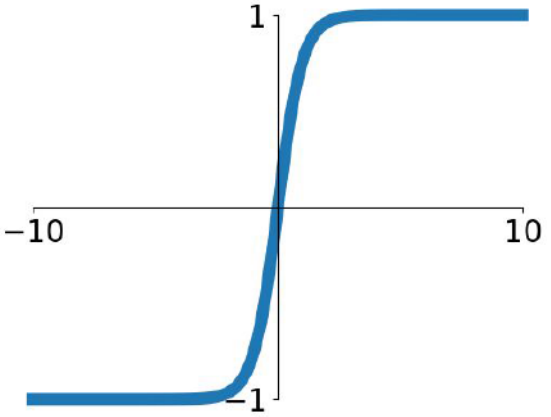


tanh(x)

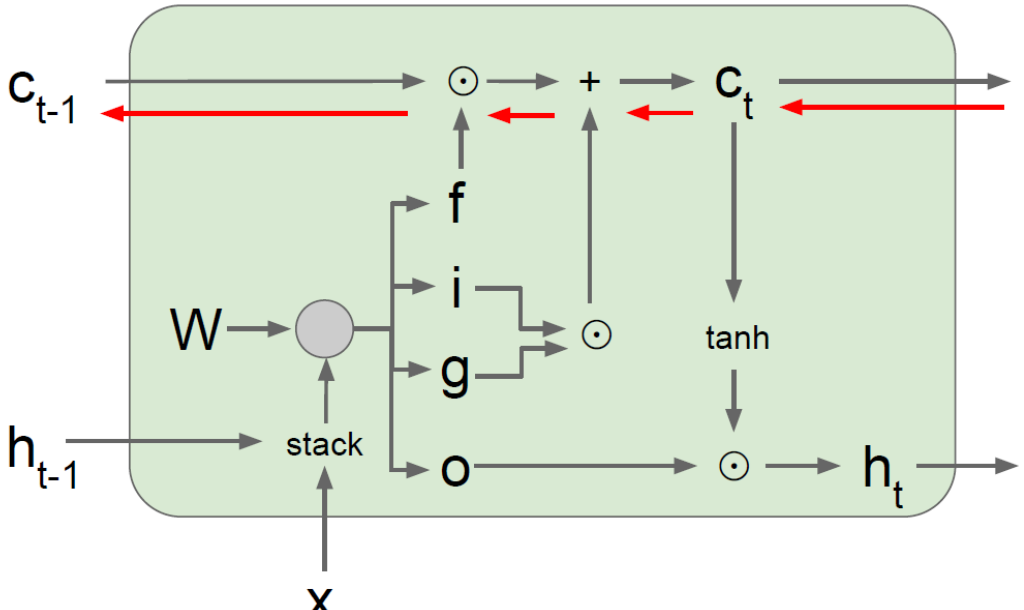
$$\tanh(x) = (e^{x} - e^{-x}) / (e^{x} + e^{-x})$$

$$= 1 - 2 / (1 + e^{2x})$$

- output range in [-1, 1]
  - zero-centered
- still kills gradients in saturation regions during training
- used in RNN (Recurrent Neural Network) such as LSTM (Long Short Term Memory)



tanh(x)



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

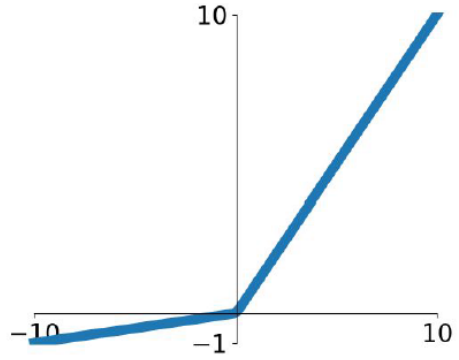
$$c_t = f \odot c_{t-1} + i \odot g$$

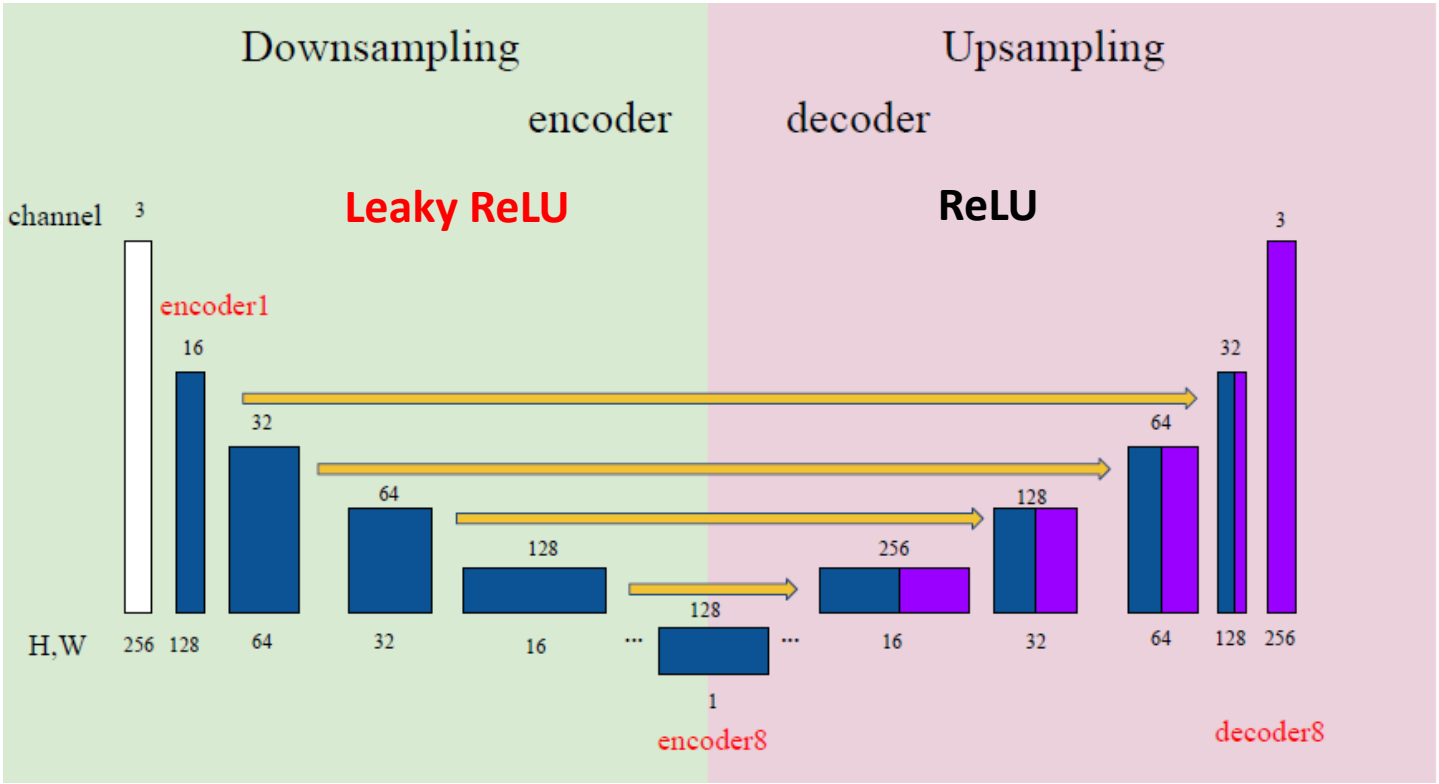
$$h_t = o \odot \tanh(c_t)$$

Leaky ReLU

Leaky $\mathrm{ReLU}(x) = \max(\alpha x, x)$

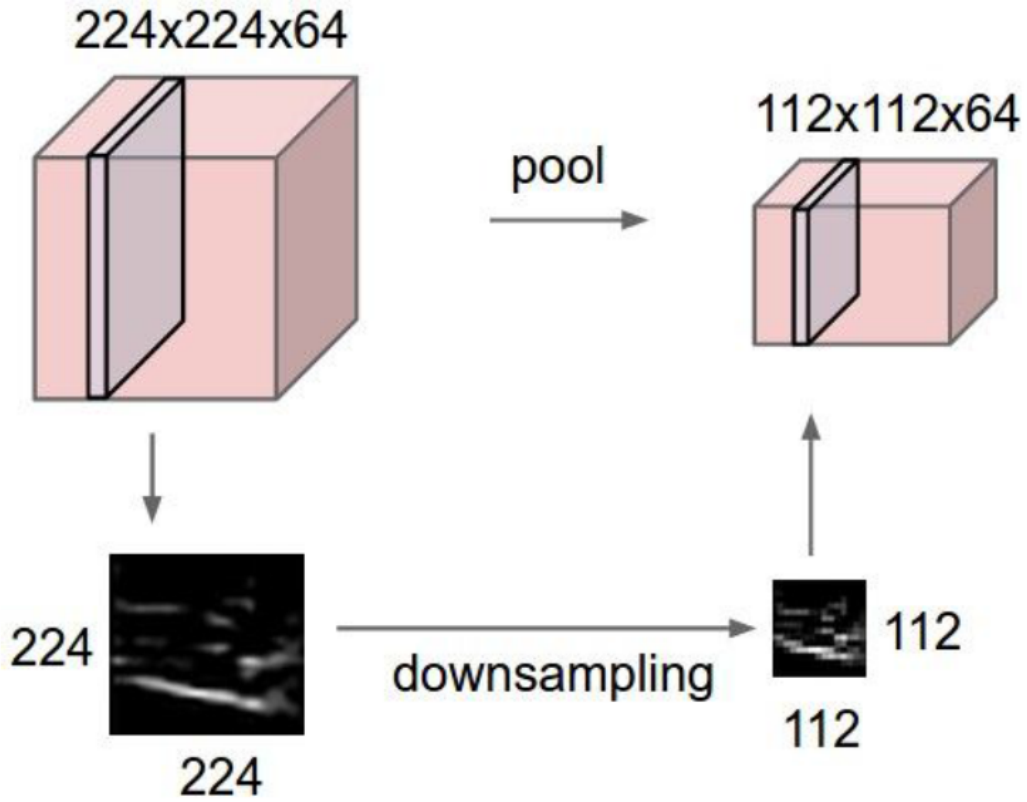
- does not saturate
- computation-efficient
- will not “die” in negative inputs
- also called Parametric ReLU (PReLU)
  - back-prop with gradient of alpha in negative x
- Leaky ReLU is used in encoder (Conv.) part of U-Net
  - decoder (DeConv) part still uses ReLU
  - also used in YOLO detector





Pooling

- invariant to small translations of the input
- reduce size of output feature maps
  - maxpooling, avgpooling
  - larger stride could have same reduction effect as pooling

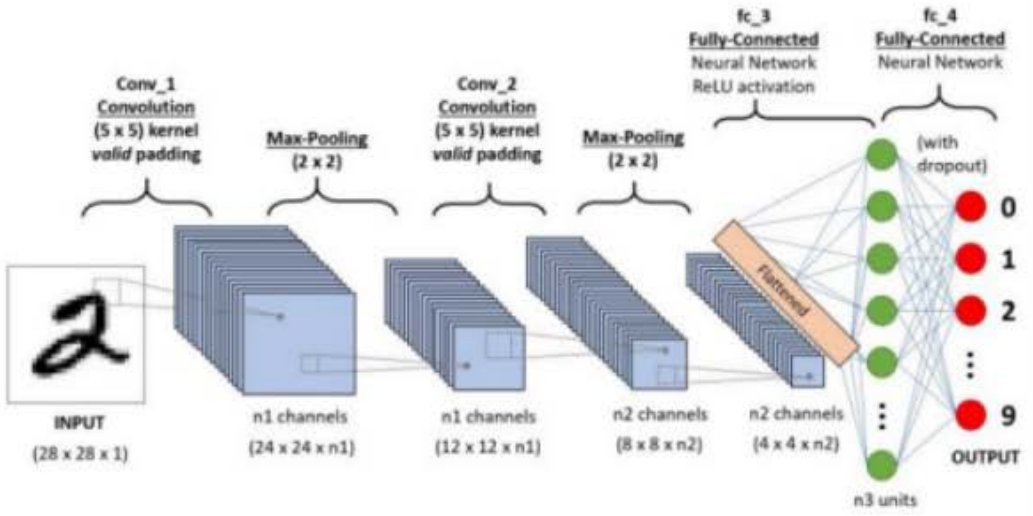


1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pooling

6	8
3	4

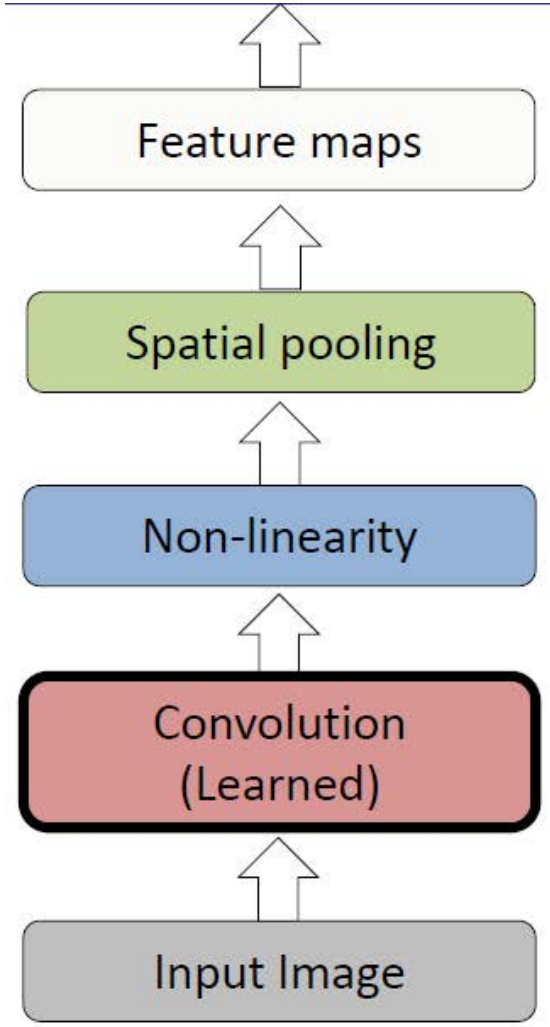
Overview of a Complete CNN Model

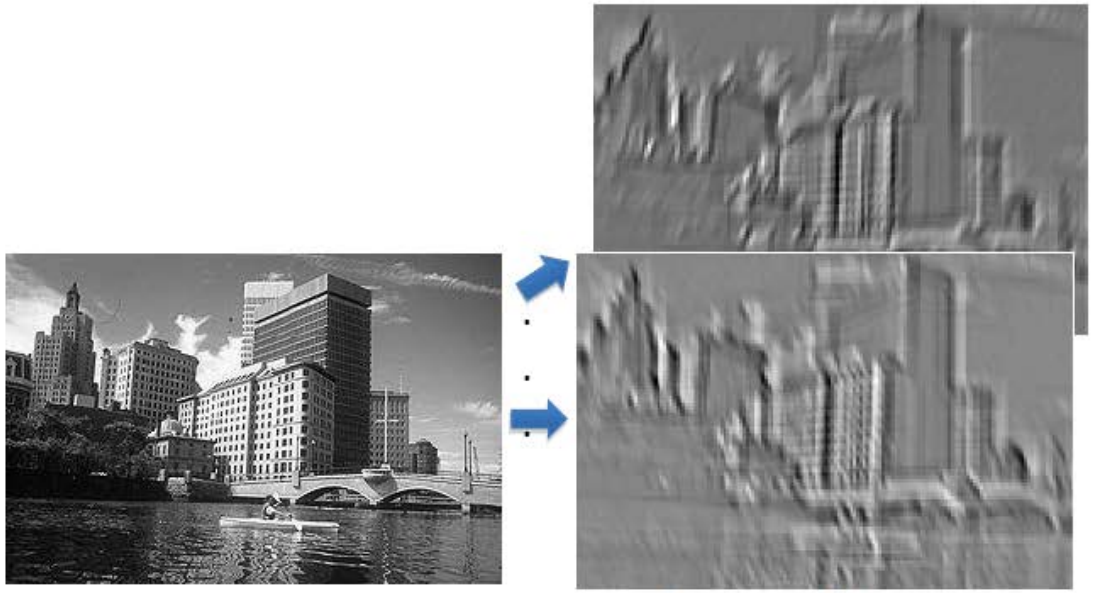


- CNNs have a layered architecture of several layers to learn hierarchical spatio-temporal features
- Convolution Layers use convolution filters to build feature maps
- Pooling Layers help in reducing dimensionality after convolutions
- Non-linear activation functions are applied in the network as usual
- Dropout or BatchNormalization Layers may be used to prevent model overfitting
- FC Layers in final stages help with flattening and prediction

