# DS 4400

## Machine Learning and Data Mining I

Alina Oprea
Associate Professor, CCIS
Northeastern University

November 6 2018

# Review

- Deep Learning has the ability to learn hierarchy of features
  - Performs better with more training data
- Neural Networks can be shallow or deep
  - Their power is given by non-linear activations
  - XOR can be learned with 1 hidden layer
- Feed-Forward architectures
  - Multi-Layer Perceptron (MLP) is fully connected
  - Convolutional Neural Networks
  - Activation functions: sigmoid, ReLU, tanh
  - Can be used with sigmoid in last layer for binary classification and softmax for multi-class classification

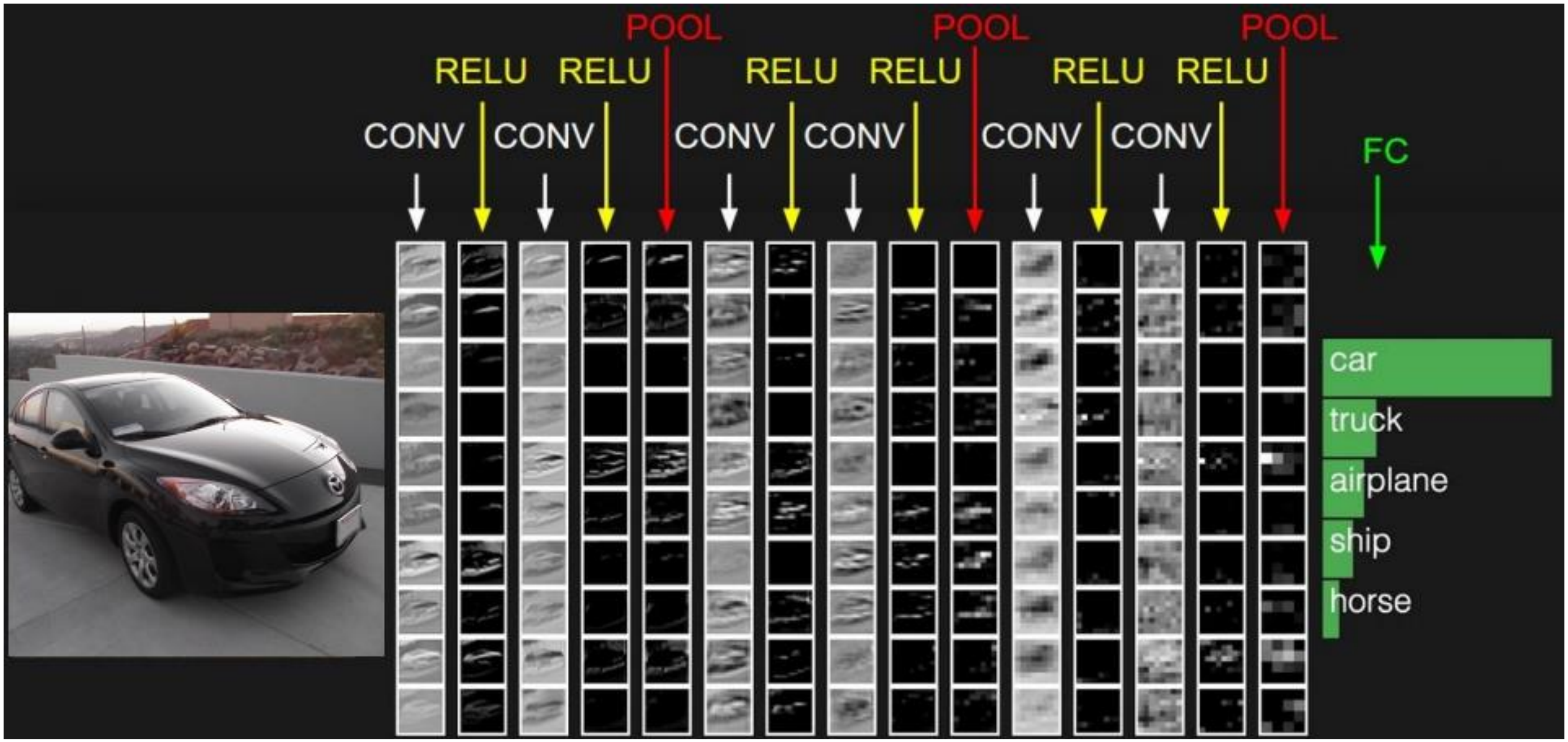
# Outline

- Convolutional Neural Networks
  - Recap: convolution layer
  - Max pooling
  - Architectures
- Training with backpropagation
  - Initialization
  - Derivation of gradients
  - Example

# Convolutional Nets

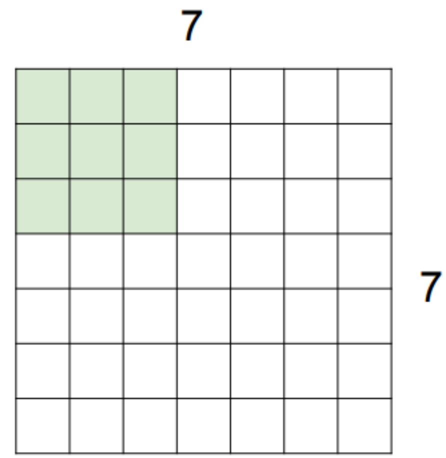
- Particular type of Feed-Forward Neural Nets
  - Invented by [LeCun 89]
- Applicable to data with natural grid topology
  - Time series
  - Images
- Use convolutions on at least one layer
  - Convolution is a linear operation
  - Also use pooling operation
  - Used for dimensionality reduction and learning hierarchical feature representations

# Convolutional Nets

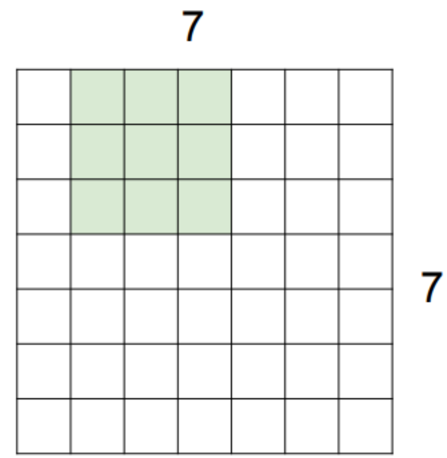


# Convolutions

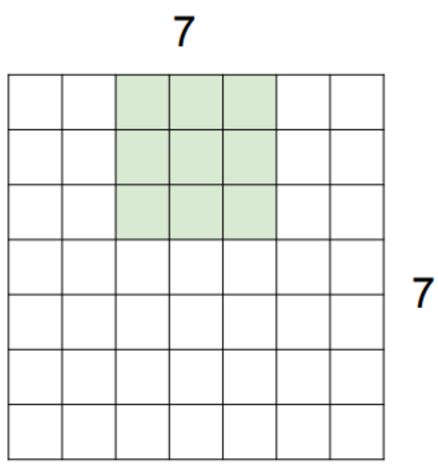
A closer look at spatial dimensions:

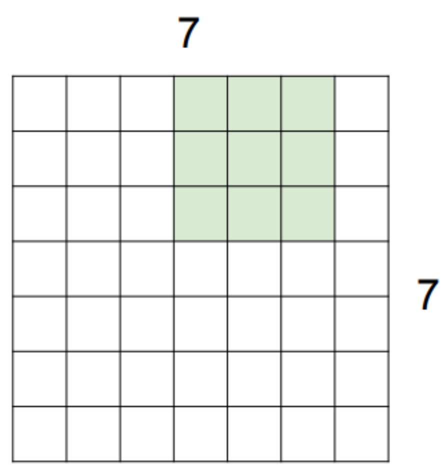


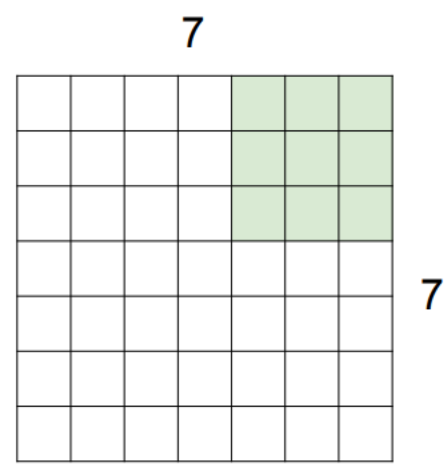

7x7 input (spatially)
assume 3x3 filter




**=> 5x5 output**

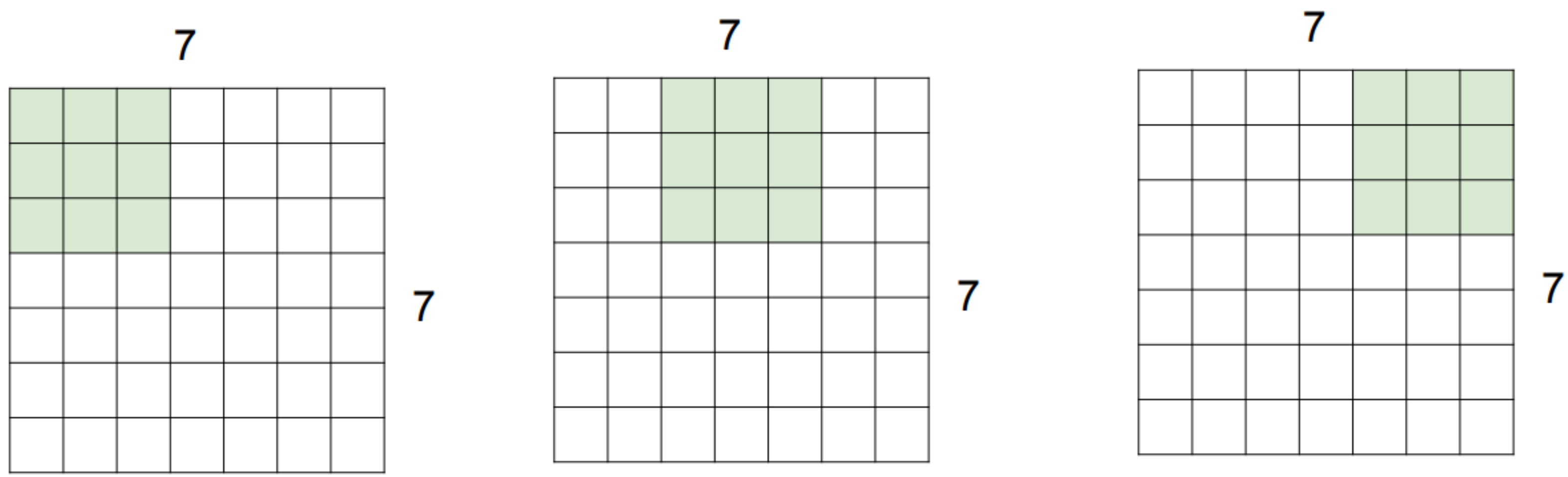


# Convolutions with stride

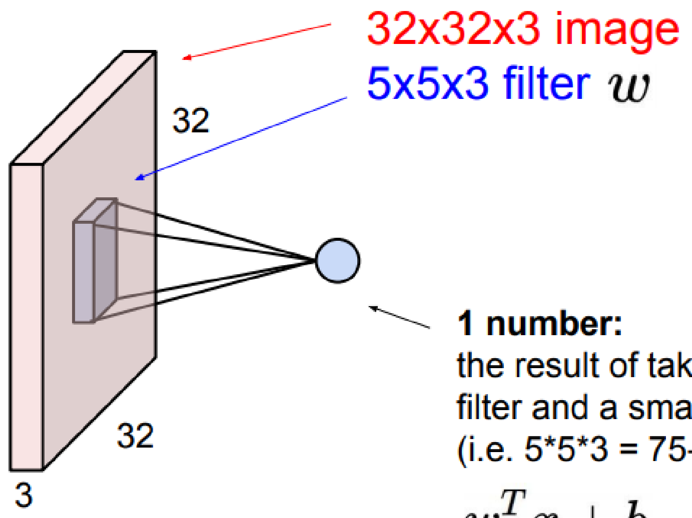
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**




. .

