SMU
SINGAPORE MANAGEMENT UNIVERSITY

Neural Networks & Deep Learning

Instructor: Steven C.H. Hoi

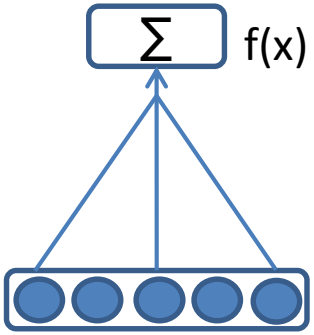
School of Information Systems

Singapore Management University

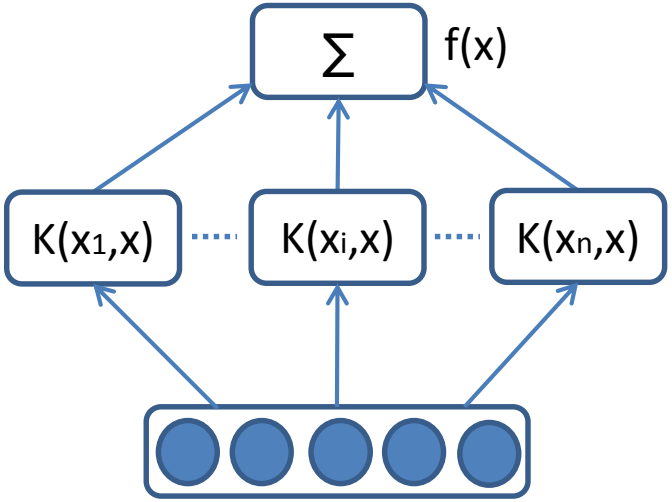
Email: chhoi@smu.edu.sg

What is Deep Learning?

Shallow Learning Architecture

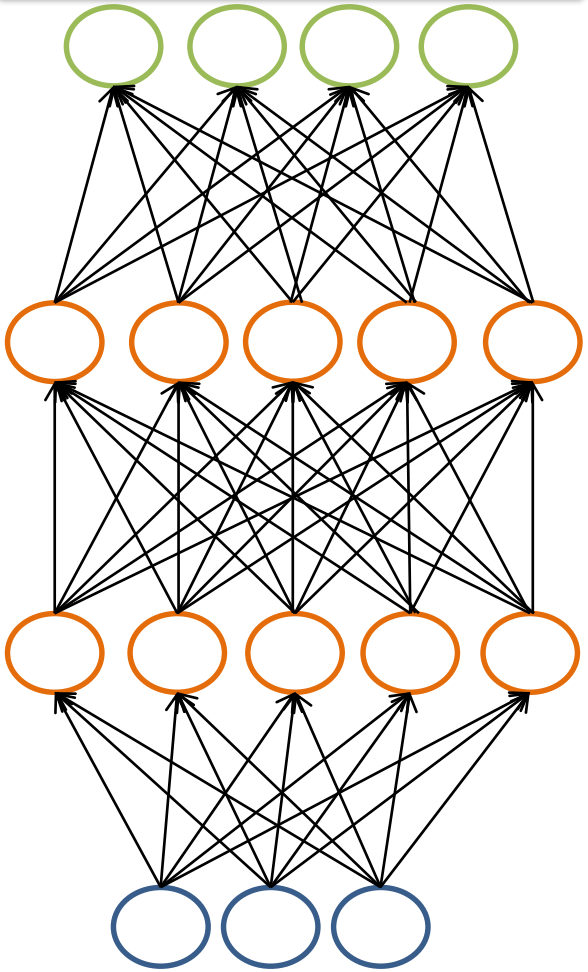


Linear models (e.g., LR)



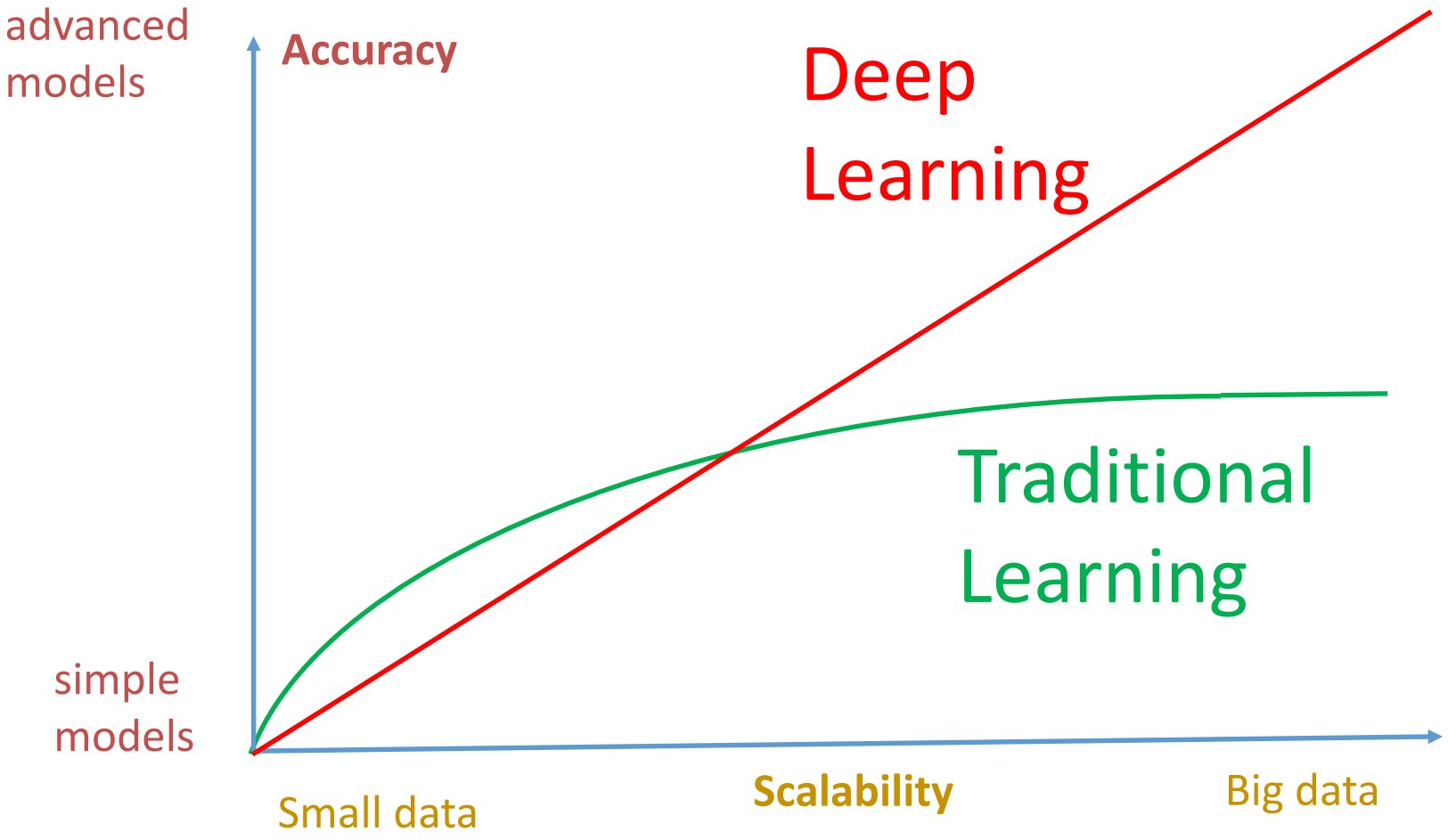
Kernel methods (e.g., SVM)

Deep Learning Architecture



Deep Learning (e.g., NN)

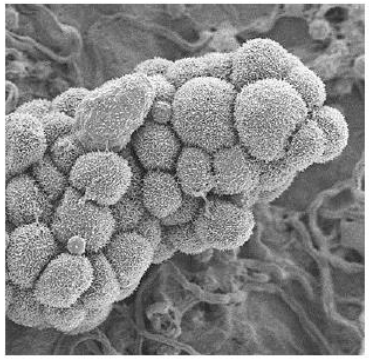
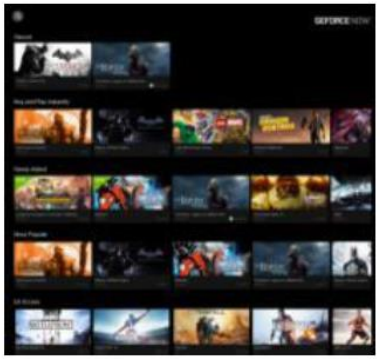
Why Deep Learning?



Deep Learning Applications

MEDICINE & BIOLOGY
Cancer Cell Detection
Diabetic Grading
Drug Discovery

SECURITY & DEFENSE
Face Detection
Video Surveillance
Satellite Imagery

INTERNET & CLOUD
Image Classification
Speech Recognition
Language Translation
Language Processing
Sentiment Analysis
Recommendation

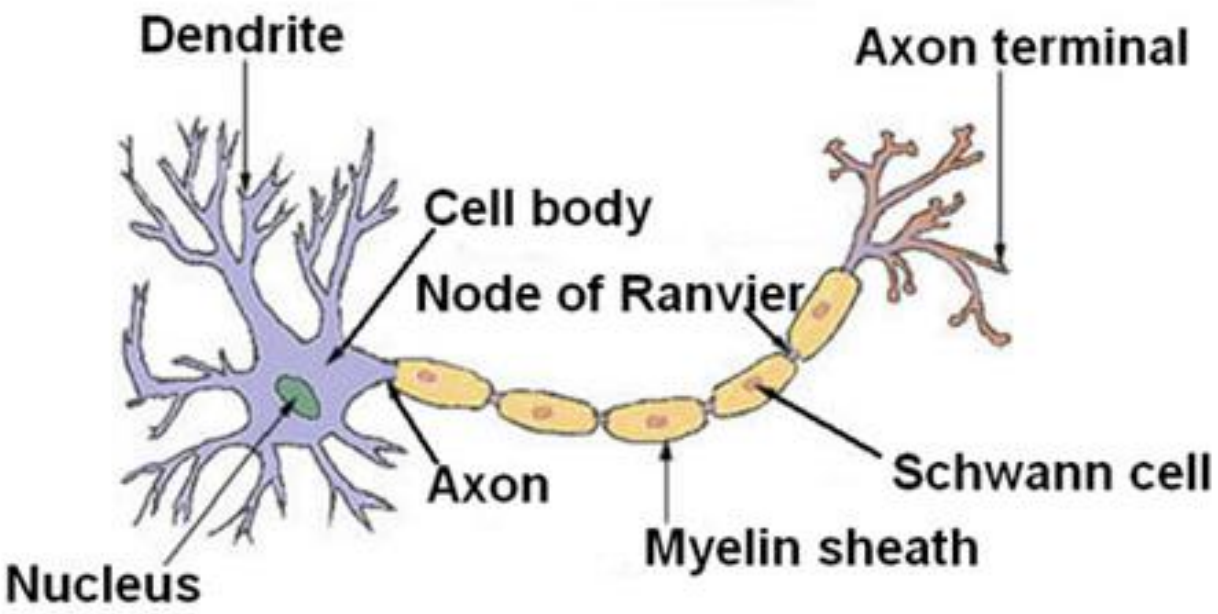
MEDIA & ENTERTAINMENT
Video Captioning
Video Search
Real Time Translation