Operations in CNN (1/3): Convolution

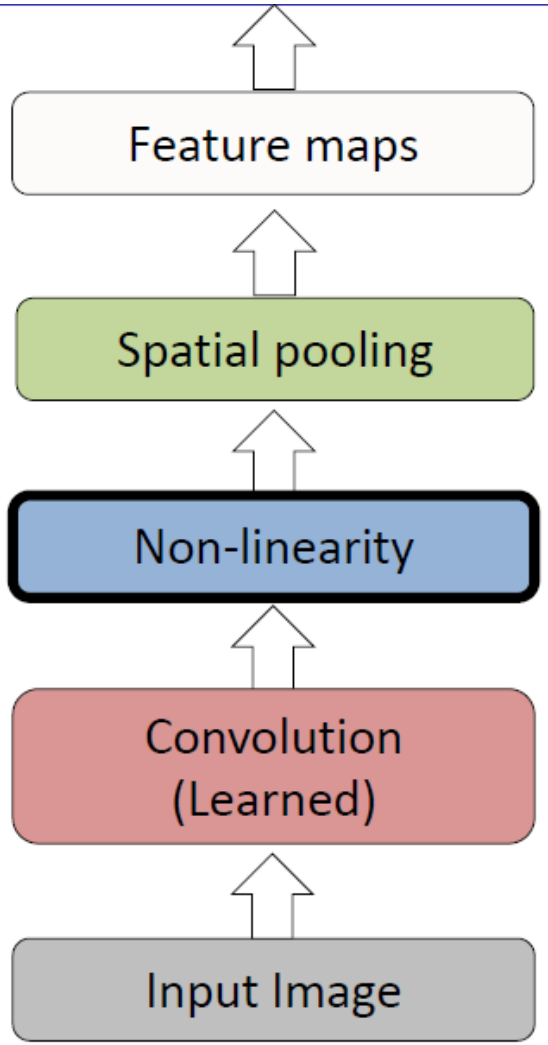
- extract features of various levels (edges, corners, …)

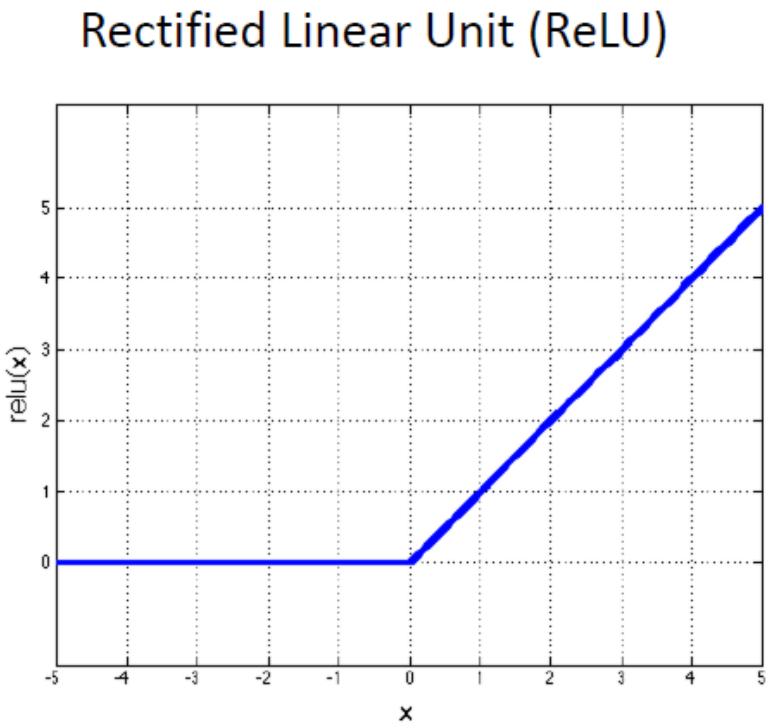




Operations in CNN (2/3): Non-Linear Activation

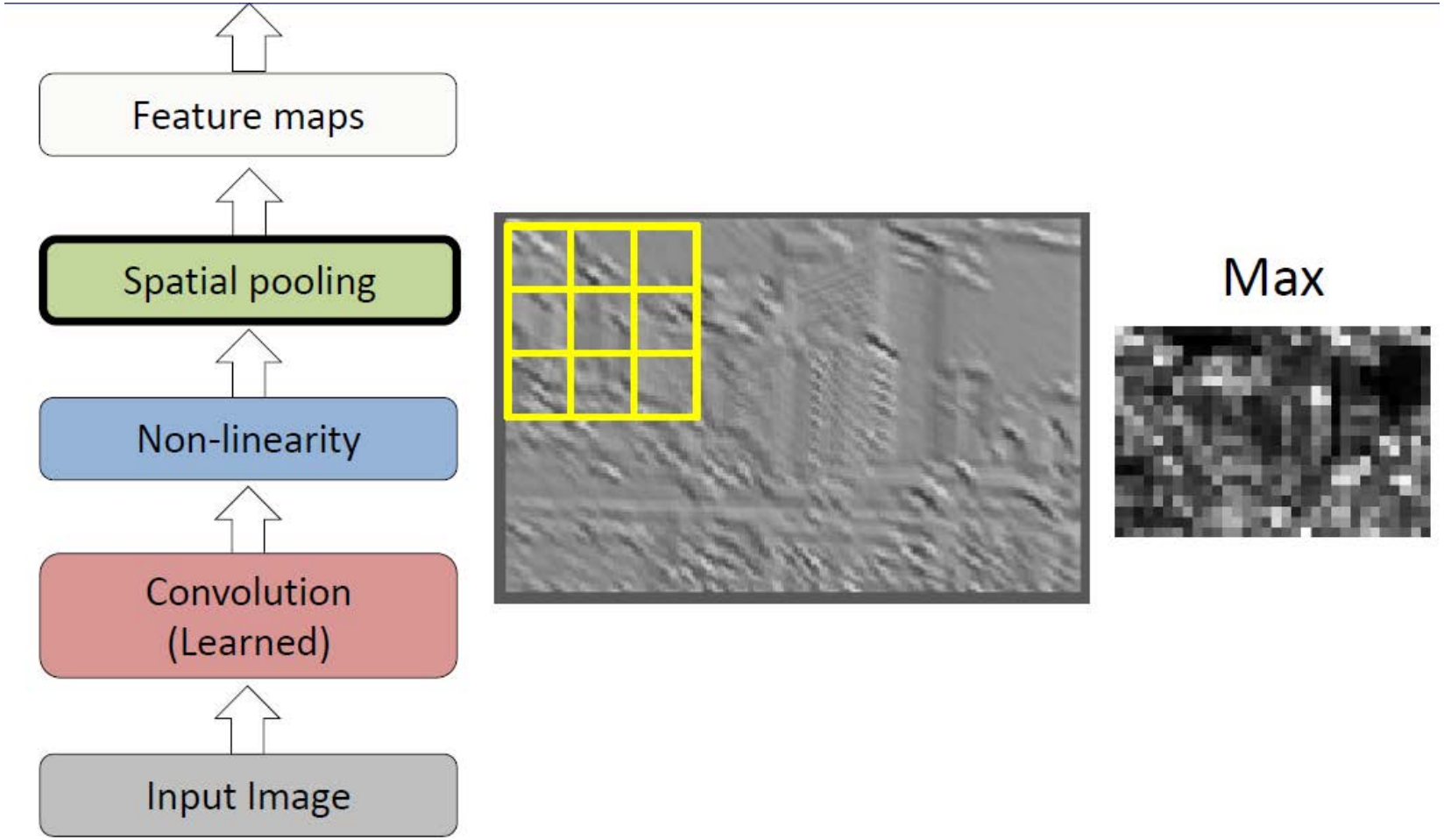
- Add non-linearity for better classification (because Sum-of-Product is linear operations)





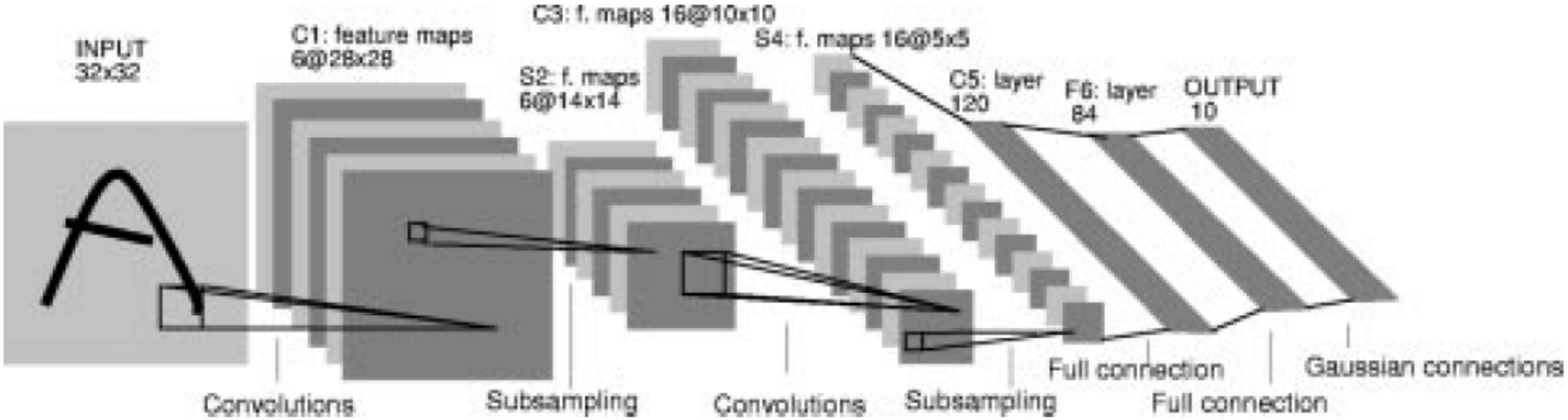
Operations in CNN (3/3): Pooling

- Reduce size of output feature maps to reduce operation complexity in the subsequent layers



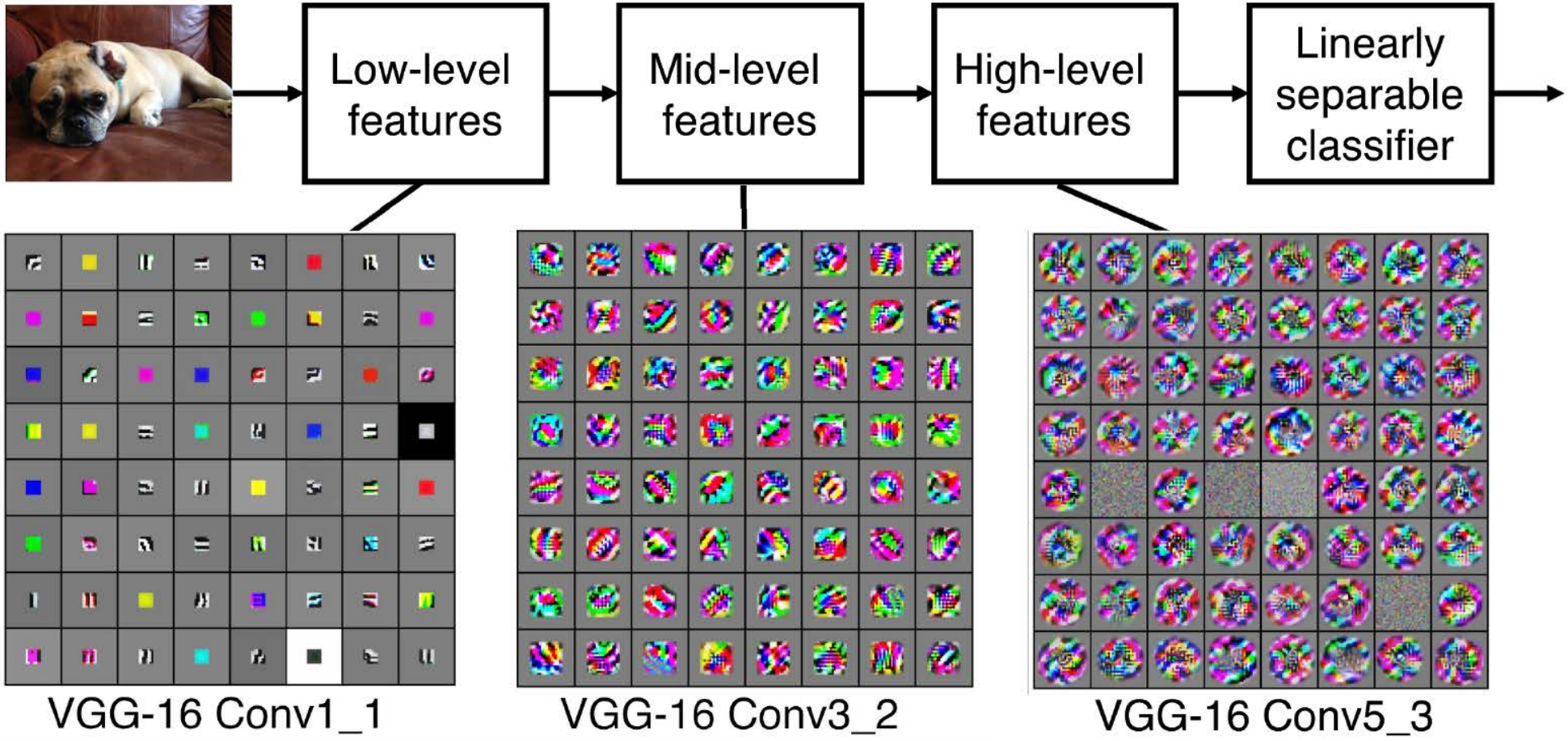
LeNet-5 [1998]