**=> 3x3 output!**

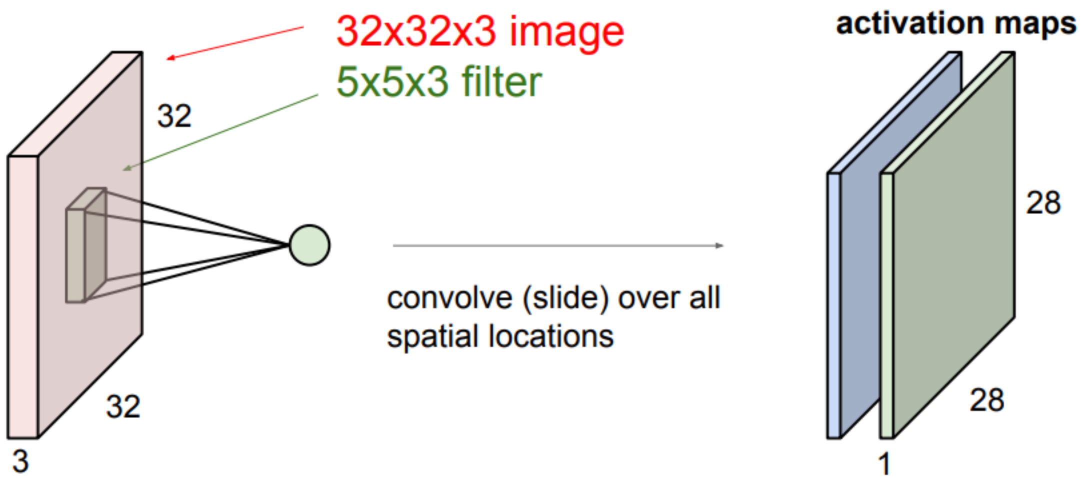
# Convolution Layer



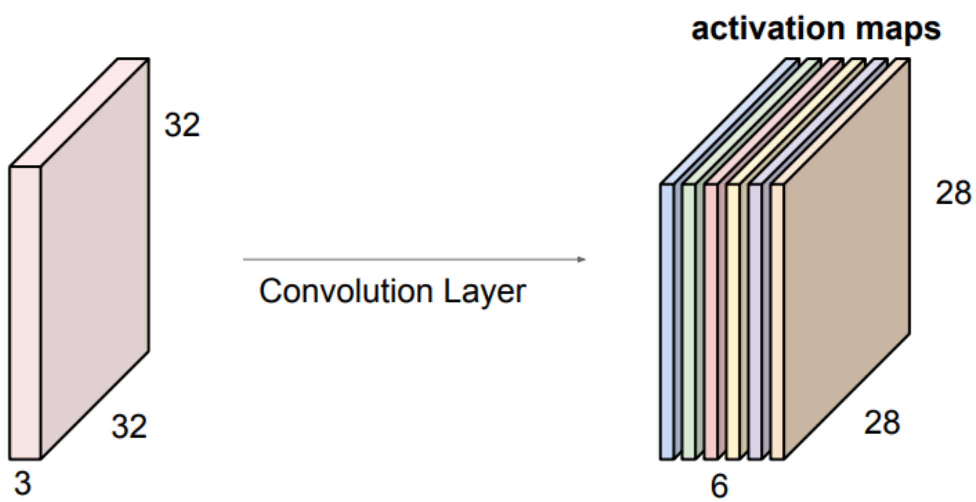

**1 number:**
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

# Convolution Layer




Second, green filter




6 filters

# Summary: Convolution Layer

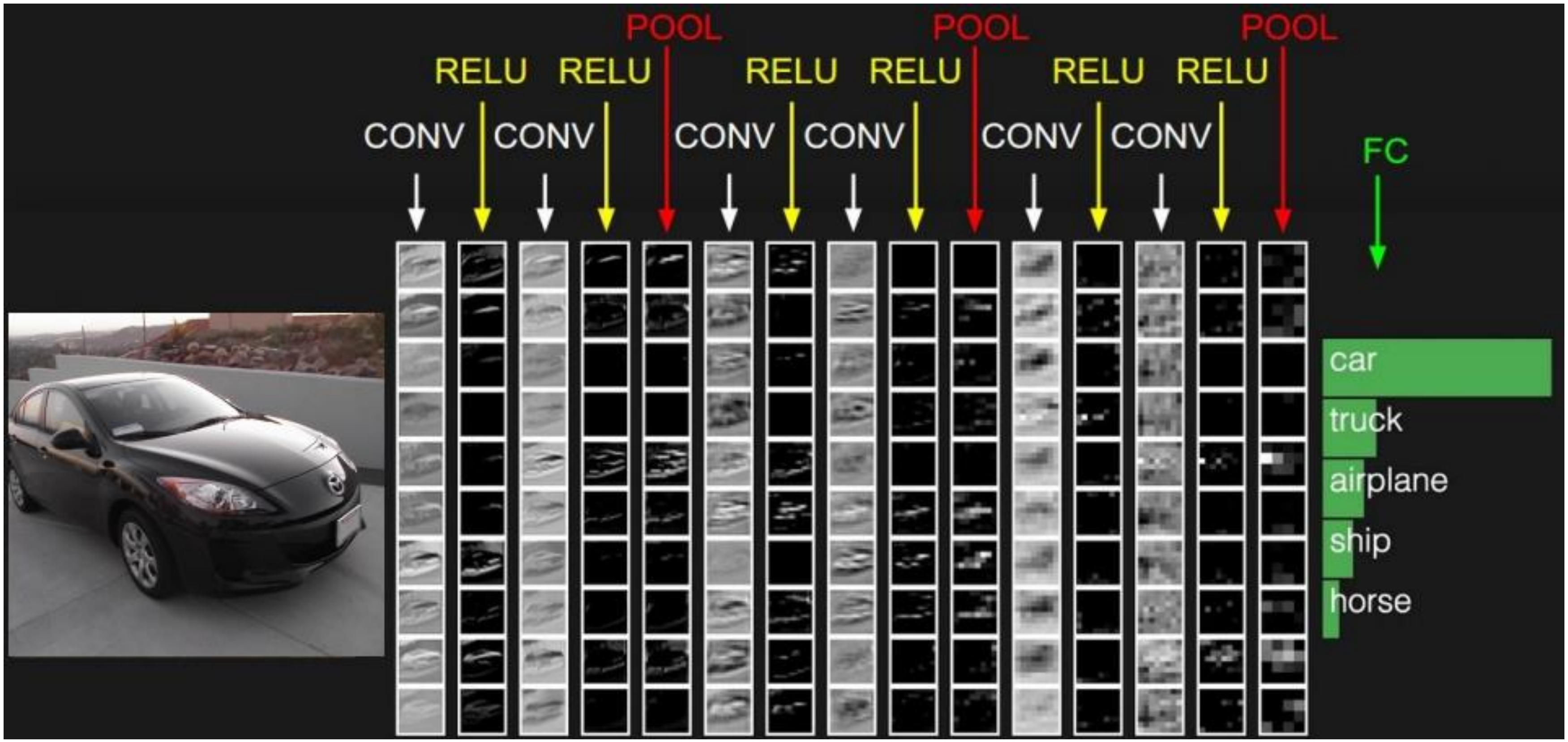
**Summary**. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
    - Number of filters $K$,
    - their spatial extent $F$,
    - the stride $S$,
    - the amount of zero padding $P$.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
    - $W_2 = (W_1 - F + 2P)/S + 1$
    - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
    - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and $K$ biases.
- In the output volume, the $d$-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the $d$-th filter over the input volume with a stride of $S$, and then offset by $d$-th bias.

# Convolution layer: Takeaways

- Convolution is a linear operation
  - Reduces parameter space of Feed-Forward Neural Network considerably
  - Capture locality of pixels in images
  - Smaller filters need less parameters
  - Multiple filters in each layer (computation can be done in parallel)
- Convolutions are followed by activation functions
  - Typically ReLU