AUTONOMOUS MACHINES
Pedestrian Detection
Lane Tracking
Recognize Traffic Sign

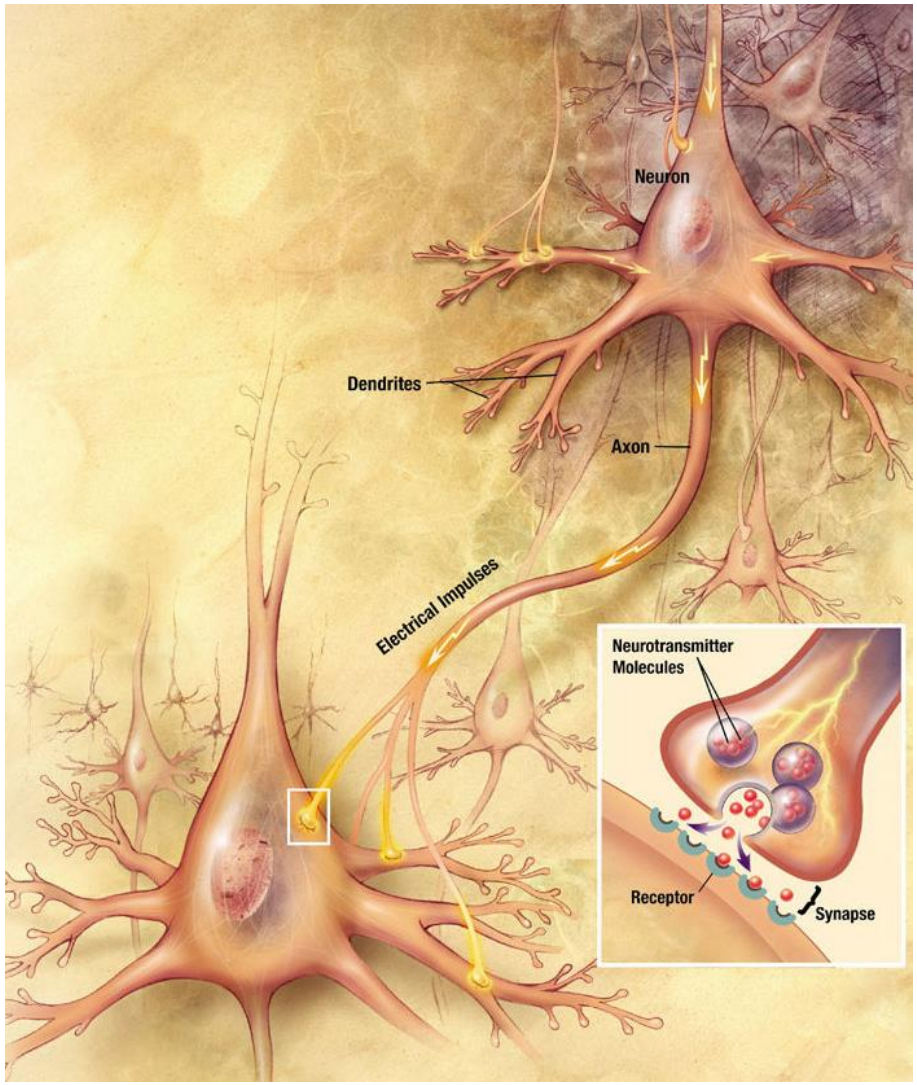
Outline

- Background
- Perceptron
- Stochastic Gradient Descent
- Multi-Layer Neural Networks
- Back-Propagation (BP)
- Convolutional Neural Networks

Neuron in the brain



Neuron in the brain



[Credit: US National Institutes of Health, National Institute on Aging]

Connectionist Models

Consider humans:

- Neuron switching time $\sim$ .001 second
- Number of neurons $\sim 10^{10}$
- Connections per neuron $\sim 10^{4-5}$
- Scene recognition time $\sim$ .1 second
- 100 inference steps doesn't seem like enough

$\Rightarrow$ Much parallel computation

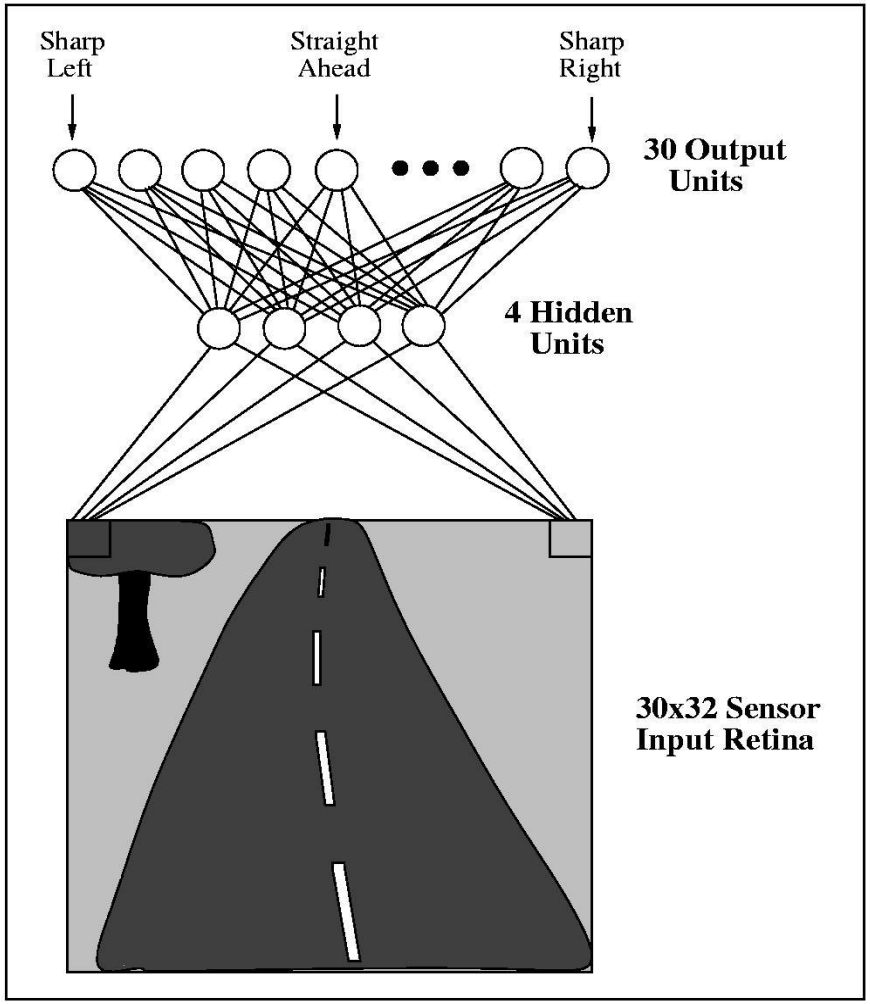
Properties of neural nets:

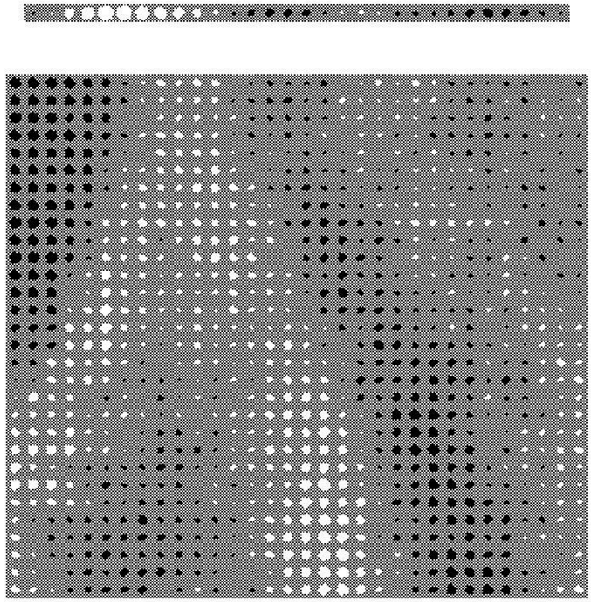
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

Example: Autonomous Vehicle

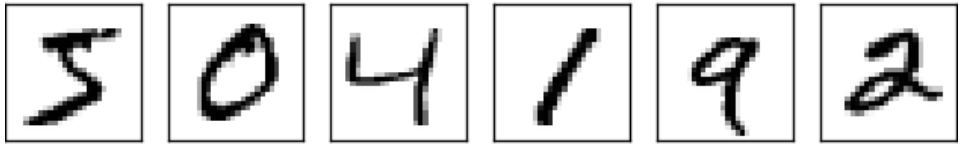
ALVINN: an autonomous land vehicle in a neural network

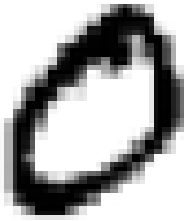
(DA Pomerleau - 1989)


Sharp
Left
Straight
Ahead
Sharp
Right
30 Output
Units
4 Hidden
Units
30x32 Sensor
Input Retina

