

# Neural Networks and Optimization

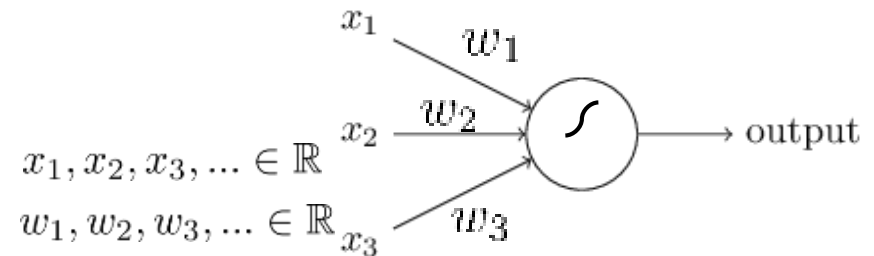
Sourangshu Bhattacharya

IIT Kharagpur

<http://cse.iitkgp.ac.in/~sourangshu/>

# Neural Network Basics

- Given several **inputs**:  
and several **weights**:  
and a **bias** value:



- A neuron produces a single output:  $b \in \mathbb{R}$

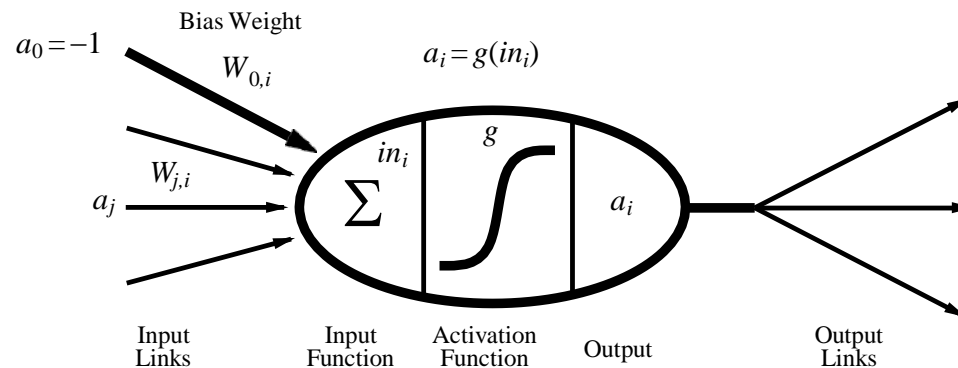
$$o_1 = s(\sum_i w_i x_i + b)$$
$$\sum_i w_i x_i + b$$

- This sum is called the **activation** of the neuron
- The function  $s$  is called the **activation function** for the neuron
- The weights and bias values are typically initialized randomly and learned during training

# McCulloch–Pitts “unit”

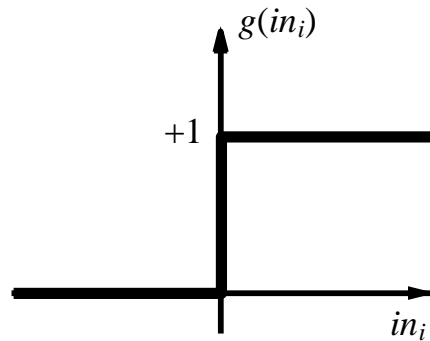
Output is a “squashed” linear function of the inputs:

$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$

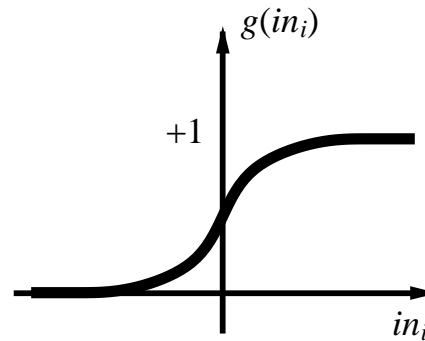


A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

# Activation functions



(a)



(b)

(a) is a **step function** or **threshold function**

(b) is a **sigmoid function**  $1/(1 + e^{-x})$

Changing the bias weight  $W_{0,i}$  moves the threshold location

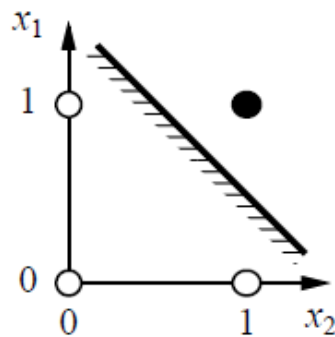
# Expressiveness of perceptrons

Consider a perceptron with  $g = \text{step function}$  (Rosenblatt, 1957, 1960)

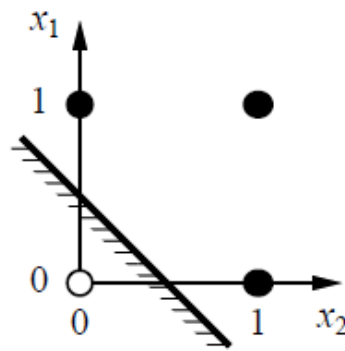
Can represent AND, OR, NOT, majority, etc., but not XOR

Represents a **linear separator** in input space:

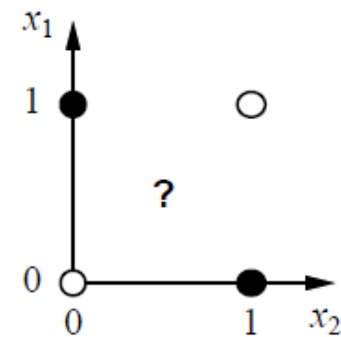
$$\sum_j W_j x_j > 0 \quad \text{or} \quad \mathbf{W} \cdot \mathbf{x} > 0$$



(a)  $x_1$  **and**  $x_2$



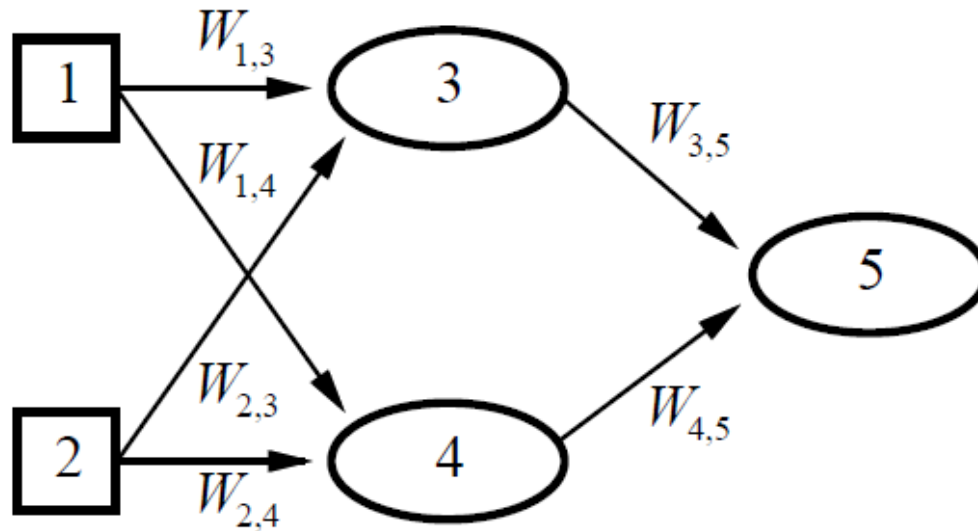
(b)  $x_1$  **or**  $x_2$



(c)  $x_1$  **xor**  $x_2$

Minsky & Papert (1969) pricked the neural network balloon

# Feed forward example



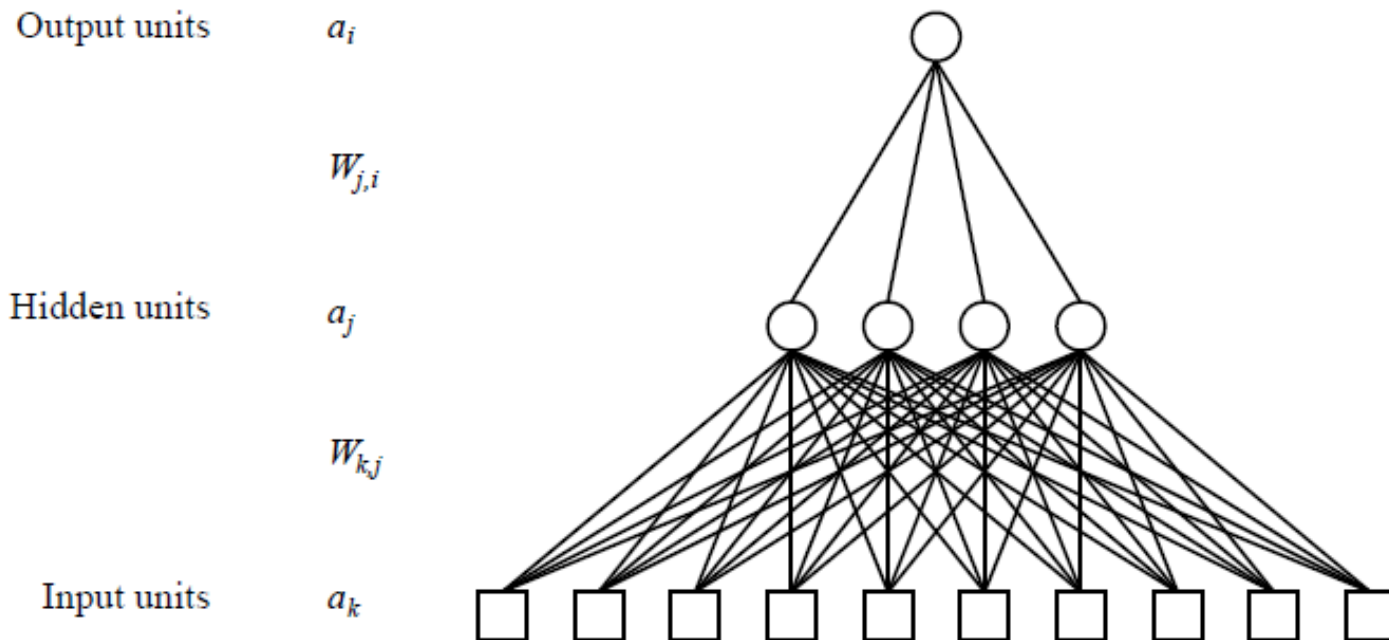
Feed-forward network = a parameterized family of nonlinear functions:

$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

Adjusting weights changes the function: do learning this way!

# Feed Forward Neural Networks

Layers are usually fully connected;  
numbers of **hidden units** typically chosen by hand



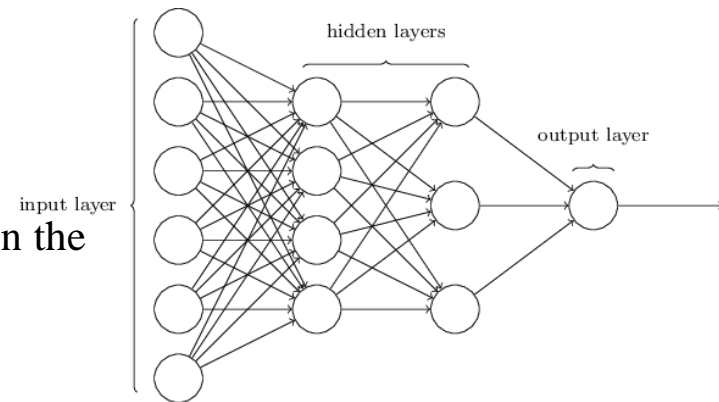
# Hidden-Layer

- The hidden layer ( $L_2$ ,  $L_3$ ) represent learned non-linear combination of input data
- For solving the XOR problem, we need a hidden layer
  - some neurons in the hidden layer will activate only for some combination of input features
  - the output layer can represent combination of the activations of the hidden neurons
- Neural network with one hidden layer **is a universal approximator**
  - Every function can be modeled as a shallow feed forward network
  - Not all functions can be represented *efficiently* with a single hidden layer  
 $\Rightarrow$  we still need deep neural networks



# Going from Shallow to Deep Neural Networks

- Neural Networks can have several hidden layers
- Initializing the weights randomly and training all layers at once does hardly work
- Instead we train layerwise on unannotated data (a.k.a. pre-training):
  - Train the first hidden layer
  - Fix the parameters for the first layer and train the second layer.
  - Fix the parameters for the first & second layer, train the third layer

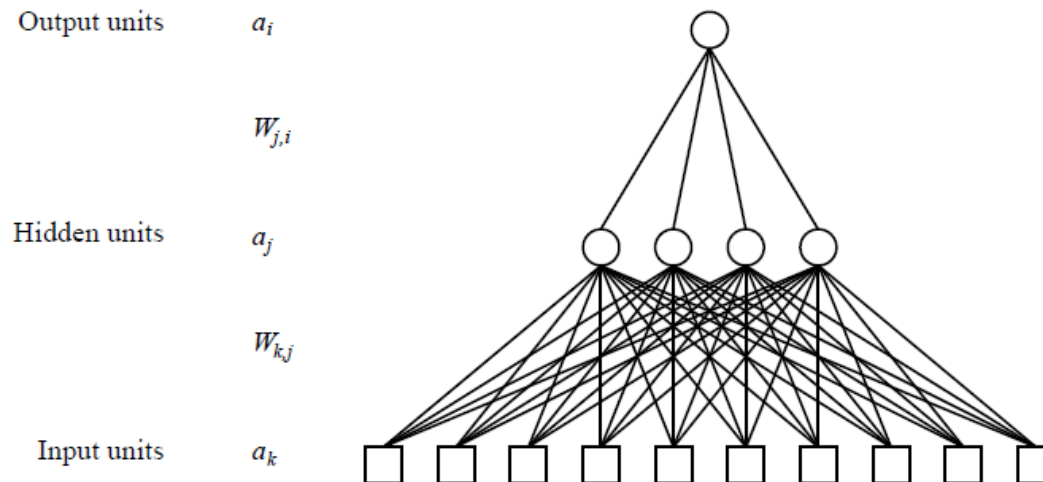


- After the pre-training, train all layers using your annotated data
- The pre-training on your unannotated data creates a high-level abstractions of the input data
- The final training with annotated data fine tunes all parameters in the network

# How to learn the weights

- Initialise the weights i.e.  $W_{k,j}$   $W_{j,i}$  with random values
- With input entries we calculate the predicted output
- We compare the prediction with the true output
- The error is calculated
- The error needs to be sent as feedback for updating the weights

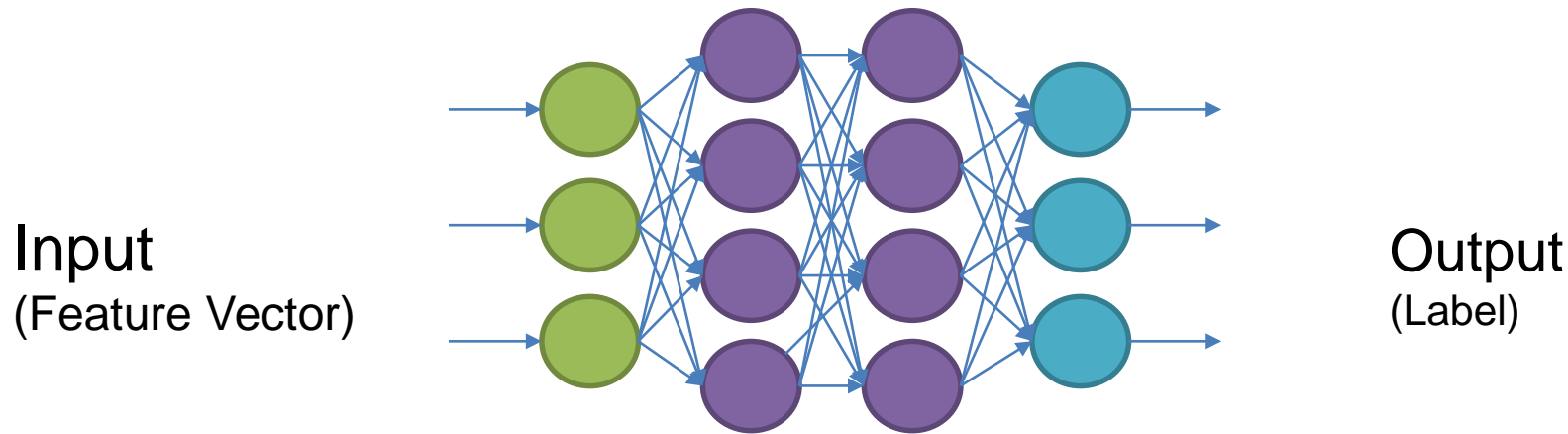
Layers are usually fully connected;  
numbers of hidden units typically chosen by hand



# BACKPROPAGATION



# How to Train a Neural Net?



- Put in Training inputs, get the output
- Compare output to correct answers: Look at loss function  $J$
- Adjust and repeat!
- Backpropagation tells us how to make a single adjustment using calculus.

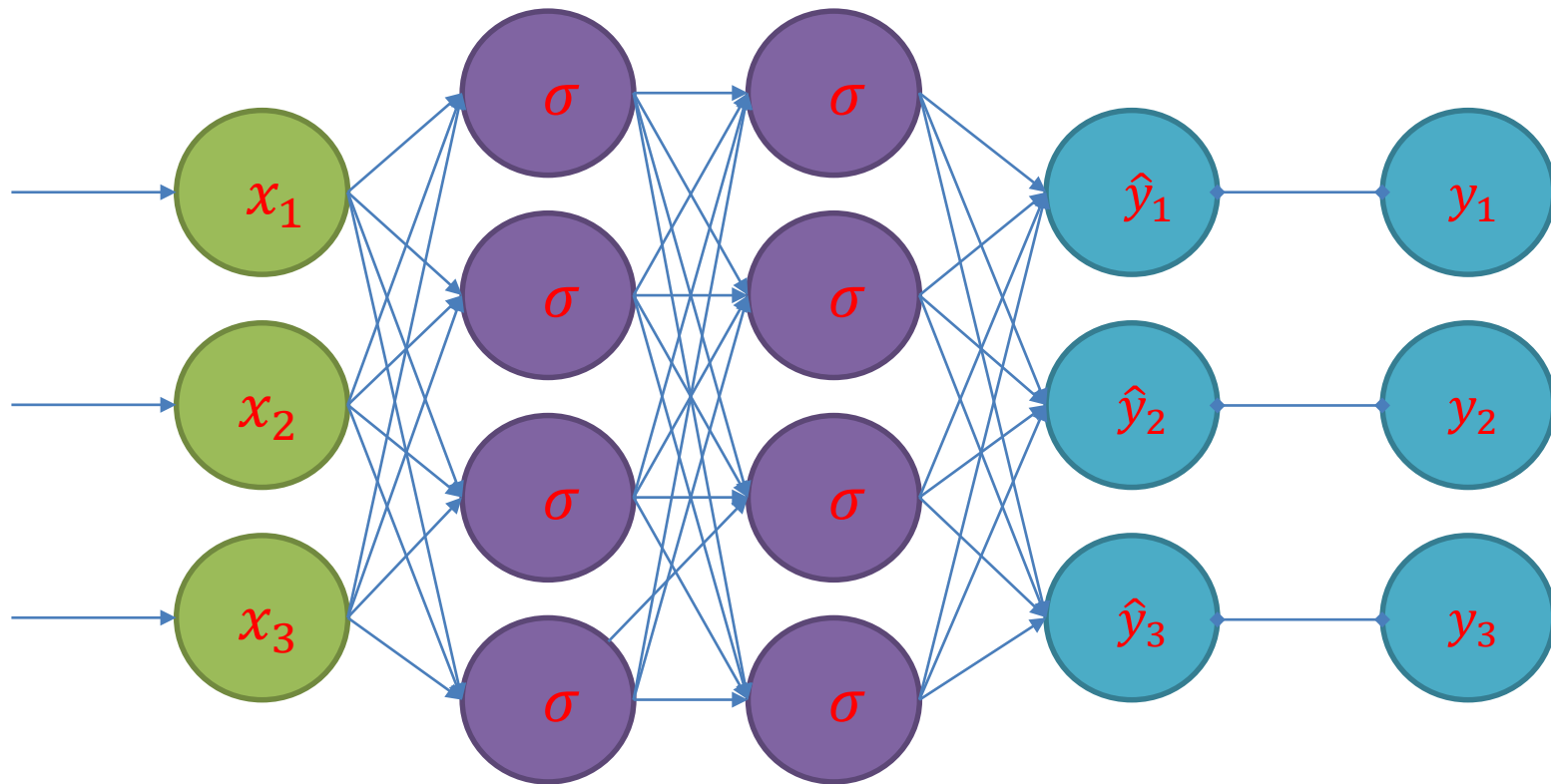


# How have we trained before?

- Gradient Descent!
  1. Make prediction
  2. Calculate Loss
  3. Calculate gradient of the loss function w.r.t. parameters
  4. Update parameters by taking a step in the opposite direction
  5. Iterate