- first CNN for digit recognition
- trained on Modified NIST (MNIST) dataset
- three convolution layers (C1, C3, C5)
  - C5 is in fact also FC
- two down-sampling (avg. pooling) layers (S2, S4)
- one fully connected layer (F6)



Y. LeCun, et al., “Gradient-Based Learning Applied to Document Recognition,” Proc. IEEE, 1998.

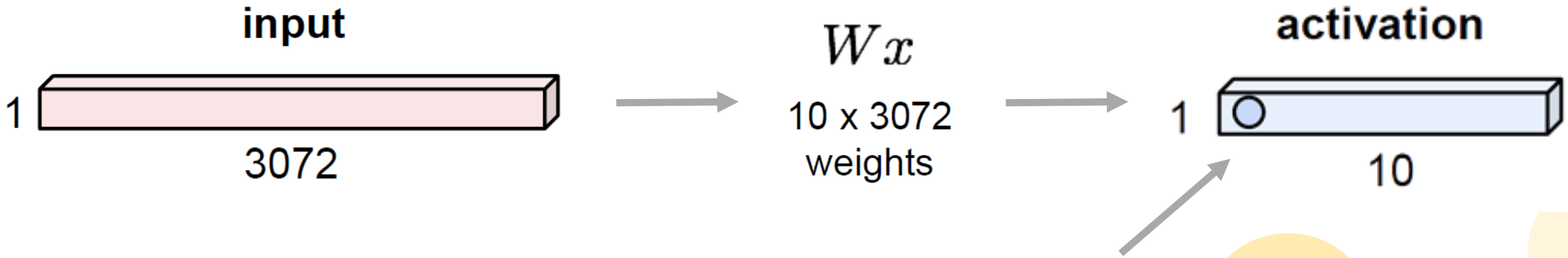
Conv Extracts Features at Different Levels



Fully Connected (FC) Layer

- each neuron looks at the full input volume
- the output of last Conv layer is **flatten**
  - each pixel in output feature maps corresponds to an input neuron in the first FC layer
- FC layer operation can be represented as matrix-vector multiplication ***Wx***
  - matrix ***W*** contains all the weights of a FC layer; vector ***x*** represents the input neurons

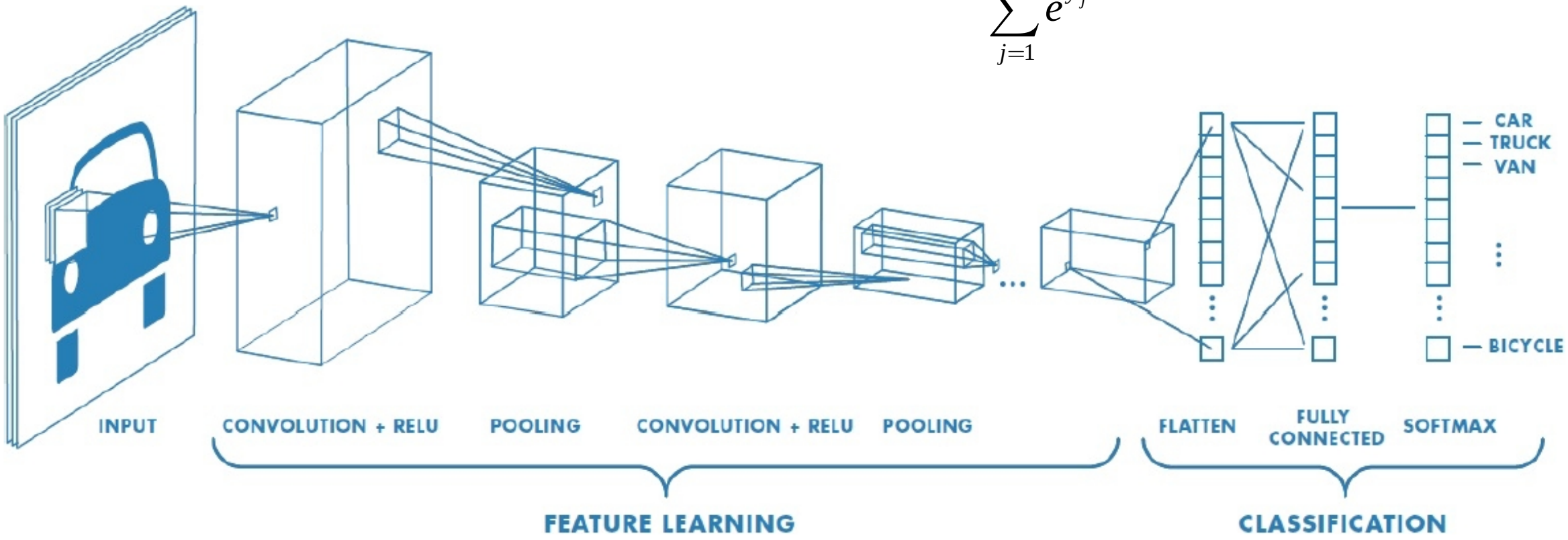
32x32x3 image -> stretch to 3072 x 1



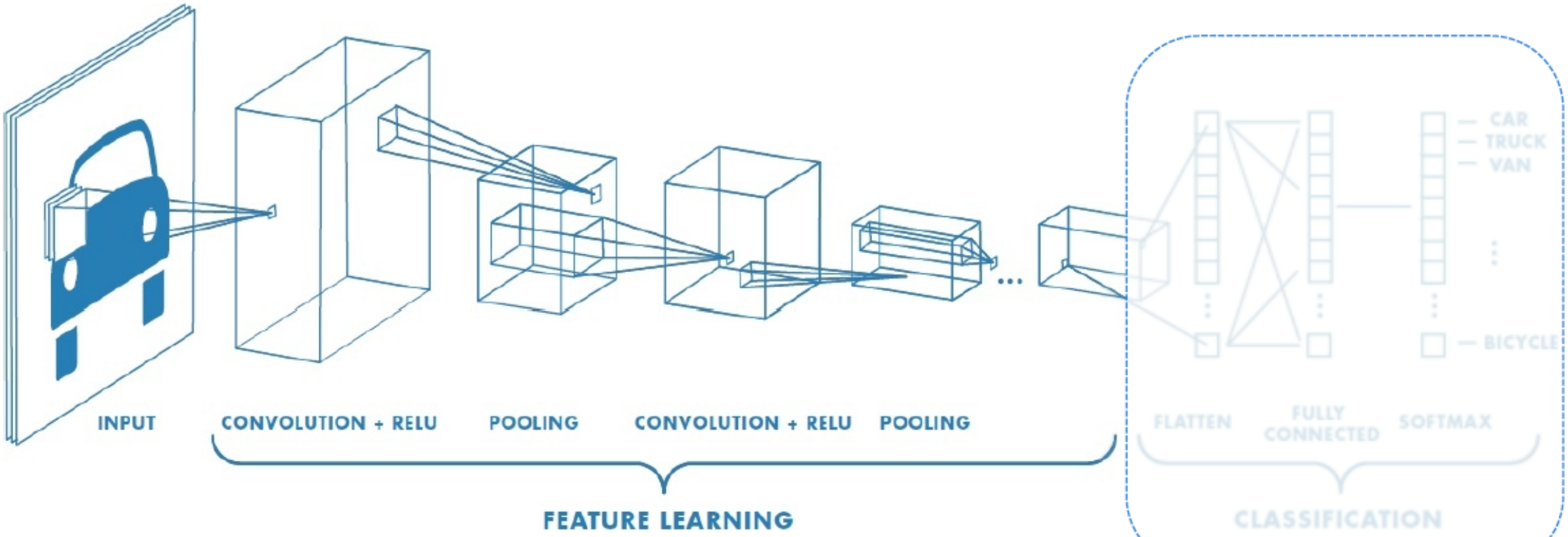
1 number :
The result of taking a dot product between a row of W and the input (a 3072-dimensional dot product)

Complete CNN for Classification

- Conv layers to extract features of different levels
- FC layers to classify
  - express output as a probability via softmax $\hat{y}_i = \frac{e^{y_i}}{\sum_{j=1}^{M} e^{y_j}}$



An Architecture for Many Applications



Different image classification domains
Object detection with R-CNNs
Segmentation with fully convolutional networks
Image captioning with RNNs

Training

Training

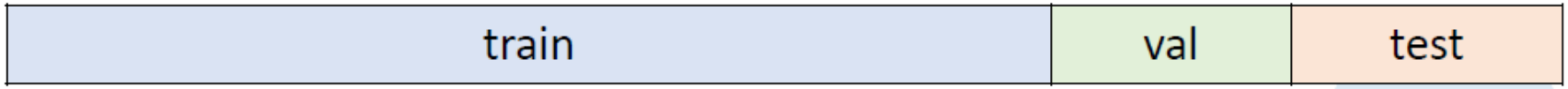
- Machine Learning = Training (Learning) + Inference (Testing, Prediction)
- Training = forward propagation + backward propagation
  - determine weights in the neural newtwork models
- Inference = forward propagation
  - use trained weights to predict for **“new” (unseen)** inputs
- Mini-batch SGD training

  loop:

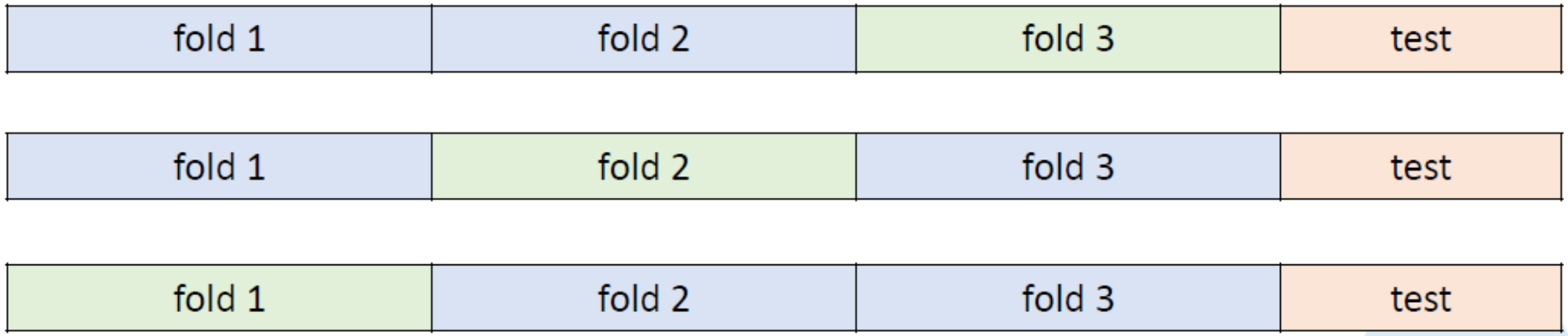
  1. **Sample** a mini-batch of data from training dataset
  2. **forward** propagate it through network, and get loss
  3. **backward** propagate to calculate gradients
  4. **update** parameters using gradients

Training, Validation, Testing (Inference)

- split dataset into data of training + validation + testing
  - training data are used for forward and backward propagation
  - validation data are used for forward propagation to run-time verify accuracy of model currently in training
  - test data are used in inference to test the accuracy of the “trained” model



- if dataset is too small, use cross-validation
  - k-fold cross-validation (e.g., k=3 below)



Deep Neural Network

- neural network model
  - different model structures (CNN, RNN, GAN, ..) and number of layers d

$$f(x; w_{1,2,\cdots,d}) = f_d(f_{d-1}(\cdots f_2(f_1(x; w_1), w_2)\cdots, w_{d-1}), w_d)$$

- learning the model's weights by minimizing loss function L

$$w^* = \underbrace{\arg\min}_{w} \sum_{(x,y)\subseteq(X,Y)} L(y, f_d(x; w_{1,2,\cdots,d}))$$

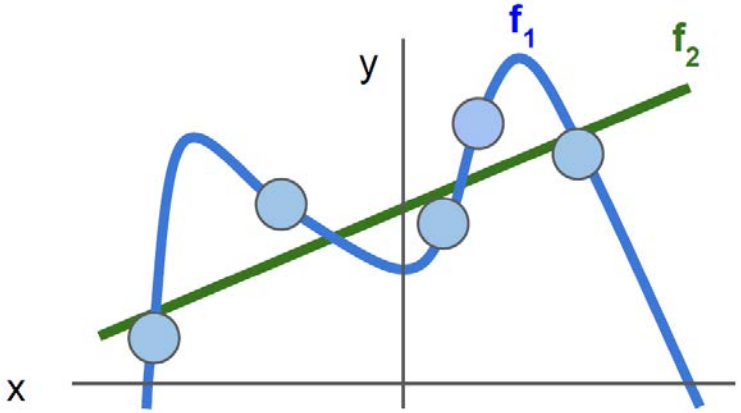
- optimizing with stochastic gradient descent (SGD)

$$w(t+1) = w(t) - \varepsilon \nabla_w L(y, w)$$

Pure Optimization vs. Machine Learning Training

- Pure Optimization
  - Find the optimal solution (usually in explicit forms) based on all the data points
  - e.g. least square (LS) estimation in normal equation

$$w^* = (X^T X)^{-1} X^T y$$



- Machine Learning Training
  - Find optimal parameters based on the training data points
  - not necessarily could be generalized to "new" data points

$$w^* = \arg\min_{w} \sum_{(x,y)\subseteq(X,Y)} L(y, f_d(x; w_{1,2,\cdots,d}))$$

$$L(x, y, w) = \|\hat{y} - y\| + \lambda \|w\|$$

Back Propagation (scalar)