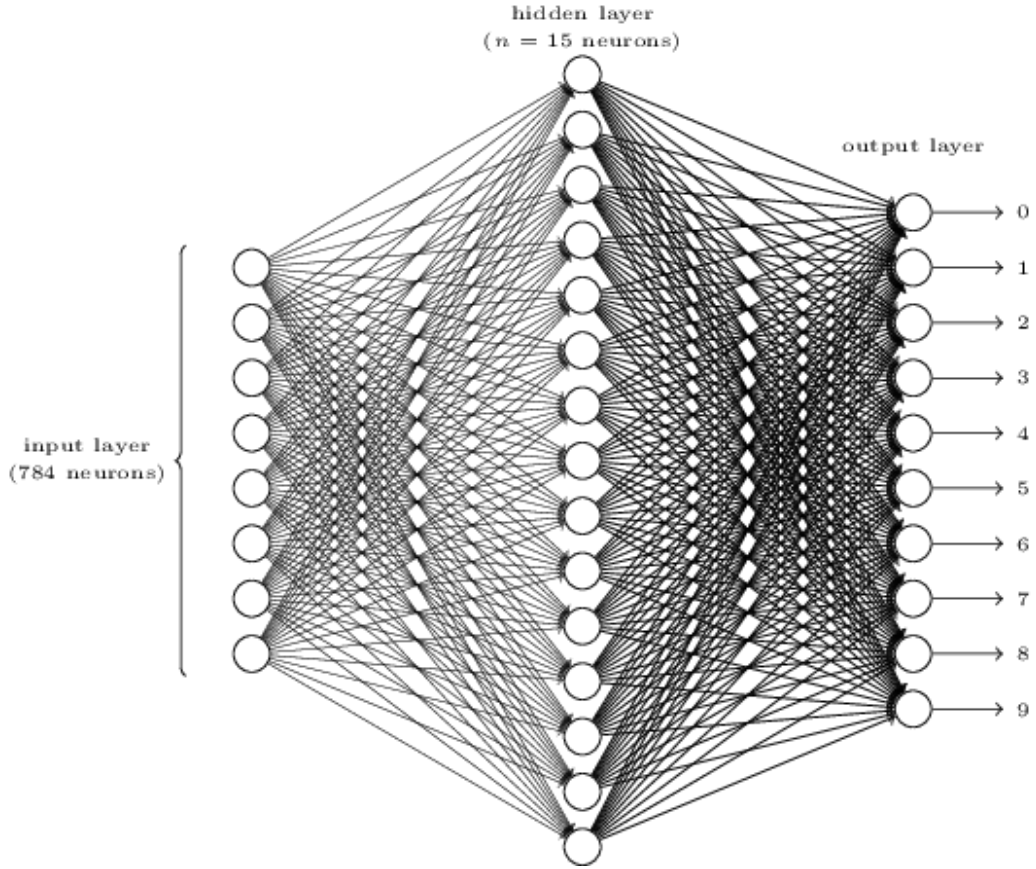


Example: Handwritten digits





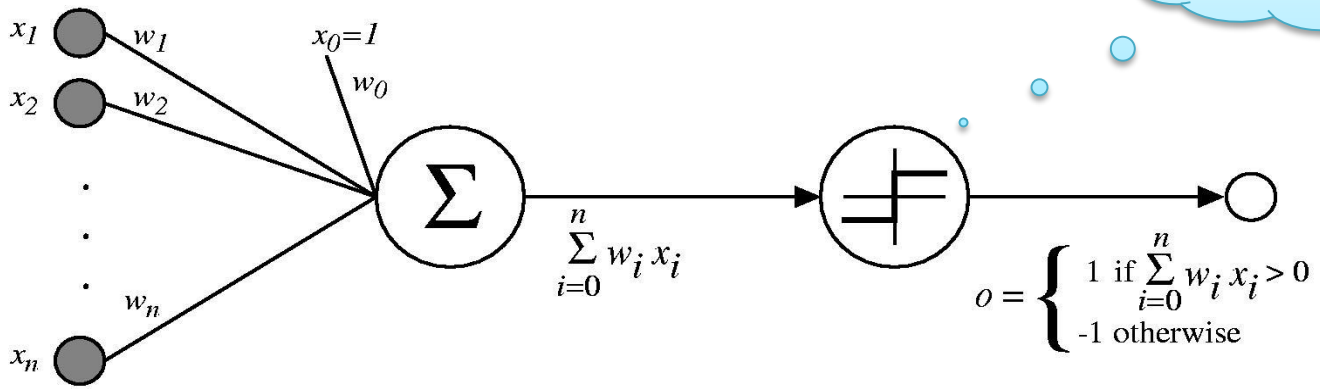
28x28 pixels



Neural Networks: Overview

- Perceptron: A Single Layer NN
- Multi-Layer Perceptron (MLP)
- Deep Multi-layer Neural Networks

Perceptron

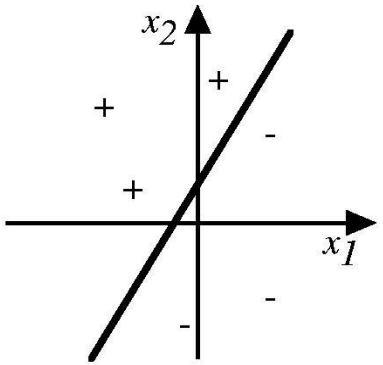


$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

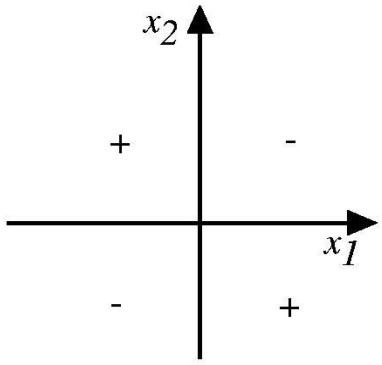
Sometimes we'll use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Decision Surface of a Perceptron



(a)



(b)

Represents some useful functions

- What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?

Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., 0.1) called *learning rate*

Perceptron Training Rule

Can prove it will converge if

- Training data is linearly separable
- η sufficiently small

Gradient Descent

To understand, consider simpler *linear unit*, where

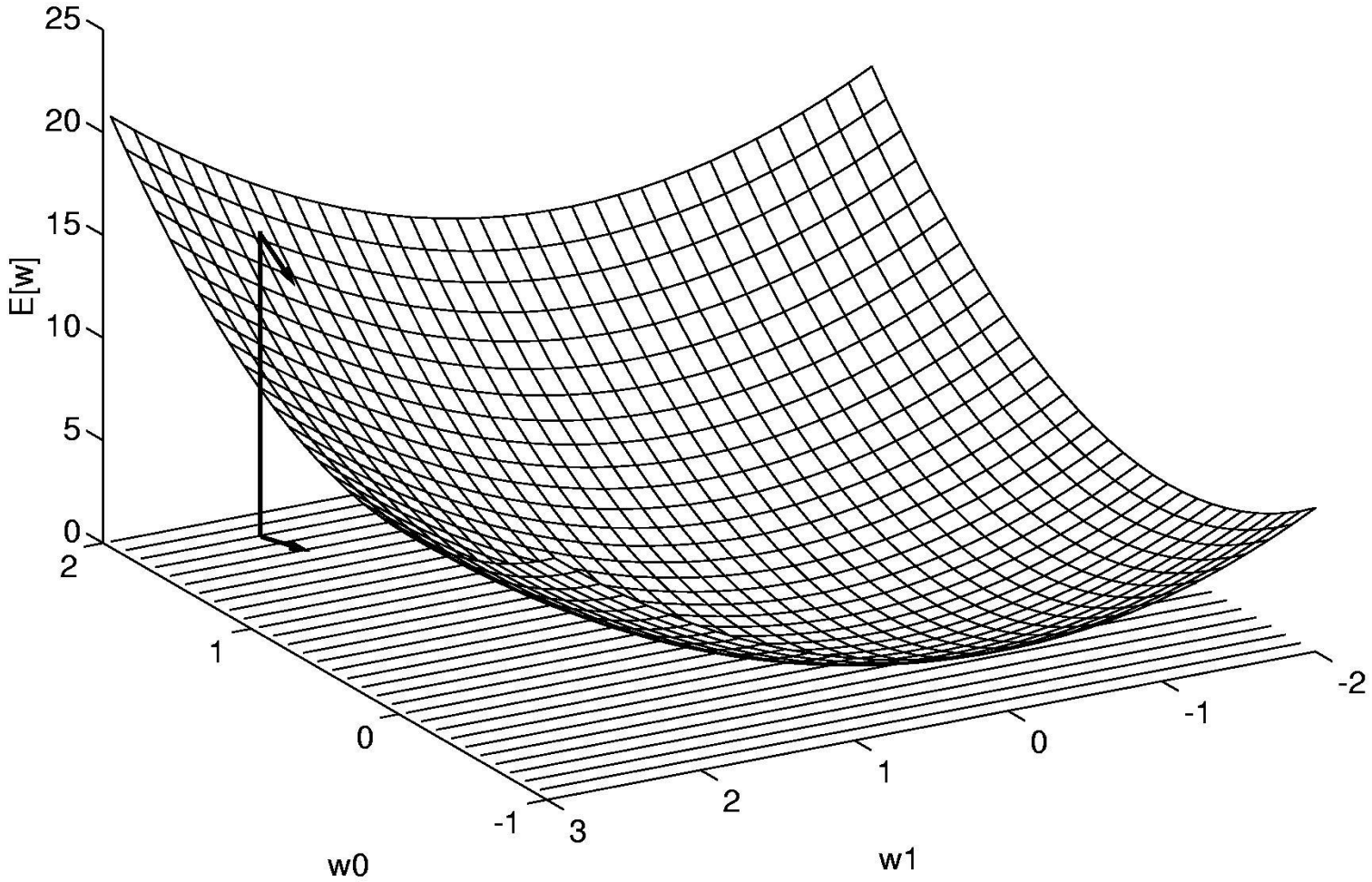
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's learn w_i's that minimize the squared error

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

Gradient Descent



Gradient:

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n} \right]$$

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

I.e.:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}
$$

Gradient Descent

GRADIENT-DESCENT($training\_examples, \eta$)

Initialize each w_i to some small random value

Until the termination condition is met, Do

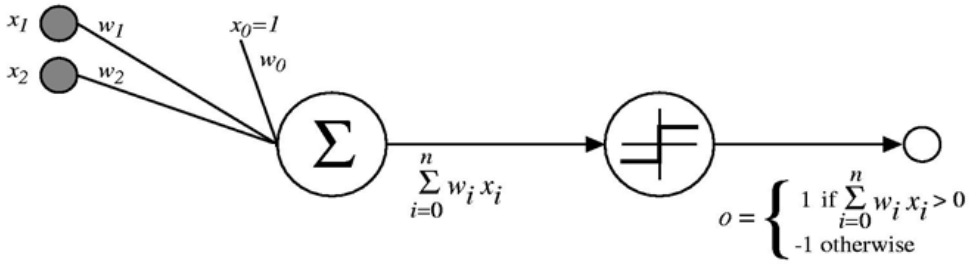
- Initialize each Δw_i to zero.
- For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do
  - Input instance $\vec{x}$ to unit and compute output o
  - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