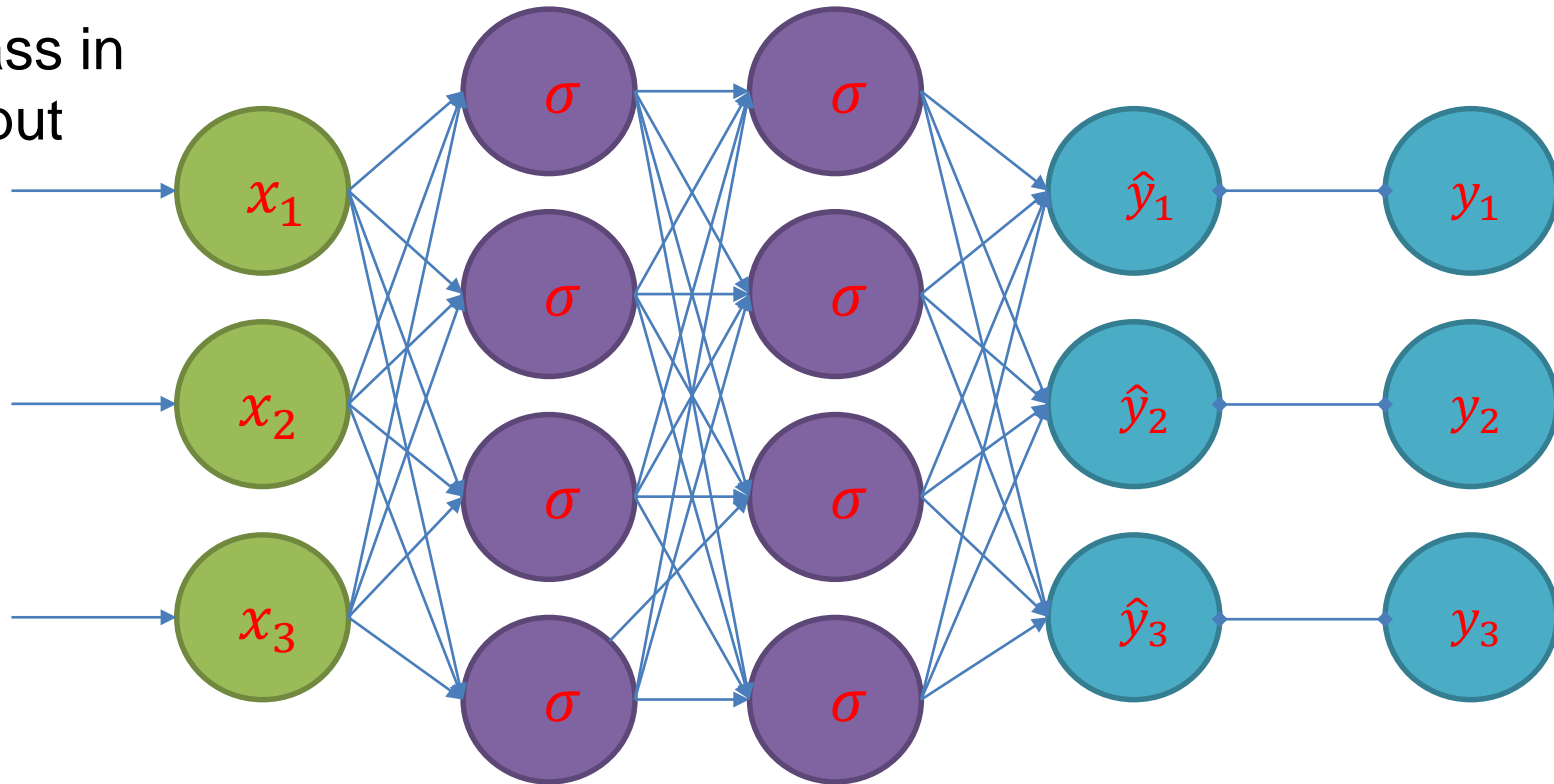
# Feedforward Neural Network





# Forward Propagation

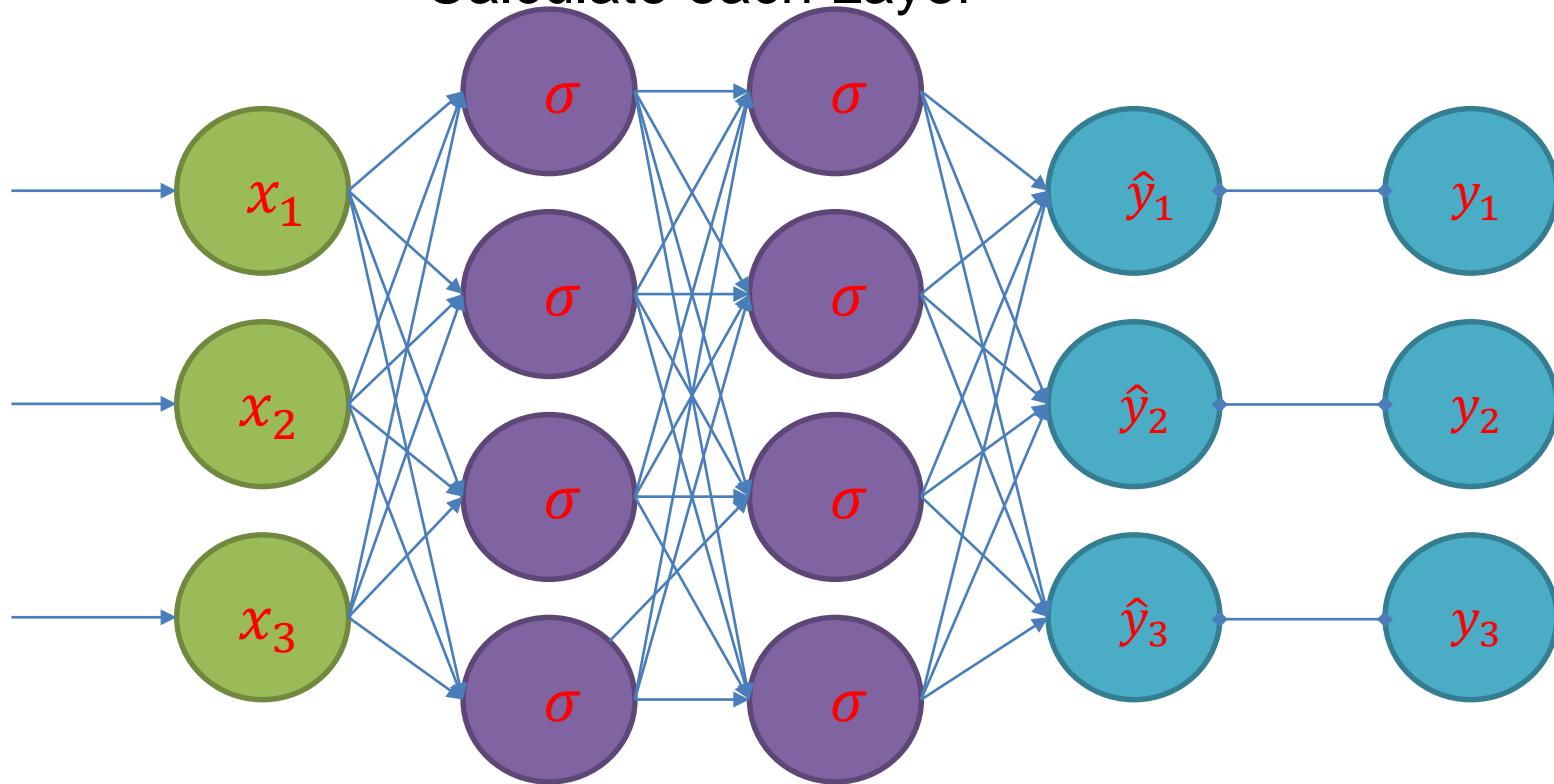
Pass in  
Input





# Forward Propagation

Calculate each Layer

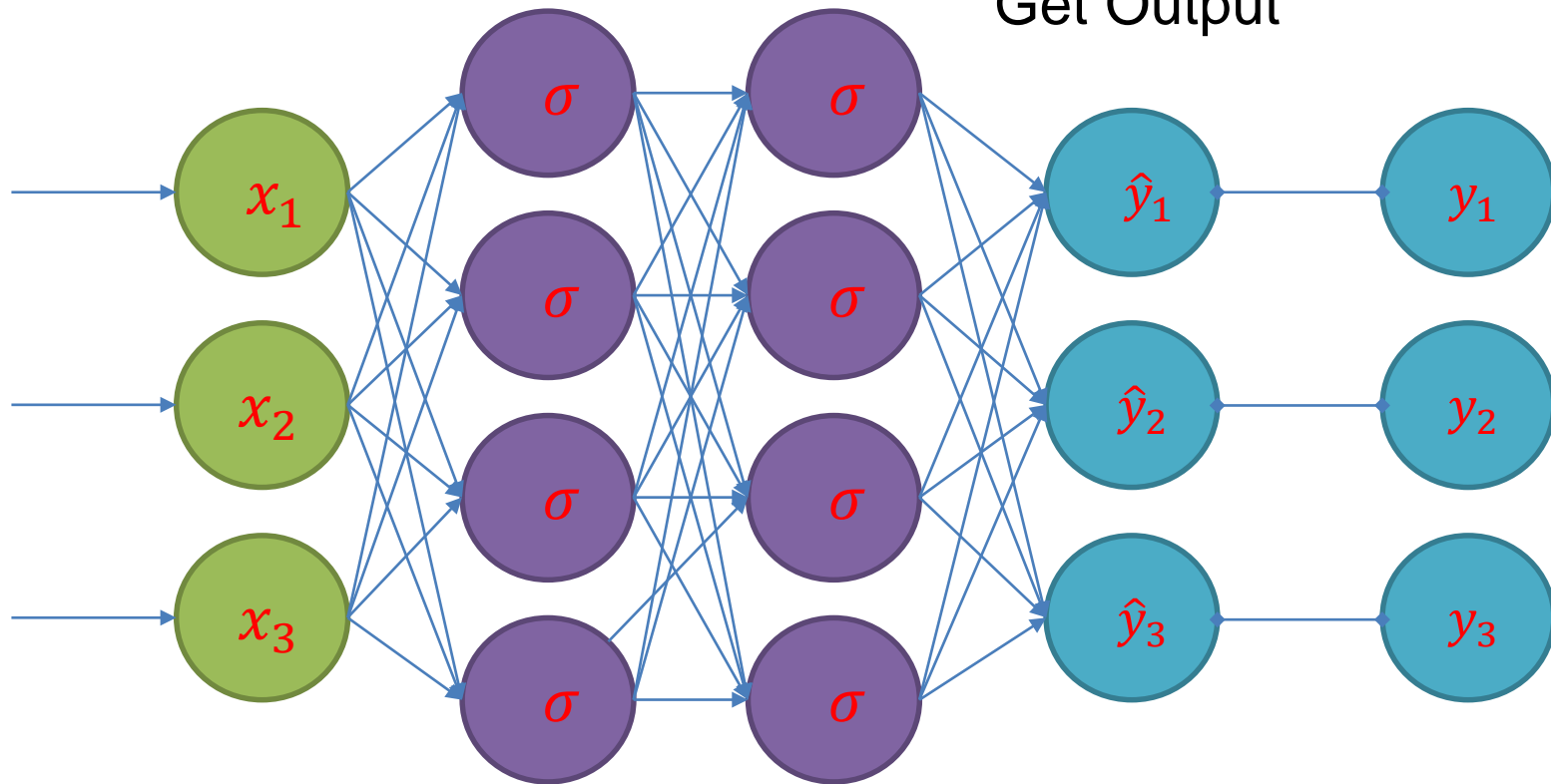






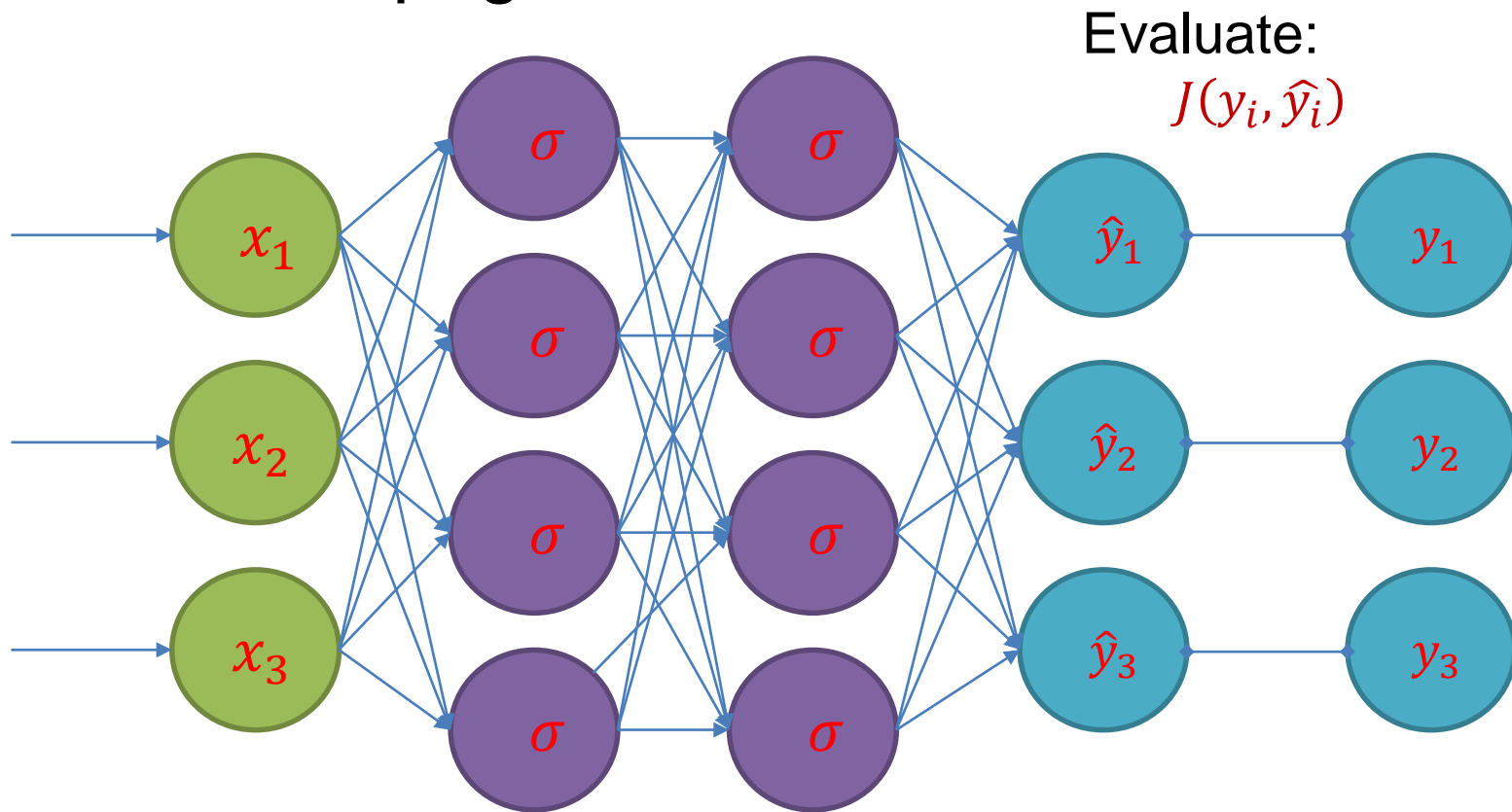
# Forward Propagation

Get Output





# Forward Propagation





# How have we trained before?

- Gradient Descent!
  1. Make prediction
  2. Calculate Loss
  3. Calculate gradient of the loss function w.r.t. parameters
  4. Update parameters by taking a step in the opposite direction
  5. Iterate



# How to Train a Neural Net?

- How could we change the weights to make our Loss Function lower?
- Think of neural net as a function  $F: X \rightarrow Y$
- $F$  is a complex computation involving many weights  $W_k$
- Given the structure, the weights “define” the function  $F$  (and therefore define our model)
- Loss Function is  $J(y, F(x))$

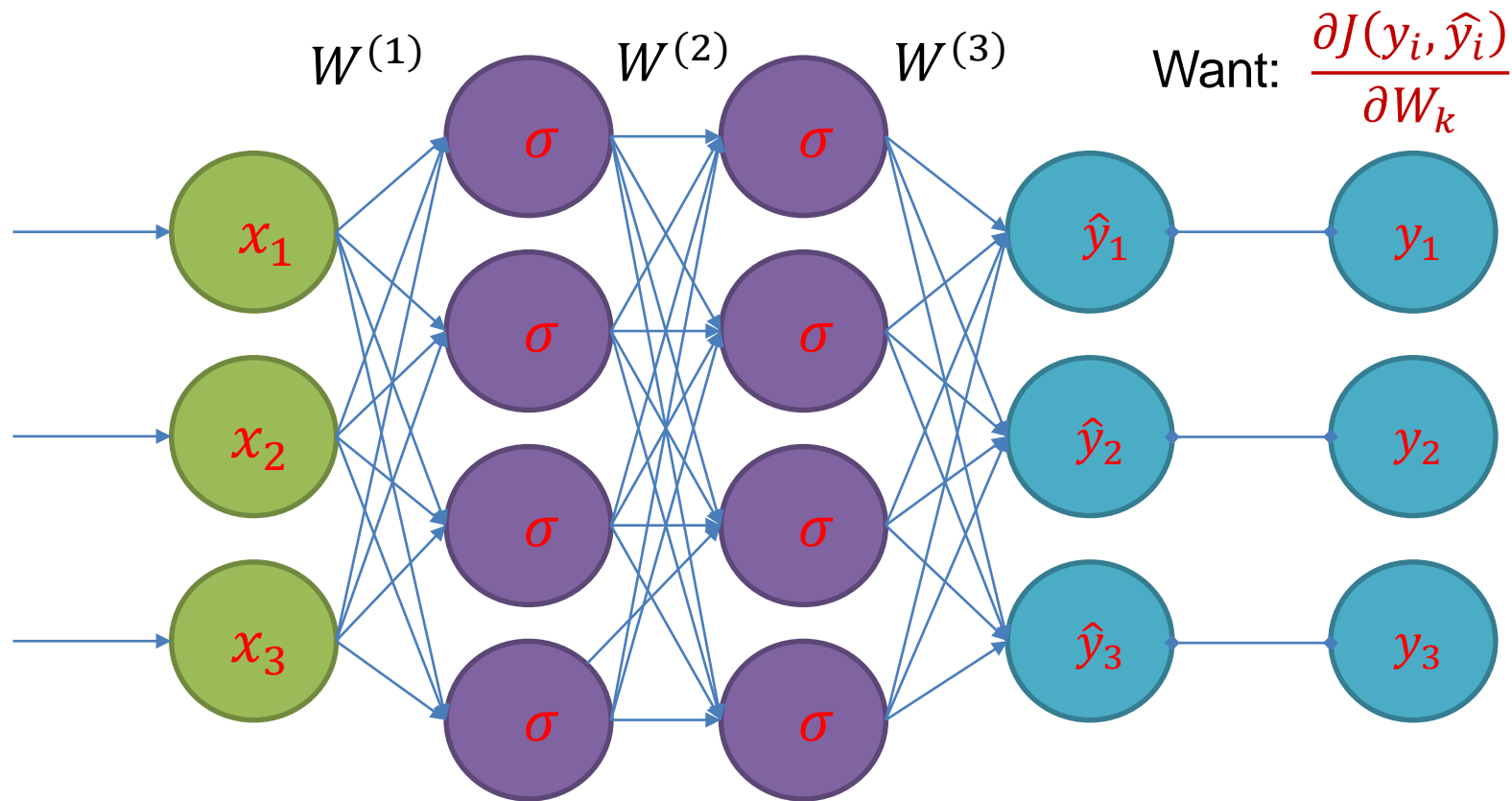


# How to Train a Neural Net?

- Get  $\frac{\partial J}{\partial W_k}$  for every weight in the network.
- This tells us what direction to adjust each  $W_k$  if we want to lower our loss function.
- Make an adjustment and repeat!



# Feedforward Neural Network





# Backpropagation

- Use calculus, chain rule.
- Functions are chosen to have derivatives
- Numerical issues to be considered

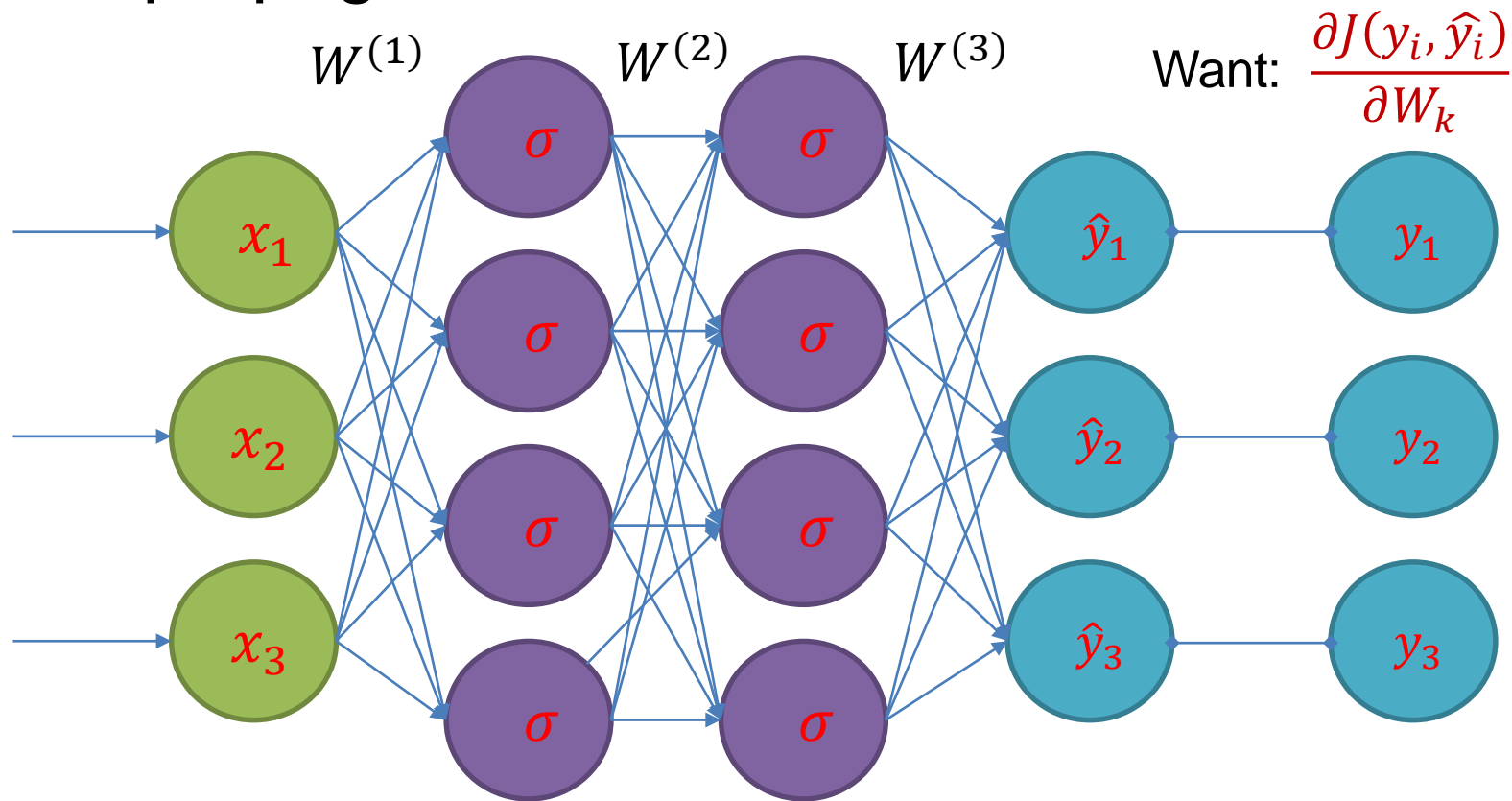
$$\frac{\partial J}{\partial W^{(3)}} = (\hat{y} - y) \cdot a^{(3)}$$

$$\frac{\partial J}{\partial W^{(2)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'(z^{(3)}) \cdot a^{(2)}$$

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'(z^{(3)}) \cdot W^{(2)} \cdot \sigma'(z^{(2)}) \cdot X$$