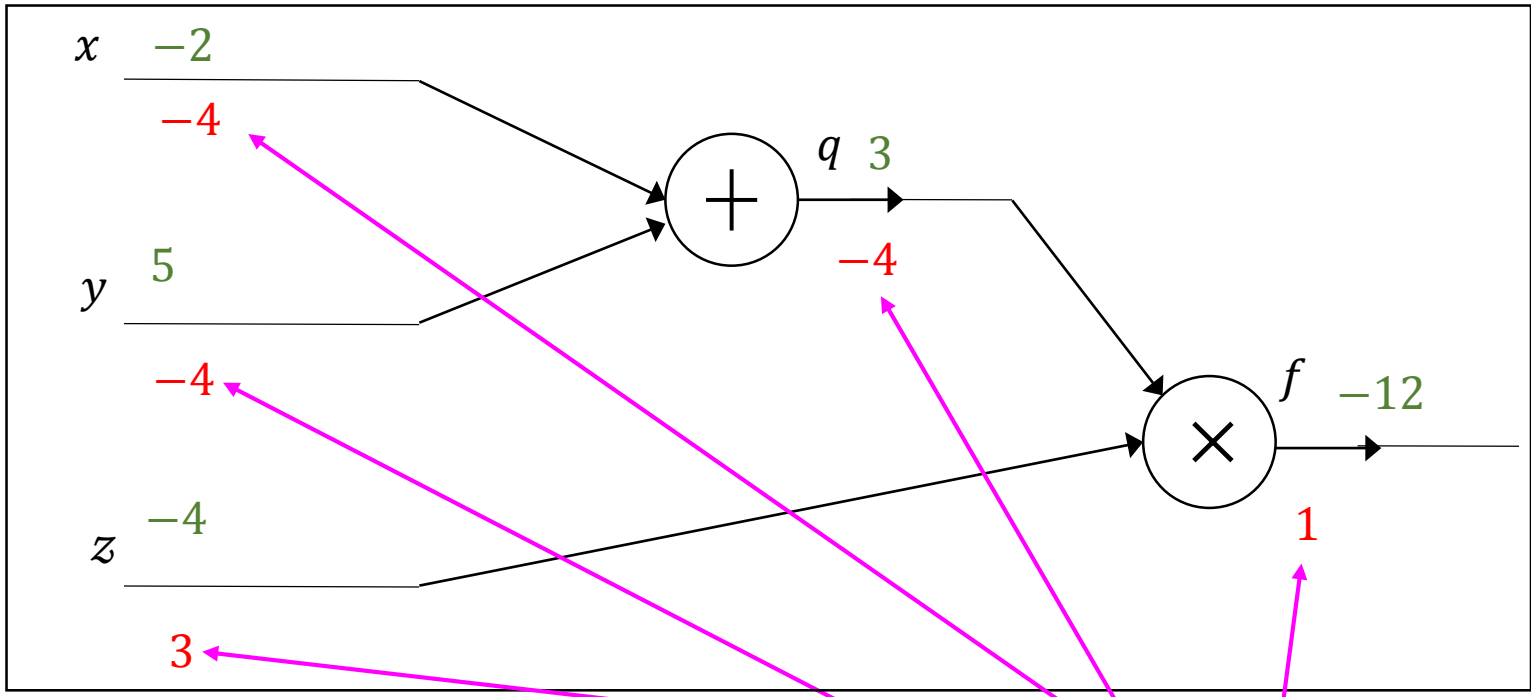
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z= -4

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want : $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain rule: Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

Upstream gradient

Local gradient

Upstream gradient

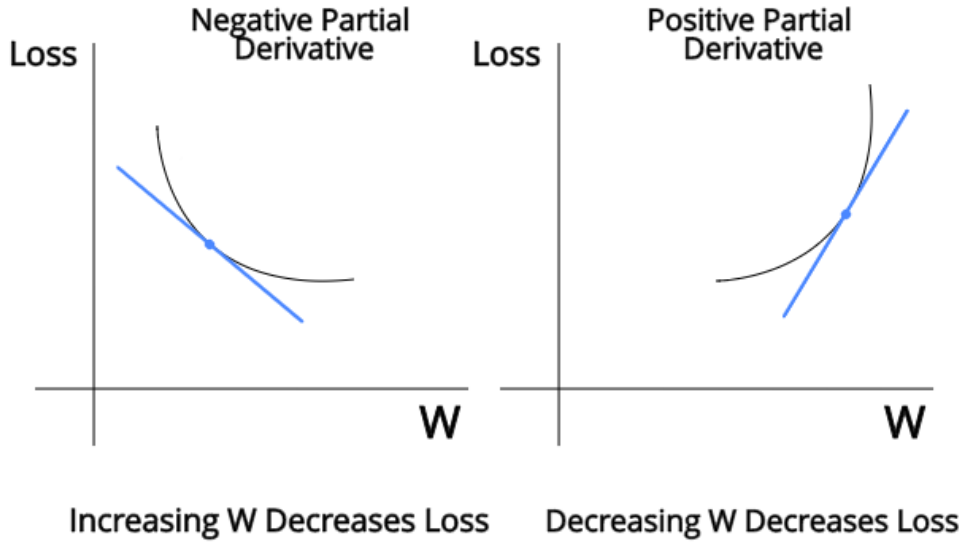
Local gradient

$\frac{\partial f}{\partial y}$ $\frac{\partial f}{\partial f}$ $\frac{\partial f}{\partial z}$

$\frac{\partial f}{\partial q}$ $\frac{\partial f}{\partial x}$

Stochastic Gradient Descent (SGD)

- use a mini-batch of *m* samples to compute the gradient



Algorithm

1 . Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

2. Loop until convergence:

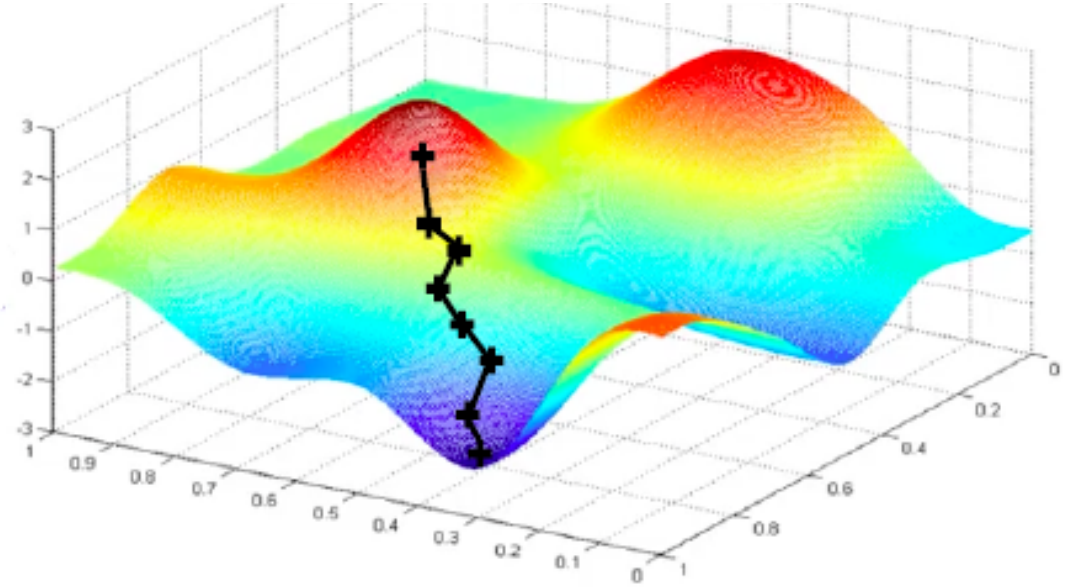
3. Pick batch of *m* data points

4. Compute gradient, $\frac{\partial L(\theta)}{\partial \theta} = \frac{1}{m} \Sigma_{k=1}^{m} \frac{\partial L_k(\theta)}{\partial \theta}$

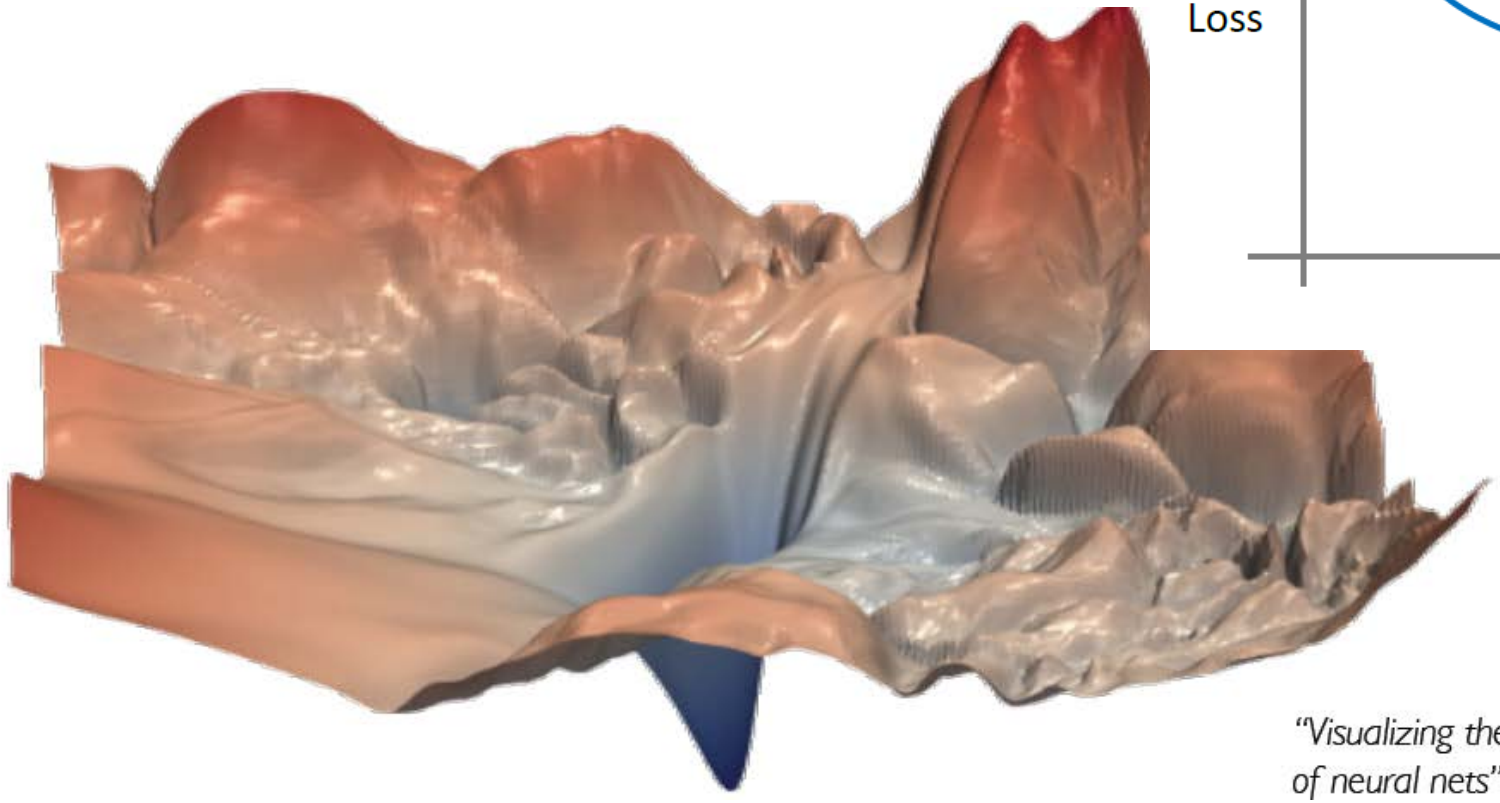
5. Update weights, $\theta \leftarrow \theta - \epsilon \frac{\partial L(\theta)}{\partial \theta}$,

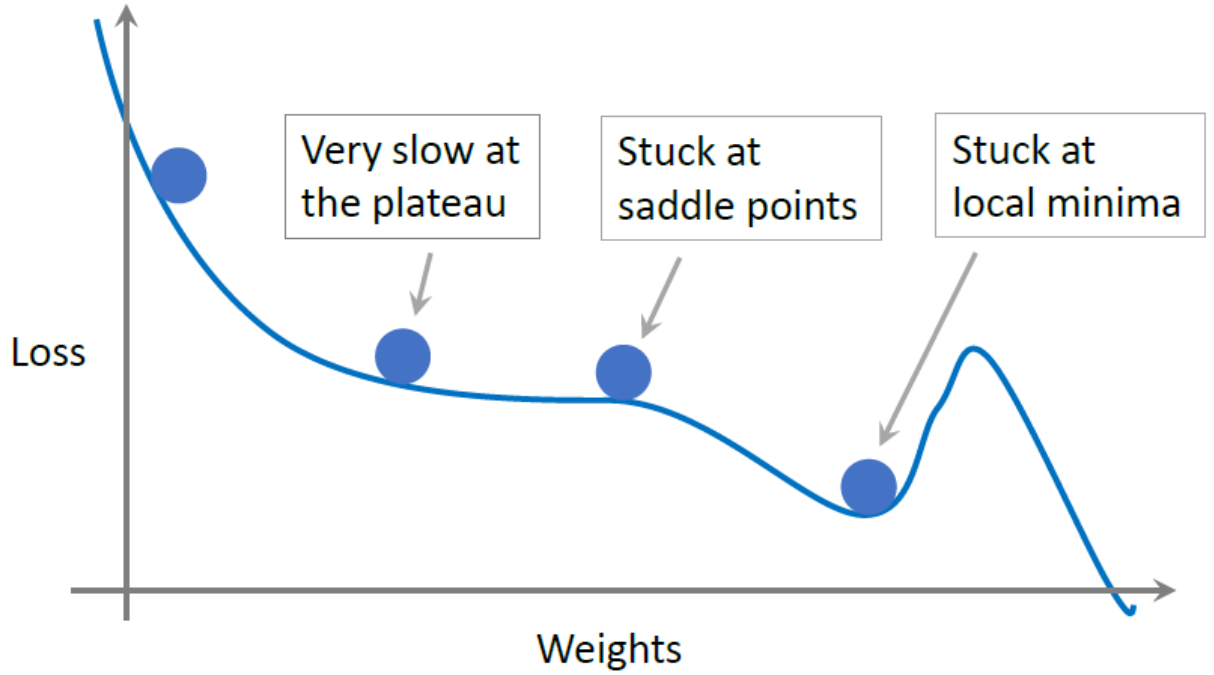
   ϵ is called **learning rate**

6. Return weights

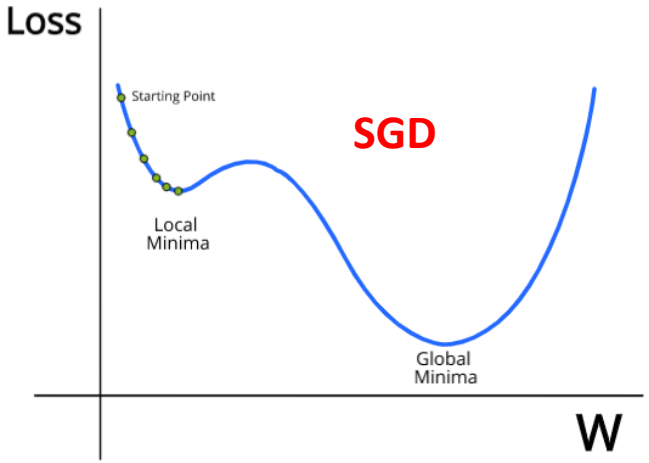


Optimization of Loss Function

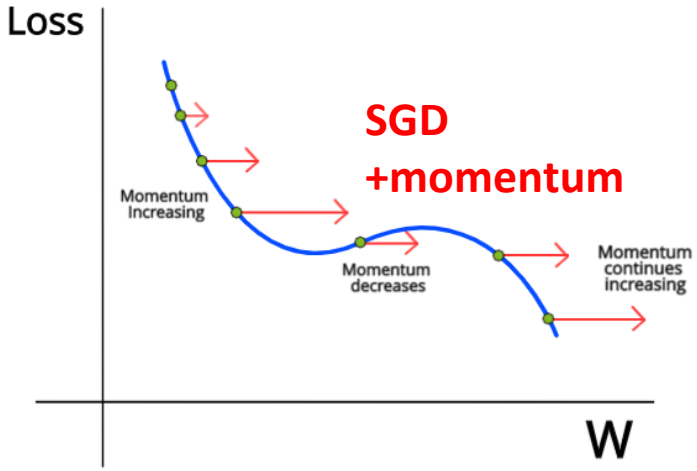




"Visualizing the loss landscape of neural nets". Dec 2017.