- For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i$$

Perceptron: Example



$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Initialization: w0=w1=w2=0

x0	x1	x2	Σ sum	o	t
1	1	1	0	-1	+1
1	-1	2	4	+1	-1

INIT: w(w0, w1, w2) = (0, 0, 0) set learning rate η = 1

$$w_i \leftarrow w_i + \Delta w_i$$

Iteration 1: Receiving example (x1,x2)=(1, 1), class label t=+1

$$\Delta w_i = \eta(t - o)x_i$$

$\eta(t - o)$ = 1x(+1 – (-1)) = 2

$w \leftarrow w + \Delta w$ = (0,0,0) + 2*(1,1,1) = (2,2,2)

Iteration 2: Receiving example (x1,x2)=(-1, 2), class label = -1

$\eta(t - o)$ = 1x((-1) – (+1)) = -2

$w \leftarrow w + \Delta w$ = (2,2,2) + -2*(1,-1,2) = (0,4,-2)

Summary

Perceptron training rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rule uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Batch vs. Incremental Gradient Descent

Batch Mode Gradient Descent:

Do until convergence

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental Mode Gradient Descent:

Do until convergence

For each training example d in D

1. Compute the gradient $\nabla E_d[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

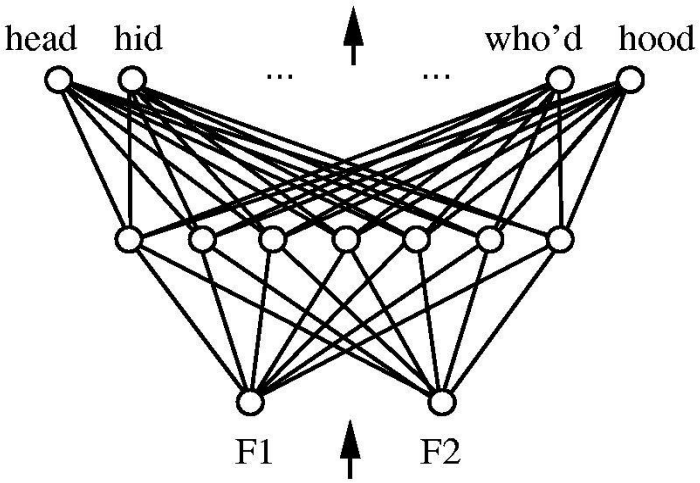
Commonly known as **"Stochastic Gradient Descent (SGD)"**

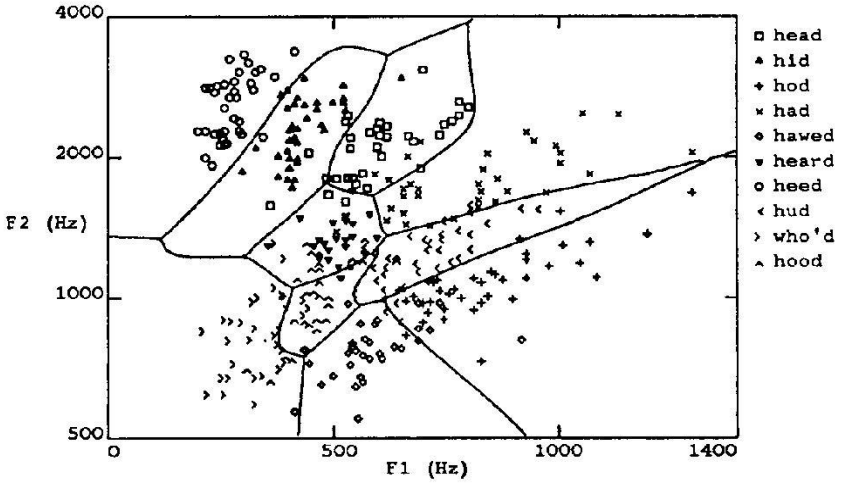
Multi-Layer Neural Networks

- Multi-Layer Neural Networks
  - Neural networks with multiple hidden layers

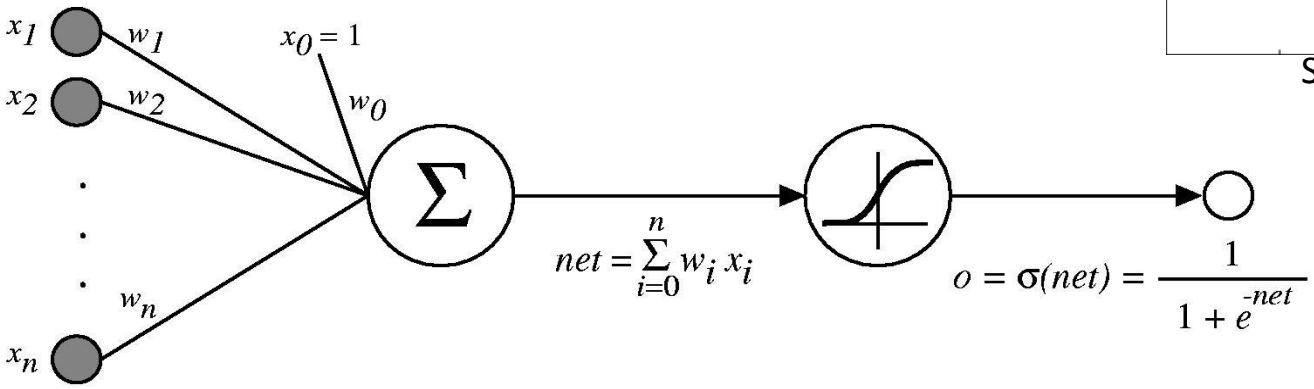
- Also known as "Multi-Layer Perceptron" (MLP), but this name is somewhat abuse because
  - It is not a Perceptron, and even
  - The activation unit used is often diff from Perceptron
  - Common activation units
    - Sigmoid Units (Logistic function)

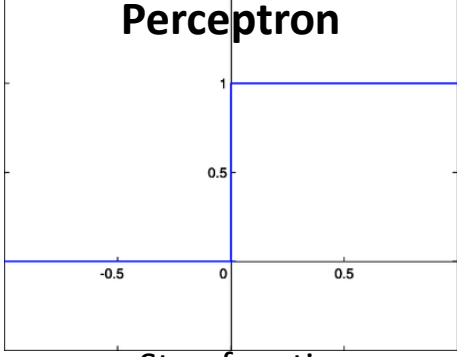
Multilayer Networks of Sigmoid Units





Sigmoid Unit



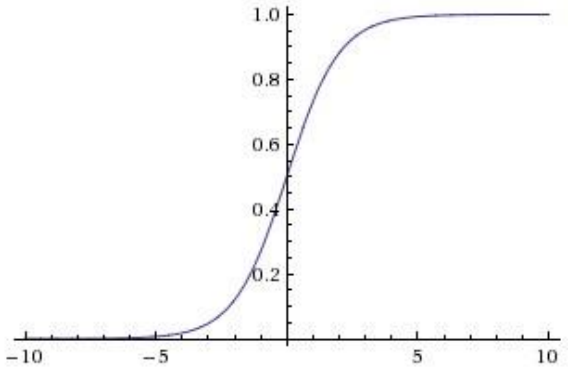


Step function

$\sigma(x)$ is the sigmoid function (the logistic function in logistic regression)

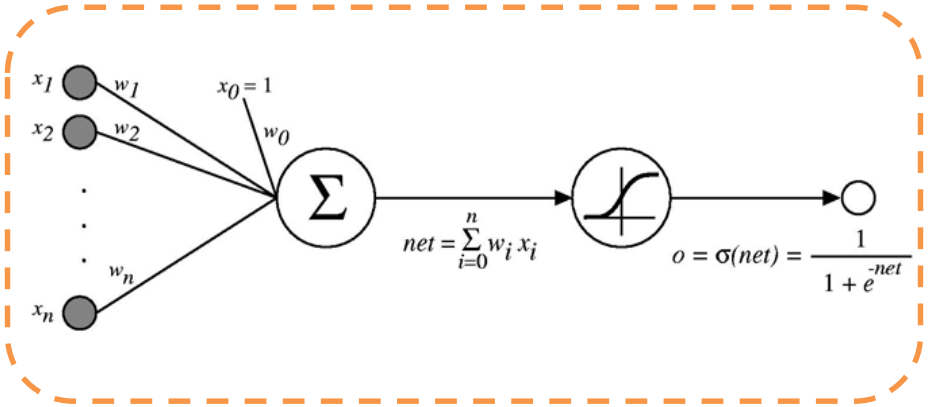
$$\frac{1}{1+e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$