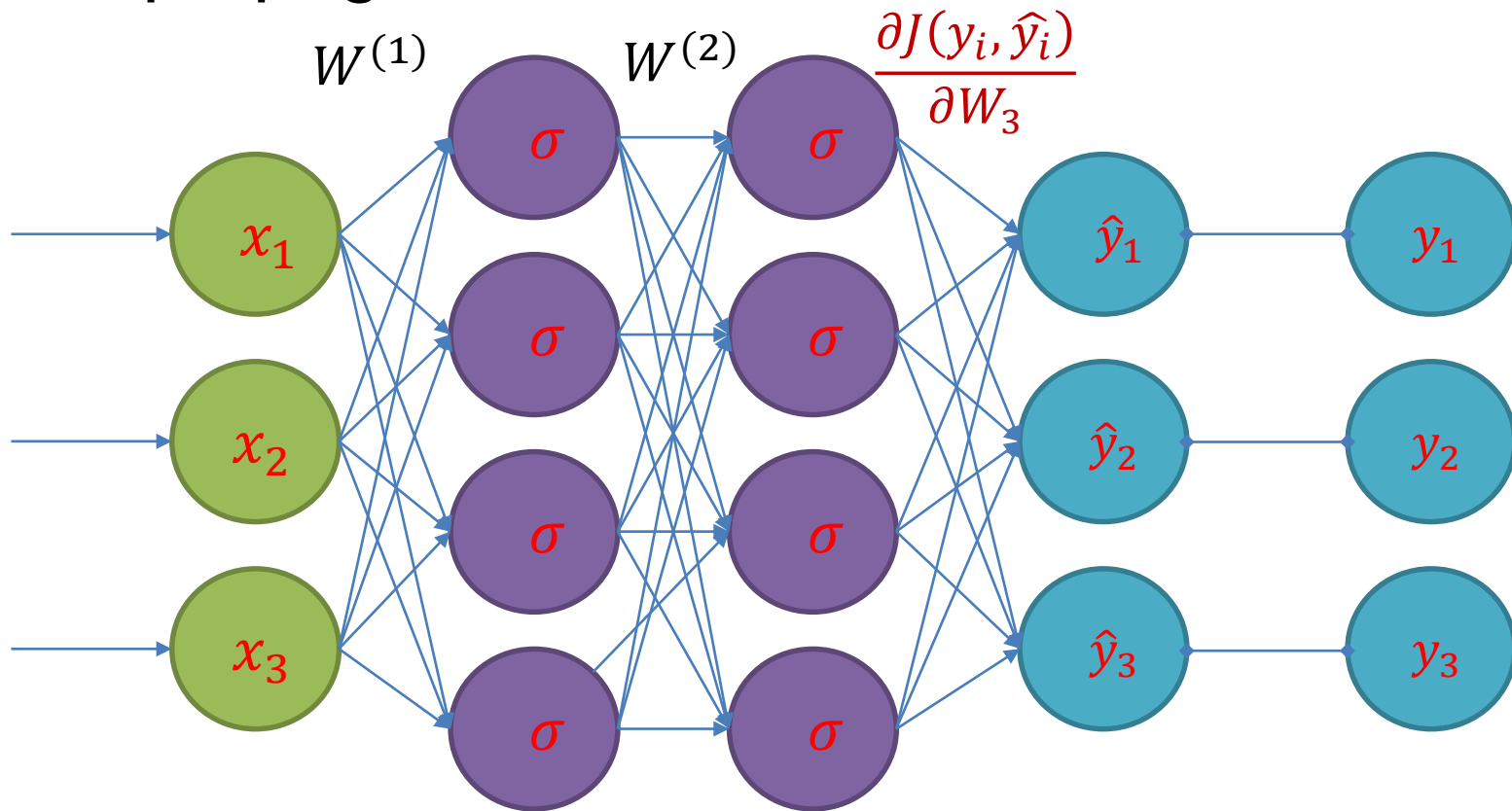
# Backpropagation





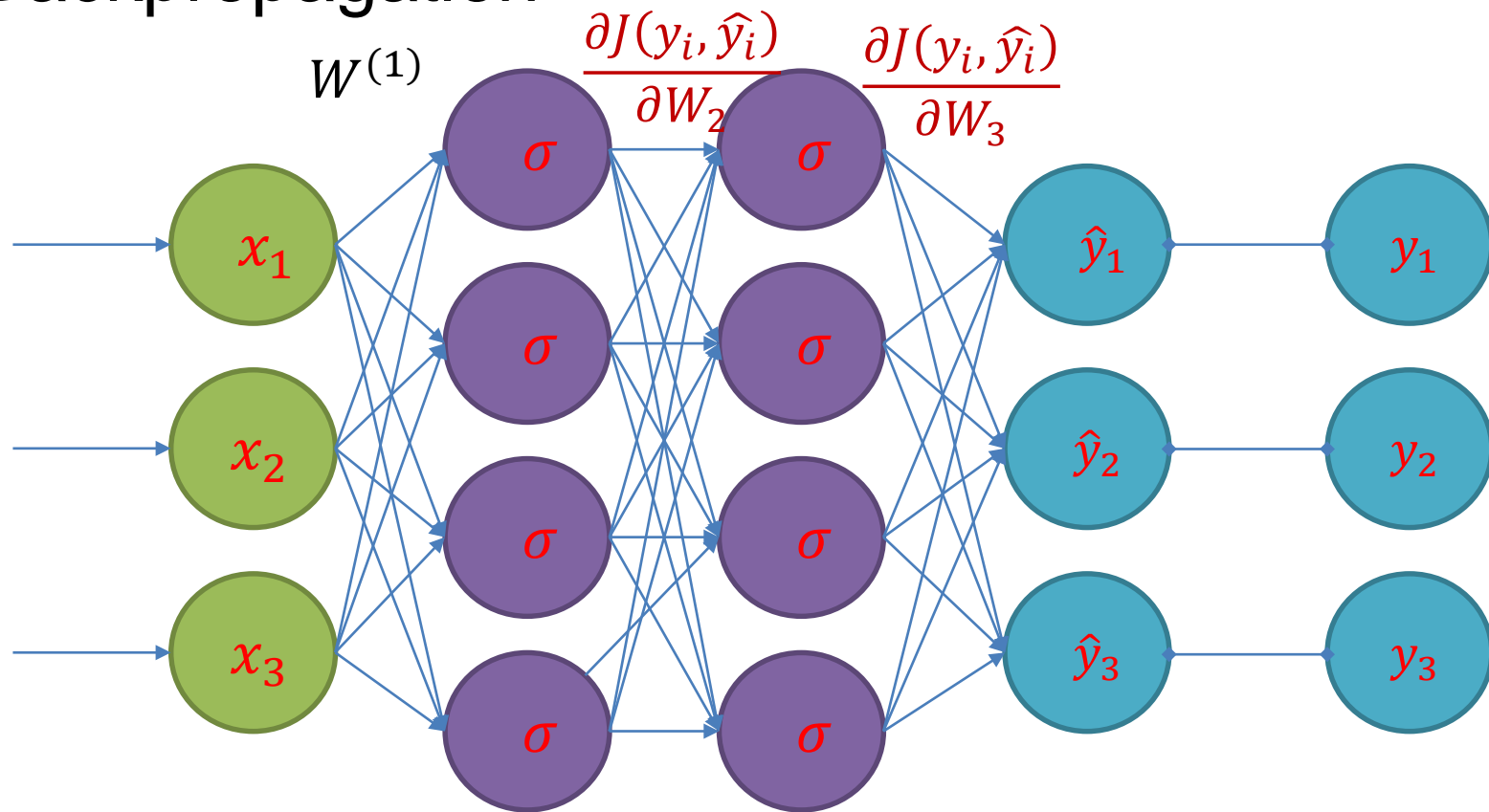


# Backpropagation



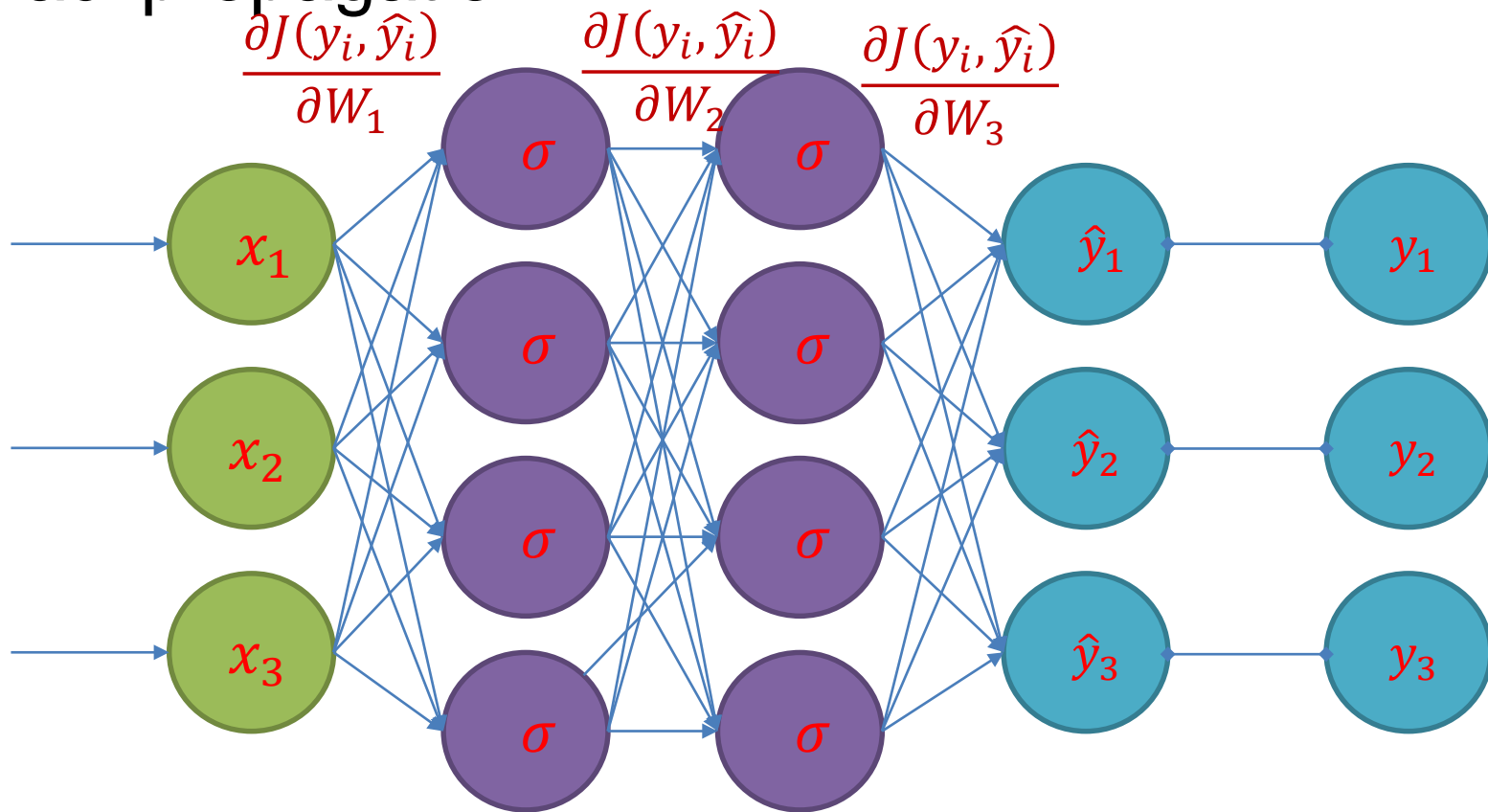


# Backpropagation





# Backpropagation





# How have we trained before?

## Gradient Descent!

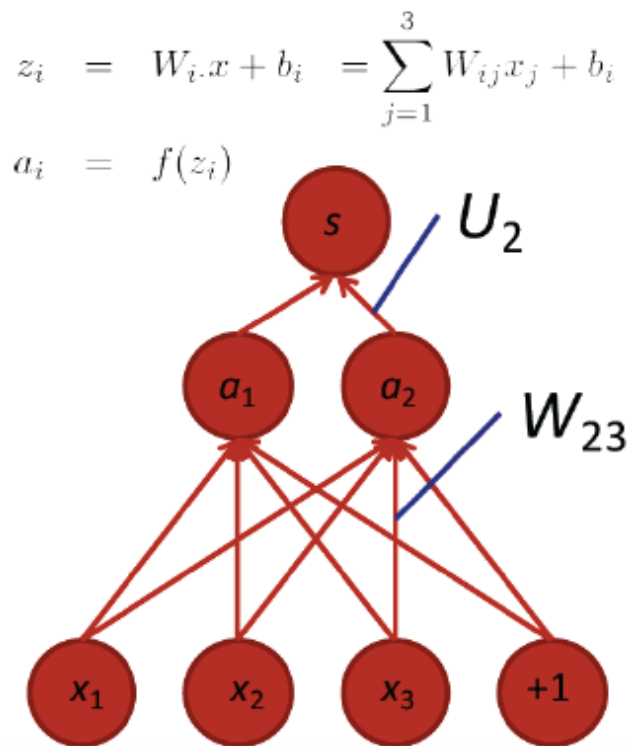
1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

# Training with backpropagation

## Derivative of weight $W_{ij}$ :

$$\frac{\partial}{\partial W_{ij}} U^T a \rightarrow \frac{\partial}{\partial W_{ij}} U_i a_i$$

$$\begin{aligned} U_i \frac{\partial}{\partial W_{ij}} a_i &= U_i \frac{\partial a_i}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} \\ &= U_i \frac{\partial f(z_i)}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} \\ &= U_i f'(z_i) \frac{\partial z_i}{\partial W_{ij}} \\ &= U_i f'(z_i) \frac{\partial W_{i \cdot} x + b_i}{\partial W_{ij}} \end{aligned}$$



## *Derivative continued ...*

$$\begin{aligned} U_i \frac{\partial}{\partial W_{ij}} a_i &= U_i f'(z_i) \frac{\partial W_{i \cdot} x + b_i}{\partial W_{ij}} \\ &= U_i f'(z_i) \frac{\partial}{\partial W_{ij}} \sum_k W_{ik} x_k \\ &= \underbrace{U_i f'(z_i)}_{\delta_i} x_j \\ &= \underbrace{\delta_i}_{\text{Local error signal}} \underbrace{x_j}_{\text{Local input signal}} \end{aligned}$$

where  $f'(z) = f(z)(1 - f(z))$  for logistic  $f$

*From single weight  $W_{ij}$  to full  $W$ :*

$$\frac{\partial s}{\partial W_{ij}} = \delta_i x_j$$

- We want all combinations of  $i = 1, 2, \dots$  and  $j = 1, 2, 3, \dots$
- Solution: Outer product

$$\frac{\partial J}{\partial W} = \delta x^T$$

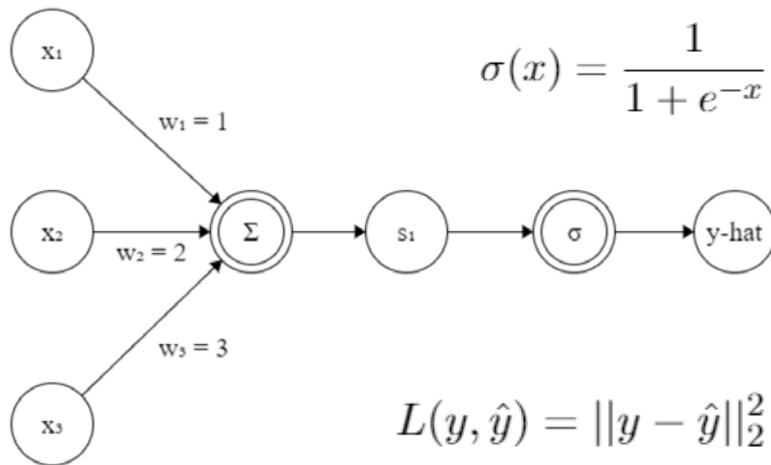
# Computational Graph

Definition: a data structure for storing gradients of variables used in computations.

- Node  $v$  represents variable
  - Stores value
  - Gradient
  - The function that created the node
- Directed edge  $(u, v)$  represents the partial derivative of  $u$  w.r.t.  $v$
- To compute the gradient  $dL/dv$ , find the unique path from  $L$  to  $v$  and multiply the edge weights.



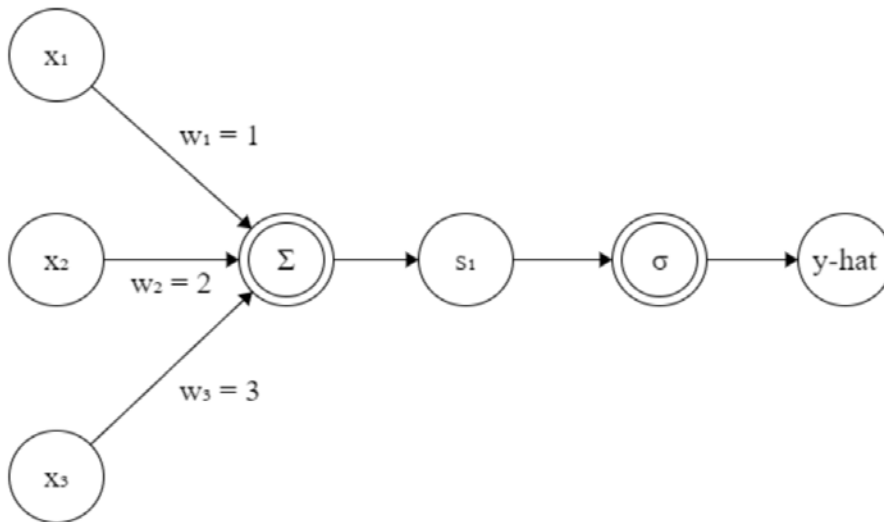
# Backpropagation for neural nets



x	$\sigma(x)$	$\sigma'(x)$
-5	0.01	0.01
-4	0.02	0.02
-3	0.05	0.05
-2	0.12	0.10
-1	0.27	0.20
0	0.50	0.25
1	0.73	0.20
2	0.88	0.10
3	0.95	0.05
4	0.98	0.02
5	0.99	0.01

Given softmax activation, L2 loss, a point  $(x_1, x_2, x_3, y) = (0.1, 0.15, 0.2, 1)$ , compute the  $\frac{\partial L}{\partial w_1}$

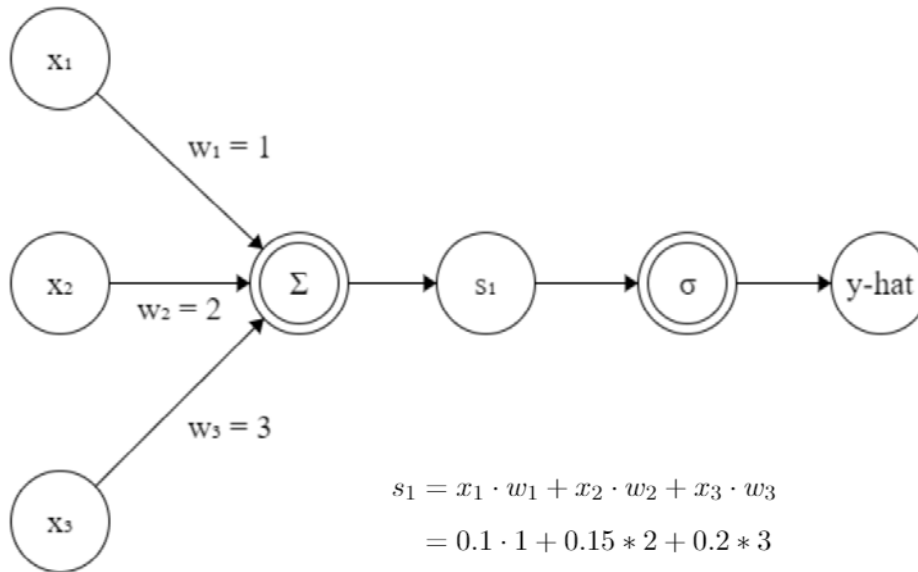
# Backpropagation for neural nets: forward pass



$$\begin{aligned} s_1 &= x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 \\ &= 0.1 \cdot 1 + 0.15 \cdot 2 + 0.2 \cdot 3 \\ &= 1 \\ \hat{y} &= \sigma(s_1) \\ &= \sigma(1) \\ &= \boxed{0.73} \end{aligned}$$



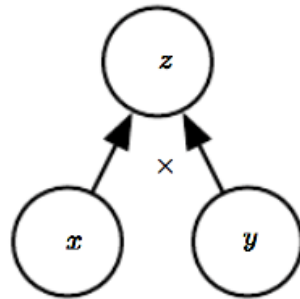
# Backpropagation for neural nets: backward pass



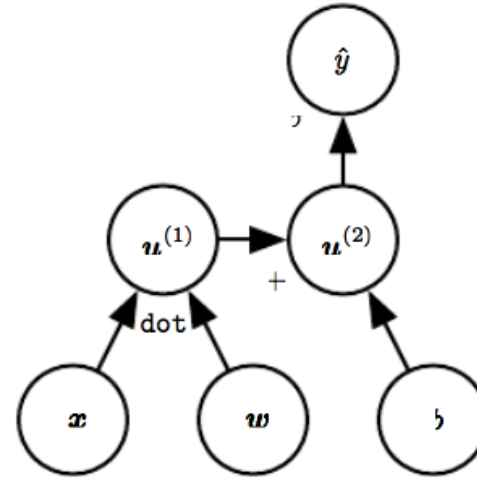
$$\begin{aligned} s_1 &= x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 \\ &= 0.1 \cdot 1 + 0.15 \cdot 2 + 0.2 \cdot 3 \\ &= 1 \\ \hat{y} &= \sigma(s_1) \\ &= \sigma(1) \\ &= \boxed{0.73} \end{aligned}$$

$$\begin{aligned} \frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial s_1} \cdot \frac{\partial s_1}{\partial w_1} \\ &= 2 \|\hat{y} - y\| \times \sigma'(s_1) \times x_1 \\ &= 2 \cdot (\sigma(1) - 1) \times \sigma'(1) \times 0.1 \\ &= \boxed{-0.0106} \end{aligned}$$

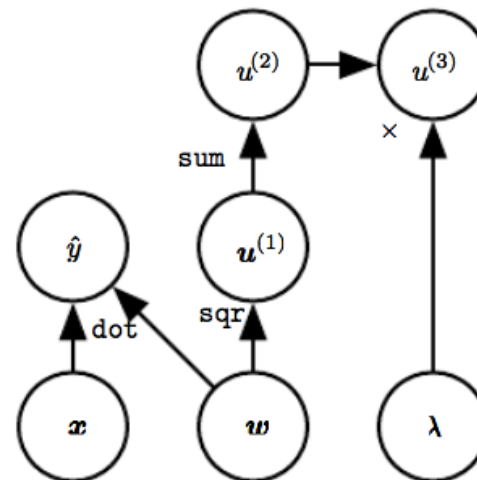
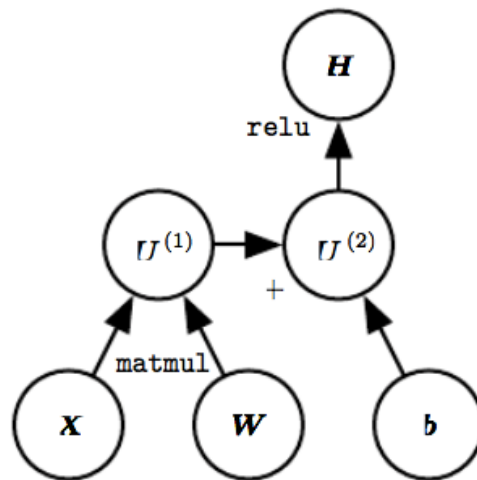
# Computation Graphs



(a)



(b)



# **CONVOLUTIONAL NEURAL NETWORKS**



## Motivation – Image Data

- So far, the structure of our neural network treats all inputs interchangeably.
- No relationships between the individual inputs
- Just an ordered set of variables
- We want to incorporate domain knowledge into the architecture of a Neural Network.



# Motivation

- Image data has important structures, such as;
- "Topology" of pixels
- Translation invariance
- Issues of lighting and contrast
- Knowledge of human visual system
- Nearby pixels tend to have similar values
- Edges and shapes
- Scale Invariance – objects may appear at different sizes in the image.