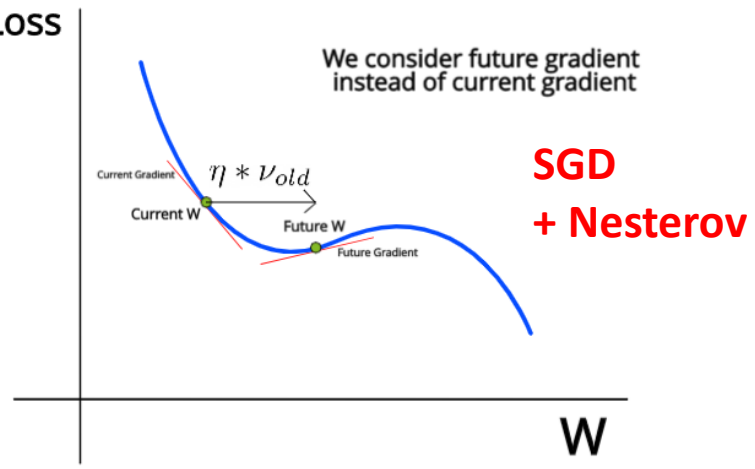


$$W_{new} = W_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$



$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

$$W_{new} = \nu_{new} + W_{old}$$



$$W_{future} = \eta * \nu_{old} + W_{old}$$

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{future})}$$

Adagrad

$$cache_{new} = cache_{old} + (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

RMSProp

$$cache_{new} = \gamma * cache_{old} + (1 - \gamma) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

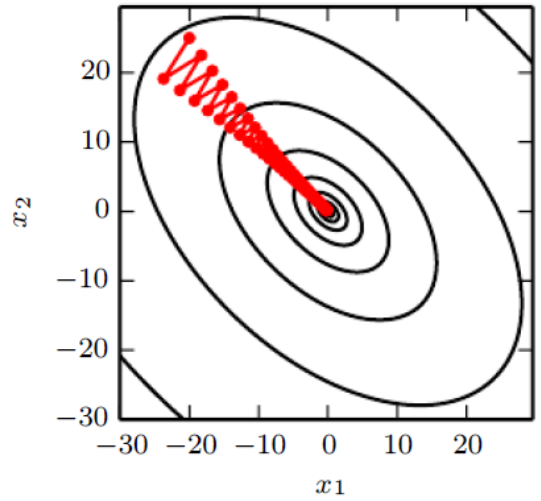
Adam

$$m_{new} = \beta_1 * m_{old} - (1 - \beta_1) * \frac{\partial(Loss)}{\partial(W_{old})}$$

$$cache_{new} = \beta_2 * cache_{old} + (1 - \beta_2) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

$$W_{new} = W_{old} - \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * m_{new}$$

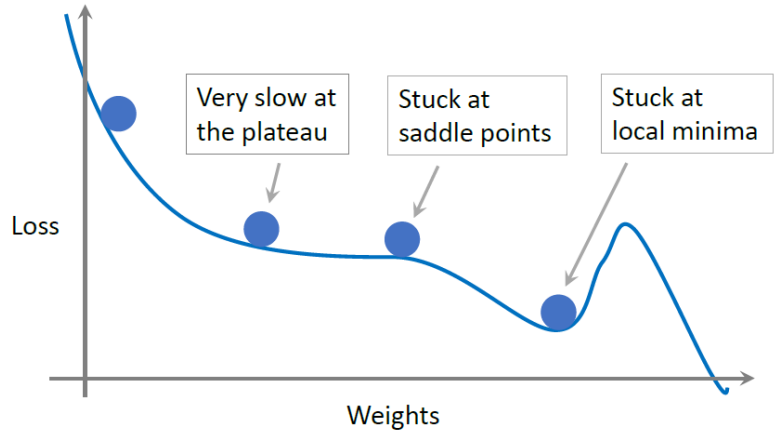
Optimization Algorithms



- **SGD** $\hat{g} \leftarrow +\frac{1}{m}\nabla_\theta \Sigma_i L\left(f\left(x^{(i)};\theta\right), y^{(j)}\right)$ $\theta \leftarrow \theta - \epsilon\hat{g}$
- **SGD+Momentum** $v \leftarrow \alpha v - \epsilon g$ $\theta \leftarrow \theta + v$
- **Nesterov Momentum** $\tilde{\theta} \leftarrow \theta + \alpha v$ $g \leftarrow \frac{1}{m}\nabla_{\tilde{\theta}} \Sigma_i L\left(f\left(x^{(i)};\tilde{\theta}\right), y^{(i)}\right)$
  $v \leftarrow \alpha v - \epsilon g$ $\theta \leftarrow \theta + v$

Local Minima Saddle points

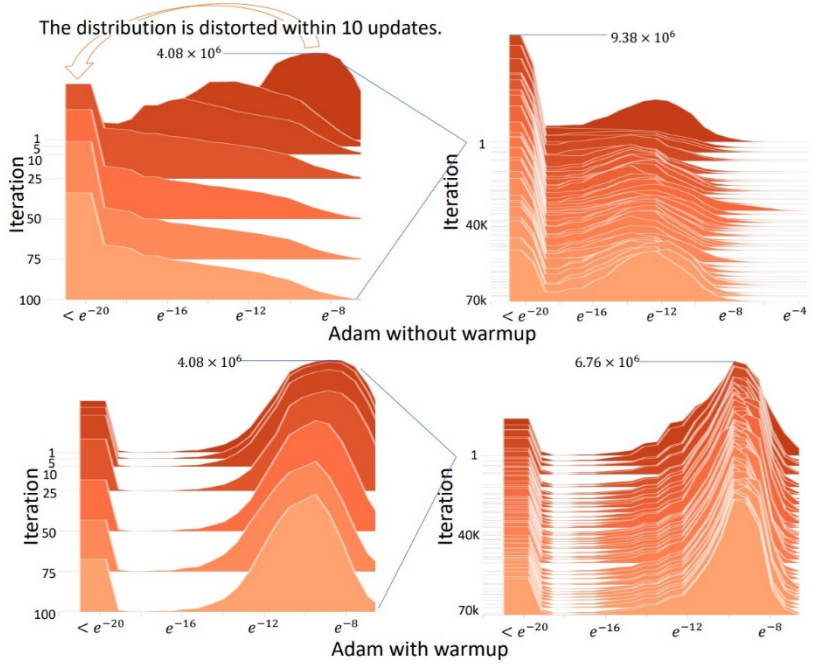
- **AddGrad** $g \leftarrow \frac{1}{m}\nabla_{\tilde{\theta}} \Sigma_i L\left(f\left(x^{(i)};\tilde{\theta}\right), y^{(i)}\right)$ $\boldsymbol{r} \leftarrow \boldsymbol{r} + g \odot g$
  $\Delta\theta \leftarrow -\frac{\epsilon}{\delta+\sqrt{r}} \odot g$ $\theta \leftarrow \theta + \Delta\theta$
- **RMSProp**
  $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho) g \odot g$
  $\Delta\theta = -\frac{\epsilon}{\sqrt{\delta+r}} \odot g$ $\theta \leftarrow \theta + \Delta\theta$

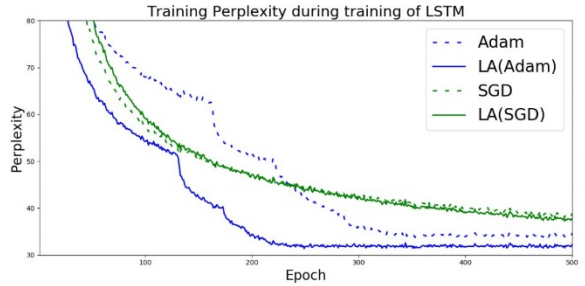


- **Adam**
  $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1-\rho_1) g$ $\tilde{s} \leftarrow \frac{s}{1-\rho_1^t}$
  $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1-\rho_2) g \odot g$ $\tilde{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
  $\Delta\theta = -\epsilon \frac{\tilde{s}}{\sqrt{\tilde{\boldsymbol{r}}} + \delta}$ $\theta \leftarrow \theta + \Delta\theta$

Recent Optimizer (2019)

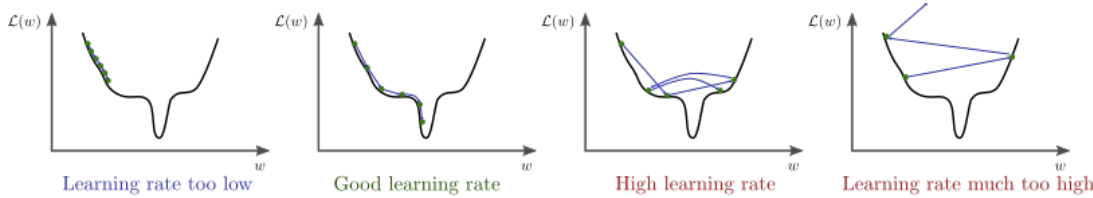
- Rectifier Adam (RAdam)
  - dynamically turns off and/or ‘tamps down’ the adaptive momentum so that it’s not jumping at full speed until the variance of the data settles down.
- LookAhead (LA)
  - maintains two sets of weights and then interpolates between them — in effect it allows a faster set of weights to ‘look ahead’ or explore while the slower weights stay behind to provide longer term stability
  - reduced variance during training, and much less sensitivity to sub-optimal hyper-parameters and reduces the need for extensive hyper-parameter tuning.
- Ranger (RAdam + LA)

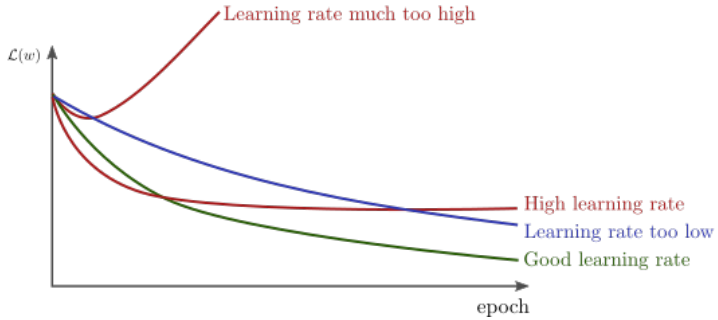




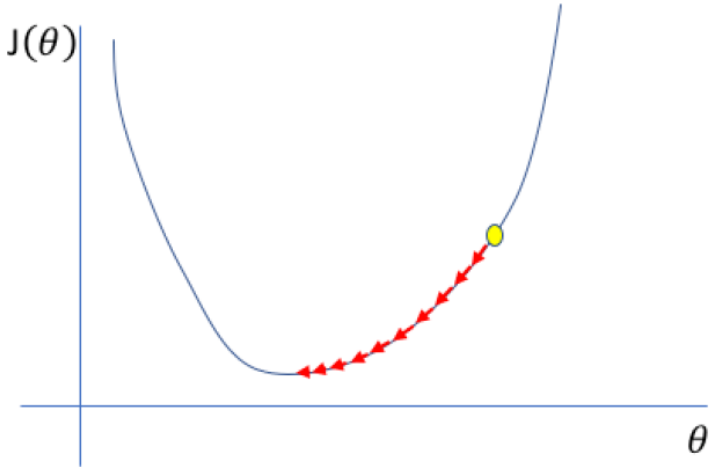
Learning Rate

- too strong: gradients overshoot and bounce
- too weak: too small gradients -> slow training
- learning rate per weight is advantageous
  - some weights are near convergence, others not



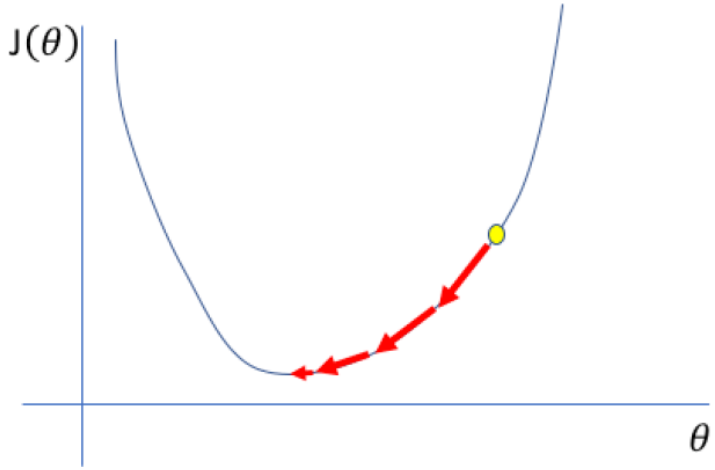


Too low



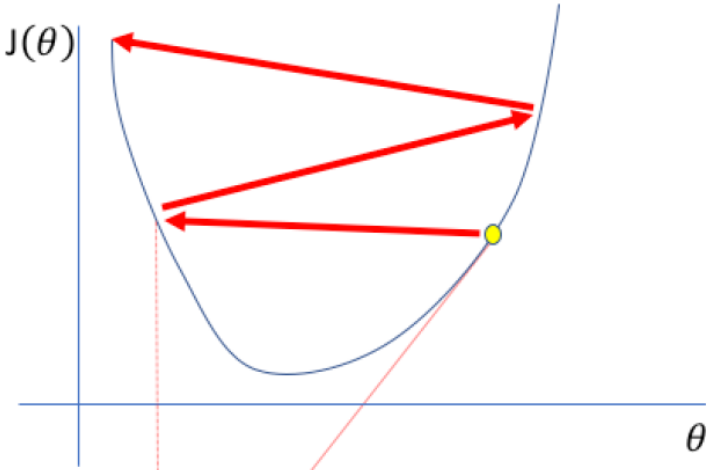
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors

learning rate

- SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as one of the hyper-parameters

SGD

$$\theta \leftarrow \theta - \epsilon \hat{g}$$

Momentum

$$v \leftarrow \alpha v - \epsilon g$$
$$\theta \leftarrow \theta + v$$

AdaGrad

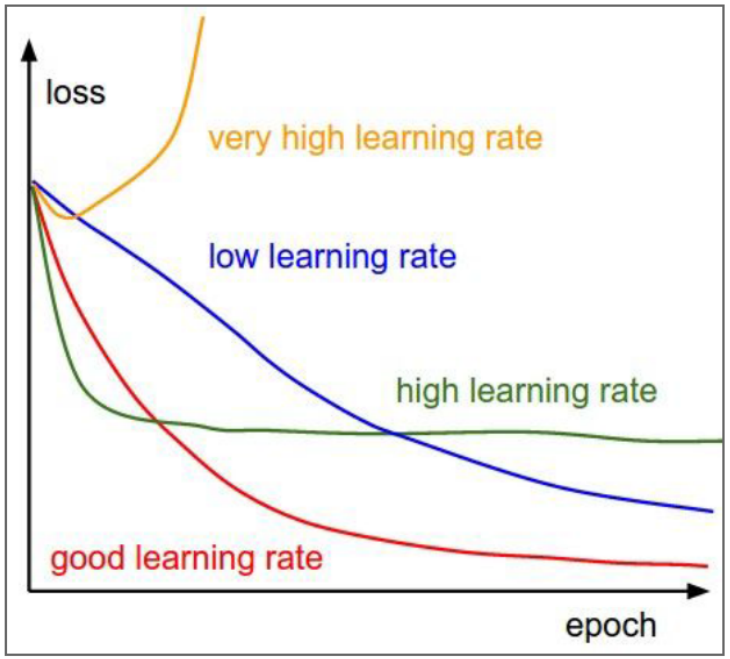
$$\boldsymbol{r} \leftarrow \boldsymbol{r} + g \odot g$$
$$\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$$
$$\theta \leftarrow \theta + \Delta\theta$$

RMSProp

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + g \odot g$$
$$\Delta\theta = -\frac{\epsilon}{\sqrt{\delta + r}} \odot g$$
$$\theta \leftarrow \theta + \Delta\theta$$

Adam

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + g \odot g$$
$$\Delta\theta = -\epsilon \frac{\tilde{s}}{\sqrt{\tilde{\boldsymbol{r}}} + \delta}$$
$$\theta \leftarrow \theta + \Delta\theta$$



=> Learning rate decay over time!