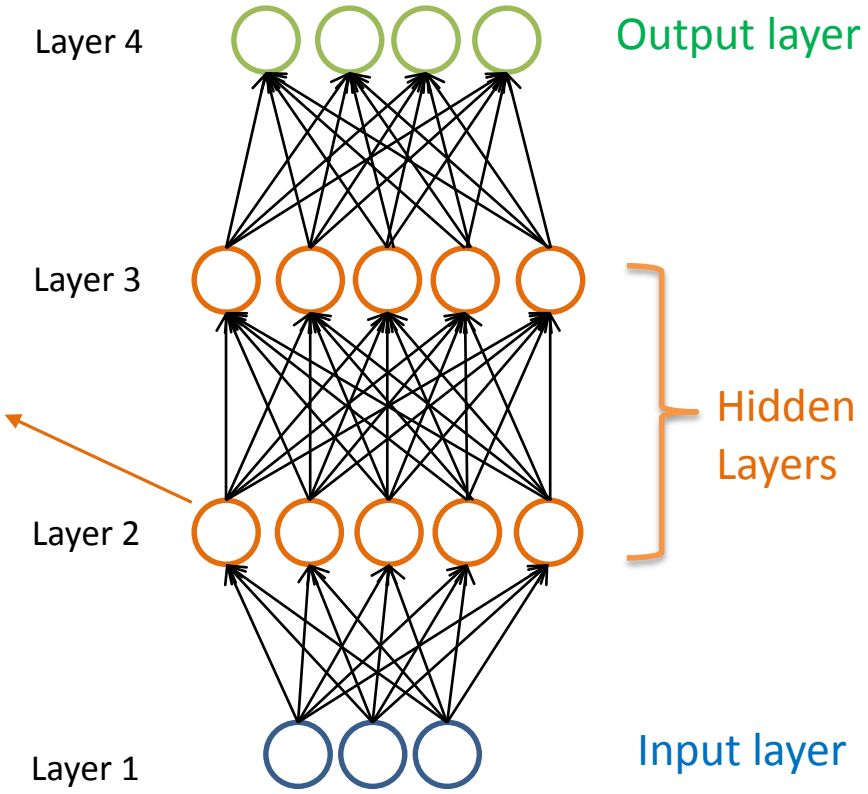


Multi-layer Neural Networks

- Multiple Layers of Sigmoid Units



Each node ("neuron") is the sigmoid unit



Forward-Propagation

$\mathbf{x}^{(l)}$ – Input vector at level l

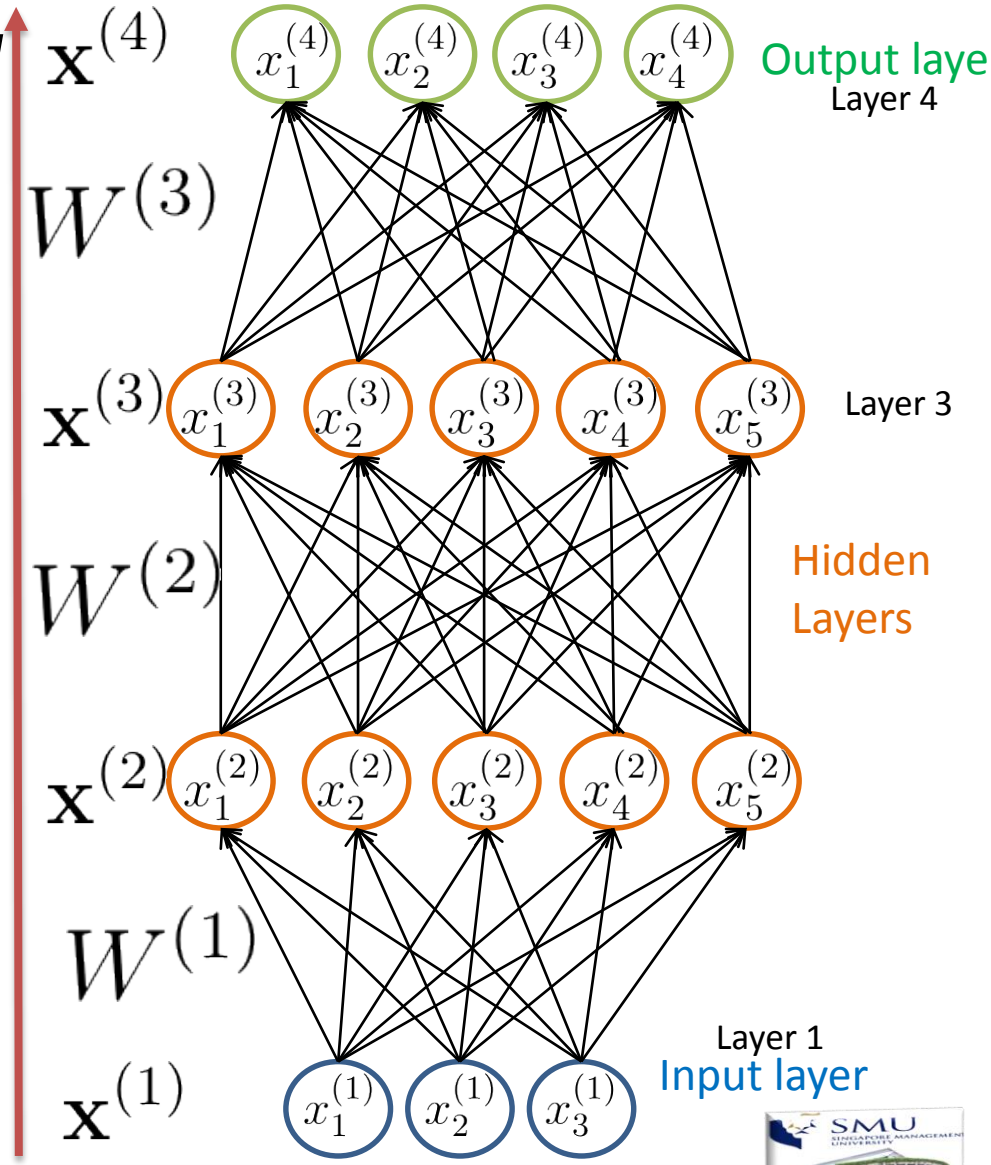
$W^{(l)}$ – Weight matrix of level l

$$\mathbf{x}^{(4)} = \sigma(W^{(3)}\mathbf{x}^{(3)})$$
$4 \times 1 \quad 4 \times 5 \quad 5 \times 1$

$$\mathbf{x}^{(3)} = \sigma(W^{(2)}\mathbf{x}^{(2)})$$
$5 \times 1 \quad 5 \times 5 \quad 5 \times 1$

$$\mathbf{x}^{(2)} = \sigma(W^{(1)}\mathbf{x}^{(1)})$$
$5 \times 1 \quad 5 \times 3 \quad 3 \times 1$

$\mathbf{x}^{(1)}$



Multilayer Neural Networks

- How to train multilayer NN, i.e., find $W^{(l)}$

We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} + \Delta W^{(l)}$$

$$\Delta W^{(l)} = \eta \boldsymbol{\delta}^{(l+1)} (\mathbf{x}^{(l)})^T$$

learning rate

error back-propagated

Back-Propagation

- Error at level l $\boldsymbol{\delta}^{(l)}$
- Error at last level (L=4)

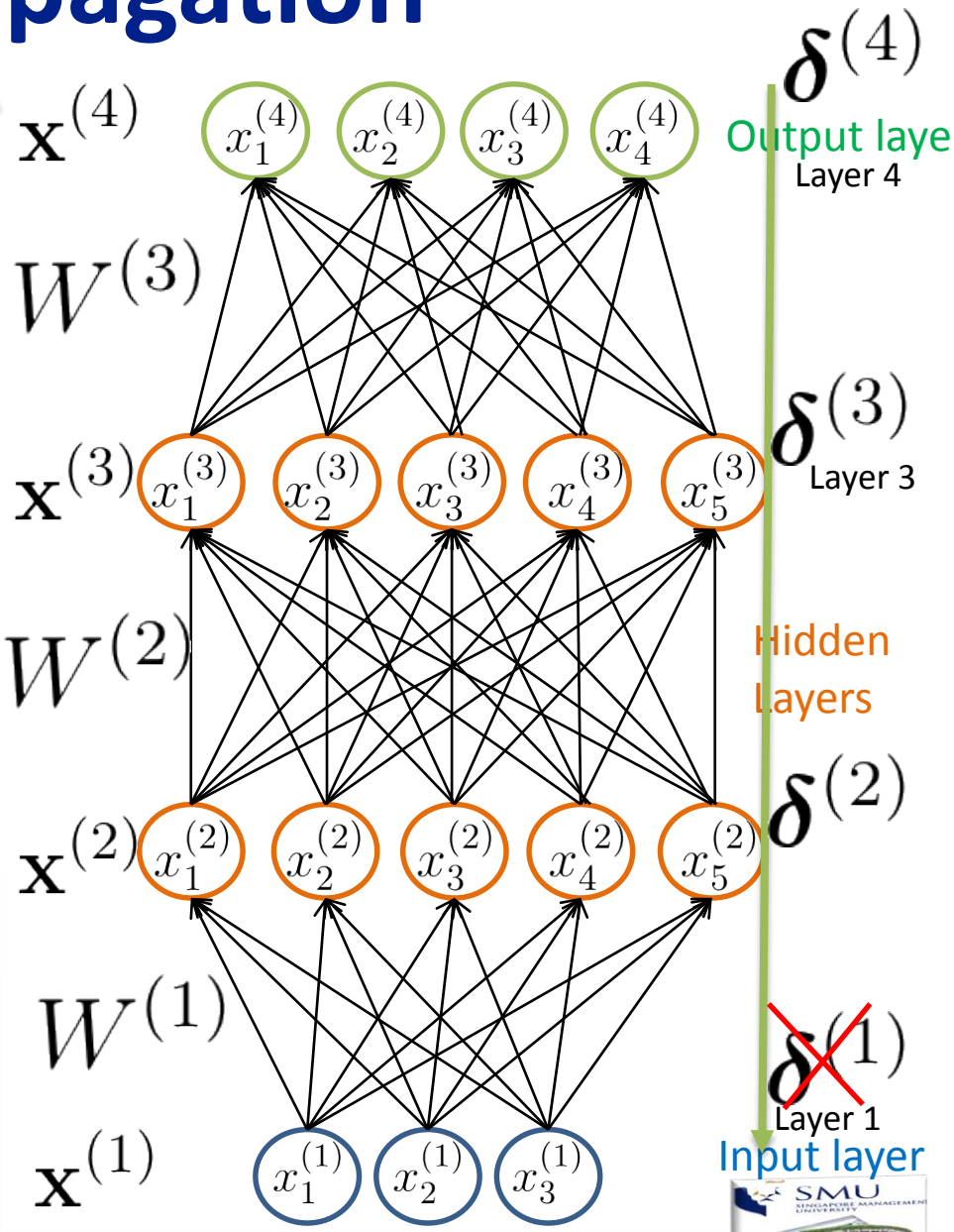
$$\boldsymbol{\delta}^{(4)} = (\mathbf{y} - \mathbf{x}^{(4)}) .* \mathbf{x}^{(4)} .* (1 - \mathbf{x}^{(4)})$$

$$\boldsymbol{\delta}^{(3)} = (W^{(3)})^T \boldsymbol{\delta}^{(4)} .* \mathbf{x}^{(3)} .* (1 - \mathbf{x}^{(3)})$$

$$\boldsymbol{\delta}^{(2)} = (W^{(2)})^T \boldsymbol{\delta}^{(3)} .* \mathbf{x}^{(2)} .* (1 - \mathbf{x}^{(2)})$$

$$\Delta W^{(l)} = \eta \boldsymbol{\delta}^{(l+1)} (\mathbf{x}^{(l)})^T$$

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} + \Delta W^{(l)}$$



Backpropagation Algorithm

Initialize all weights to small random numbers

Until convergence, Do

For each training example, Do

Forward-propagation

1. Input it to network and compute network outputs

Back-propagation

2. For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. For each hidden unit h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

4. Update each network weight $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

The details of derivations are given in appendix!