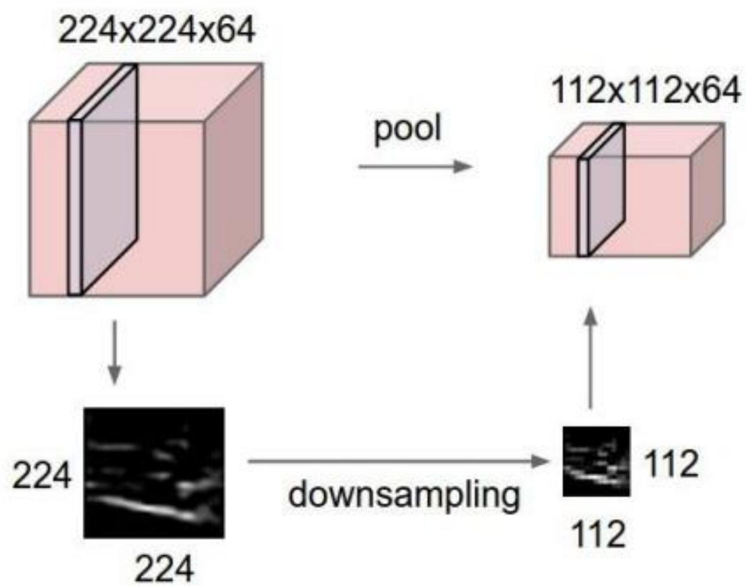
# Convolutional Nets



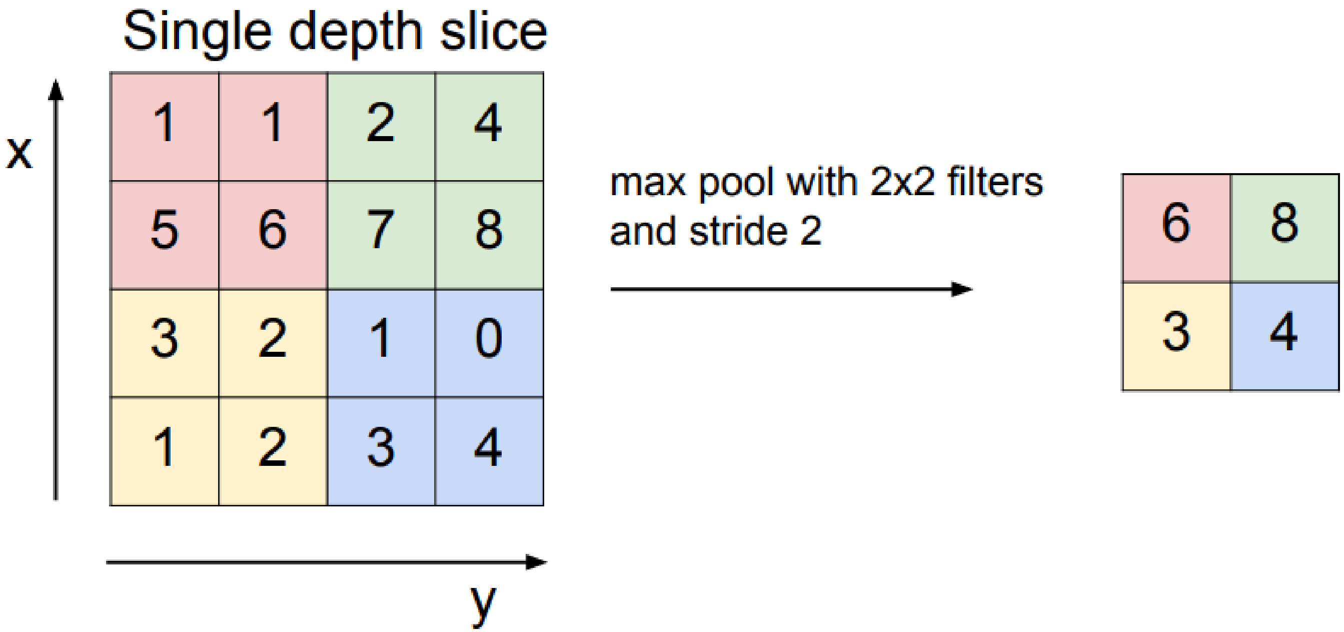
# Pooling layer

## Pooling layer

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



# Max Pooling

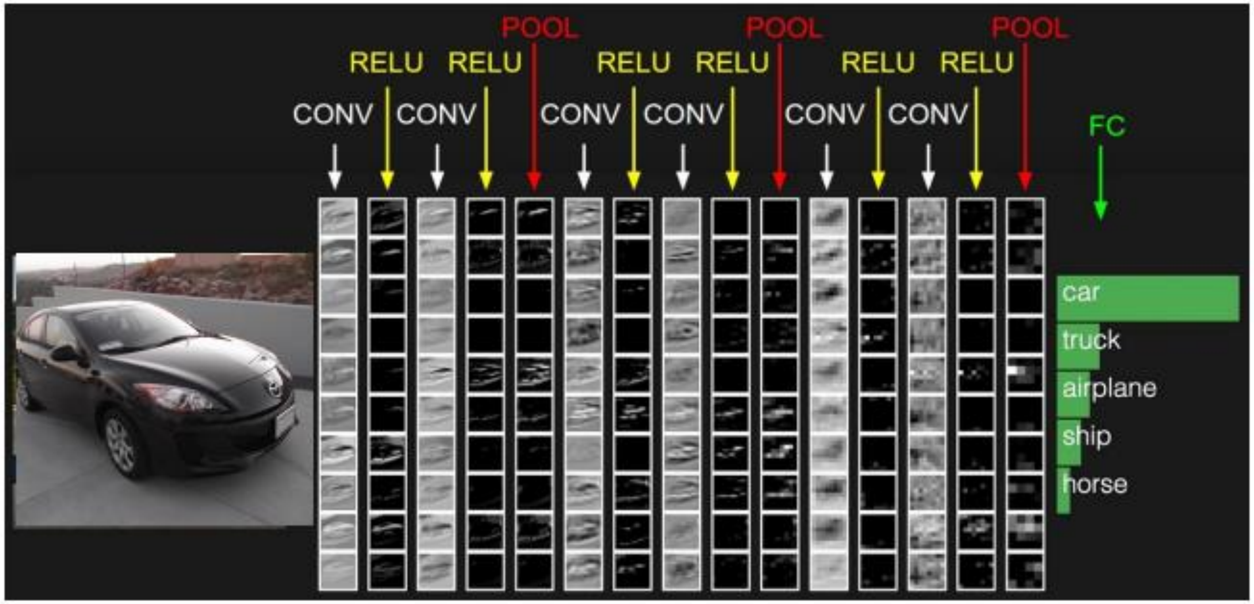



- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent $F$,
  - the stride $S$,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

# Convolutional Nets

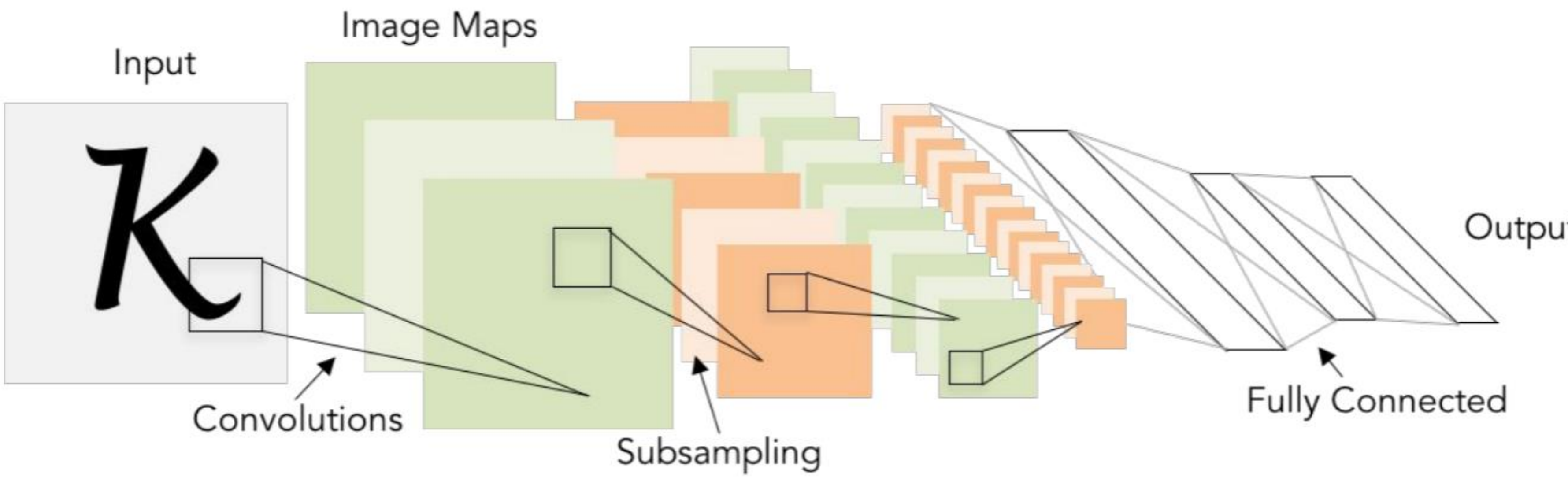
## Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