# Motivation – Image Data

- Fully connected would require a vast number of parameters
- MNIST images are small (32 x 32 pixels) and in grayscale
- Color images are more typically at least (200 x 200) pixels x 3 color channels (RGB) = 120,000 values.
- A single fully connected layer would require  $(200 \times 200 \times 3)^2 = 14,400,000,000$  weights!
- Variance (in terms of bias-variance) would be too high
- So we introduce “bias” by structuring the network to look for certain kinds of patterns





# Motivation

- Features need to be “built up”
- Edges -> shapes -> relations between shapes
- Textures
  
- Cat = two eyes in certain relation to one another + cat fur texture.
- Eyes = dark circle (pupil) inside another circle.
- Circle = particular combination of edge detectors.
- Fur = edges in certain pattern.



# Kernels

- A *kernel* is a grid of weights “overlaid” on image, centered on one pixel
- Each weight multiplied with pixel underneath it
- Output over the centered pixel is  $\sum_{p=1}^P W_p \cdot pixel_p$
- Used for traditional image processing techniques:
  - Blur
  - Sharpen
  - Edge detection
  - Emboss



# Kernel: 3x3 Example

Input

3	2	1
1	2	3
1	1	1

Kernel

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

Output




# Kernel: 3x3 Example

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

Output




# Kernel: 3x3 Example

Input

3	2	1
1	2	3
1	1	1

Kernel

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

Output

	2	

$$\begin{aligned} &= (3 \cdot -1) + (2 \cdot 0) + (1 \cdot 1) \\ &+ (1 \cdot -2) + (2 \cdot 0) + (3 \cdot 2) \\ &+ (1 \cdot -1) + (1 \cdot 0) + (1 \cdot 1) \end{aligned}$$

$$= -3 + 1 - 2 + 6 - 1 + 1 = 2$$



# Kernel: Example

1	1	1
1	1	1
1	1	1

Unweighted 3x3  
smoothing kernel

0	1	0
1	4	1
0	1	0

Weighted 3x3 smoothing  
kernel with Gaussian blur

0	-1	0
-1	5	-1
0	-1	0

Kernel to make  
image sharper

-1	-1	-1
-1	9	-1
-1	-1	-1

Intensified sharper  
image



Gaussian Blur



Sharpened image

# Kernels as Feature Detectors

Can think of kernels as a "local feature detectors"

Vertical Line Detector

-1	1	-1
-1	1	-1
-1	1	-1

Horizontal Line Detector

-1	-1	-1
1	1	1
-1	-1	-1

Corner Detector

-1	-1	-1
-1	1	1
-1	1	1



# Convolutional Neural Nets

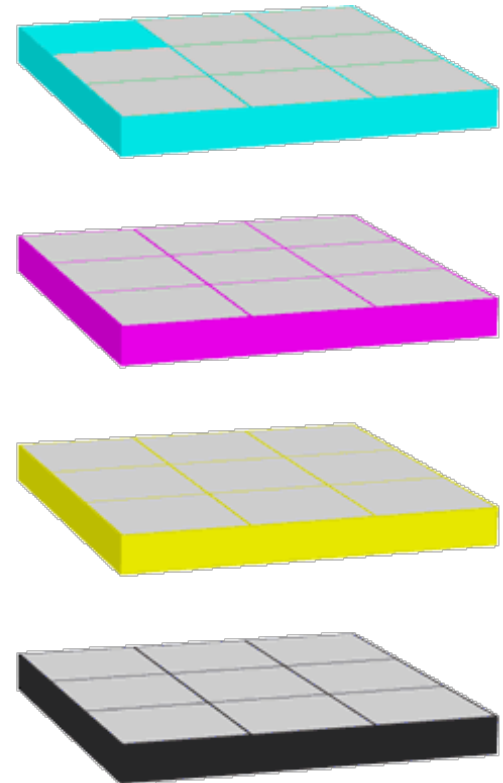
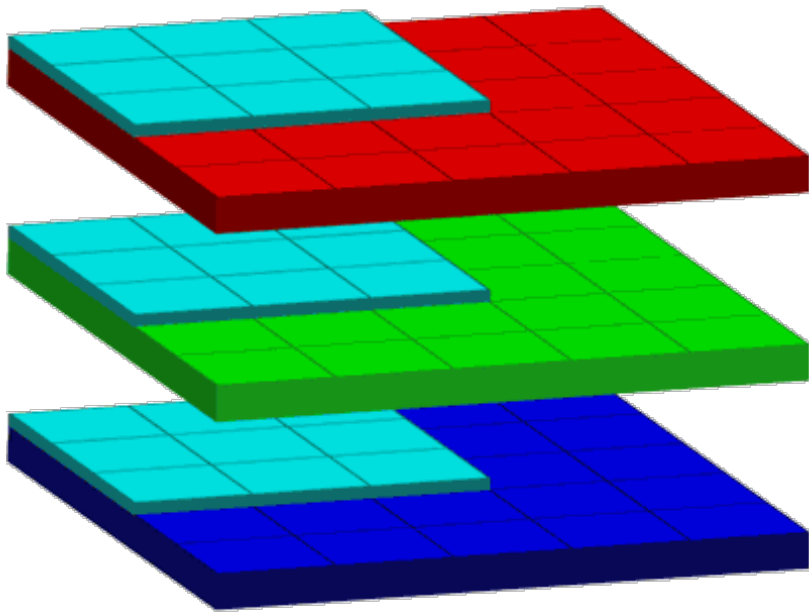
Primary Ideas behind Convolutional Neural Networks:

- Let the Neural Network learn which kernels are most useful
- Use same set of kernels across entire image (translation invariance)
- Reduces number of parameters and “variance” (from bias-variance point of view)





# Convolutions



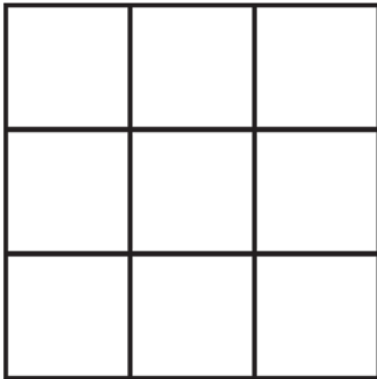


# Convolution Settings – Grid Size

## **Grid Size (Height and Width):**

- The number of pixels a kernel “sees” at once
- Typically use odd numbers so that there is a “center” pixel
- Kernel does not need to be square

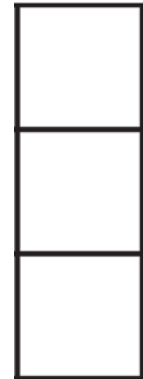
Height: 3, Width: 3



Height: 1, Width: 3



Height: 3, Width: 1





# Convolution Settings - Padding

## **Padding**

- Using Kernels directly, there will be an “edge effect”
- Pixels near the edge will not be used as “center pixels” since there are not enough surrounding pixels
- Padding adds extra pixels around the frame
- So every pixel of the original image will be a center pixel as the kernel moves across the image
- Added pixels are typically of value zero (zero-padding)



# Without Padding

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

input

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

kernel

-2		

output



# With Padding

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

input

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

kernel

-1				

output

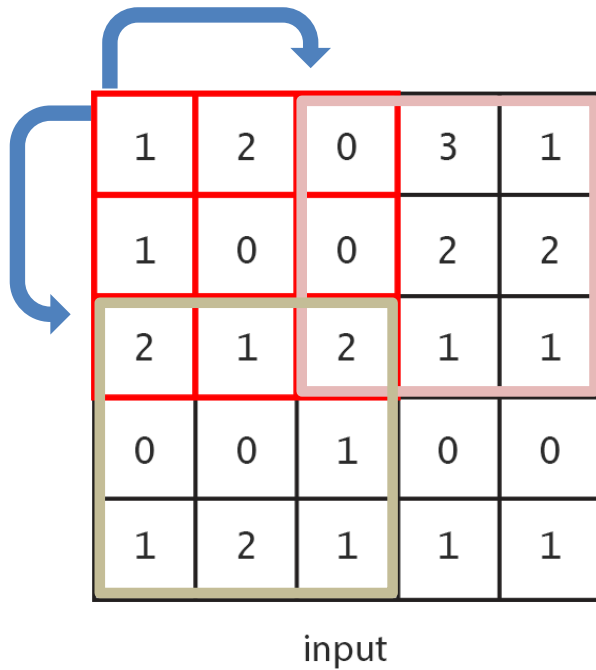


# Convolution Settings

## **Stride**

- The "step size" as the kernel moves across the image
- Can be different for vertical and horizontal steps (but usually is the same value)
- When stride is greater than 1, it scales down the output dimension

# Stride 2 Example – No Padding



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

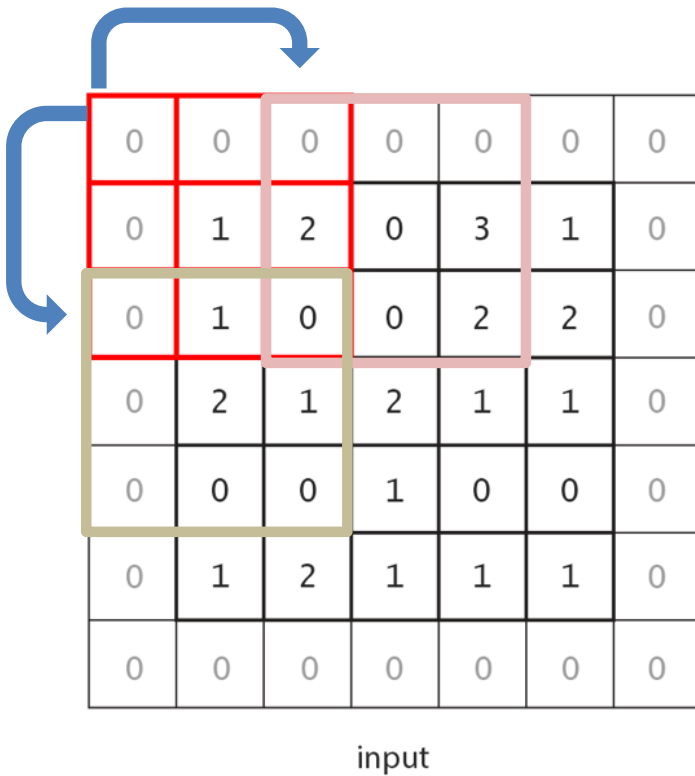
kernel

-2	3
0	

output



# Stride 2 Example – With Padding



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

kernel

-1	2	
3		

output





# Convolutional Settings - Depth

- In images, we often have multiple numbers associated with each pixel location.
- These numbers are referred to as “channels”
  - RGB image – 3 channels
  - CMYK – 4 channels
- The number of channels is referred to as the “depth”
- So the kernel itself will have a “depth” the same size as the number of input channels
- Example: a 5x5 kernel on an RGB image
  - There will be  $5 \times 5 \times 3 = 75$  weights

# Convolutional Settings - Depth

- The output from the layer will also have a depth
- The networks typically train many different kernels
- Each kernel outputs a single number at each pixel location
- So if there are 10 kernels in a layer, the output of that layer will have depth 10.

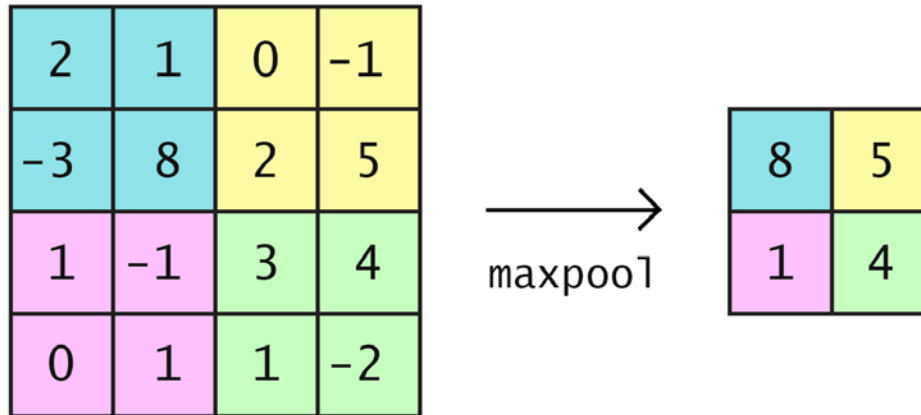


# Pooling

- Idea: Reduce the image size by mapping a patch of pixels to a single value.
- Shrinks the dimensions of the image.
- Does not have parameters, though there are different types of pooling operations.

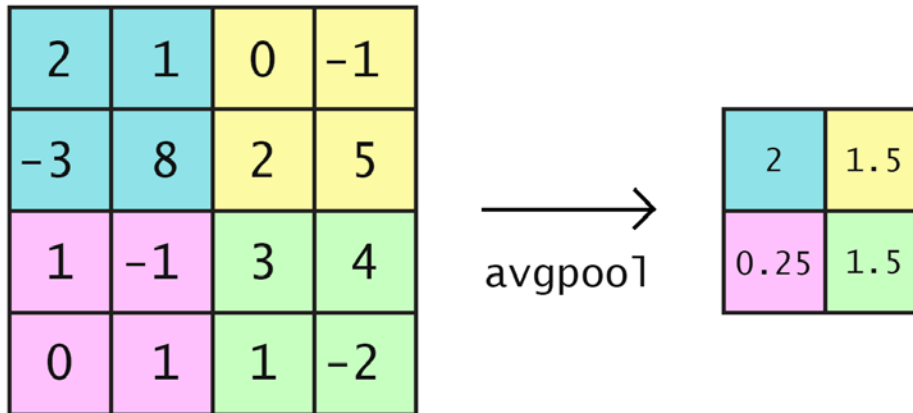
# Pooling: Max-pool

- For each distinct patch, represent it by the maximum
- 2x2 maxpool shown below

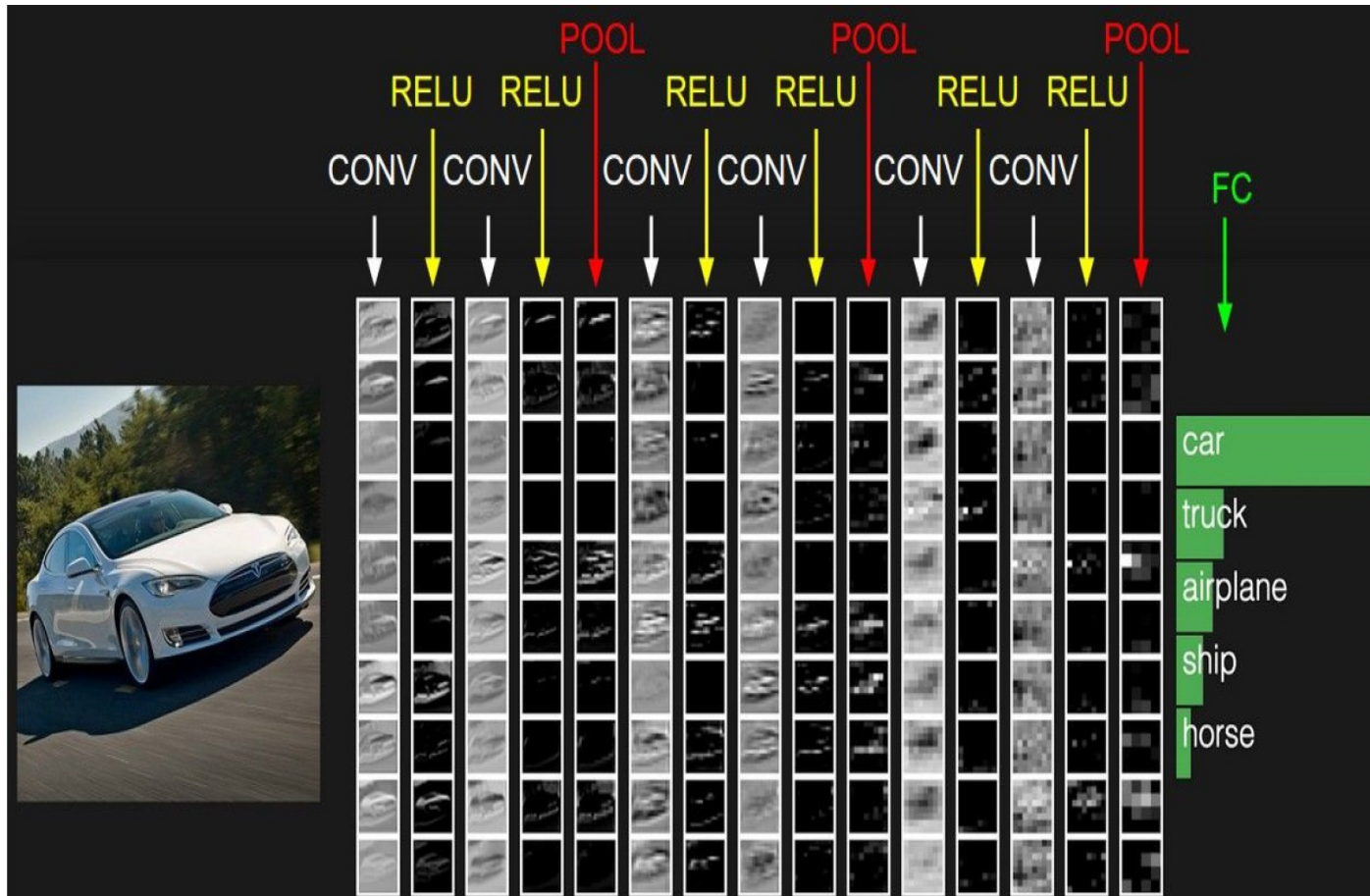


# Pooling: Average-pool

- For each distinct patch, represent it by the average
- 2x2 avgpool shown below.

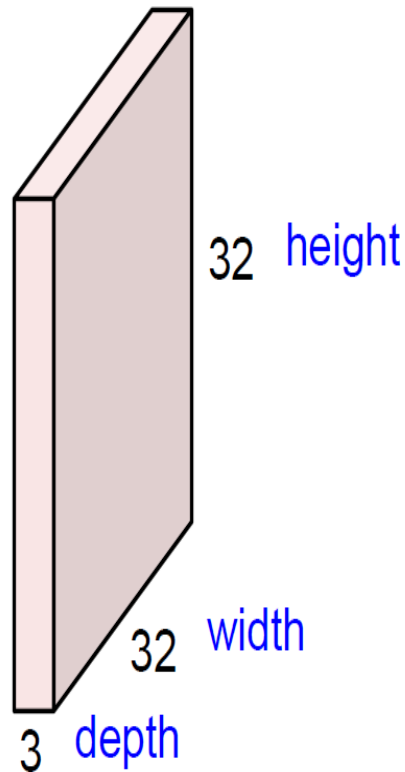


# ConvNet: CONV, RELU, POOL and FC Layers



# Convolution Layer

32x32x3 image



Filters always extend the full depth of the input volume

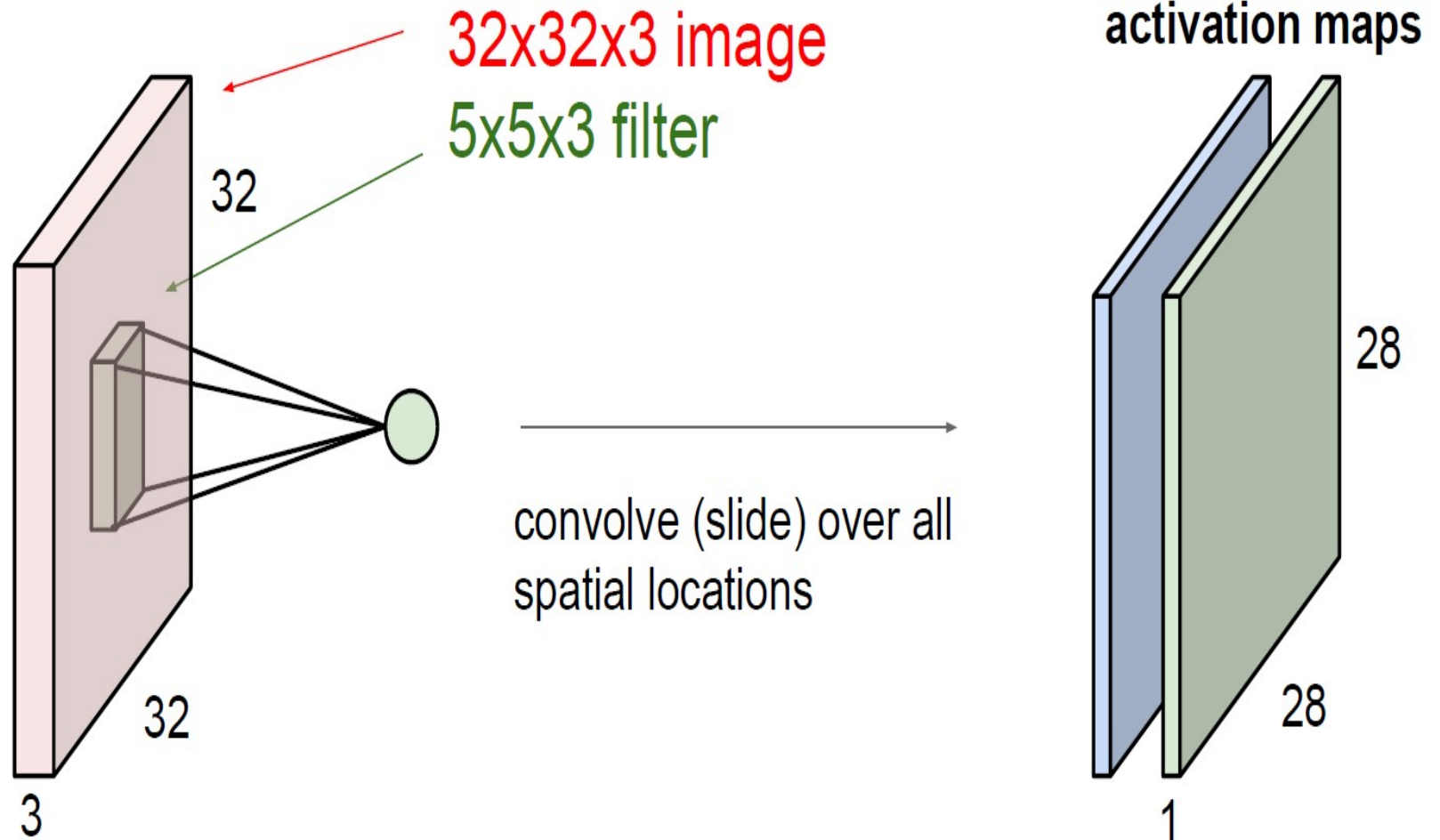
5x5x3 filter



**Convolve** the filter with the image  
i.e. “slide over the image spatially,  
computing dot products”

# Convolution Layer

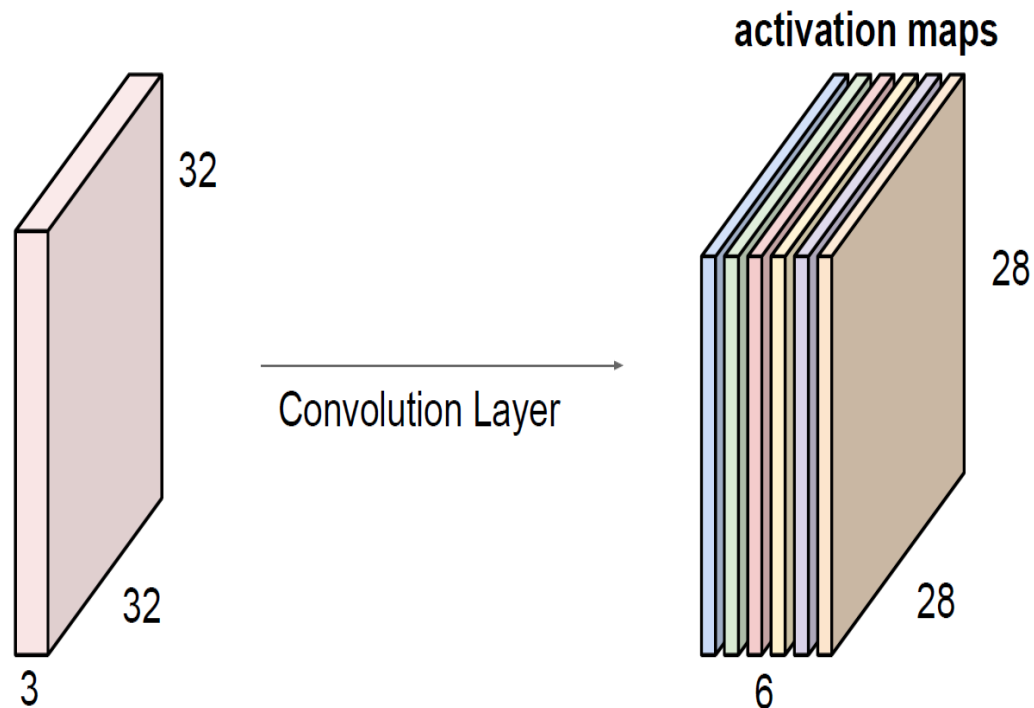
consider a second,  
green filter





# Convolution Layer

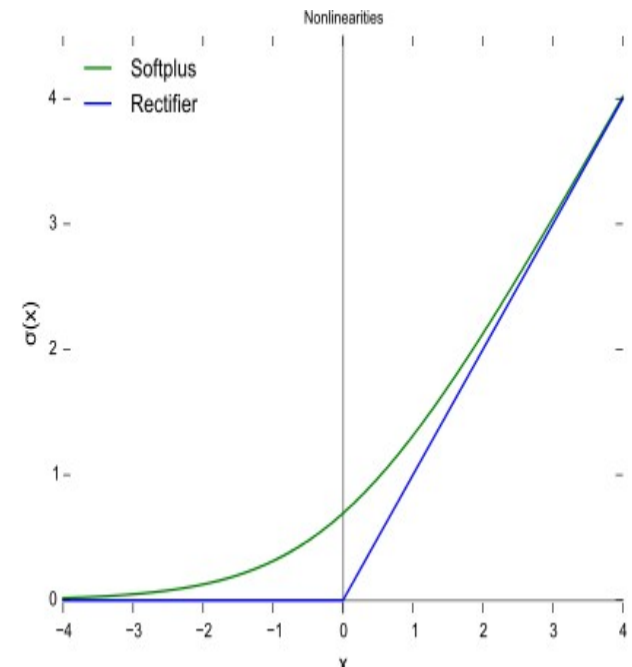
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a “new image” of size 28x28x6!