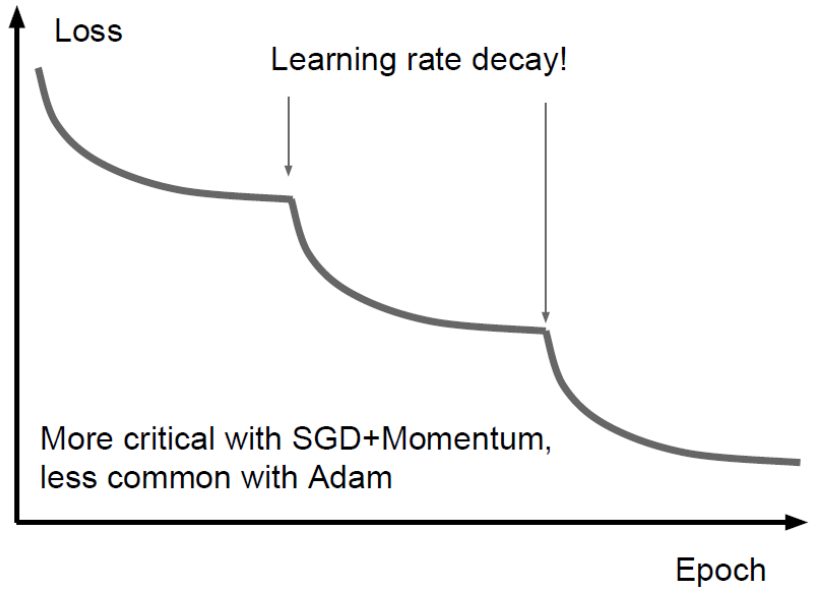
step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

$$\alpha = \alpha_0 \boldsymbol{e}^{-kt}$$

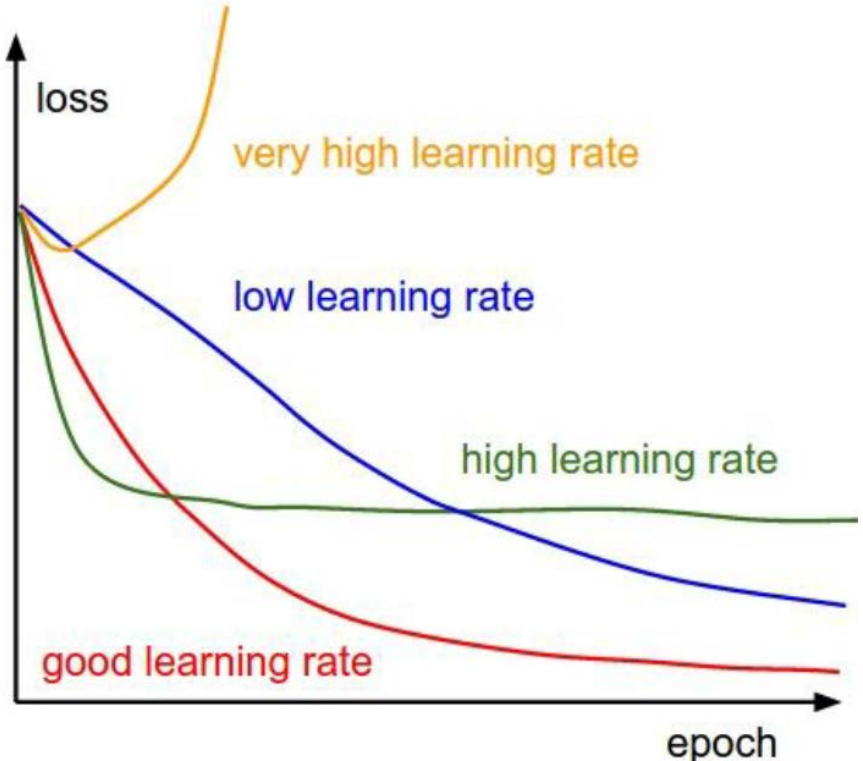
1/t decay:

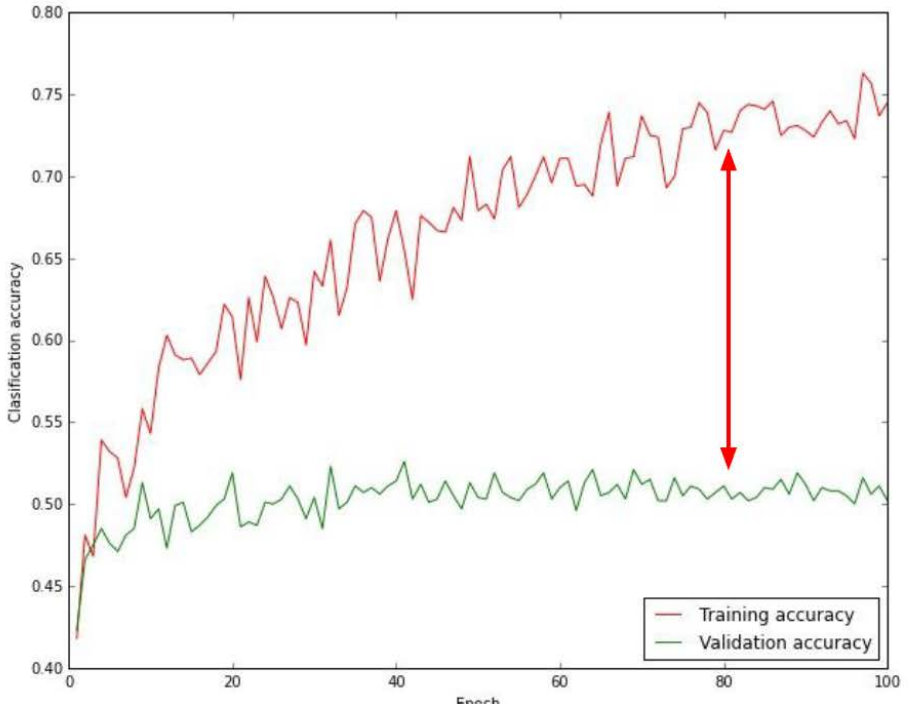
$$\alpha = \alpha_0 / (1 + kt)$$



Hyper-parameter Optimization

- hyper-parameters
  - network architecture
  - learning rate, its decay schedule, update type
  - regularization (L1/L2, dropout strength), weight decay



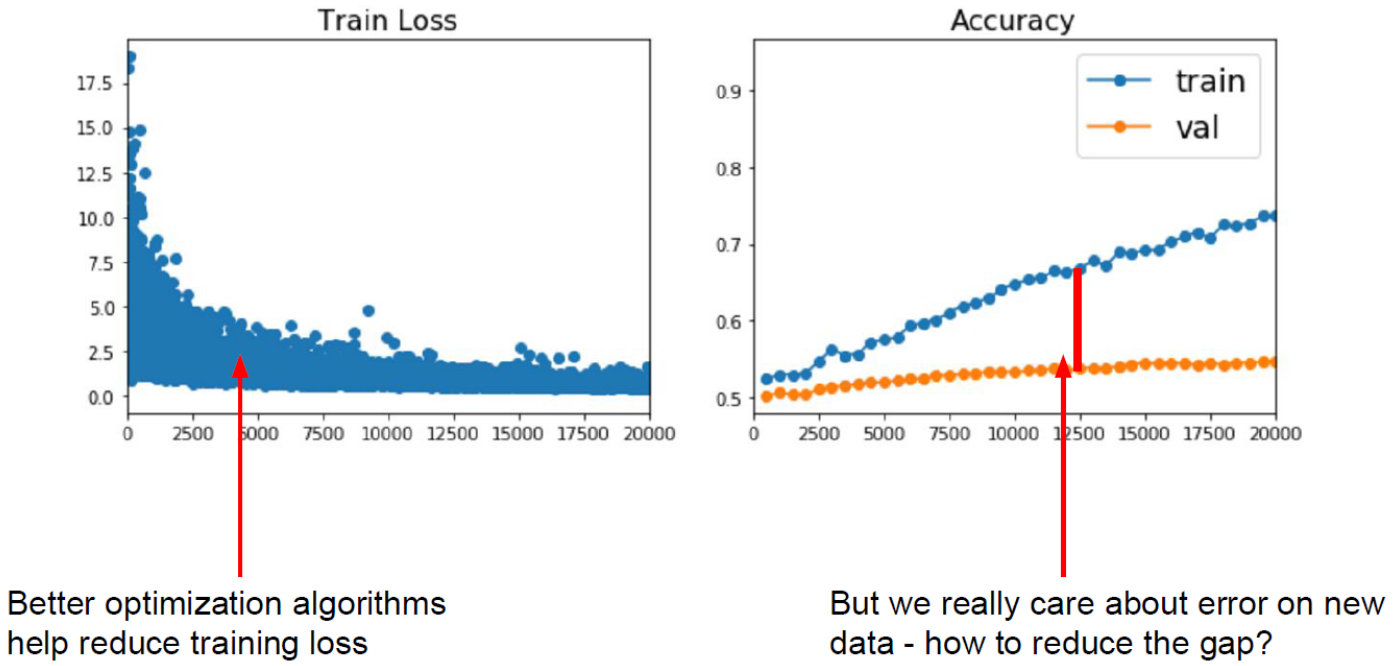


big gap = overfitting
=> increase regularization strength?

no gap
=> increase model capacity?

Summary of Optimization Algorithms

- **Adam** is a good default choice in many cases
- **SGD + Momentum** with learning rate decay often outperforms Adam by a bit, but requires more tuning
- But we care about error rate on “new” data
  - **regularization** methods



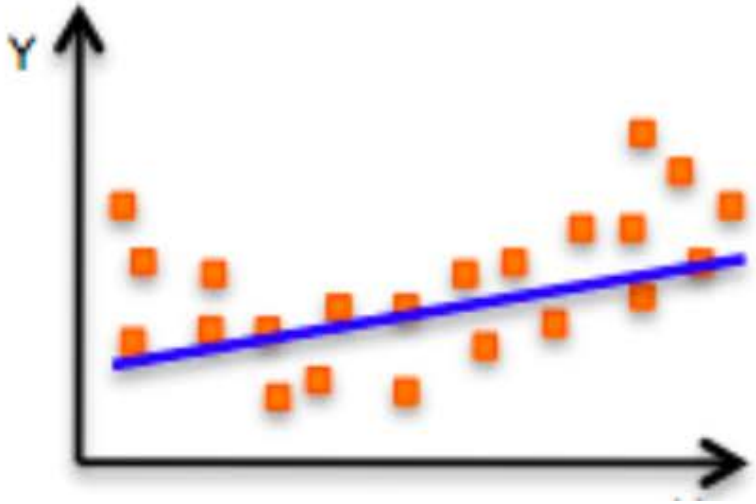
Regularization

- early stop
- L1/L2 regularization term
- dropout
- batch normalization
- data augmentation

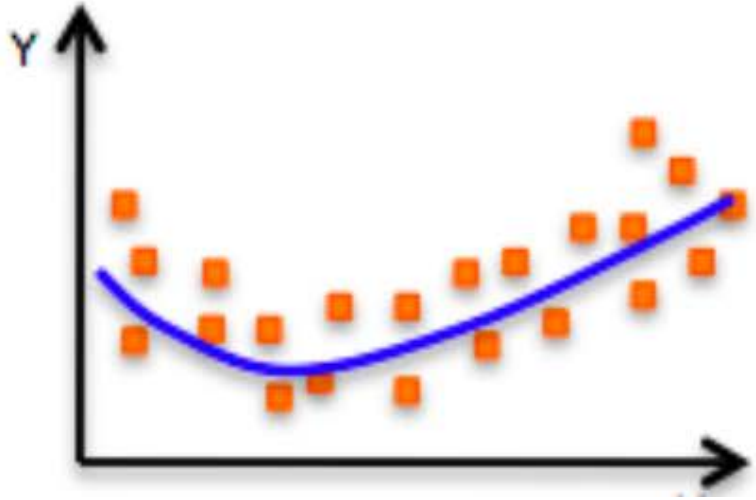
Problem of Overfitting

- too complex CNN models with too small training dataset causes overfitting
  - low accuracy in inference with “new” test data

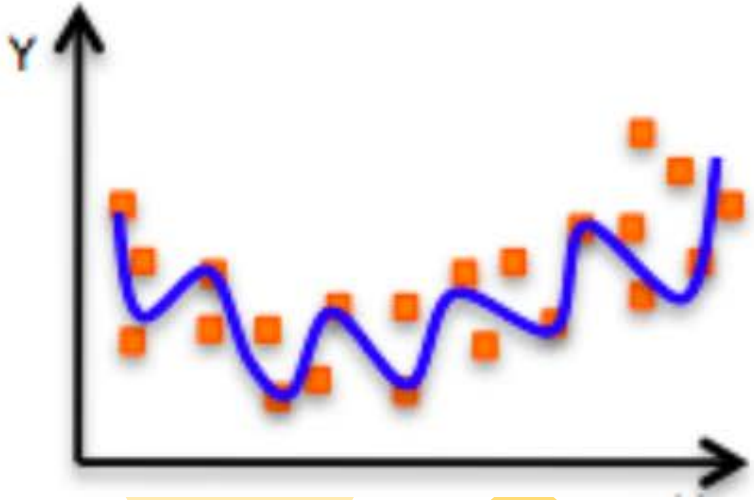
Under-fitting:
model does not have capacity to fully learn the data



Ideal-fitting:



Over-fitting:
too complex, extra parameters does not generalize



Early Stop

- stop training before overfitting

add regularization term to loss function

- proper weight regularization to avoid overfitting

$$L = \frac{1}{N}\Sigma_{i=1}^{N}\Sigma_{j\neq y_i} max\left(0, f(x_i\ ;\ W)_j - f(x_i\ ;\ W)_{y_i} + 1\right) + \boxed{\lambda R(W)}$$