# LeNet 5

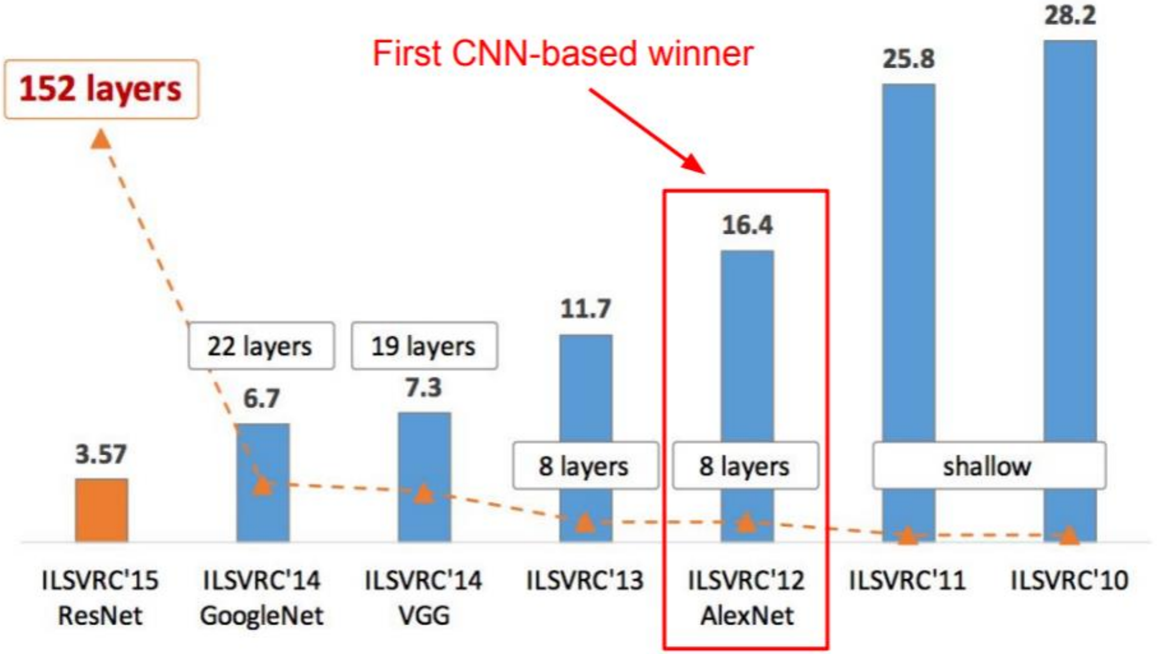
[LeCun et al., 1998]




Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

# History

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# VGGNet

## Case Study: VGGNet

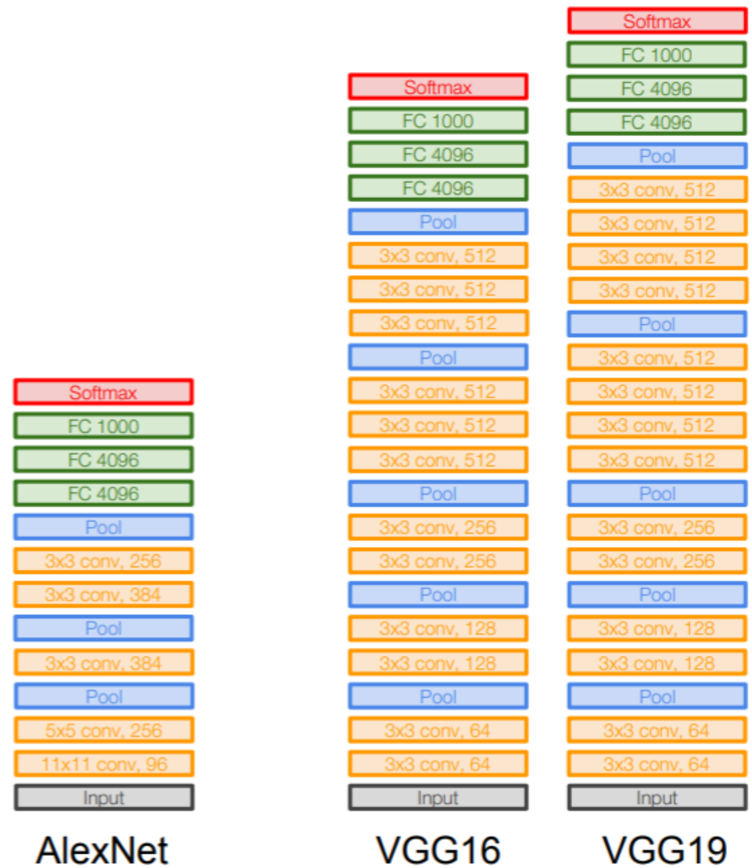
*[Simonyan and Zisserman, 2014]*

Small filters, Deeper networks

8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

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

11.7% top 5 error in ILSVRC'13
(ZFNet)
-> 7.3% top 5 error in ILSVRC'14




138 million parameters

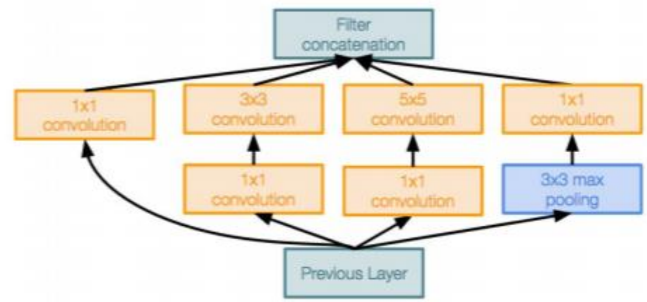
# GoogLeNet

## Case Study: GoogLeNet

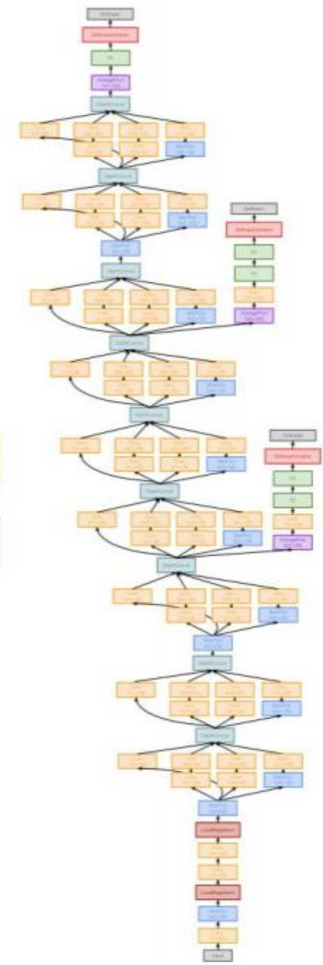
*[Szegedy et al., 2014]*

Deeper networks, with computational efficiency

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




Inception module



# Summary CNNs

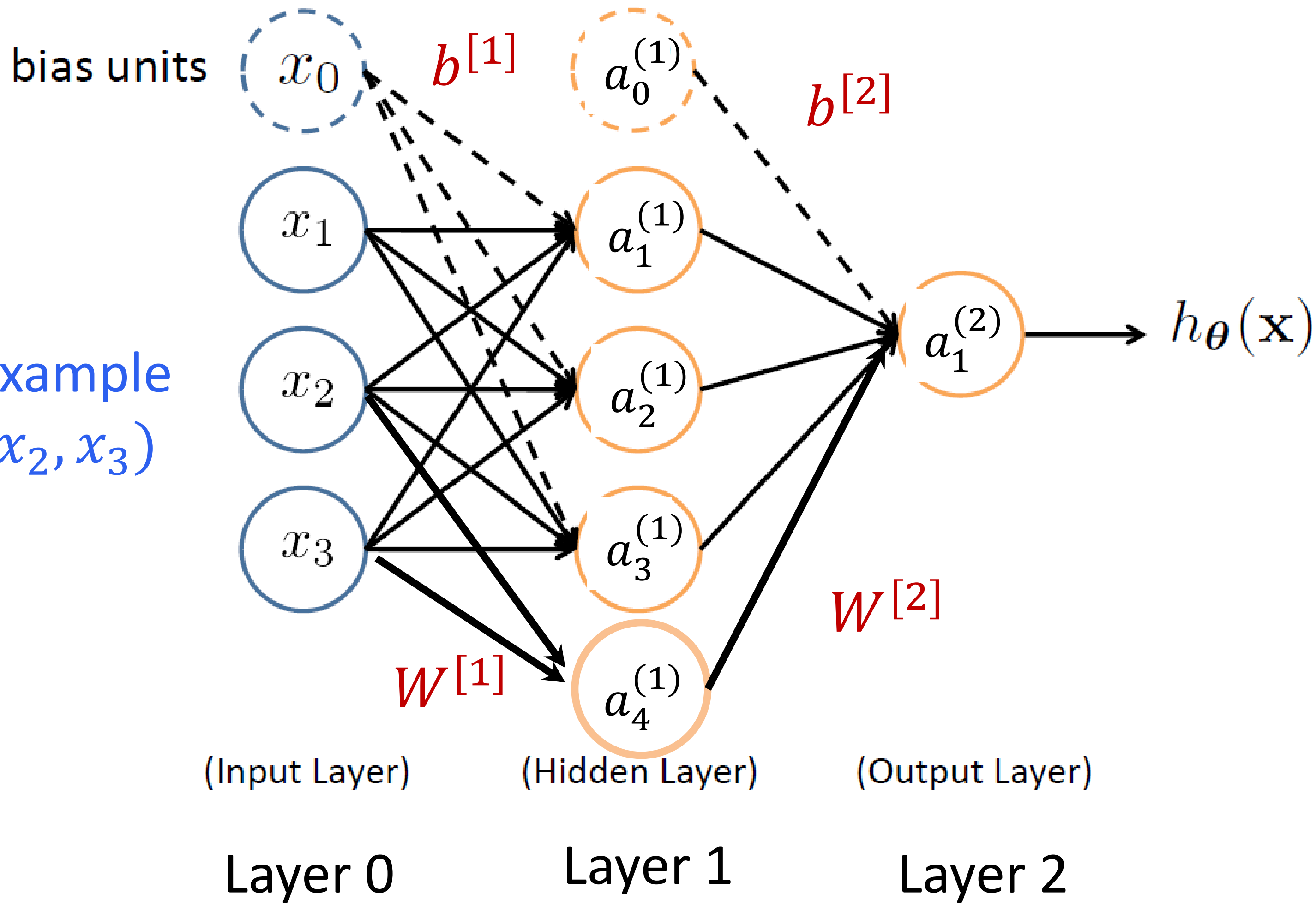
- Convolutional Nets are Feed-Forward Networks with at least one convolution layer and optionally max pooling layers
- Convolutions enable dimensionality reduction
- Much fewer parameters relative to Feed-Forward Neural Networks
  - Deeper networks with multiple small filters at each layer is a trend
- Fully connected layer at the end (fewer parameters)
- Learn hierarchical feature representations
  - Data with natural grid topology (images, maps)
- Reached human-level performance in ImageNet in 2014

# Outline

- Convolutional Neural Networks
  - Recap: convolution layer
  - Max pooling
  - Architectures
- Training with backpropagation
  - Initialization
  - Derivation of gradients
  - Example

# Feed-Forward Neural Network

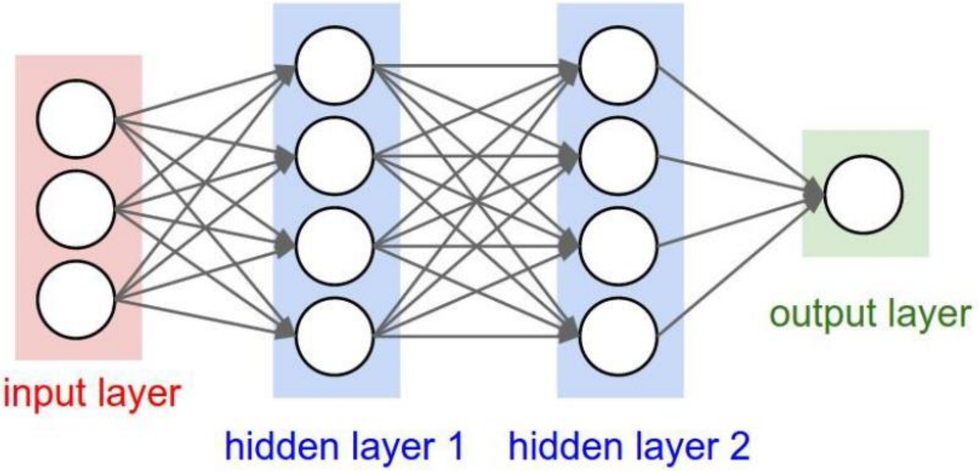
Training example
$x = (x_1, x_2, x_3)$




No cycles

# Forward Propagation

- The input neurons first receive the data features of the object. After processing the data, they send their output to the first hidden layer.
- The hidden layer processes this output and sends the results to the next hidden layer.
- This continues until the data reaches the final output layer, where the output value determines the object's classification.
- This entire process is known as Forward Propagation, or Forward prop.



# Perceptron Learning

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(y - h(\mathbf{x}))\mathbf{x}$$

Equivalent to the intuitive rules:

- If output is correct, don't change the weights
- If output is low ($h(\mathbf{x}) = 0$, $y = 1$), increment weights for all the inputs which are 1
- If output is high ($h(\mathbf{x}) = 1$, $y = 0$), decrement weights for all inputs which are 1

**Perceptron Convergence Theorem**:

- If there is a set of weights that is consistent with the training data (i.e., the data is linearly separable), the perceptron learning algorithm will converge [Minicksy & Papert, 1969]

# Batch Perceptron

Given training data $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$

Let $\boldsymbol{\theta} \leftarrow [0, 0, \ldots, 0]$

Repeat:

  Let $\boldsymbol{\Delta} \leftarrow [0, 0, \ldots, 0]$

  for $i = 1 \ldots n$, do

    if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} \leq 0$ // prediction for $\text{i}^{th}$ instance is incorrect

      $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta} + y^{(i)}\boldsymbol{x}^{(i)}$

  $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta}/n$ // compute average update

  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{\Delta}$

Until $\|\boldsymbol{\Delta}\|_2 < \epsilon$

- Simplest case: $\alpha = 1$ and don't normalize, yields the fixed increment perceptron
- Each increment of outer loop is called an **epoch**

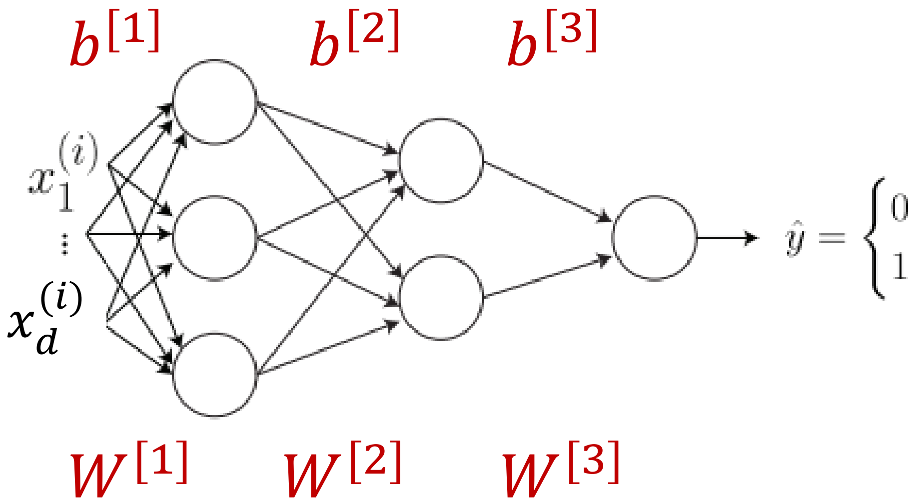
# Learning in NN: Backpropagation

- Similar to the perceptron learning algorithm, we cycle through our examples
  - If the output of the network is correct, no changes are made
  - If there is an error, weights are adjusted to reduce the error