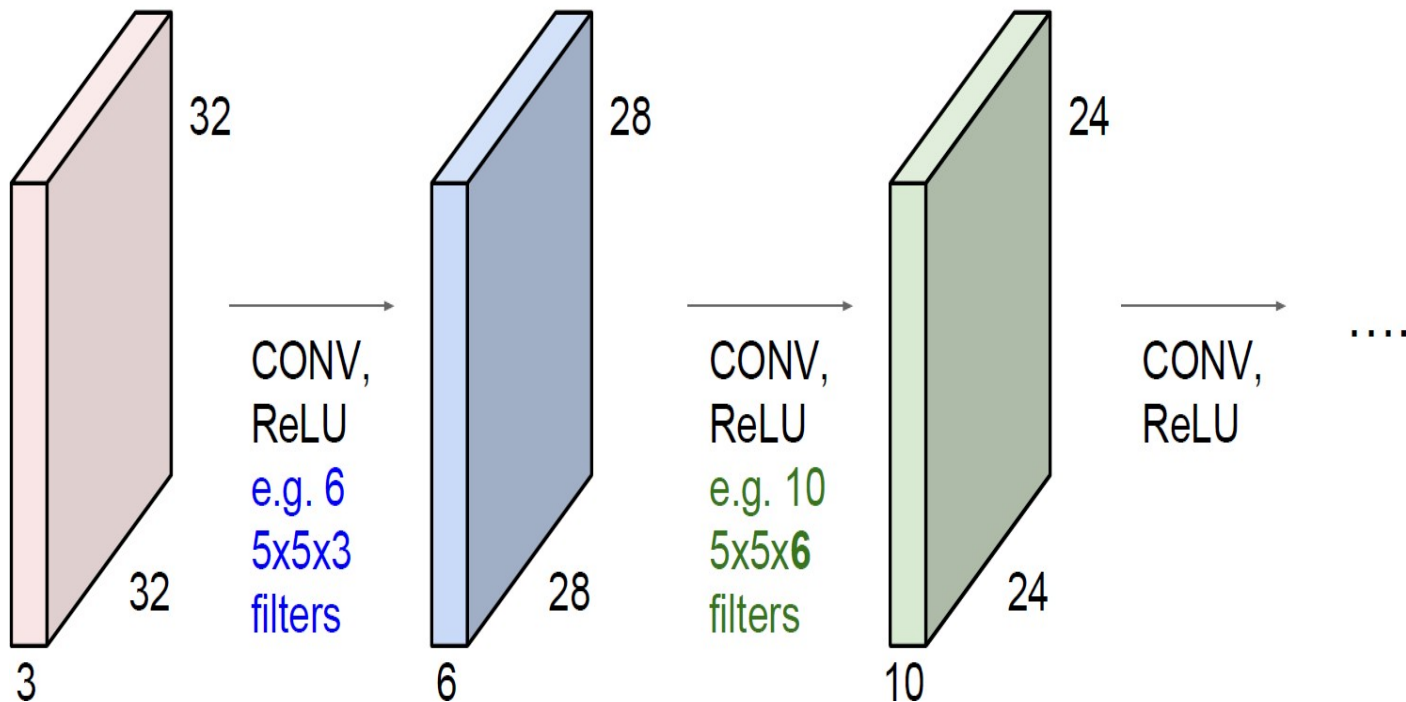
# ReLU (Rectified Linear Units) Layer

- This is a layer of neurons that applies the activation function  $f(x)=\max(0,x)$ .
- It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer.
- Other functions are also used to increase nonlinearity, for example the hyperbolic tangent  $f(x)=\tanh(x)$ , and the sigmoid function.
- This is also known as a ramp function.



# A Basic ConvNet

**Preview:** ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



# What is convolution of an image with a filter

1	1	1	0	0
0	1	1	1	0
0	0 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	1
0	0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	0
0	1 <sub>x1</sub>	1 <sub>x0</sub>	0 <sub>x1</sub>	0

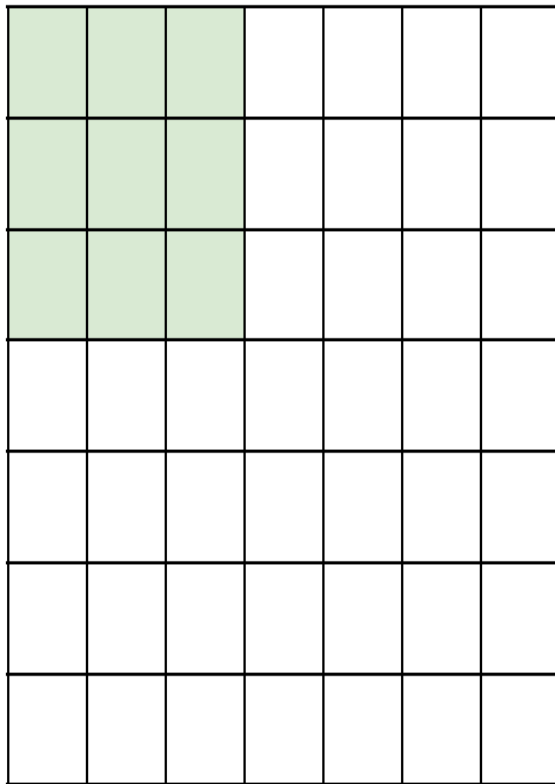
Image

4	3	4
2	4	3
2	3	

Convolved  
Feature

# Details about the convolution layer

7

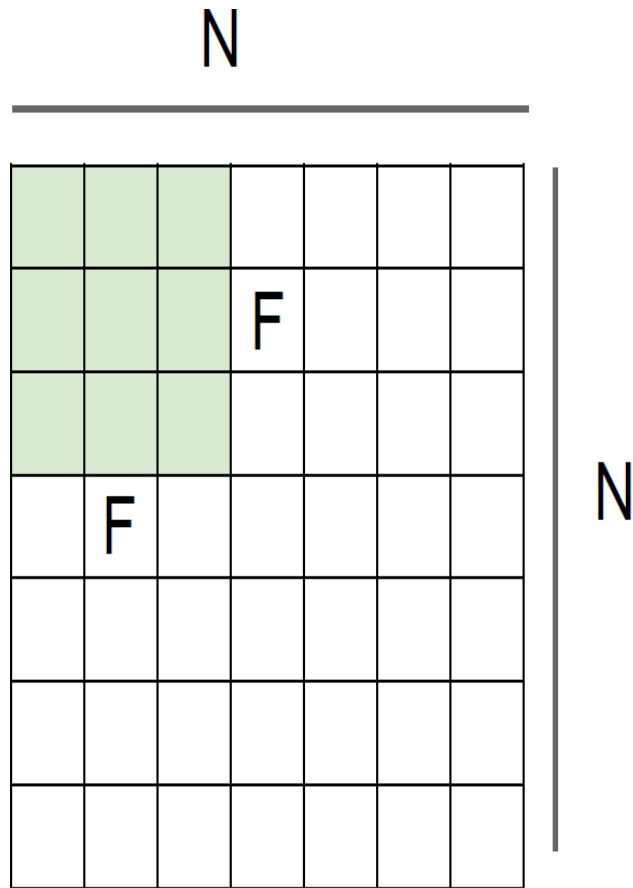


7

7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 3?**

**doesn't fit!**  
cannot apply 3x3 filter on  
7x7 input with stride 3.

# Details about the convolution layer



Output size:  
 $(N - F) / \text{stride} + 1$

e.g.  $N = 7, F = 3$ :

stride 1  $\Rightarrow (7 - 3) / 1 + 1 = 5$

stride 2  $\Rightarrow (7 - 3) / 2 + 1 = 3$

stride 3  $\Rightarrow (7 - 3) / 3 + 1 = 2.33$

# Details about the convolution layer

In practice: Common to zero pad the border

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

**3x3 filter**, applied with **stride 1**

**pad with 1 pixel** border => what is the output?

**7x7 output!**

in general, common to see CONV layers with stride 1, filters of size  $F \times F$ , and zero-padding with  $(F-1)/2$ . (will preserve size spatially)

e.g.  $F = 3 \Rightarrow$  zero pad with 1

$F = 5 \Rightarrow$  zero pad with 2

$F = 7 \Rightarrow$  zero pad with 3

# Convolution layer examples

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Output volume size: ?

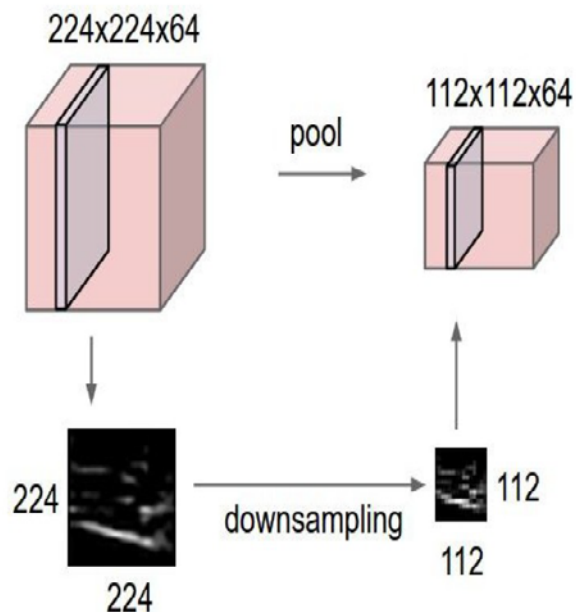
$(32 + 2 * 2 - 5) / 1 + 1 = 32$  spatially, so

**32x32x10**



# Pooling Layer

makes the representations smaller and more manageable  
operates over each activation map independently:



Single depth slice

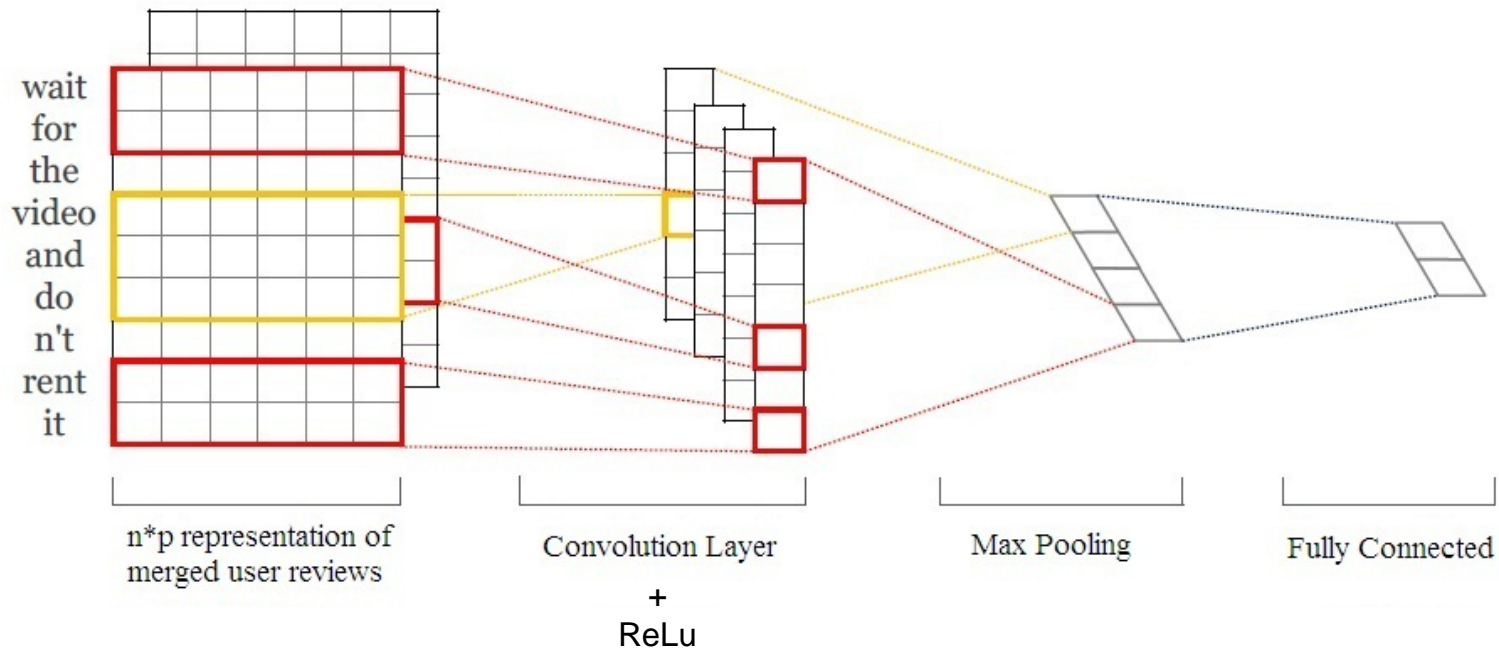
1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters  
and stride 2

6	8
3	4

- Invariance to image transformation and increases compactness to representation.
- Pooling types: Max, Average, L2 etc.

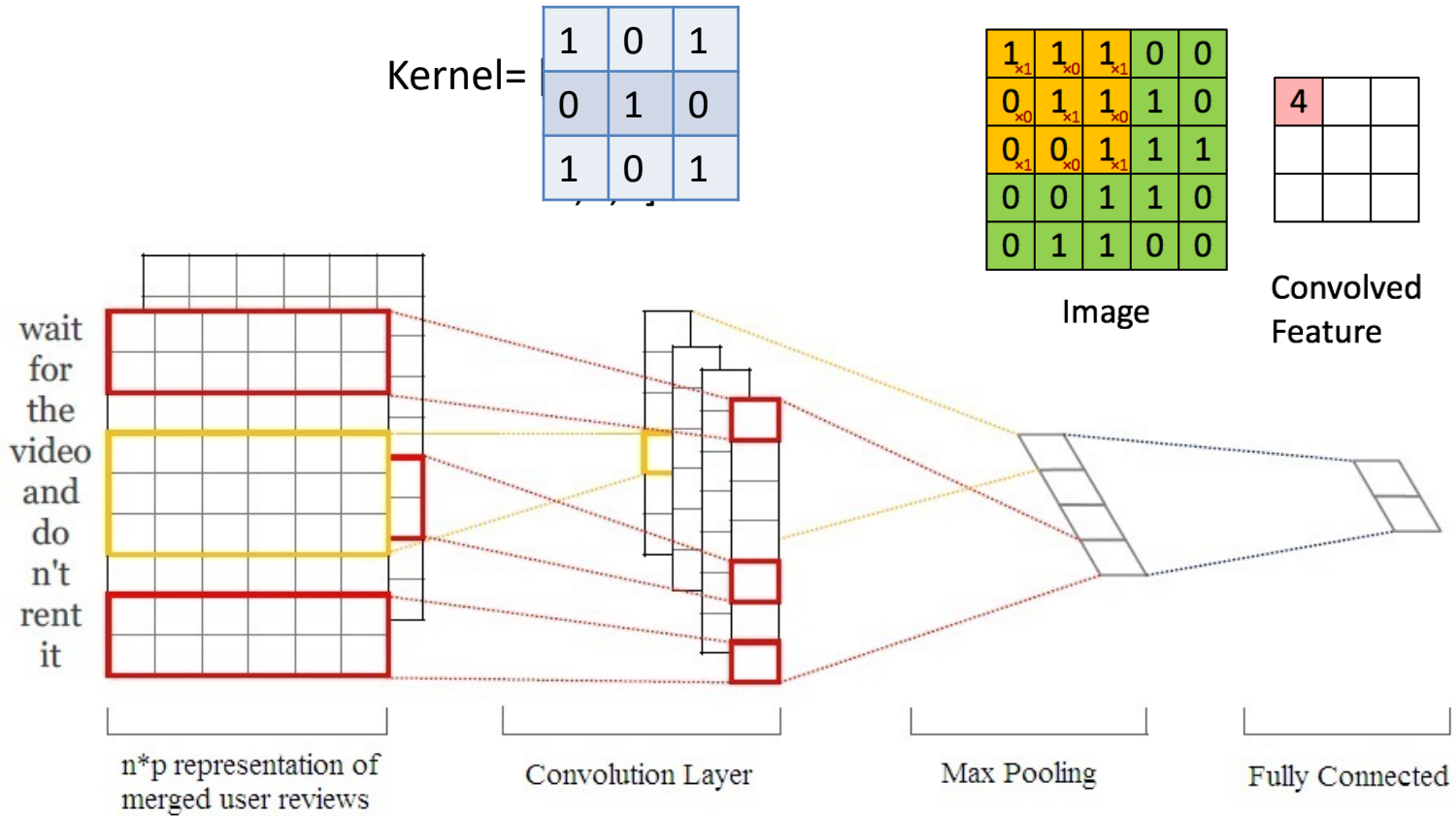
# Convolutional Neural Networks



$$z_j = f(V_{1:n}^u * K_j + b_j)$$

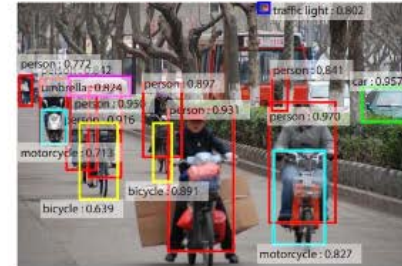
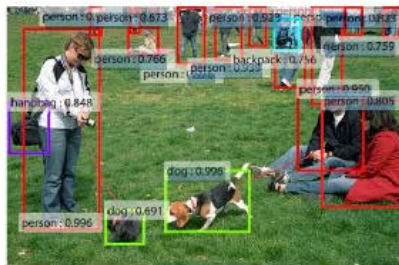
Where ReLu is used as  $f$ .