More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different inital weights

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Expressiveness of Neural Nets

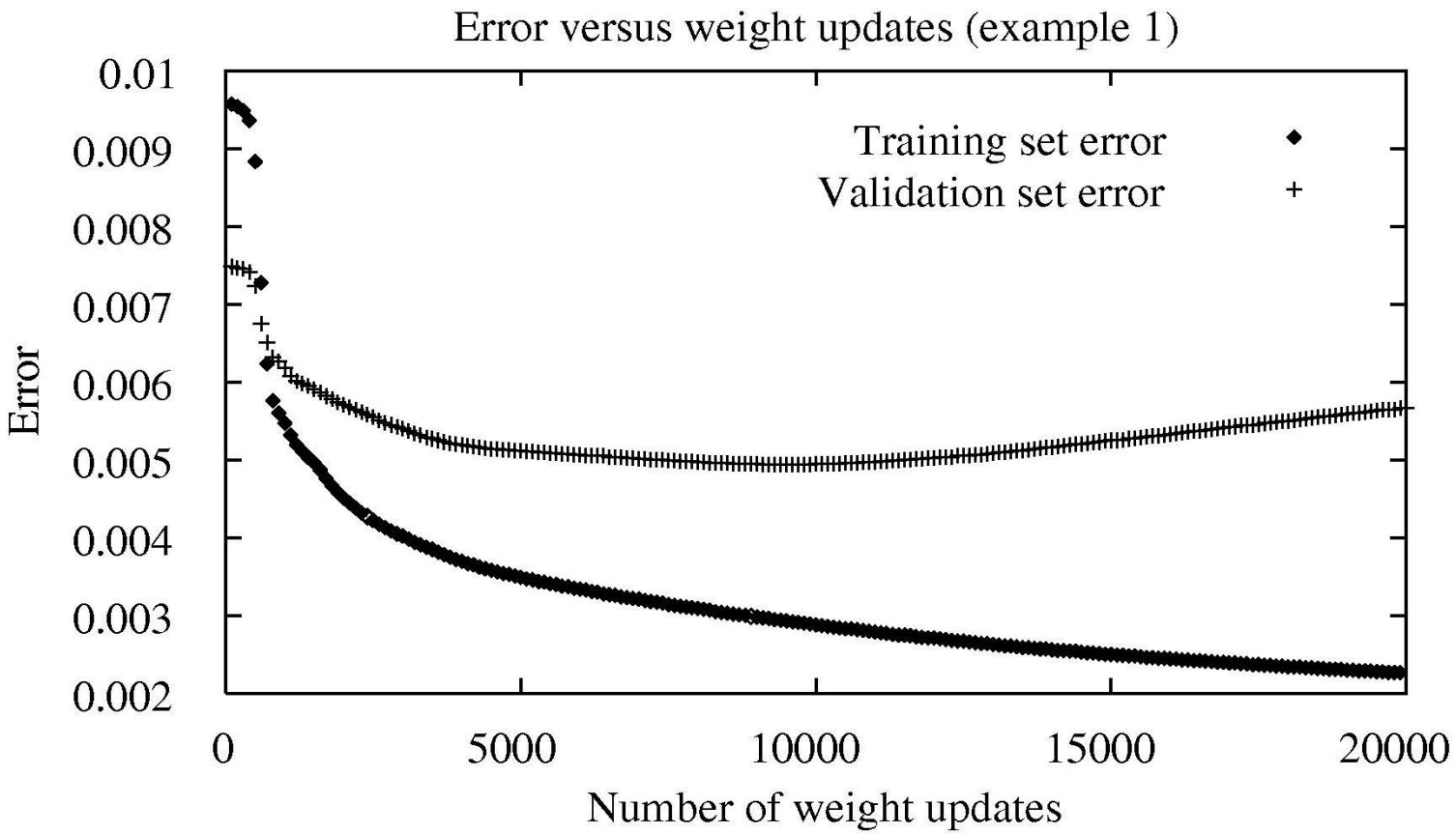
Boolean functions:

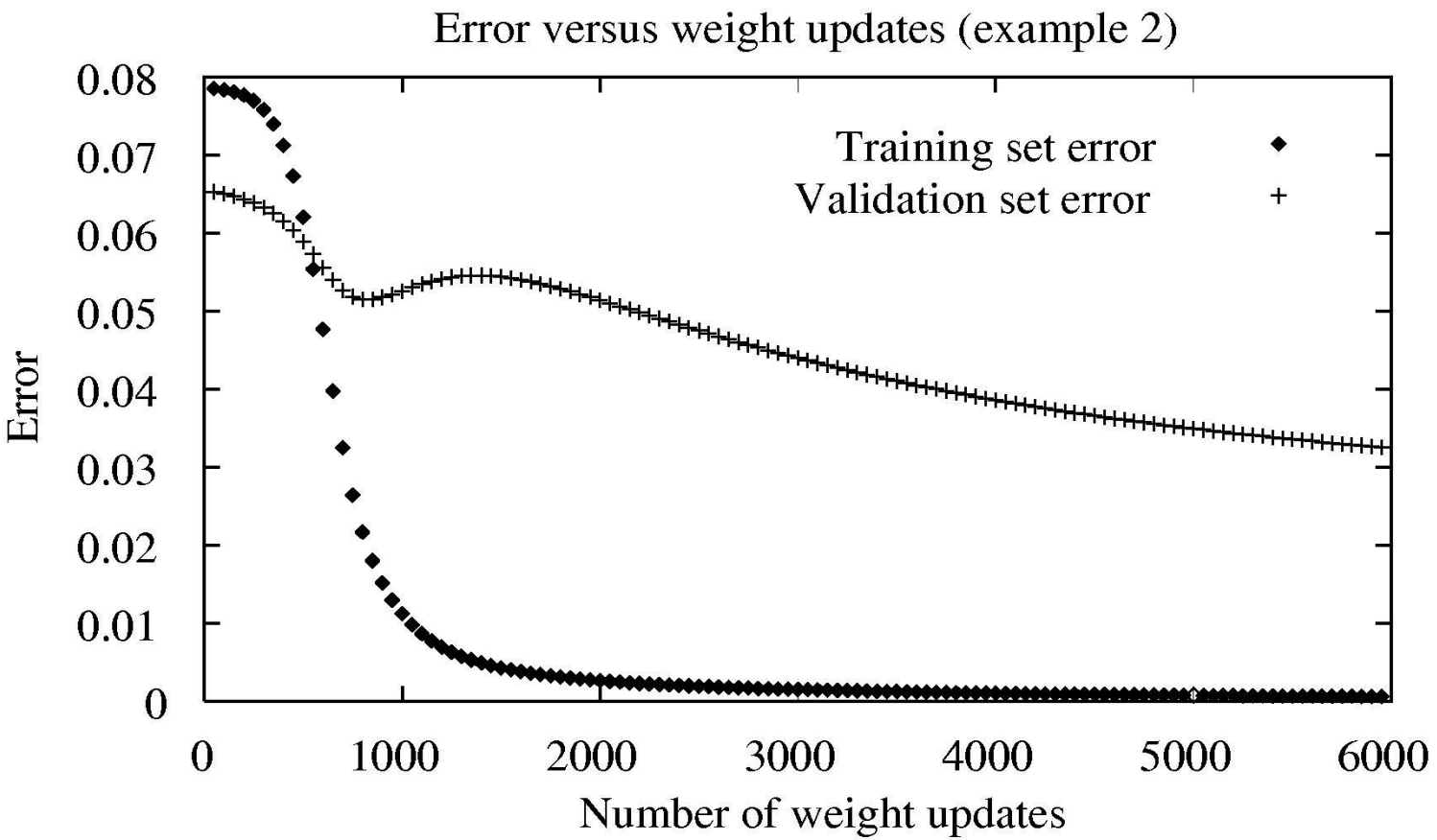
- Every Boolean function can be represented by network with single hidden layer
- But might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers

Overfitting in Neural Nets




Error versus weight updates (example 2)
Training set error
Validation set error
Error
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
1000
2000
3000
4000
5000
6000
Number of weight updates

Overfitting Avoidance

Penalize large weights:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

Train on target slopes as well as values:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

Weight sharing

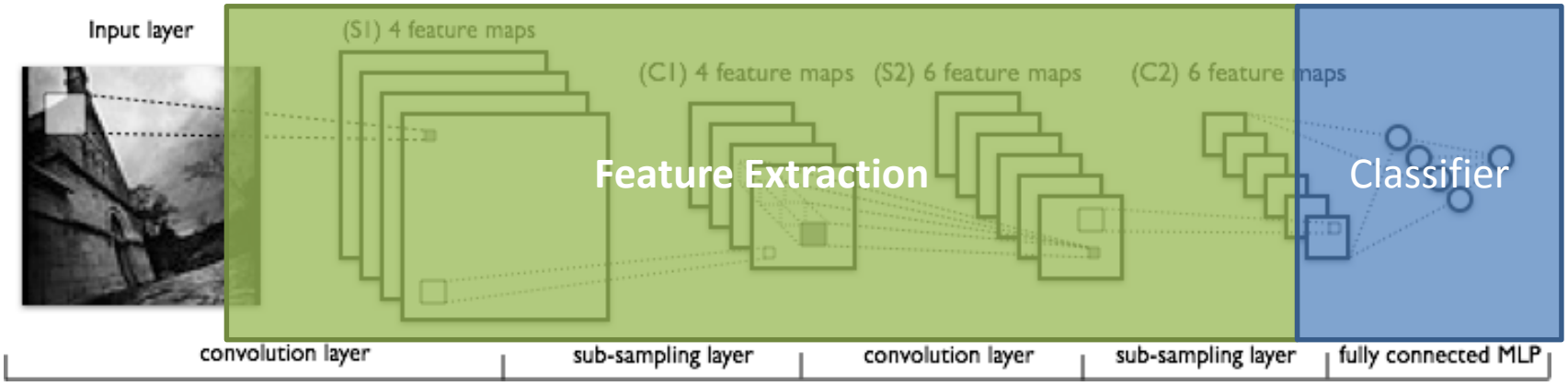
Early stopping

Convolutional Neural Networks (CNN)

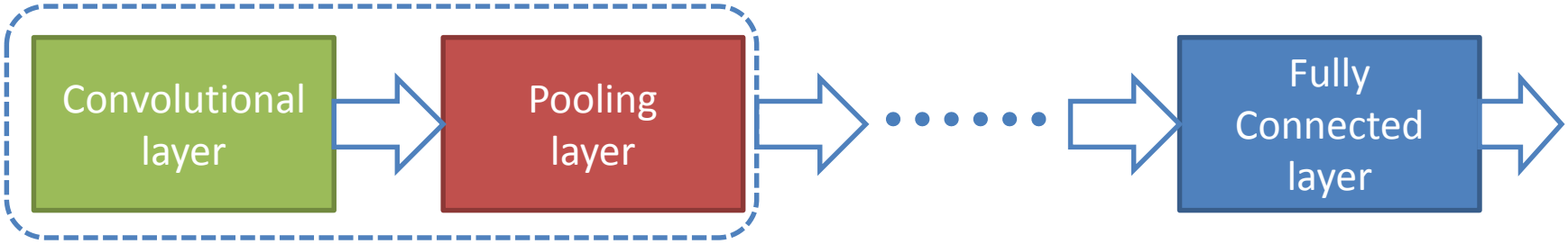
Yann Lecun (1989)

- Convolutional Neural Networks (CNN) are biologically-inspired variants of MLPs
- CNNs are designed to recognize visual patterns directly from pixel images with minimal preprocessing.
- Visual cortex contains a complex arrangement of cells, which are sensitive to small sub-regions of the visual field, called a ***Local Receptive Field***.
  - Simple cells respond maximally to specific edge-like patterns within their receptive field.
  - Complex cells have larger receptive fields and are locally invariant to the exact position of the pattern.