- The trick is to assess the blame for the error and divide it among the contributing weights

# Example

Training data
Dimension d




$$z^{[1]} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1]} = g(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = g(z^{[2]})$$
$$z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$$
$$\hat{y}^{(i)} = a^{[3]} = g(z^{[3]})$$

# Parameter Initialization

- How about we set all W and b to 0?
- First layer
  - $z^{[1]} = \mathrm{W}^{[1]}x + \mathrm{b}^{[1]}$= (0,…0)
  - $a^{[1]} = \mathrm{g}\left(\mathrm{z}^{[1]}\right) = \left(\frac{1}{2}, \ldots, \frac{1}{2}\right)$
- Second layer
  - $z^{[2]} = \mathrm{W}^{[2]}x + \mathrm{b}^{[2]}$= (0,…0)
  - $a^{[2]} = \mathrm{g}\left(\mathrm{z}^{[2]}\right) = \left(\frac{1}{2}, \ldots, \frac{1}{2}\right)$
- Third layer
  - $z^{[3]} = \mathrm{W}^{[3]}x + \mathrm{b}^{[3]}$= (0,…0)
  - $a^{[3]} = \mathrm{g}\left(\mathrm{z}^{[3]}\right) = \left(\frac{1}{2}, \ldots, \frac{1}{2}\right)$
- Initialize with random values instead!

# Training

- Training data $x^{(1)}, \mathrm{y}^{(1)}, \ldots x^{(N)}, \mathrm{y}^{(\mathrm{N})}$
- One training example $x^{(i)} = \left(x_1^{(i)}, \ldots x_d^{(i)}\right)$, label $y$
- One forward pass through the network
  - Compute prediction $\hat{y}$
- Loss function for one example
  - $L(\hat{y}, y) = -[(1-y)\log(1-\hat{y}) + y\log\hat{y}]$

Cross-entropy loss

- Loss function for training data
  - $J(W, b) = \frac{1}{N}\sum_i L\left(\hat{y}^{(i)}, y^{(i)}\right) + \lambda R(W, b)$

# Reminder: Logistic Regression

$$J(\boldsymbol{\theta}) = -\sum_{i=1}^{N}\left[y^{(i)} \log h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) + \left(1 - y^{(i)}\right) \log \left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right)\right]$$

- Cost of a single instance:

$$\text{cost}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}), y\right) = \begin{cases} -\log(h_{\boldsymbol{\theta}}(\boldsymbol{x})) & \text{if } y = 1 \\ -\log(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x})) & \text{if } y = 0 \end{cases}$$

- Can re-write objective function as

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} \underbrace{\text{cost}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)}\right)}_{\text{Cross-entropy loss}}$$

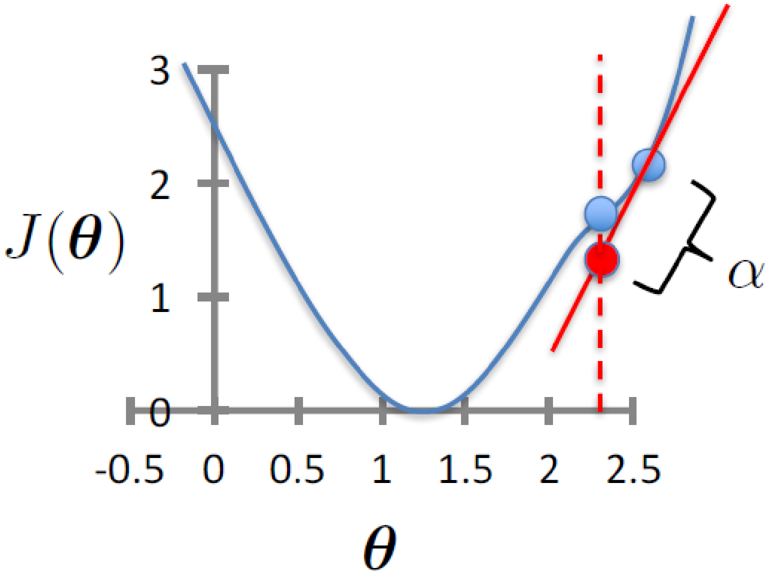
# Gradient Descent

- Initialize $\boldsymbol{\theta}$ $\qquad \boldsymbol{\theta} = (W, b)$
- Repeat until convergence

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta})$$

simultaneous update
for j = 0 ... d

learning rate (small)
e.g., α = 0.05

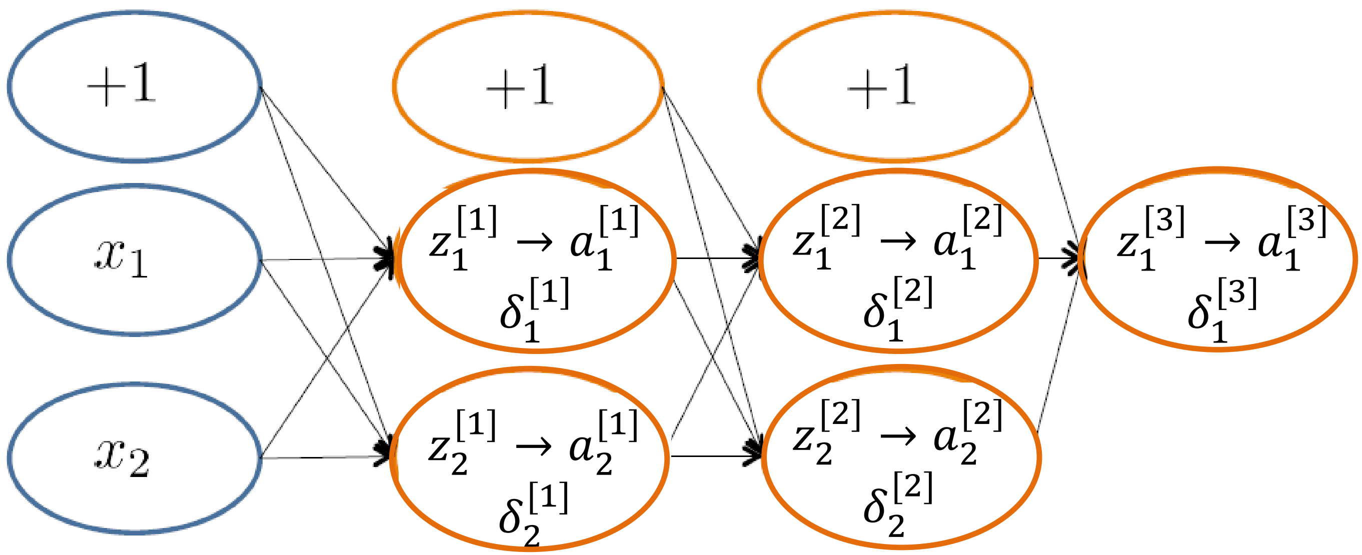



- Converges for convex objective
- Could get stuck in local minimum for non-convex objectives