# Convolutional Neural Networks



# Applications

## Localization and Detection



Results from Faster R-CNN, Ren et al 2015

# Applications

## Computer Vision Tasks

**Classification**



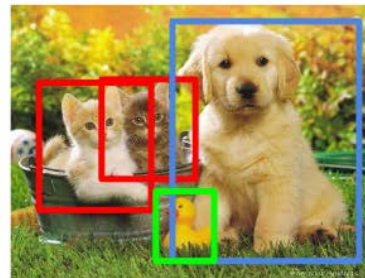
CAT

**Classification  
+ Localization**



CAT

**Object Detection**



CAT, DOG, DUCK

**Instance  
Segmentation**



CAT, DOG, DUCK

Single object

Multiple objects

# **CNN ARCHITECTURES**

# ImageNet Challenge

ImageNet Large Scale Visual Recognition Challenge ([ILSVRC](#))

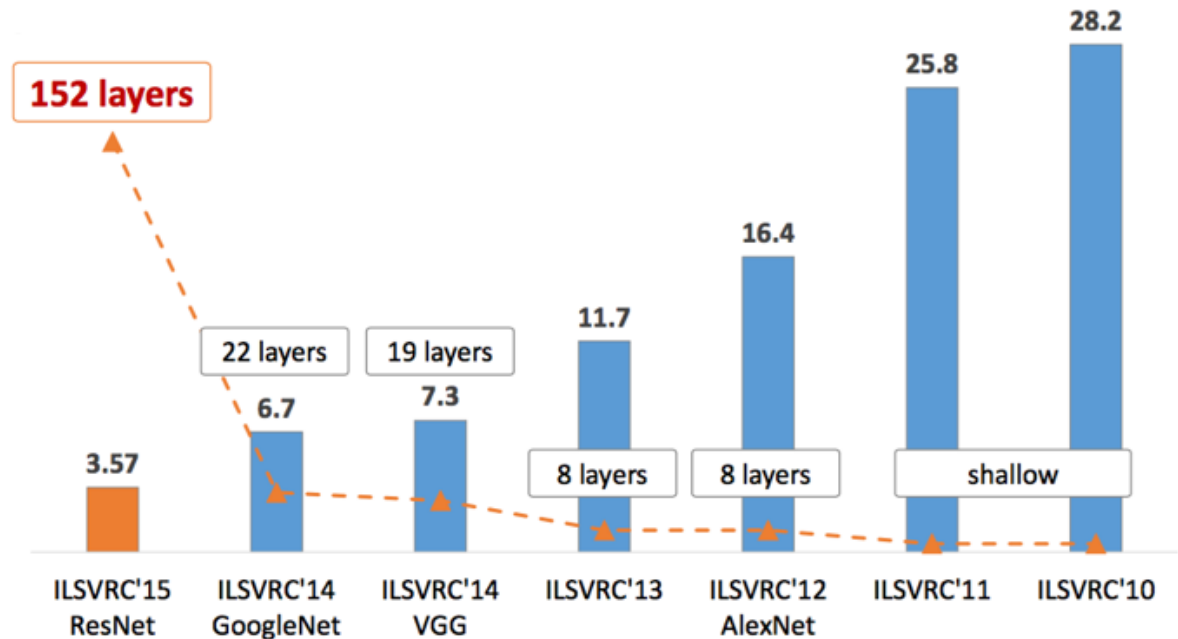
Image Classification

Classes – 1000

Train images - 1,281,167  
(732–1300)

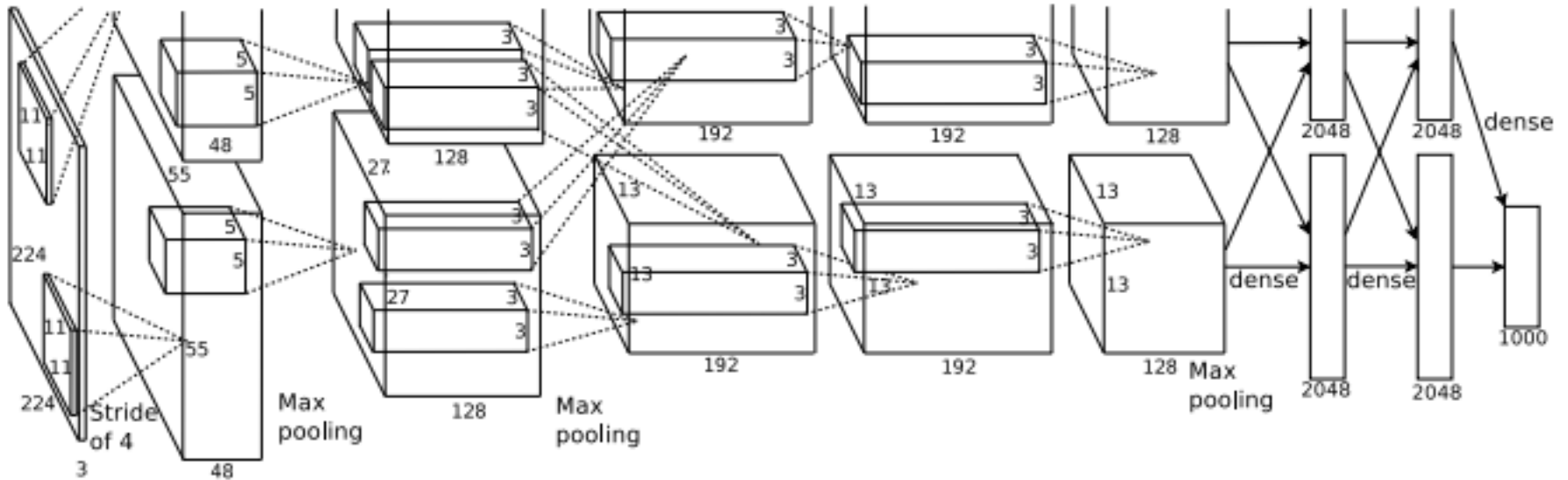
Val images - 50,000 (50)

Test images - 100,000  
(100)



# AlexNet (2012)

[Krizhevsky et al. 2012]



## Architecture:

CONV1 -> MAX POOL1 -> NORM1 -> CONV2 -> MAX POOL2 -> NORM2 ->  
CONV3 -> CONV4 -> CONV5 -> Max -> POOL3 -> FC6 -> FC7 -> FC8



# AlexNet (2012)

[227x227x3] **INPUT**

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)

Other details:

- first use of ReLU
- used Norm layers (not common anymore) - heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2

# VGG Net (2014)

*[Simonyan and Zisserman, 2014]*

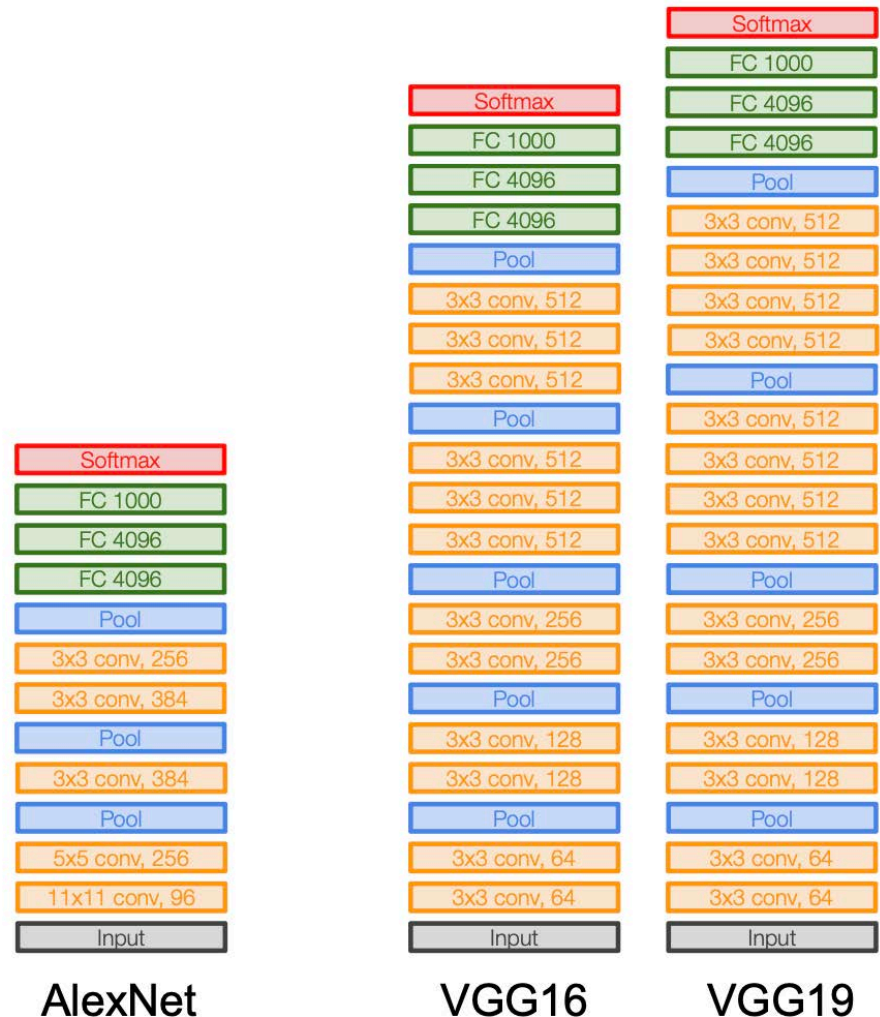
Small filters, Deeper networks

8 layers (AlexNet)

-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1  
and 2x2 MAX POOL stride 2

7.3% top 5 error in ILSVRC'14



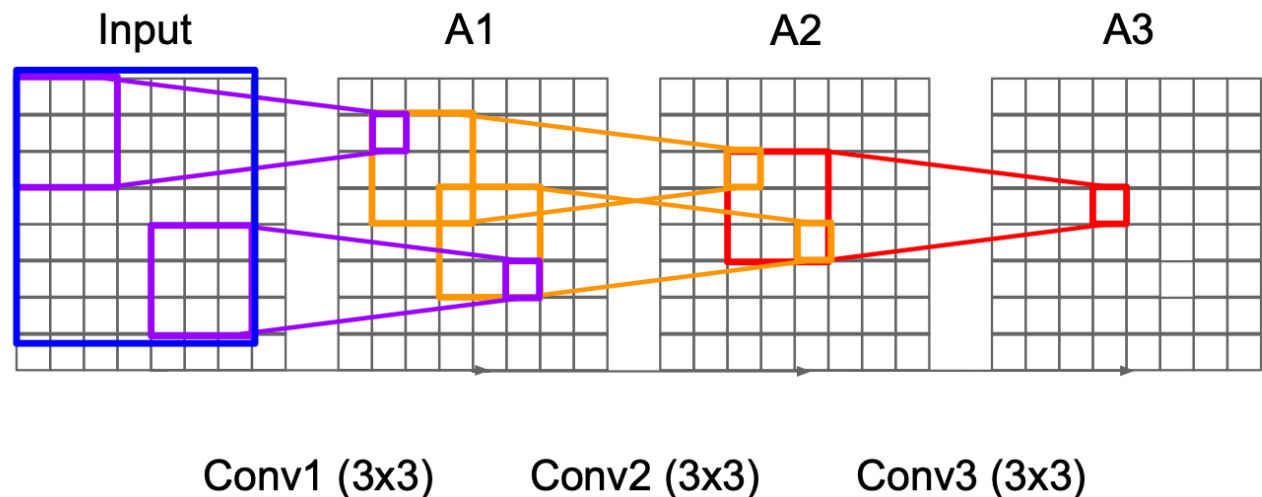
# VGG Net (2014)

## Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters:  $3 * (32C^2)$  vs.  $72C^2$  for  $C$  channels per layer

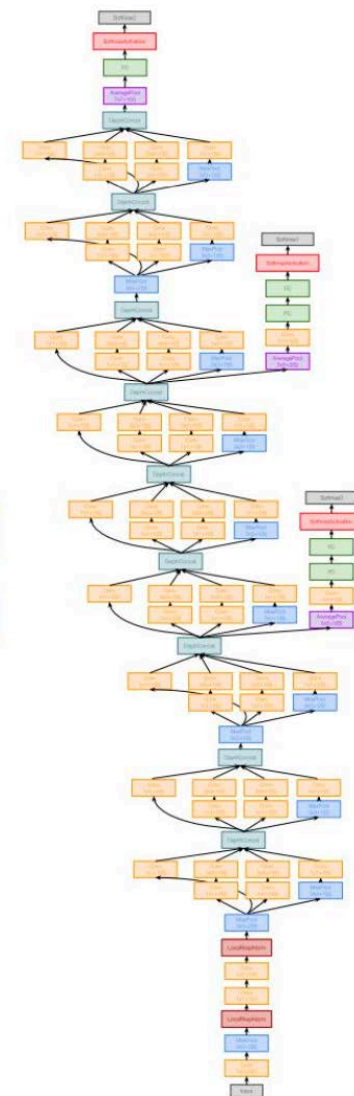
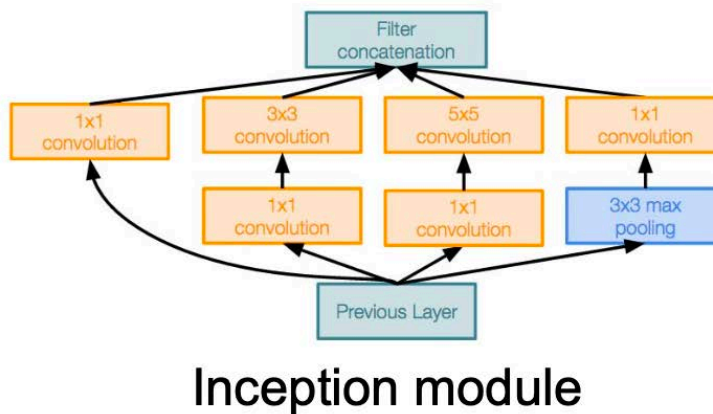


# GoogleNet (2014)

[Szegedy et al., 2014]

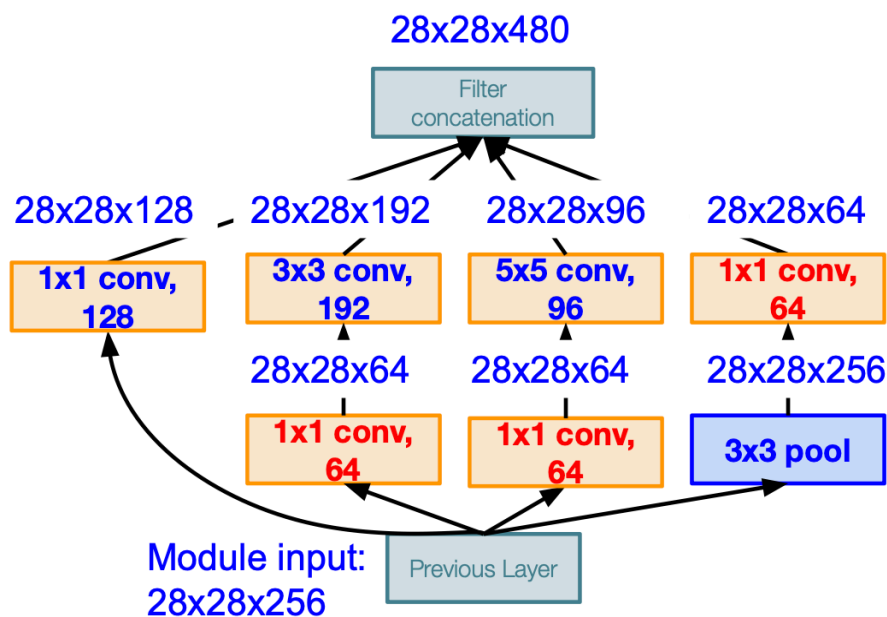
Deeper networks, with computational efficiency

- ILSVRC'14 classification winner (6.7% top 5 error)
- 22 layers
- Only 5 million parameters!  
12x less than AlexNet  
27x less than VGG-16
- Efficient “Inception” module
- No FC layers



# GoogleNet (2014)

## Computational Efficiency



Inception module with dimension reduction

## Without 1x1 convolutions

### Conv Ops:

[1x1 conv, 128] 28x28x128x1x1x256

[3x3 conv, 192] 28x28x192x3x3x256

[5x5 conv, 96] 28x28x96x5x5x256

**Total: 854M ops**

### Conv Ops:

[1x1 conv, 64] 28x28x64x1x1x256

[1x1 conv, 64] 28x28x64x1x1x256

[1x1 conv, 128] 28x28x128x1x1x256

[3x3 conv, 192] 28x28x192x3x3x64

[5x5 conv, 96] 28x28x96x5x5x64

[1x1 conv, 64] 28x28x64x1x1x256

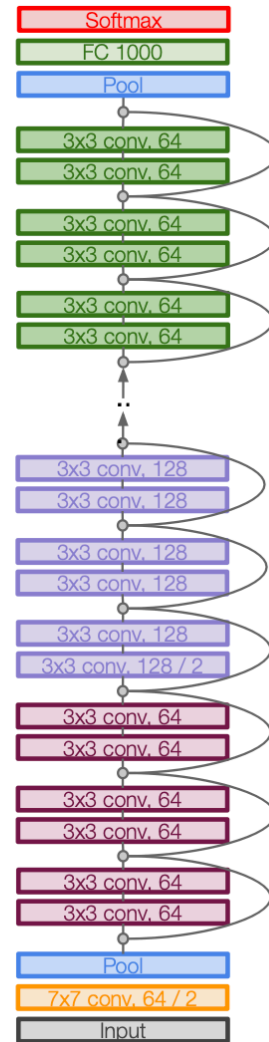
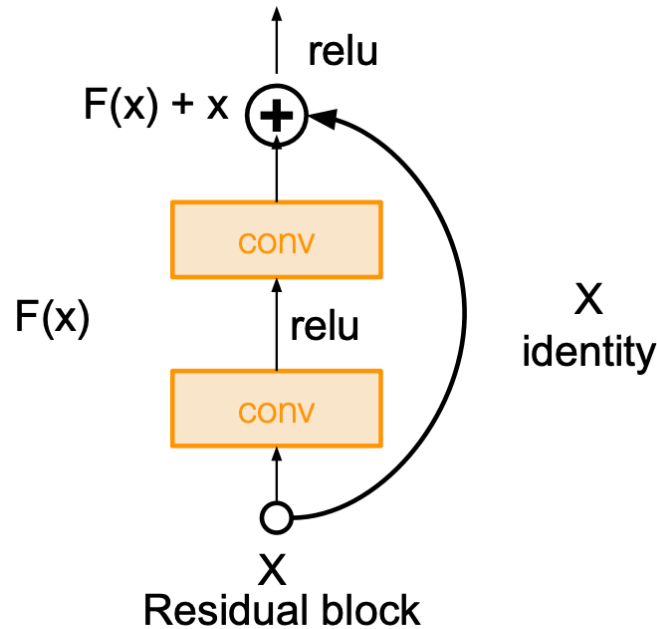
**Total: 358M ops**

# ResNet (2015)

*[He et al., 2015]*

## Very deep networks using residual connections

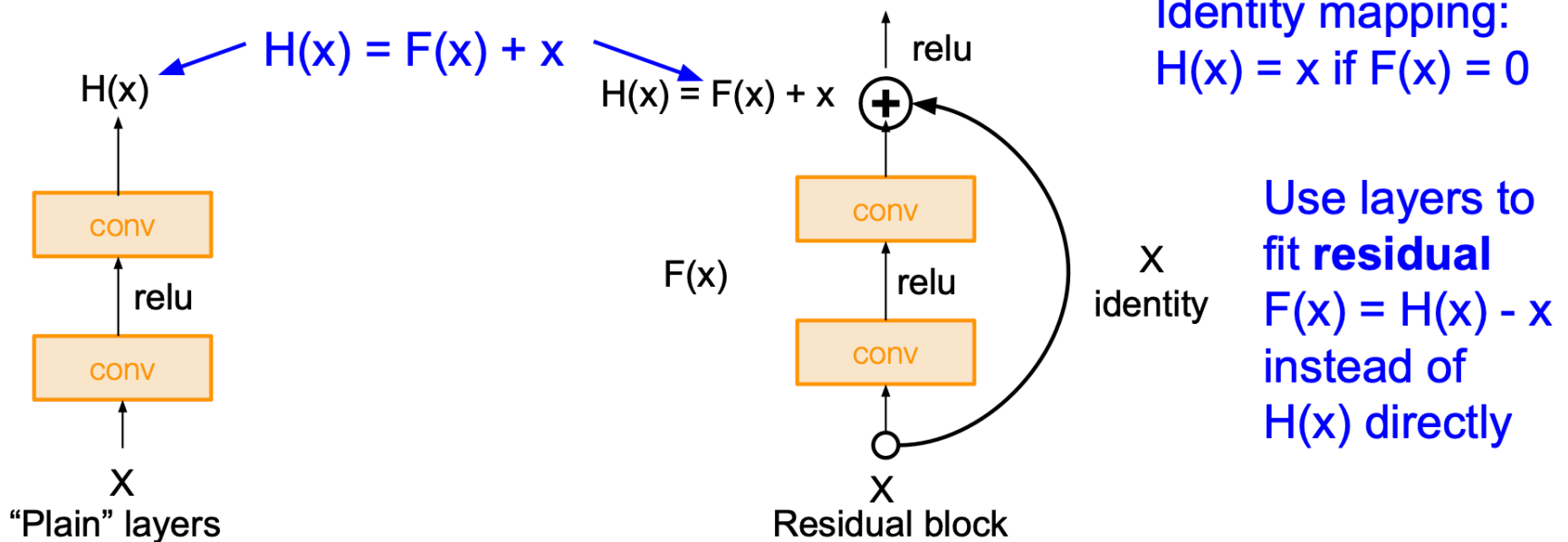
- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!



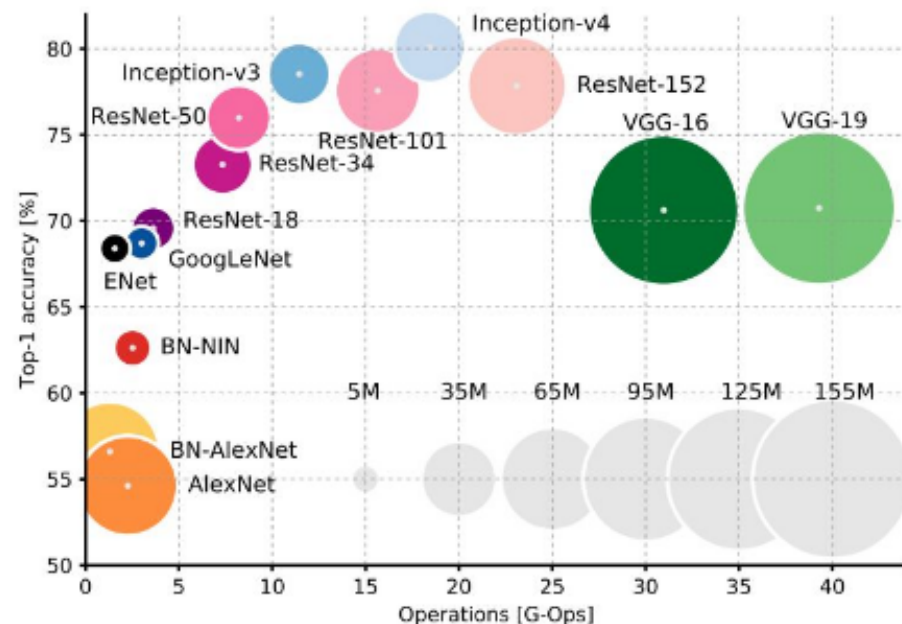
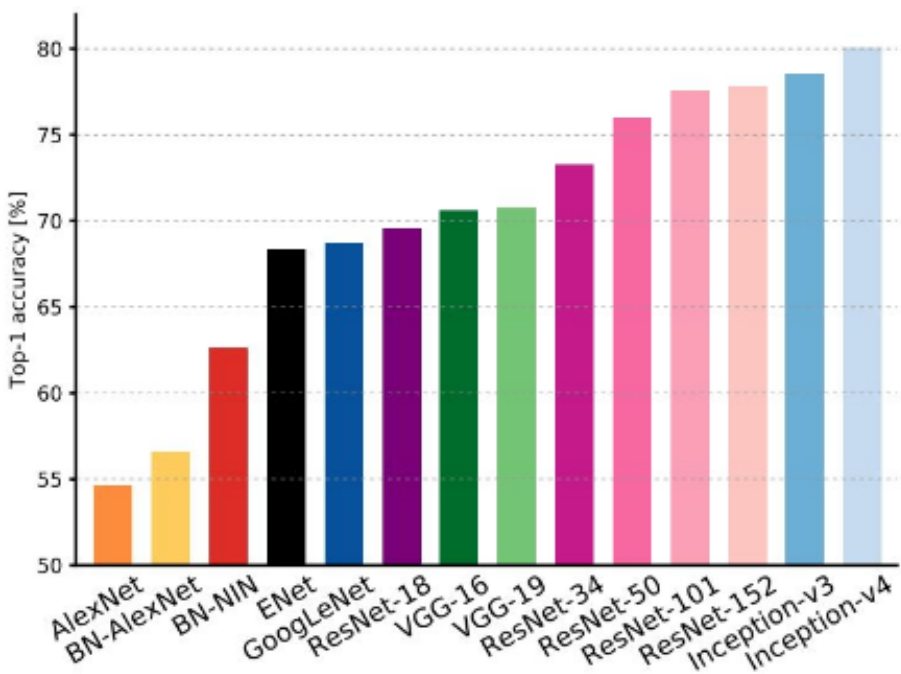
# ResNet (2015)

**Key Problem:** Training very deep neural networks

**Solution:** Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping.



# Summary of Results



An Analysis of Deep Neural Network Models for Practical Applications, 2017.