Deep CNN Example: LeNet

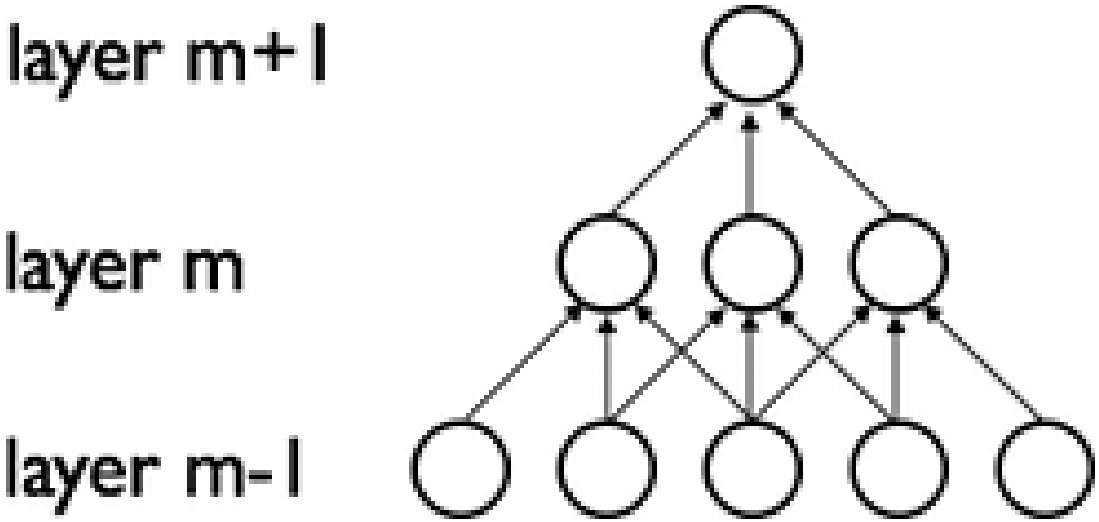


Stacked building blocks

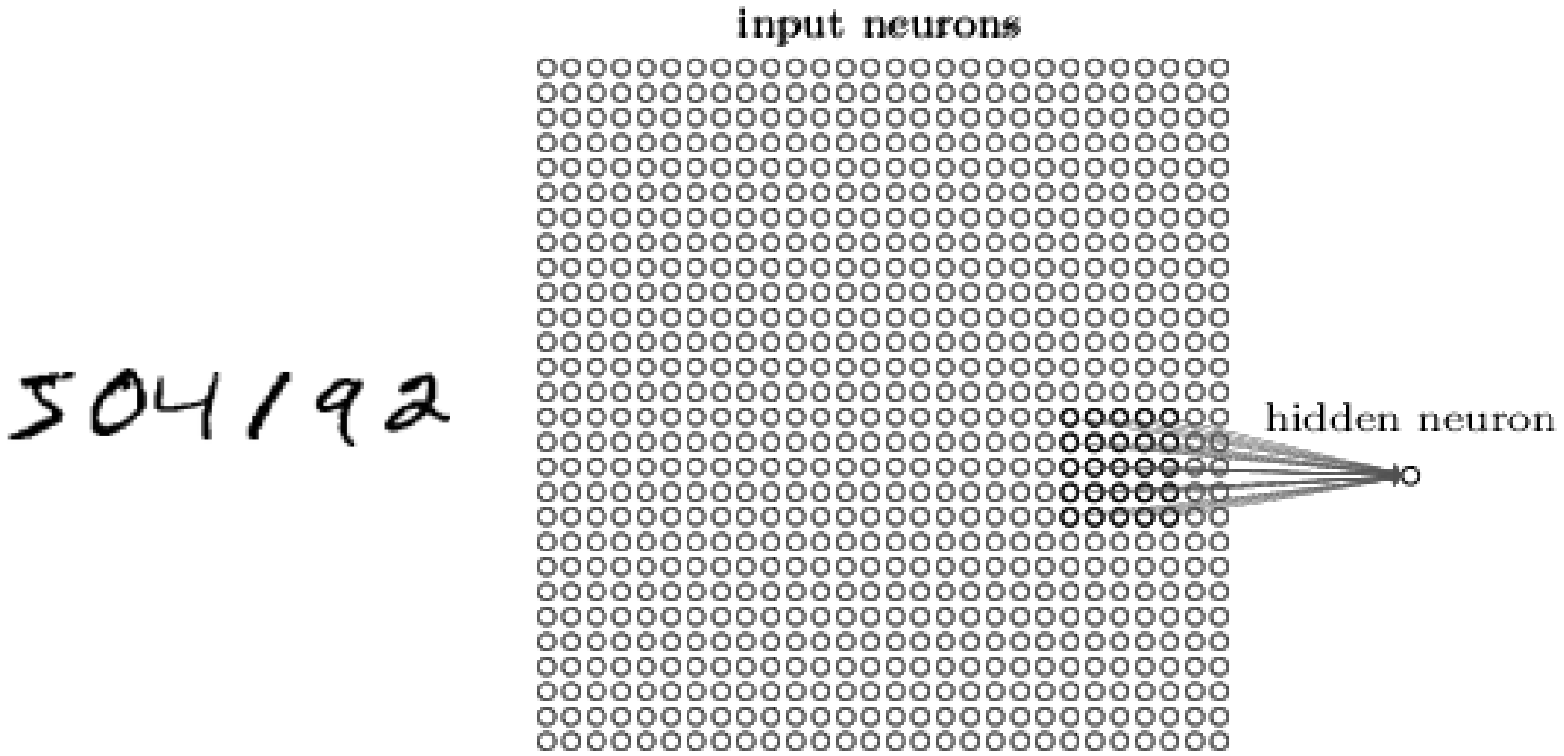


Local (Sparse) Connectivity

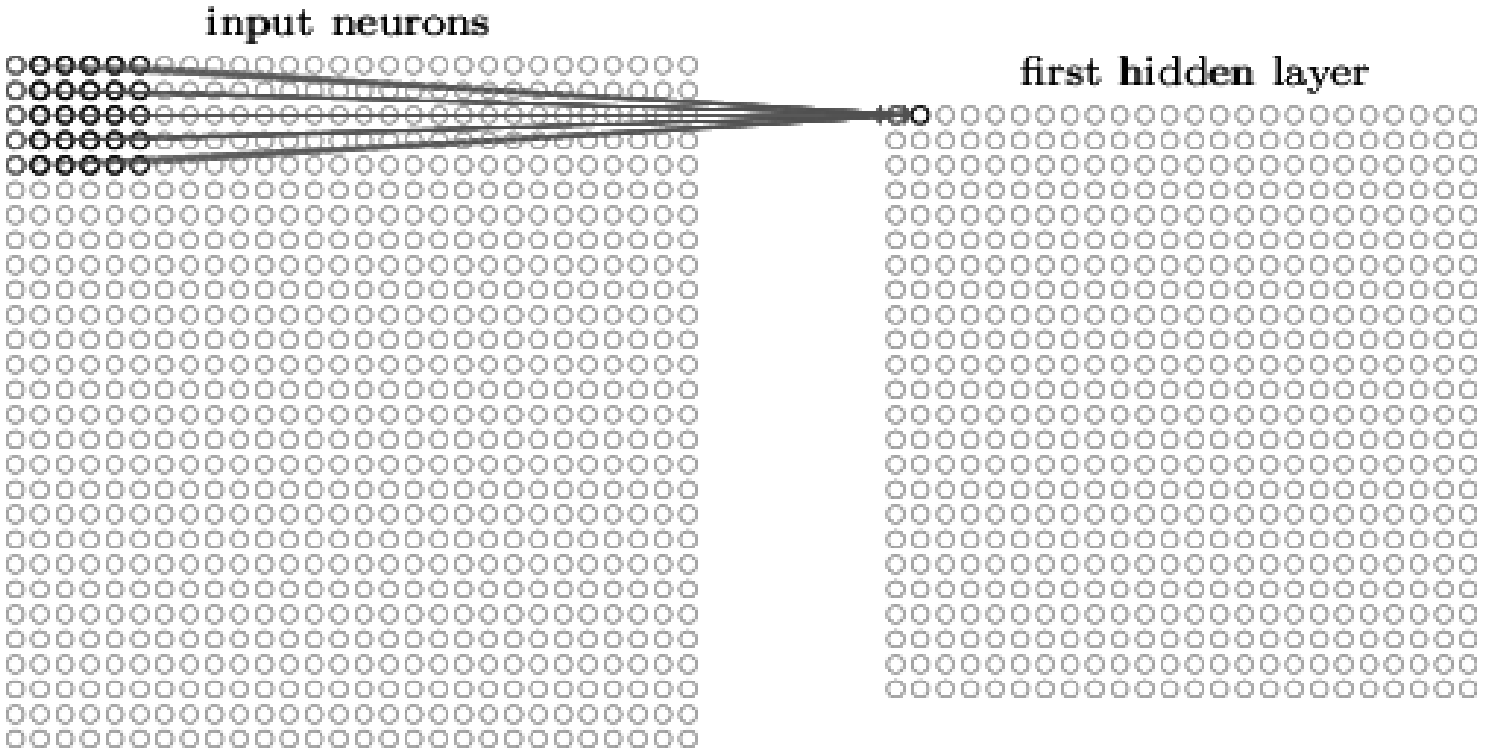
- CNNs exploit **spatially-local** correlation by enforcing a **local** connectivity pattern between neurons of adjacent layers.