# GD for Neural Networks

- Initialization
  - For all layers $\ell$
    - Set $W^{[\ell]}, b^{[\ell]}$ at random
- Backpropagation
  - Fix learning rate $\alpha$
  - For all layers $\ell$ (starting backwards)
    - $W^{[\ell]} = W^{[\ell]} - \alpha \sum_{i=1}^{N} \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial W^{[\ell]}}$
    - $b^{[\ell]} = b^{[\ell]} - \alpha \sum_{i=1}^{N} \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial b^{[\ell]}}$

# Backpropagation Intuition

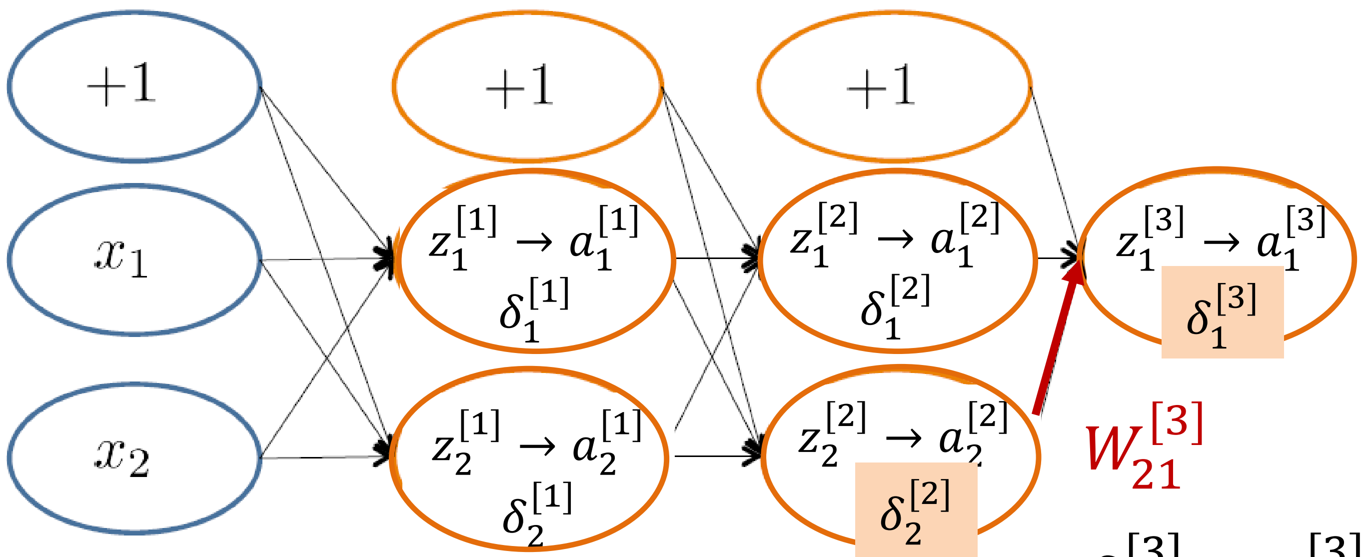



$\delta_j^{(l)}$ = "error" of node $j$ in layer $l$

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}\ (x^{(i)})$

$$cost(x^{(i)}) = y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

# Backpropagation Intuition




$$\delta_1^{[3]} \approx a_1^{[3]} - y$$

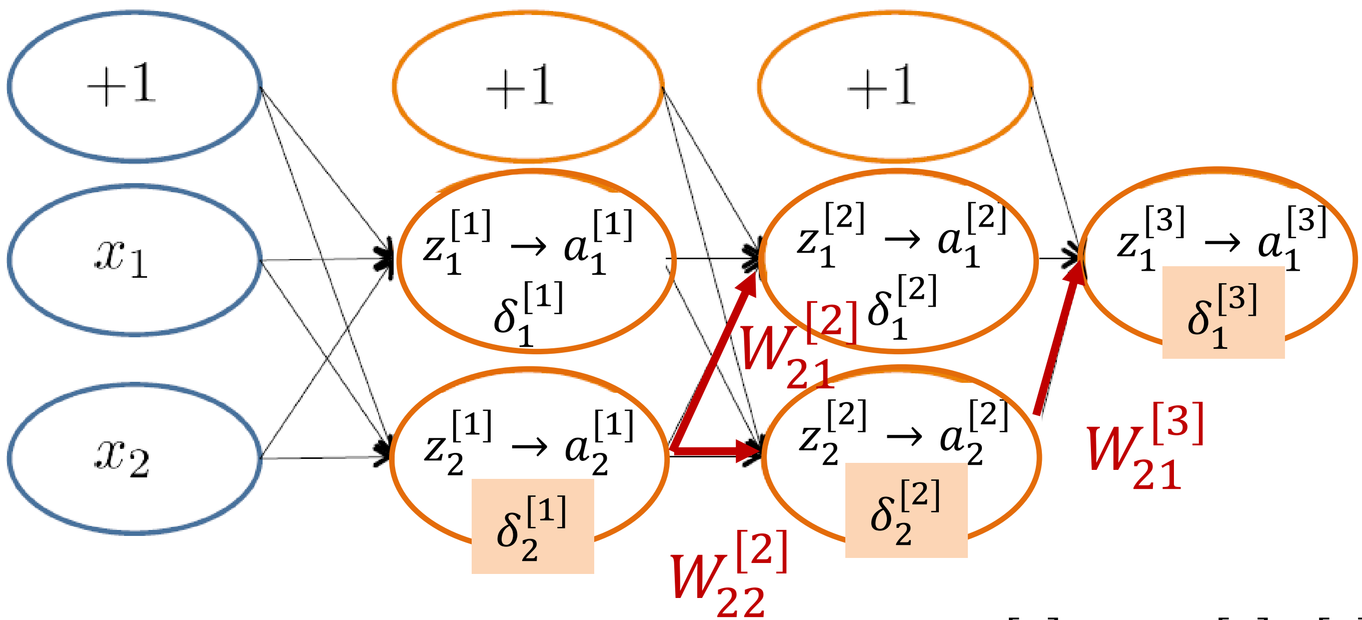
$$\delta_2^{[2]} \approx \delta_1^{[3]} W_{21}^{[3]}$$

$\delta_j^{(l)}$ = "error" of node $j$ in layer $l$

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost} \, (x^{(i)})$

$$cost\left(x^{(i)}\right) = y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)$$

# Backpropagation Intuition




$$\delta_2^{[1]} \approx W_{21}^{[2]}\delta_1^{[2]} + W_{22}^{[2]}\delta_2^{[2]}$$

$\delta_j^{(l)}$ = "error" of node $j$ in layer $l$

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}\ (x^{(i)})$

$$cost(x^{(i)}) = y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))$$

# Materials

- Stanford tutorial on training Multi-Layer Neural Networks
  - http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/
- Notes on backpropagation by Andrew Ng
  - http://cs229.stanford.edu/notes/cs229-notes-backprop.pdf
- Deep learning notes by Andrew Ng
  - http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf

# Review

- To train neural networks, need to decide first on architecture
  - Number of layers, number of hidden units, connections between neurons, activation functions
- Randomly initialize parameters
- For each training example, use forward propagation to compute prediction
- Use backpropagation to propagate the error from last layer back into the network

# Acknowledgements


- Slides made using resources from:
  - Yann LeCun
  - Andrew Ng
  - Eric Eaton
  - David Sontag
  - Andrew Moore
- Thanks!