**OPTIMIZATION**

# Much of ML is optimization

## Linear Classification

$$\begin{aligned} \arg \min_w \quad & \sum_{i=1}^n \|w\|^2 + C \sum_{i=1}^n \xi_i \\ \text{s.t.} \quad & 1 - y_i x_i^T w \leq \xi_i \\ & \xi_i \geq 0 \end{aligned}$$

## Maximum Likelihood

$$\arg \max_{\theta} \sum_{i=1}^n \log p_{\theta}(x_i)$$

## K-Means

$$\arg \min_{\mu_1, \mu_2, \dots, \mu_k} J(\mu) = \sum_{j=1}^k \sum_{i \in C_j} \|x_i - \mu_j\|^2$$

# Stochastic optimization

- Goal of machine learning :
  - Minimize expected loss

$$\min_h L(h) = \mathbf{E} [\text{loss}(h(x), y)]$$

given samples  $(x_i, y_i) \ i = 1, 2 \dots m$

- This is Stochastic Optimization
  - Assume loss function is convex

# Batch (sub)gradient descent for ML

- Process all examples together in each step

$$w^{(k+1)} \leftarrow w^{(k)} - \eta_t \left( \frac{1}{n} \sum_{i=1}^n \frac{\partial L(w, x_i, y_i)}{\partial w} \right)$$

where  $L$  is the regularized loss function

- Entire training set examined at each step
- Very slow when  $n$  is very large

# Stochastic (sub)gradient descent

- “Optimize” one example at a time
- Choose examples randomly (or reorder and choose in order)
  - Learning representative of example distribution

for  $i = 1$  to  $n$ :

$$w^{(k+1)} \leftarrow w^{(k)} - \eta_t \frac{\partial L(w, x_i, y_i)}{\partial w}$$

where  $L$  is the regularized loss function

# Stochastic (sub)gradient descent

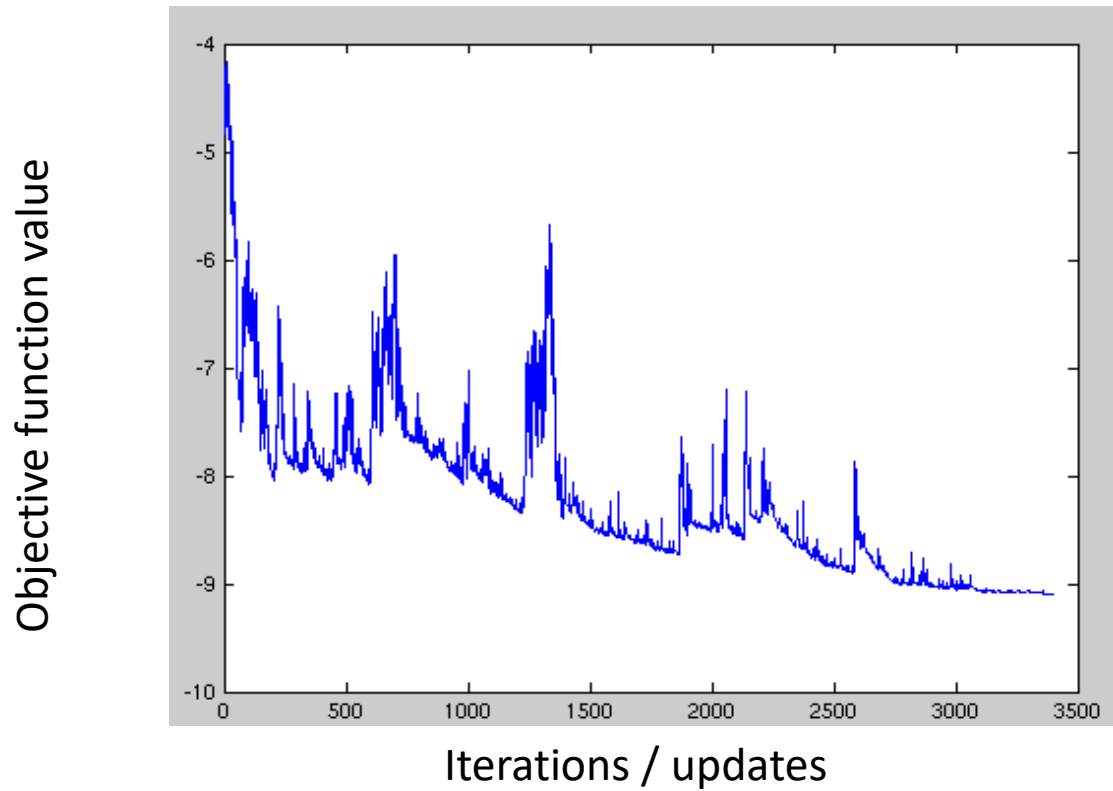
for  $i = 1$  to  $n$ :

$$w^{(k+1)} \leftarrow w^{(k)} - \eta_t \frac{\partial L(w, x_i, y_i)}{\partial w}$$

where  $L$  is the regularized loss function

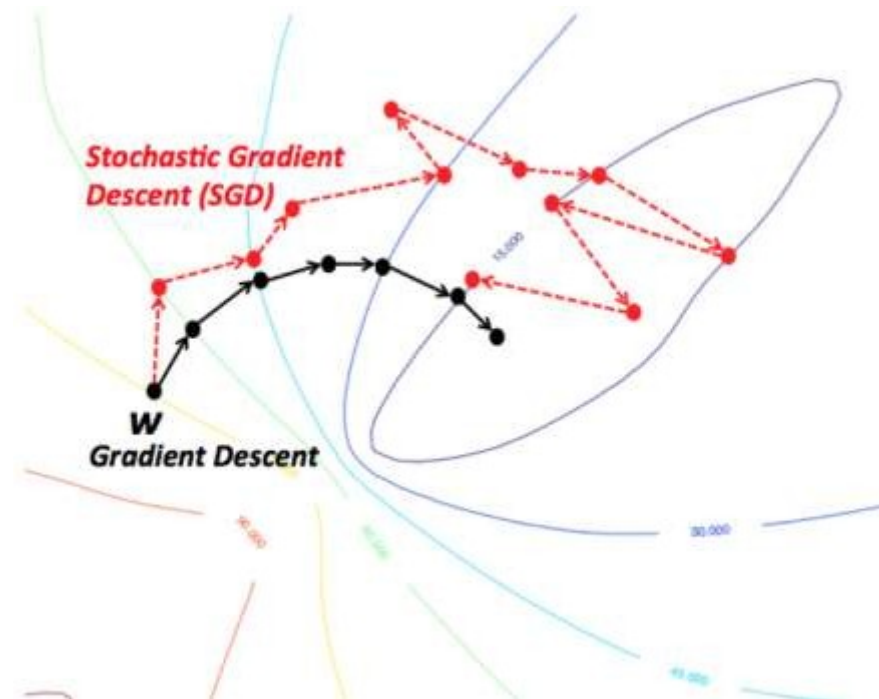
- Equivalent to online learning (the weight vector  $w$  changes with every example)
- Convergence guaranteed for convex functions (to local minimum)

# SGD convergence



# SGD - Issues

- Convergence very sensitive to learning rate  
( $\eta_t$ ) (oscillations near solution due to probabilistic nature of sampling)
  - Might need to decrease with time to ensure the algorithm converges eventually
- Basically – SGD good for machine learning with large data sets!





# Mini-batch SGD

- Stochastic – 1 example per iteration
- Batch – All the examples!
- Mini-batch SGD:
  - Sample  $m$  examples at each step and perform SGD on them
- Allows for parallelization, but choice of  $m$  based on heuristics

# Example: Text categorization

- **Example by Leon Bottou:**
  - **Reuters RCV1** document corpus
    - Predict a category of a document
      - One **vs.** the rest classification
  - **$n = 781,000$**  training examples (documents)
  - 23,000 test examples
  - **$d = 50,000$**  features
    - One feature per word
    - Remove stop-words
    - Remove low frequency words

# Example: Text categorization

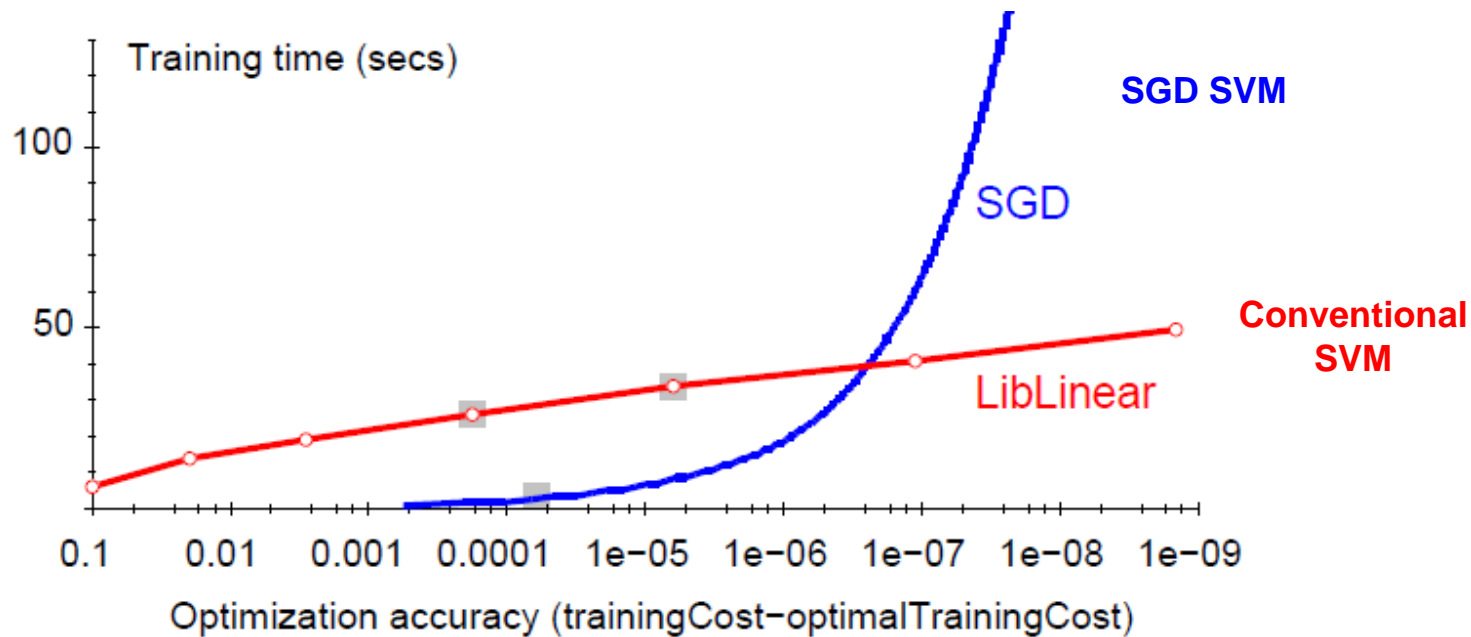
- **Questions:**

- (1) Is **SGD** successful at minimizing  $f(\mathbf{w}, \mathbf{b})$ ?
- (2) How quickly does **SGD** find the min of  $f(\mathbf{w}, \mathbf{b})$ ?
- (3) What is the error on a test set?

	<i>Training time</i>	<i>Value of <math>f(\mathbf{w}, \mathbf{b})</math></i>	<i>Test error</i>
Standard SVM	23,642 secs	0.2275	6.02%
“Fast SVM”	66 secs	0.2278	6.03%
<b>SGD SVM</b>	1.4 secs	0.2275	6.02%

- (1) SGD-SVM is successful at minimizing the value of  $f(\mathbf{w}, \mathbf{b})$
- (2) SGD-SVM is super fast
- (3) SGD-SVM test set error is comparable

# Optimization “Accuracy”



Optimization quality:  $| f(w,b) - f(w^{opt},b^{opt}) |$

For optimizing  $f(w,b)$  within reasonable quality SGD-SVM is super fast

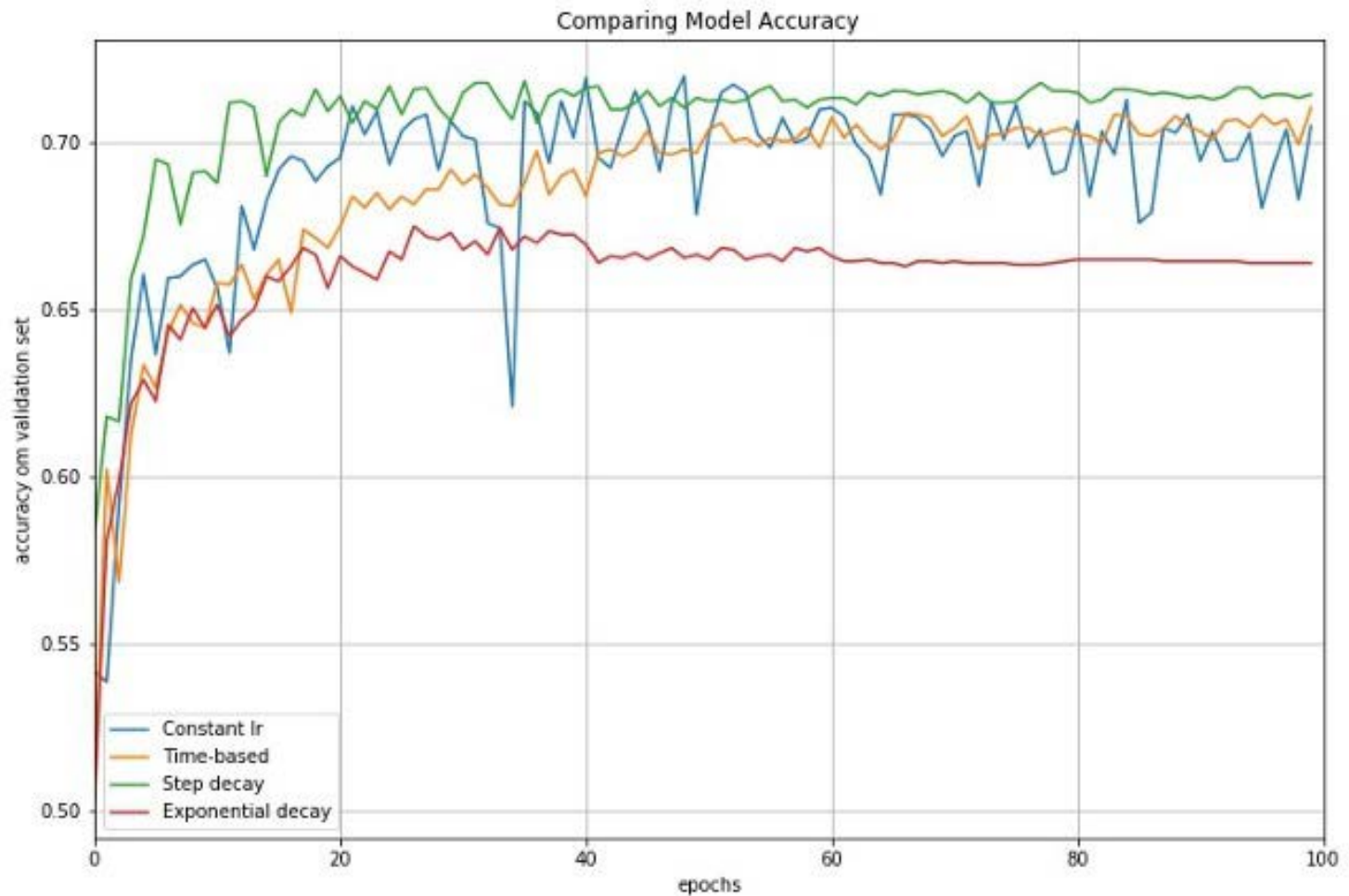
# Practical Considerations

- Need to choose learning rate  $\eta$  and  $t_0$

$$w_{t+1} \leftarrow w_t - \frac{\eta_0}{t + t_0} \left( w_t + C \frac{\partial L(x_i, y_i)}{\partial w} \right)$$

- Leon suggests:
  - Choose  $t_0$  so that the expected initial updates are comparable with the expected size of the weights
  - Choose  $\eta$ :
    - Select a **small subsample**
    - Try various rates  $\eta$  (e.g., 10, 1, 0.1, 0.01, ...)
    - Pick the one that most reduces the cost
    - Use  $\eta$  for next 100k iterations on the full dataset

# Learning rate comparison



# **ACCELERATED GRADIENT DESCENT**

# Stochastic gradient descent

Idea: Perform a parameter update for each training example  $x(i)$  and label  $y(i)$

Update:  $\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x(i), y(i))$

Performs redundant computations for large datasets



# Momentum gradient descent

- Idea: Overcome ravine oscillations by momentum

- 

Update:

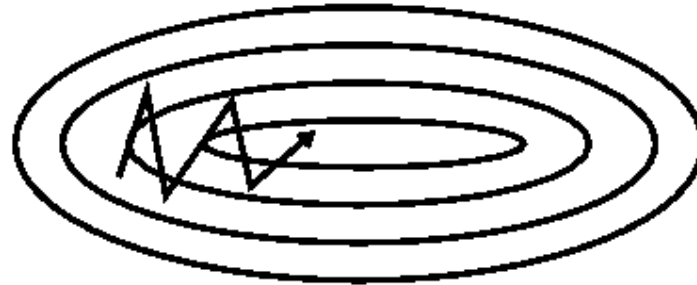
- $V_t = \gamma V_{t-1} + \eta \cdot \nabla_{\theta} J(\theta)$

- $\theta = \theta - V_t$

SGD

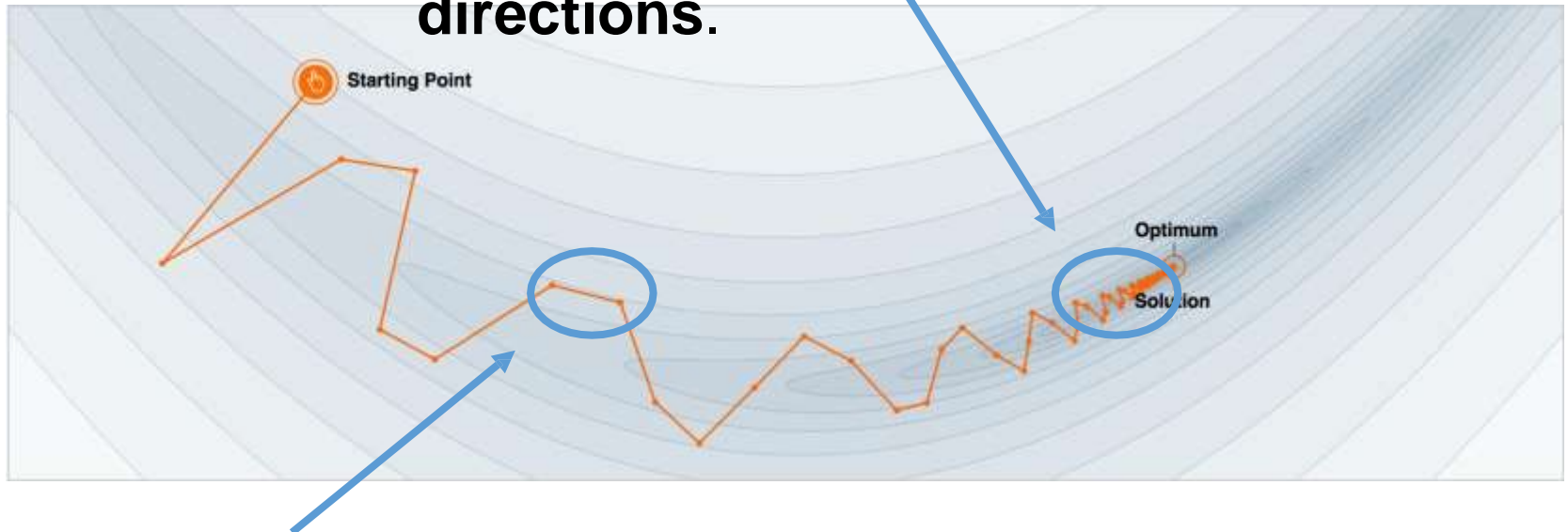


SGD with  
momentum



# Why Momentum Really Works

The momentum term **reduces updates for dimensions whose gradients change directions.**



The momentum term **increases for dimensions whose gradients point in the same directions.**

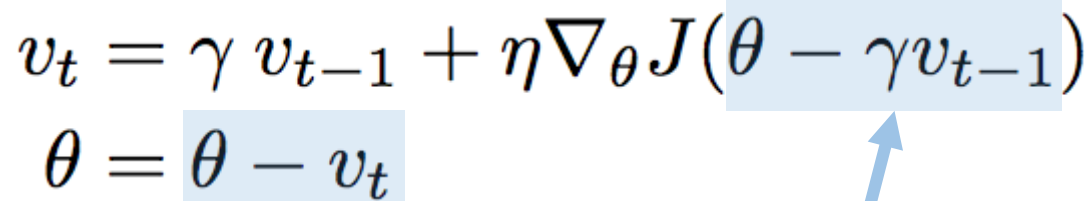
Demo :

<http://distill.pub/2017/momentum/>

# Nesterov accelerated gradient

- However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory.
- We would like to have a smarter ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.
- **Nesterov accelerated gradient** gives us a way of it.