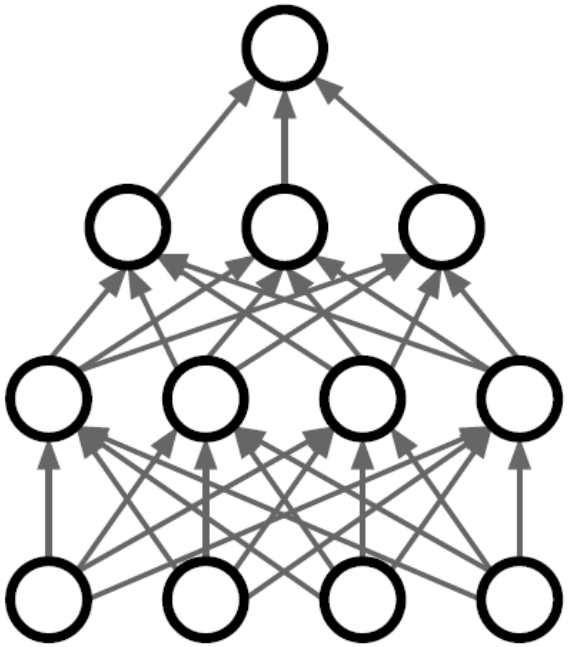
In common use :

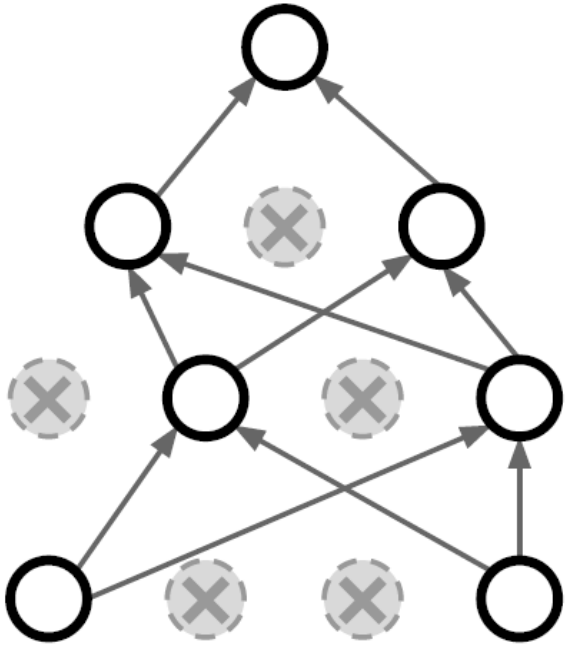
L2 regularization	$R(W) = \Sigma_k \Sigma_l W_{k,l}^2$ (Weight decay)
L1 regularization	$R(W) = \Sigma_k \Sigma_l \left\|W_{k,l}\right\|$
Elastic net (L1 + L2)	$R(W) = \Sigma_k \Sigma_l \beta W_{k,l}^2 + \left\|W_{k,l}\right\|$

dropout

- force network to not rely on any one node

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyper-parameter; 0.5 is common

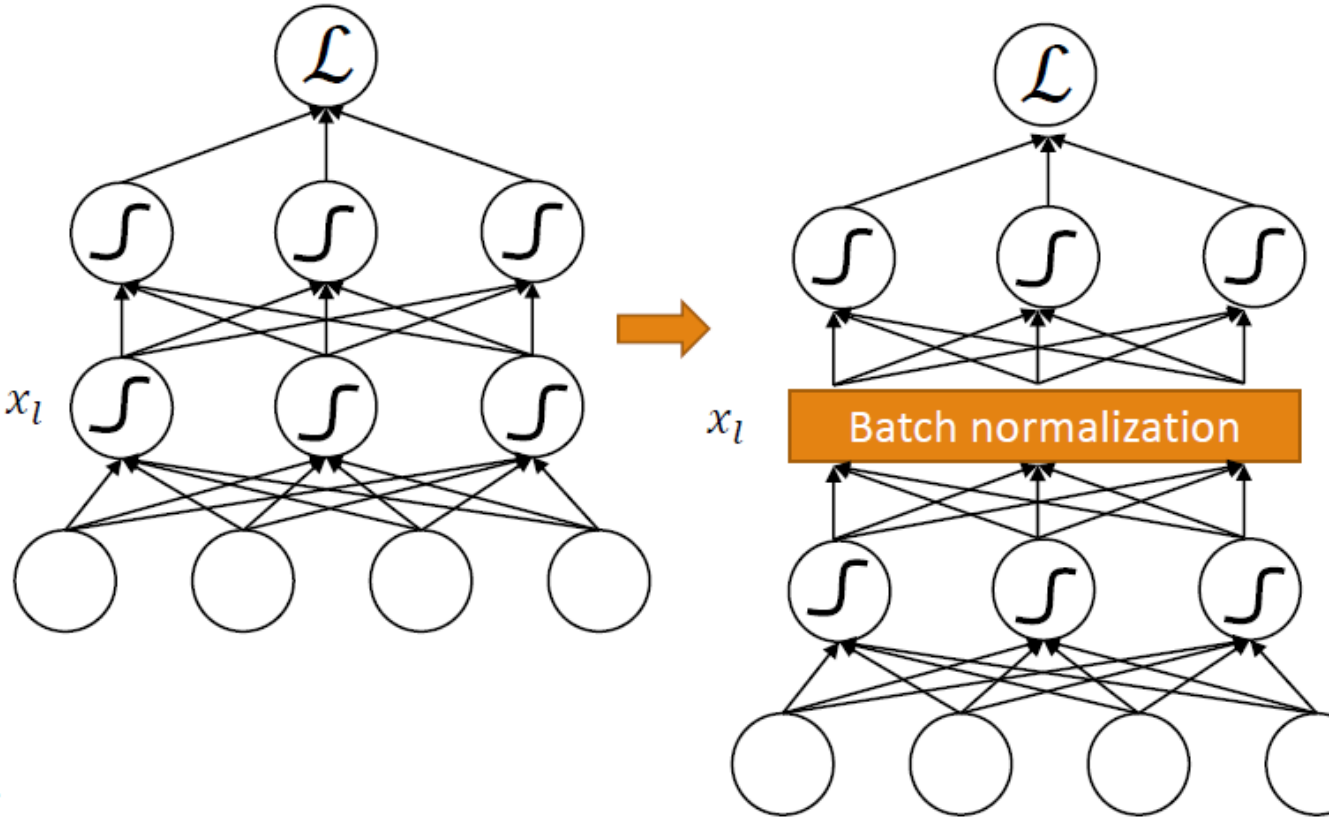


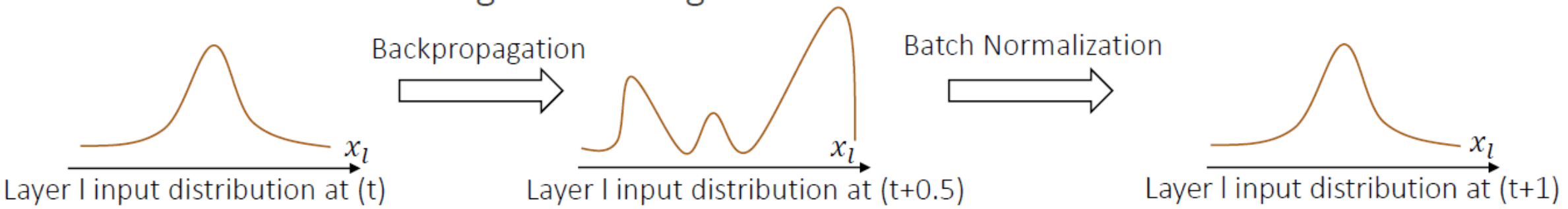


Srivastava et al, “Dropout : A simple way to prevent neural networks from overfitting”, JMLR 2014

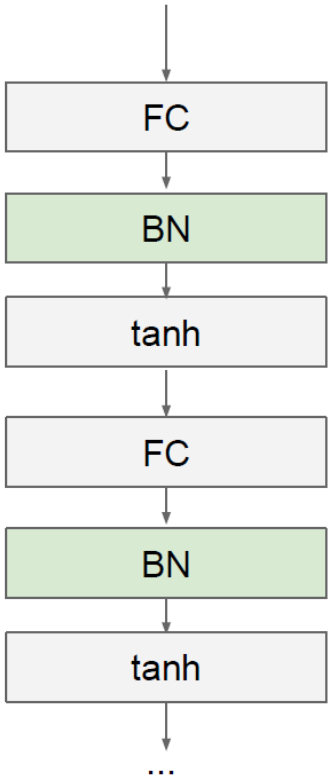
Batch Normalization

- Weights change → the distribution of the layer inputs changes per round
- Normalize the layer inputs with batch normalization
  - Roughly speaking, normalize x_l to $N(0,1)$, then rescale
  - Rescaling is so that the model decides itself the scaling and shifting





Batch Normalization (BN)



Usually inserted
after Fully Connected or
Convolutional layers,

and before nonlinearity.

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Input: Values of x over a mini-batch: $\boldsymbol{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

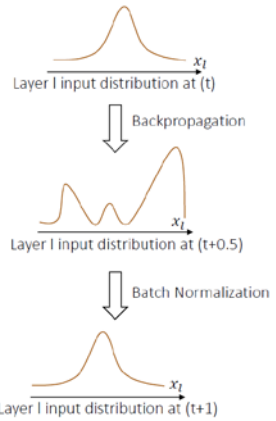
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\boldsymbol{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{//mini-batch mean}$$

$$\sigma^2_{\boldsymbol{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}\left(x_i - \mu_{\boldsymbol{B}}\right)^2 \quad \text{//mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\boldsymbol{B}}}{\sqrt{\sigma^2_{\boldsymbol{B}} + \varepsilon}} \quad \text{// normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \quad \text{//scale and shift}$$

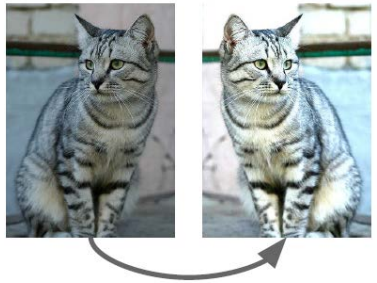


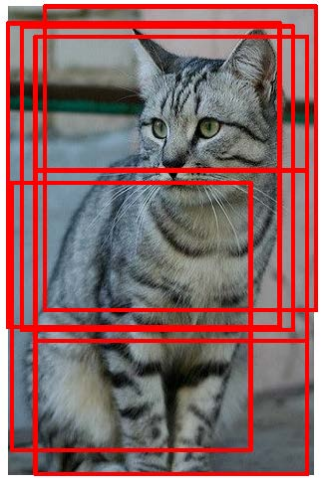
Note: at test time BatchNorm layer functions differently:

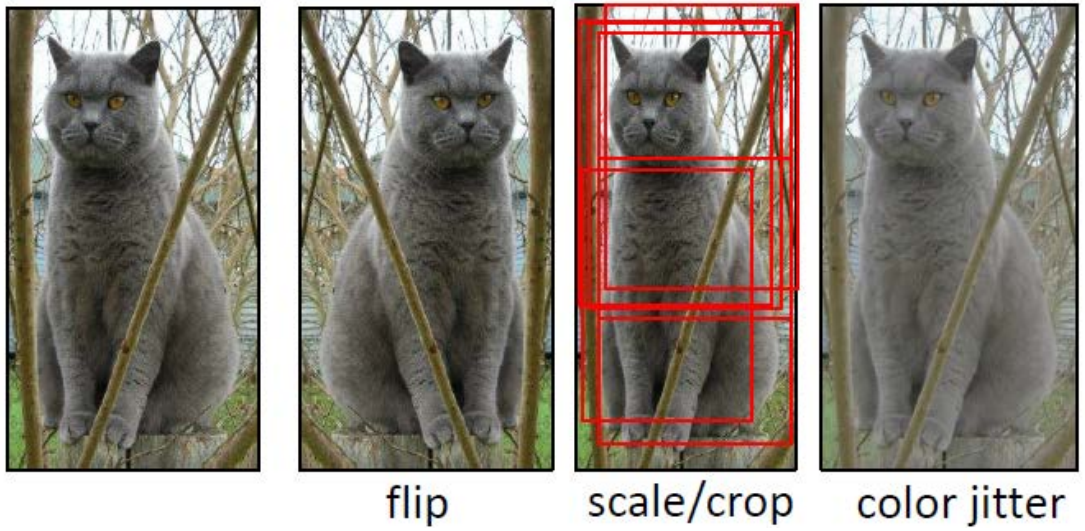
The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.
(e.g. can be estimated during training with running averages)