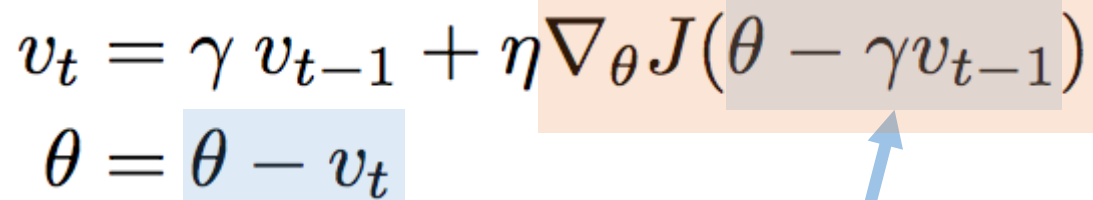
# Nesterov accelerated gradient

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$


**Approximation of the next  
position of the  
parameters(predict)**

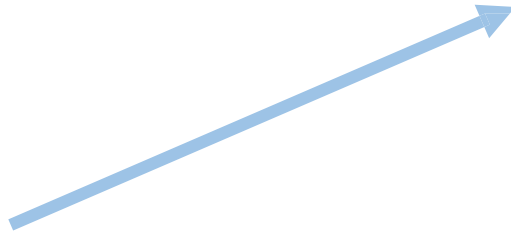
# Nesterov accelerated gradient

Approximation of the next  
position of the parameters'  
gradient(**correction**)

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$


Approximation of the next  
position of the  
parameters(**predict**)

# Nesterov accelerated gradient



**Blue line :**  
**predict**

**Red line :**  
**correction**

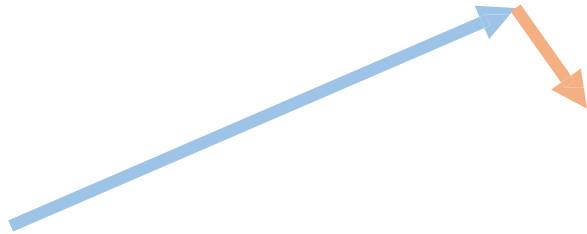
**Green line :accumulated**  
**gradient**

Approximation of the next  
position of the parameters'  
gradient(**correction**)

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next  
position of the  
parameters(**predict**)

# Nesterov accelerated gradient



**Blue line :**  
**predict**

**Red line :**  
**correction**

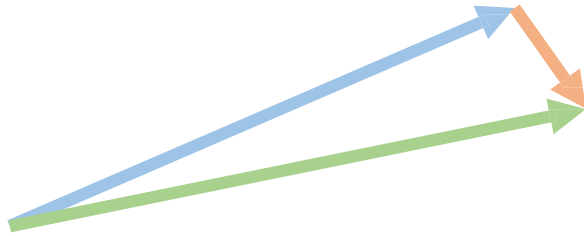
**Green line :accumulated**  
**gradient**

Approximation of the next  
position of the parameters'  
gradient(**correction**)

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next  
position of the  
parameters(**predict**)

# Nesterov accelerated gradient



**Blue line :**  
**predict**

**Red line :**  
**correction**

**Green line :accumulated**  
**gradient**

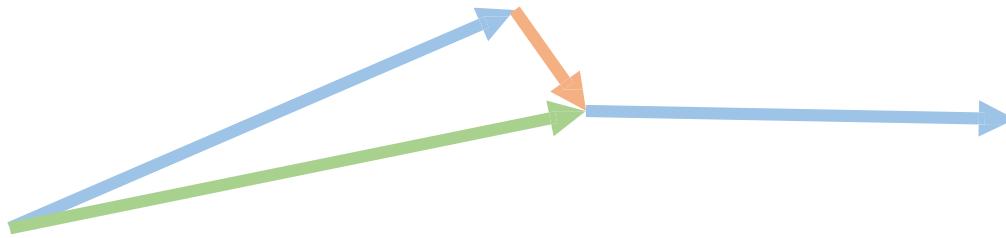
Approximation of the next  
position of the parameters'  
gradient(**correction**)

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next  
position of the  
parameters(**predict**)



# Nesterov accelerated gradient



**Blue line :**  
**predict**

**Red line :**  
**correction**

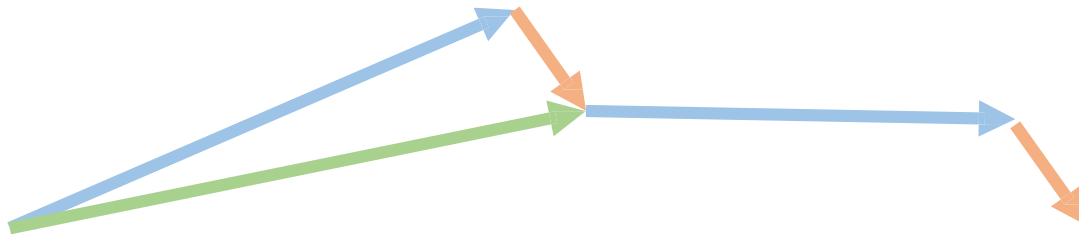
**Green line :accumulated**  
**gradient**

Approximation of the next  
position of the parameters'  
gradient(**correction**)

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next  
position of the  
parameters(**predict**)

# Nesterov accelerated gradient



**Blue line :**  
**predict**

**Red line :**  
**correction**

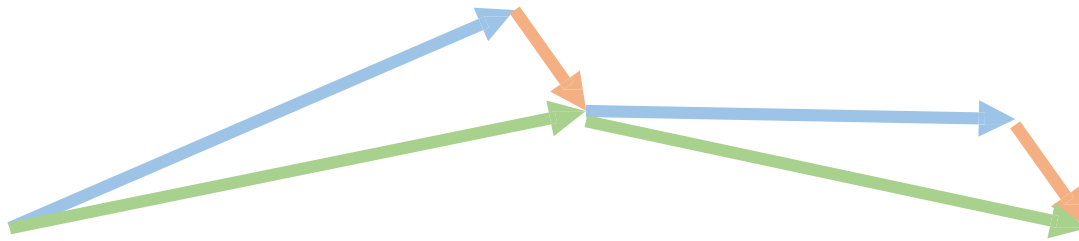
**Green line :accumulated**  
**gradient**

Approximation of the next  
position of the parameters'  
gradient(**correction**)

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next  
position of the  
parameters(**predict**)

# Nesterov accelerated gradient



**Blue line :**  
**predict**

**Red line :**  
**correction**

**Green line :accumulated**  
**gradient**

Approximation of the next  
position of the parameters'  
gradient(**correction**)

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next  
position of the  
parameters(**predict**)

# Nesterov accelerated gradient

- This anticipatory update **prevents** us from **going too fast** and **results in increased responsiveness**.
- Now , we can adapt our updates to the slope of our error function and **speed up SGD** in turn.

# What's next...?

- We also want to **adapt** our updates to each individual parameter to perform larger or smaller updates **depending on their importance**.
  - Adagrad
  - Adadelta
  - RMSprop
  - Adam

# Adagrad

- Adagrad adapts the learning rate to the parameters
  - Performing larger updates for infrequent
  - Performing smaller updates for frequent parameters.
- Ex.
  - Training large-scale neural nets at Google that learned to recognize cats in Youtube videos.

# Different learning rate for every parameter

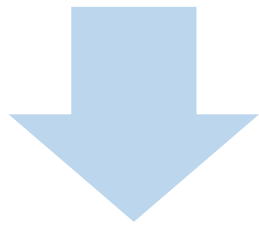
- Previous methods :
  - **we used the same learning rate  $\eta$  for all parameters  $\theta$**
- Adagrad :
  - It uses a different learning rate for every parameter  $\theta_i$  at every time step  $t$

# Adagrad

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$$G_t = \mathbb{R}^{d \times d} = \begin{pmatrix} \square & \dots & \circ & \dots & \circ \\ \vdots & & \vdots & & \vdots \\ \circ & \dots & \square & \dots & \circ \\ \vdots & & \vdots & & \vdots \\ \circ & \dots & \circ & \dots & \square \end{pmatrix}$$



Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_{\theta} J(\theta_i)$$

Vectorize

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$



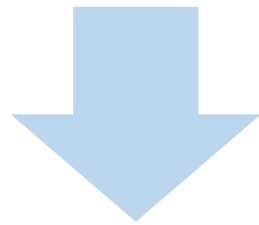
# Adagrad

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$\mathbb{R}^{d \times d}$

$$G_t = \begin{pmatrix} \square & \dots & \circ & \dots & \circ \\ \vdots & & \vdots & & \vdots \\ \circ & \dots & \square & \dots & \circ \\ \vdots & & \vdots & & \vdots \\ \circ & \dots & \circ & \dots & \square \end{pmatrix}$$



Adagrad modifies the general learning rate  $\eta$  based on the **past gradients that have been computed for  $\theta_i$**

Adagrad

d

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_{\theta} J(\theta_i)$$

Vectorize

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

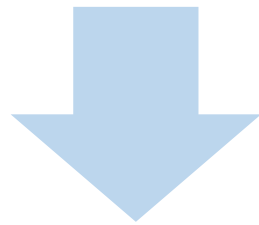
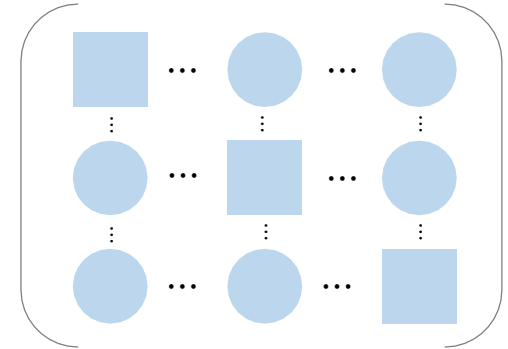
# Adagrad

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$\mathbb{R}^{d \times d}$

$$G_t =$$



$G_t$  is a diagonal matrix where each diagonal element  $(i,i)$  is the sum of the squares of the gradients  $\theta_i$  up to time step  $t$ .

Adagrad

d

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}} + \epsilon} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_{\theta} J(\theta_i)$$

Vectorize

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t} + \epsilon} \odot g_t.$$

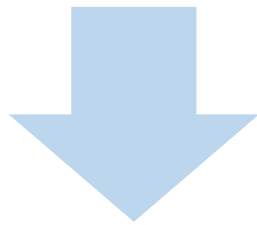
# Adagrad

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$\mathbb{R}^{d \times d}$

$$G_t = \begin{pmatrix} \square & \dots & \circ & \dots & \circ \\ \vdots & & \vdots & & \vdots \\ \circ & \dots & \square & \dots & \circ \\ \vdots & & \vdots & & \vdots \\ \circ & \dots & \circ & \dots & \square \end{pmatrix}$$



$\epsilon$  is a smoothing term that avoids division by zero (usually on the order of  $1e-8$ ).

Adagrad

d

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_{\theta} J(\theta_i)$$

Vectorize

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

# Adagrad's advantages

- Advantages :
  - It is well-suited for dealing with sparse data.
  - It greatly improved the robustness of SGD.
  - It eliminates the need to manually tune the learning rate.

# Adagrad's disadvantage

- Disadvantage :
  - Main weakness is its accumulation of the squared gradients in the denominator.

# Adagrad's disadvantage

- The disadvantage causes the learning rate to shrink and become infinitesimally small. The algorithm can no longer acquire additional knowledge.
- The following algorithms aim to resolve this flaw.
  - Adadelta
  - RMSprop
  - Adam

# Adadelta : extension of Adagrad

- Adadelta is an extension of Adagrad.
- Adagrad :
  - It accumulate all past squared gradients.
- Adadelta :
  - It restricts the window of accumulated past gradients to some fixed size  $w$ .

# Adadelta

- Instead of inefficiently storing, the sum of gradients is recursively defined as a decaying average of all past squared gradients.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2$$

- $E[g^2]_t$  : The running average at time step  $t$ .
- $\gamma$  : A fraction similarly to the Momentum term, around 0.9



# Adadelta

Adagrad

d

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

SGD

$$\begin{aligned}\Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t\end{aligned}$$



Adadelta

a

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

# Adadelta

Adagrad

d

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

SGD

$$\begin{aligned}\Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t\end{aligned}$$



Replace the diagonal matrix  $G_t$  with the decaying average over past squared gradients  $E[g^2]_t$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

# Adadelta

Adagrad

d

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

SGD

$$\begin{aligned}\Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t\end{aligned}$$



Replace the diagonal matrix  $G_t$  with the decaying average over past squared gradients  $E[g^2]_t$

Adadelt

a

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$




Adadelt

a

$$\Delta\theta_t = -\frac{\eta}{\text{RMS}[g]_t} g_t$$

# Adadelta

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2$$


$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma)\Delta\theta_t^2$$



$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

**Adadelt**

**a**

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}g_t$$

**Adadelt**

**a**

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t}g_t$$

# Adadelta

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2$$

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma)\Delta\theta_t^2$$



We approximate learning rate with the RMS of parameter updates until the previous time step.

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

**Adadelt**

a

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}g_t$$

**Adadelt**

a

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t}g_t$$

# Adadelta update rule

- Replacing the learning rate  $\eta$  in the previous update rule with  $RMS[\Delta\theta]_{t-1}$  finally yields the Adadelta update rule:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

- Note : **we do not even need to set a default learning rate**

# Adam

- Adam's feature :
  - Storing an exponentially decaying average of past squared gradients  $v_t$  like Adadelta and RMSprop
  - Keeping an exponentially decaying average of past gradients  $m_t$ , similar to momentum.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \rightarrow$$

The first moment (the mean)

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \rightarrow$$

The second moment (the uncentered

# Adam

- As  $m_t$  and  $v_t$  are initialized as vectors of 0's, they are biased towards zero.
  - Especially during the initial time steps
  - Especially when the decay rates are small
    - (i.e.  $\beta_1$  and  $\beta_2$  are close to 1).
- Counteracting these biases in Adam

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

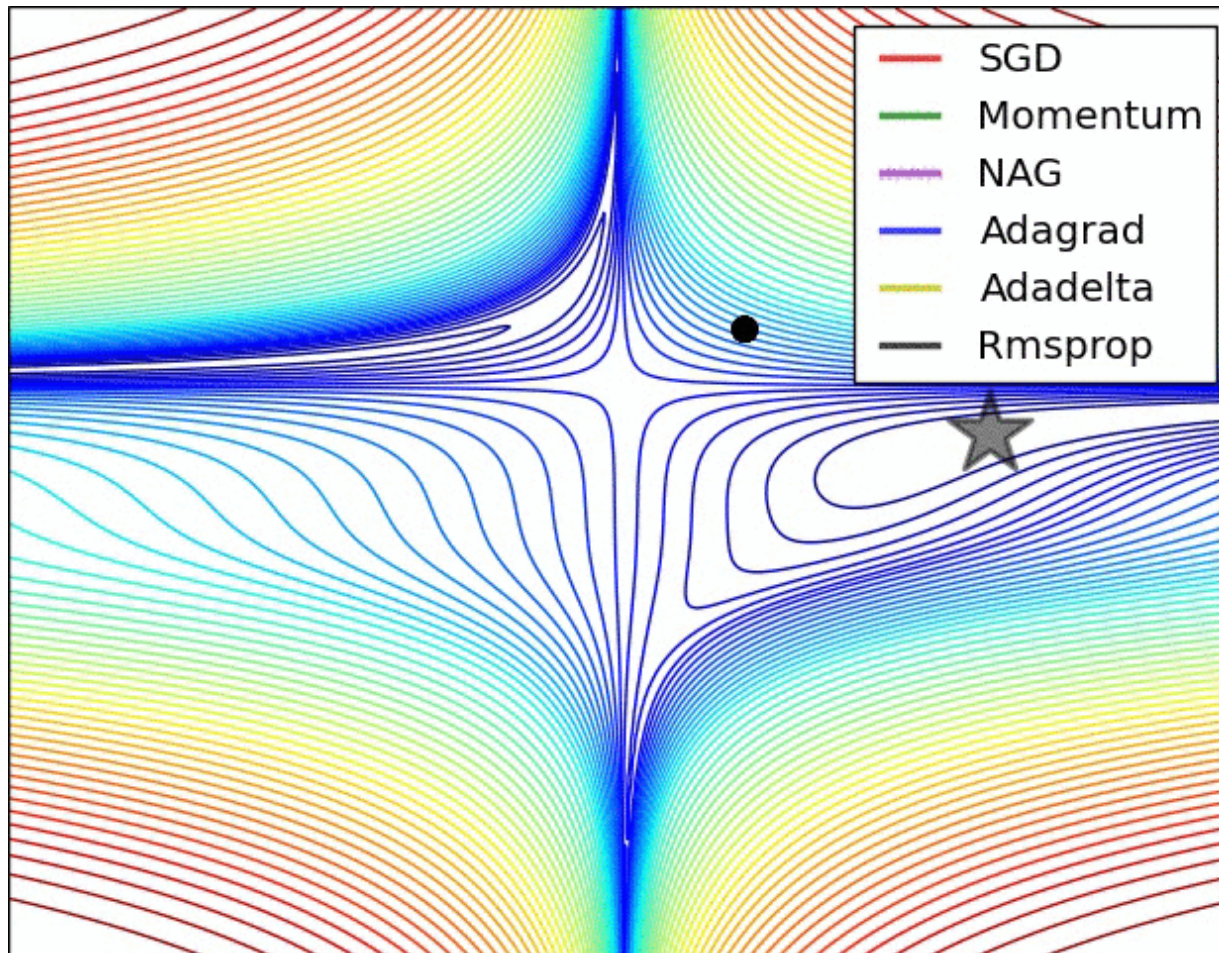
Adam

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

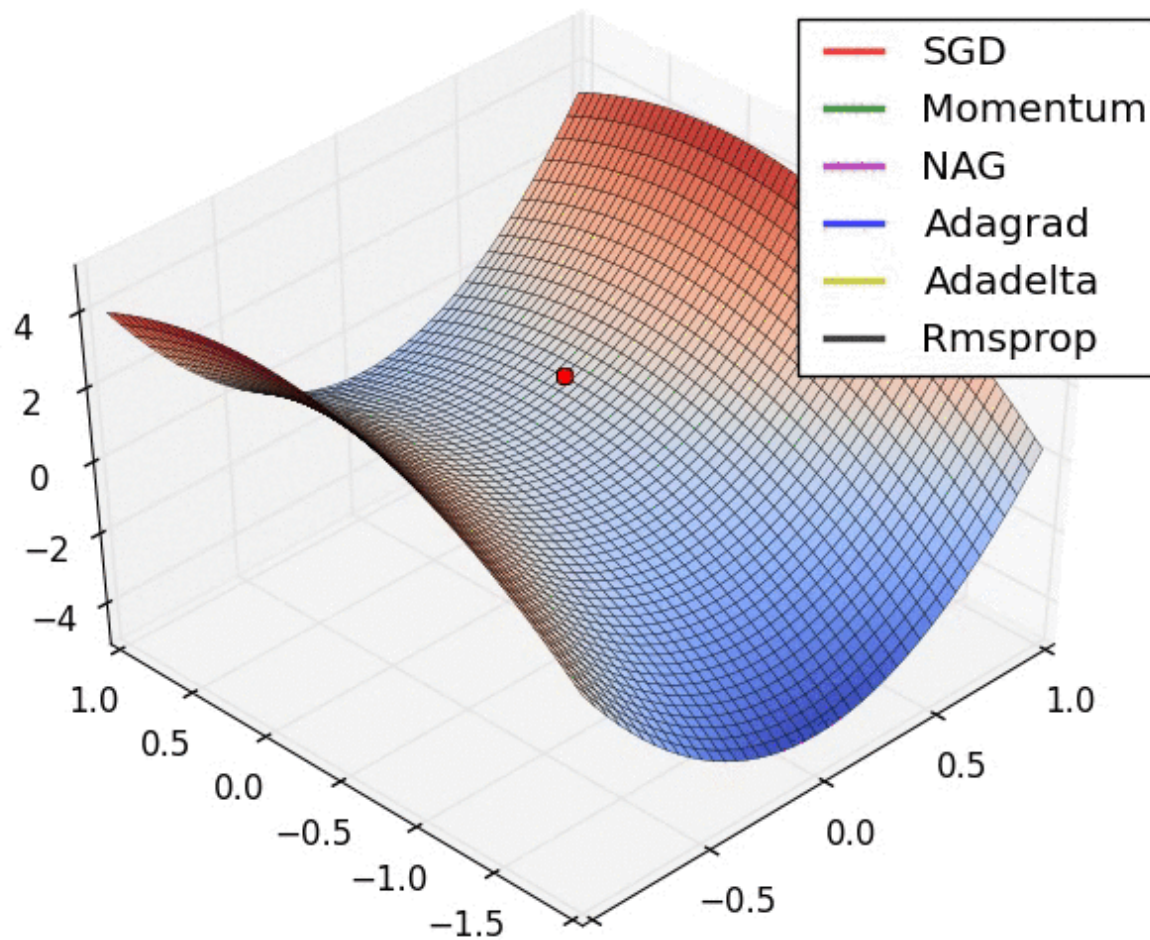
Note : default values of 0.9 for  $\beta_1$ , 0.999 for  $\beta_2$ , and  $10^{-8}$  for  $\epsilon$



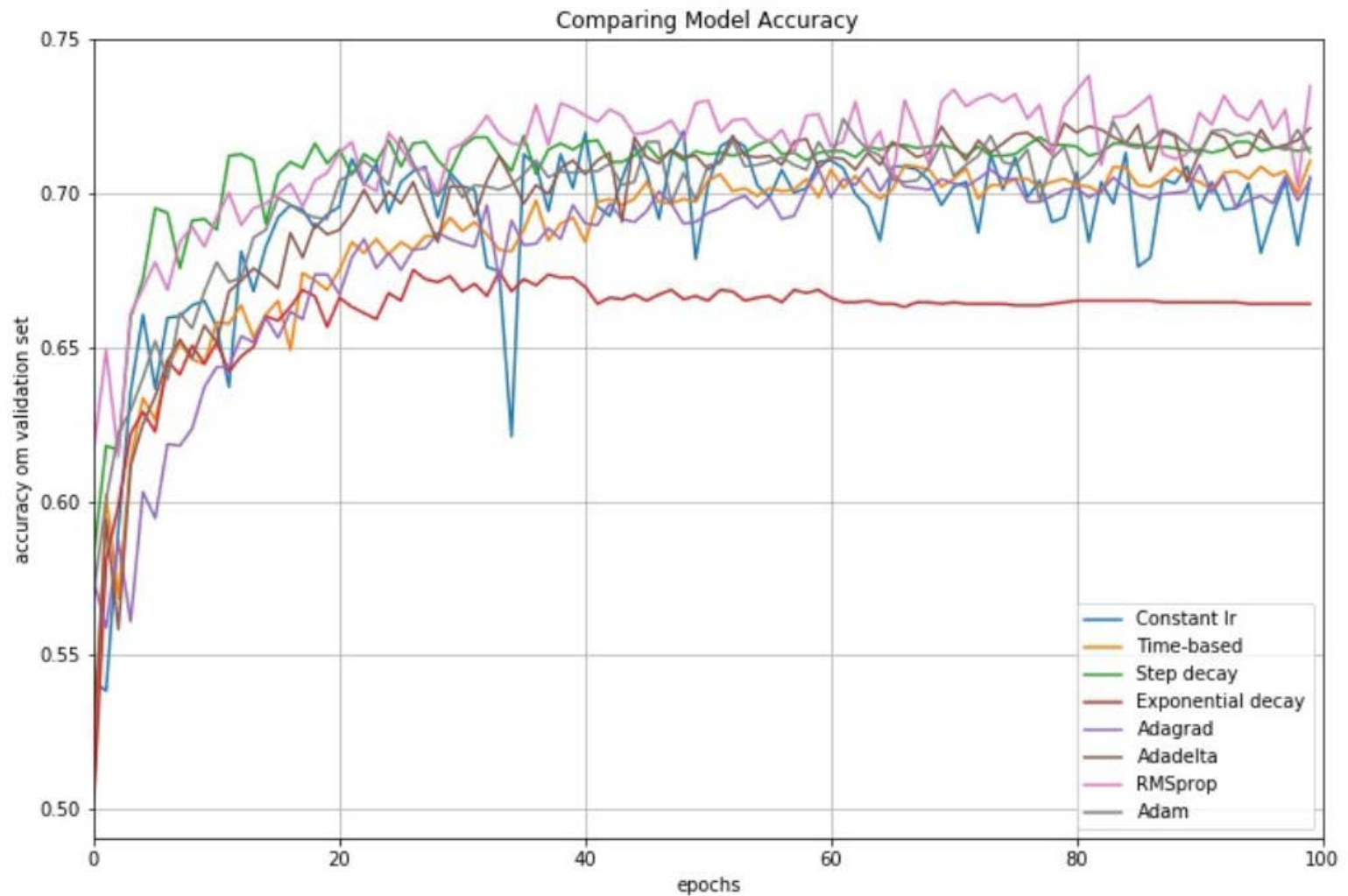
# Visualization



# Visualization



# Enhancements comparison



# Summary

- There are two main ideas at play:
  - **Momentum** : Provide consistency in update directions by incorporating past update directions.
  - **Adaptive gradient** : Scale the scale updates to individual variables using the second moment in that direction.
  - This also relates to adaptively altering step length for each direction.

# References

- [CS231n: Convolutional Neural Networks for Visual Recognition. Andrej Karpathy](http://cs231n.github.io/convolutional-networks/)
- <https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5>