data augmentation

- horizontal flip

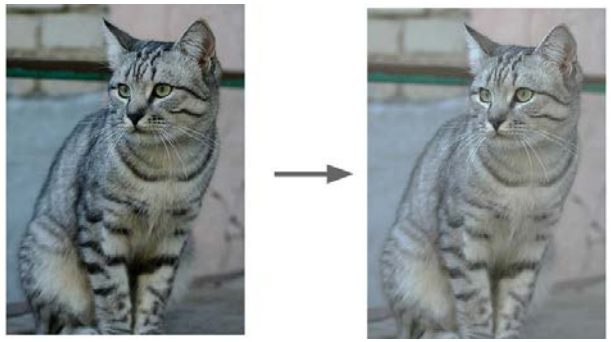






- random crops and scales

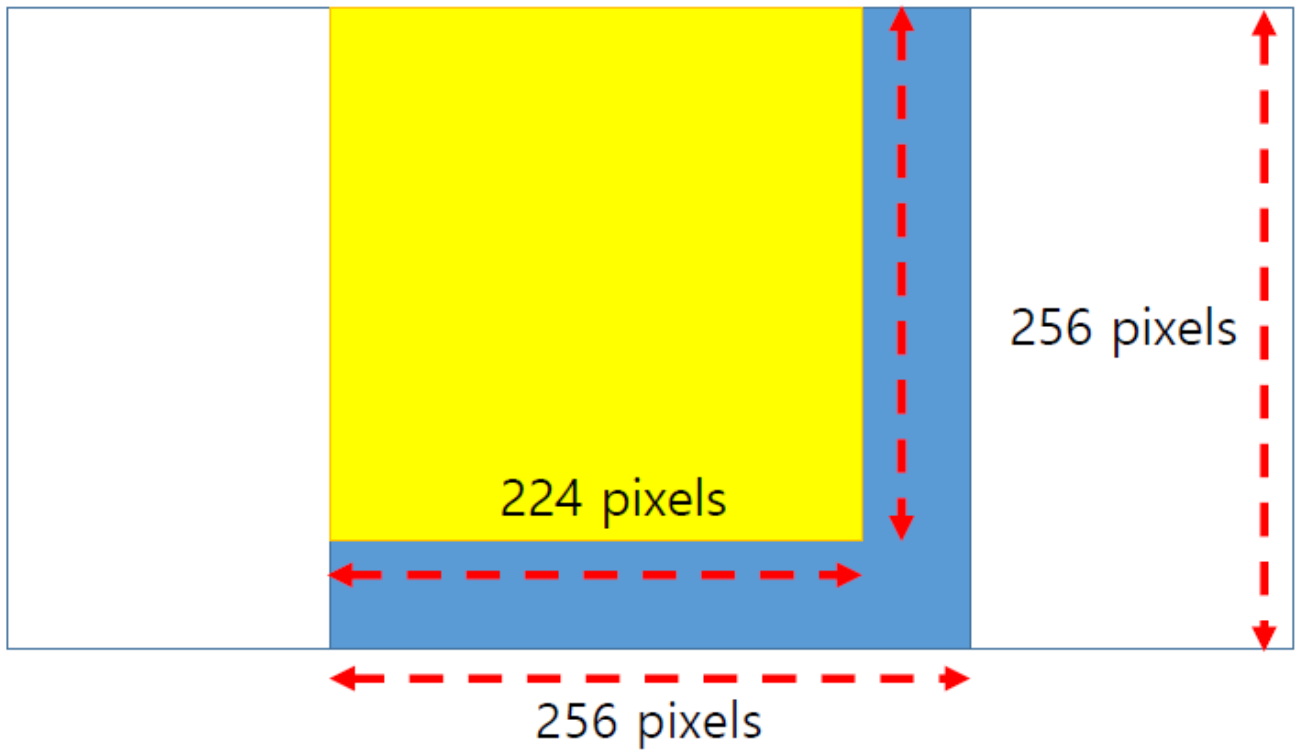
- color jitters

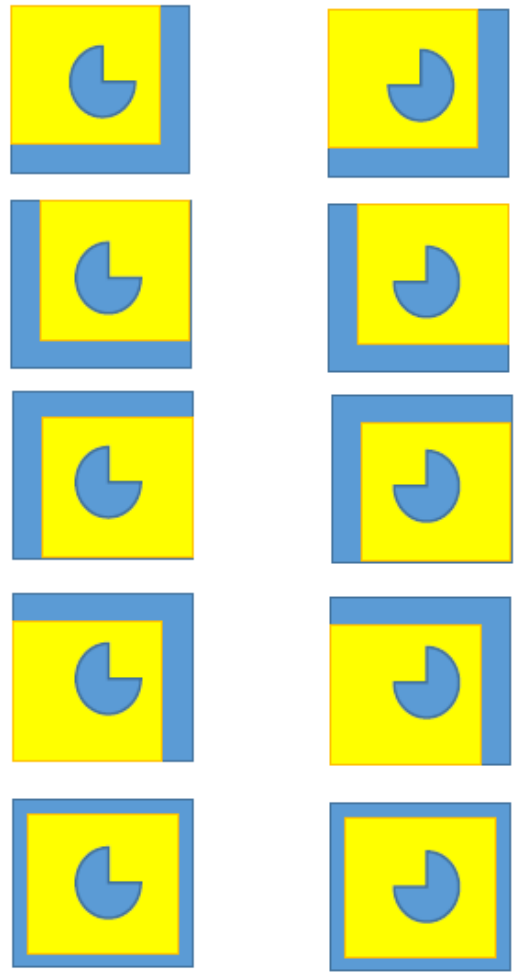


- translation, rotation, stretching, shearing, lens distortion, …

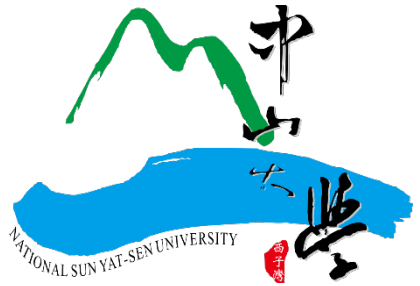
Data Augmentation in AlexNet

- Given an original image, first resize it to fit 256 pixel to the smaller side
- Crop 10 224x224 patches (4 corner and 1 center crops)
- Prediction based on the average of 10 softmax outputs





Transfer Learning

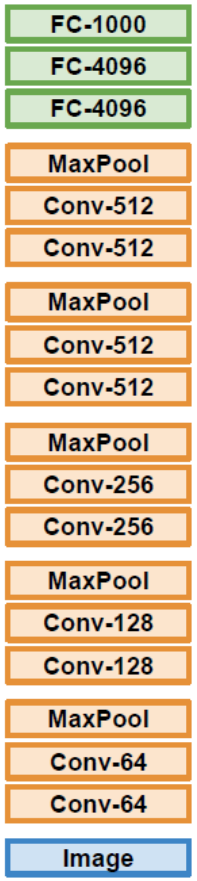


transfer learning summary

Transfer Learning with CNNs

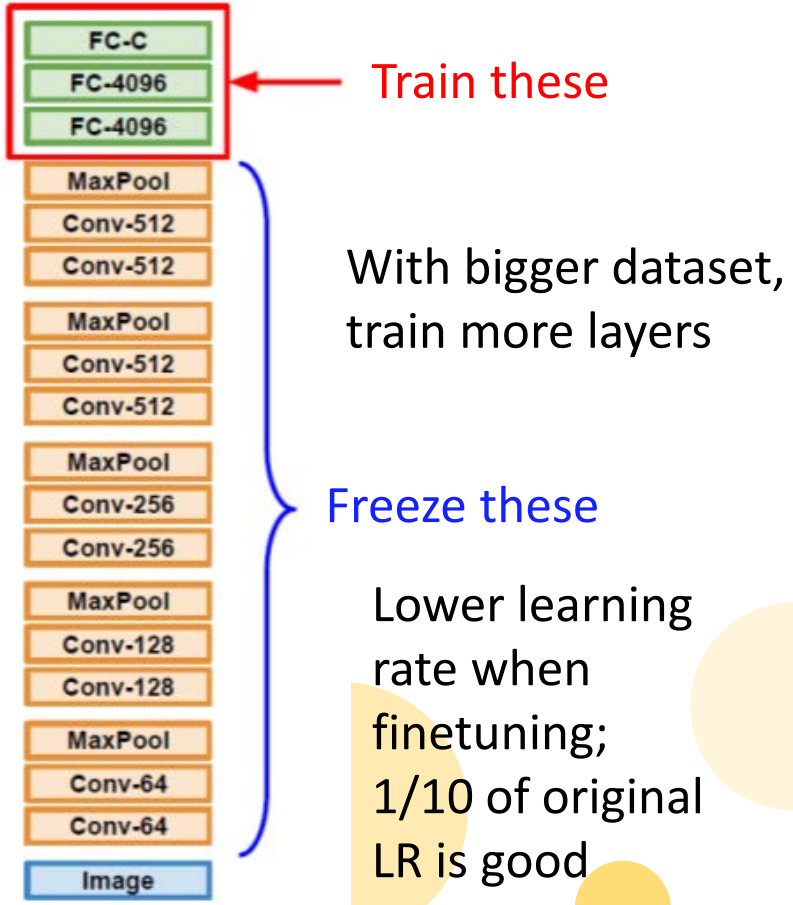
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1.Train on Imagenet



2.Small Dataset (C classes)

3.Bigger dataset

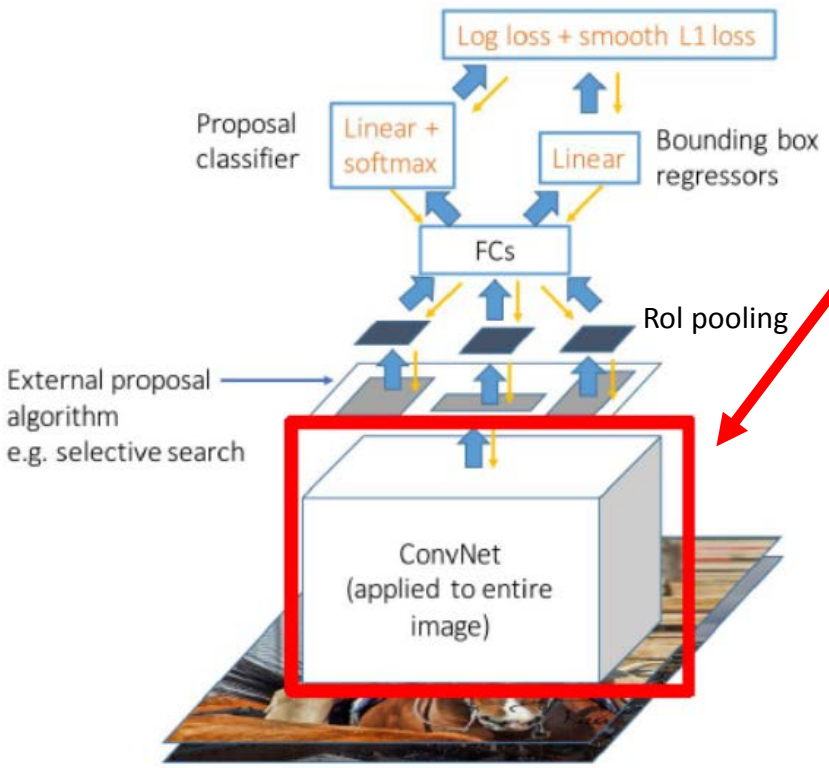


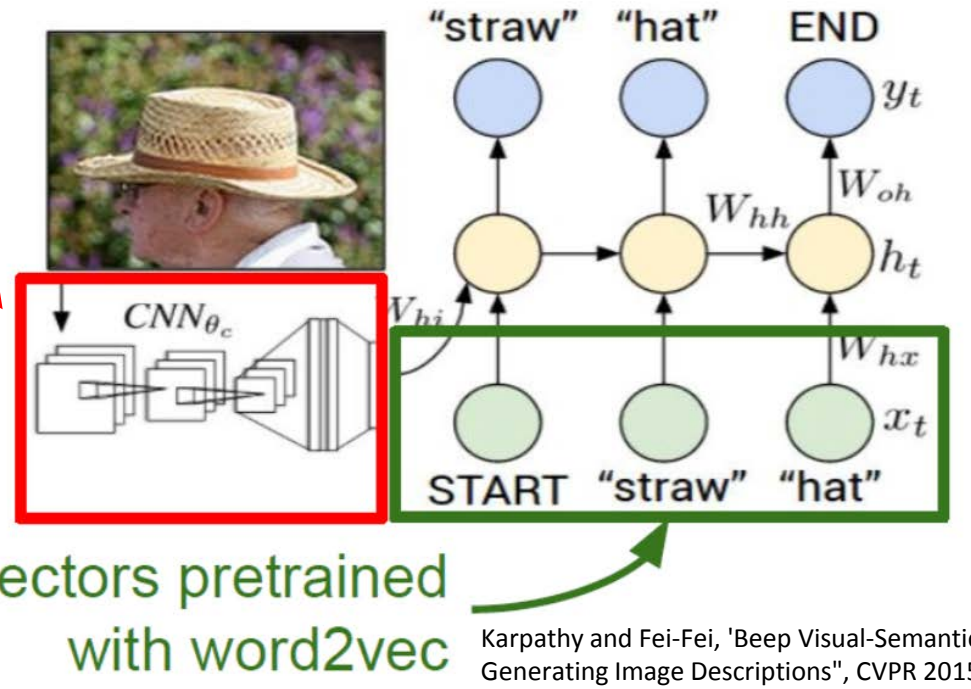
Transfer learning with CNNs is common

Object Detection
(Faster R-CNN, YOLO)

CNN pre-trained
on ImageNet

Image Captioning: CNN + RNN





Word vectors pretrained
with word2vec

Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015.Reproduced with permission.

Karpathy and Fei-Fei, 'Beep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Popular Datasets

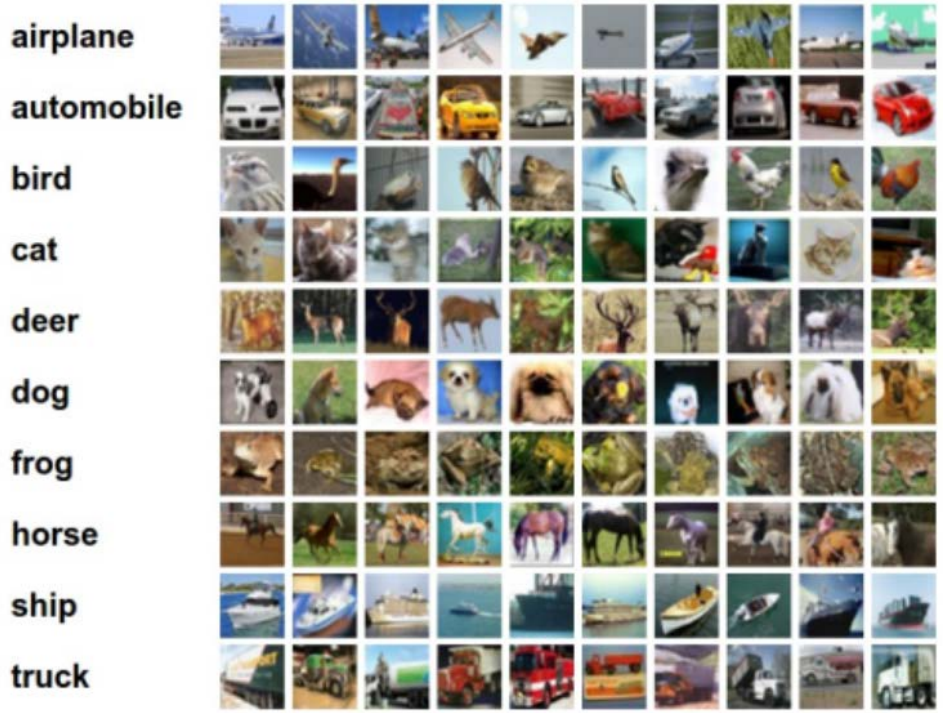
- MNIST
- CIFAR-10/100
- ImageNet
- VOC
- COCO (Microsoft)
- SVHN (Google)
- more datasets from Kaggle (acquired by Google)
- …

dataset	image size	# of samples	# of categories	image types	training set	test set
MNIST	28x28 gray	70K	10	digits	60K	10K
ImageNet	256x256 color	> 15 M	> 22,000	vision	1.2M	150K
Caltech-101	300x200 color	>9K	101	vision	5K	3K
CIFAR-10/100	32x32 color	6 K	10, 100	vision	60K	10K
PASCAL VOC	500*334 color	>10K	21	vision	10K	5K
LabelMe (MIT)	-	>2K	7	vision	2K	1K
SVHN	32x32 color	>70K	10	s	600K	26K
COCO	-	330K	80	vision	10K	5K
CompCars	-	136K	-	cars	-	-
Stanford Cars	-	16K	196 car classes	cars	8K	8K
KITTI	64*32 color	>15K	5	pedestrian	>7K	>7K
INRIA	64*128 color	>0.8K	1	pedestrian	614	288

MNIST, CIFAR-10, ImageNet Datasets

- MNIST: 10 categories, 60,000 training images of 28x28, 10,000 test images
- CIFAR-10: 10 catogories, 60,000 training images of 32x32x3, 10,000 test images
- ImageNet: 1000 categories, 1,200,000 training images of 256x256x3, 150,000 test images

MNIST: handwritten digits



ImageNet:
22K categories. 14M images.

Demo of Training + Classification

- Training (MNIST, CIFAR-10): https://cs.stanford.edu/people/karpathy/convnetjs

Classify **MNIST** digits with a Convolutional Neural Network

Interactively regress toy 1-D data

Classify **CIFAR-10** with Convolutional Neural Network

Train an MNIST digits Autoencoder