Local receptive fields

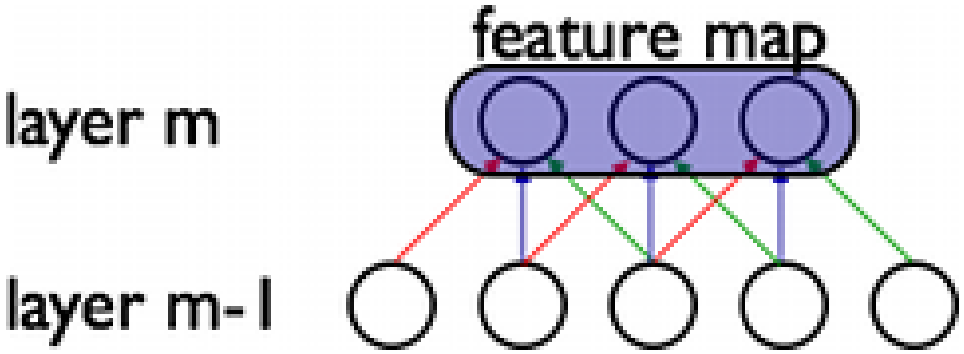


Convolution

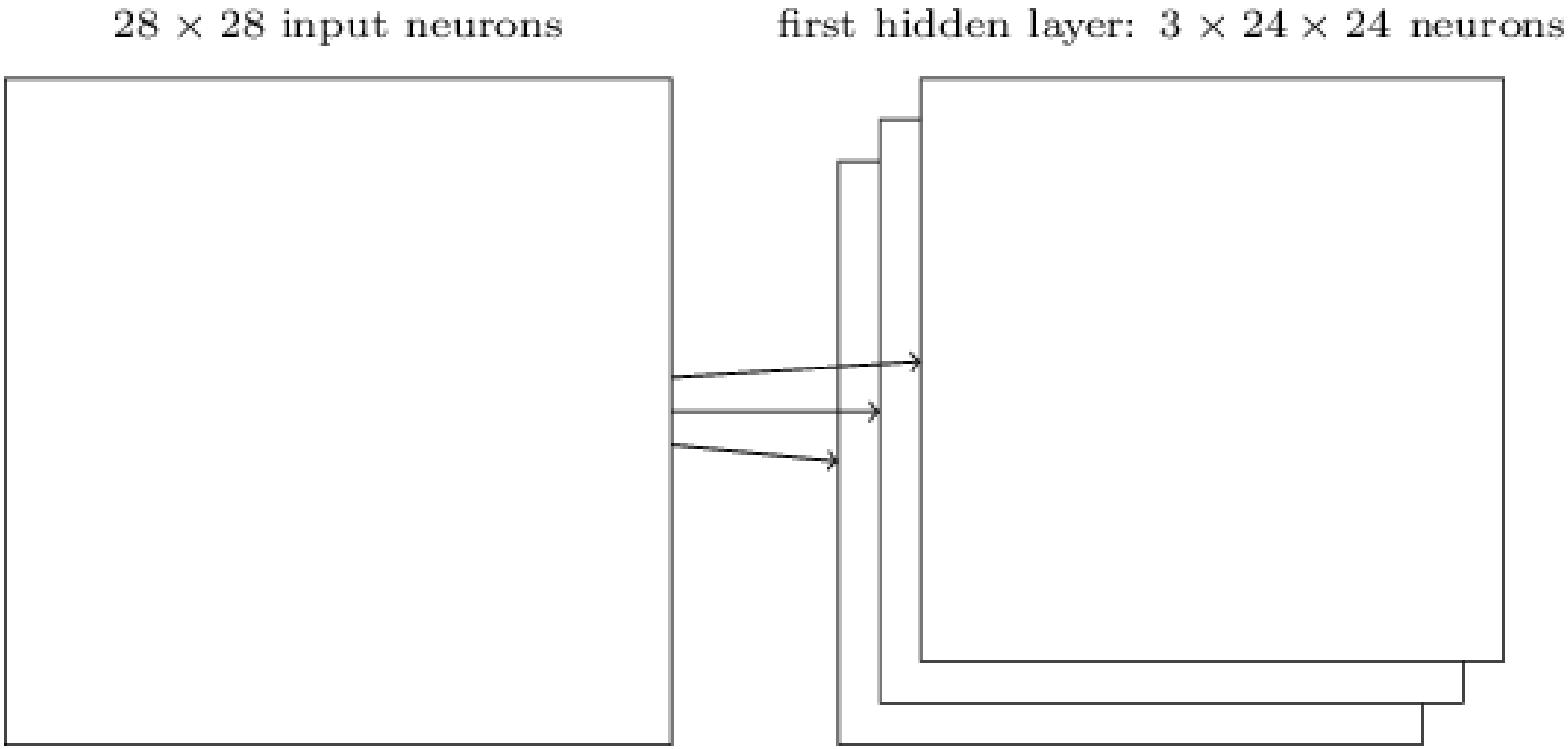


Filter: shared weights and bias

- **Feature map**: the map from input layer to the hidden layer
- **Shared weights:** the weights defining the feature map
- **Shared bias**: the bias defining the feature map
- **Filter or kernel**: the shared weights and bias
- In CNNs, each filter is replicated across the entire visual field.
- These replicated units share the same parameterization (weight vector and bias) and form a ***feature map***.



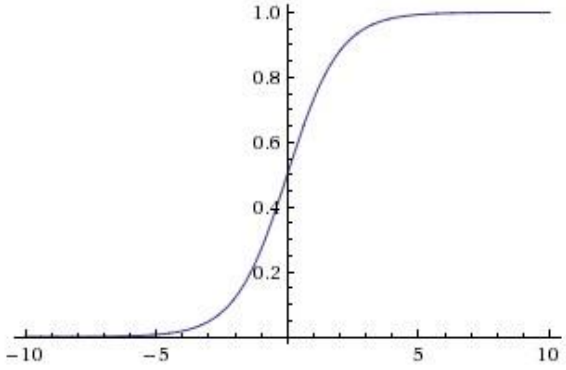
Convolution layer and Feature Maps



Activation (unit/layer)

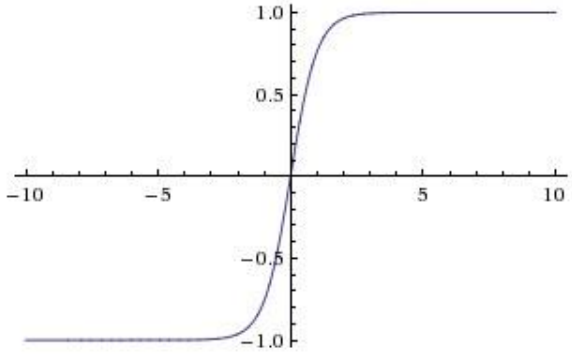
- Examples of activation functions

Sigmoid



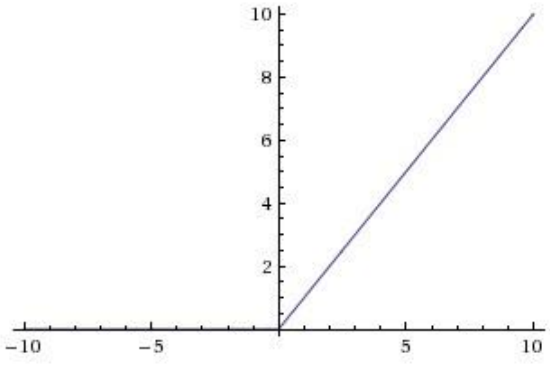
$$\sigma(x) = 1/(1+e^{-x})$$

Tanh



$$\tanh(x) = 2\sigma(2x) - 1$$

Rectified Linear Unit (ReLU)



$$f(x) = \max(0, x)$$

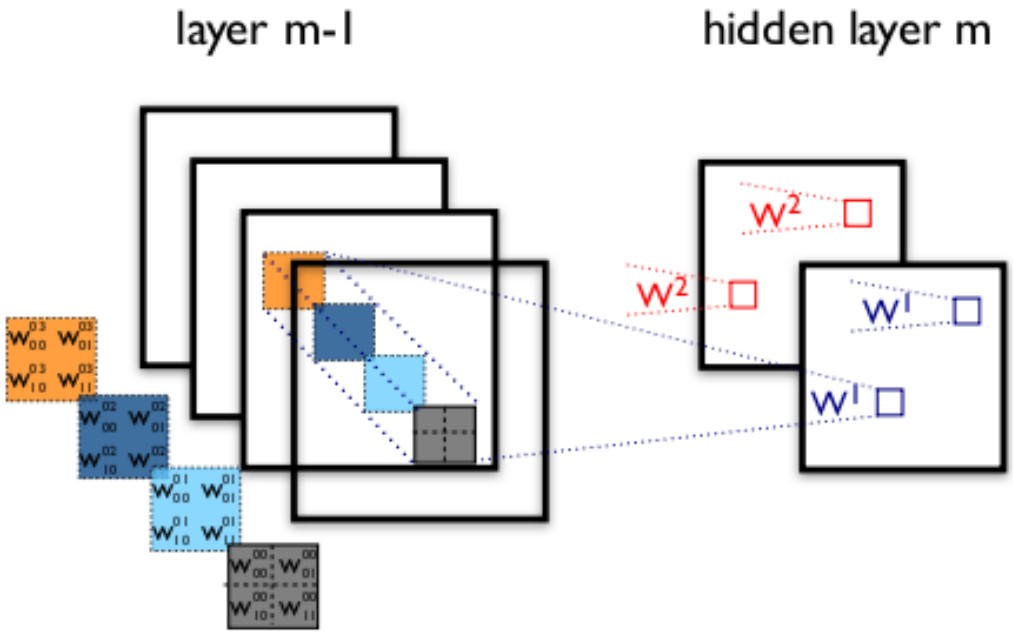
Convolution layer

- Feature map

activation

$$h_{ij}^k = \tanh((W^k * x)_{ij} + b_k)$$

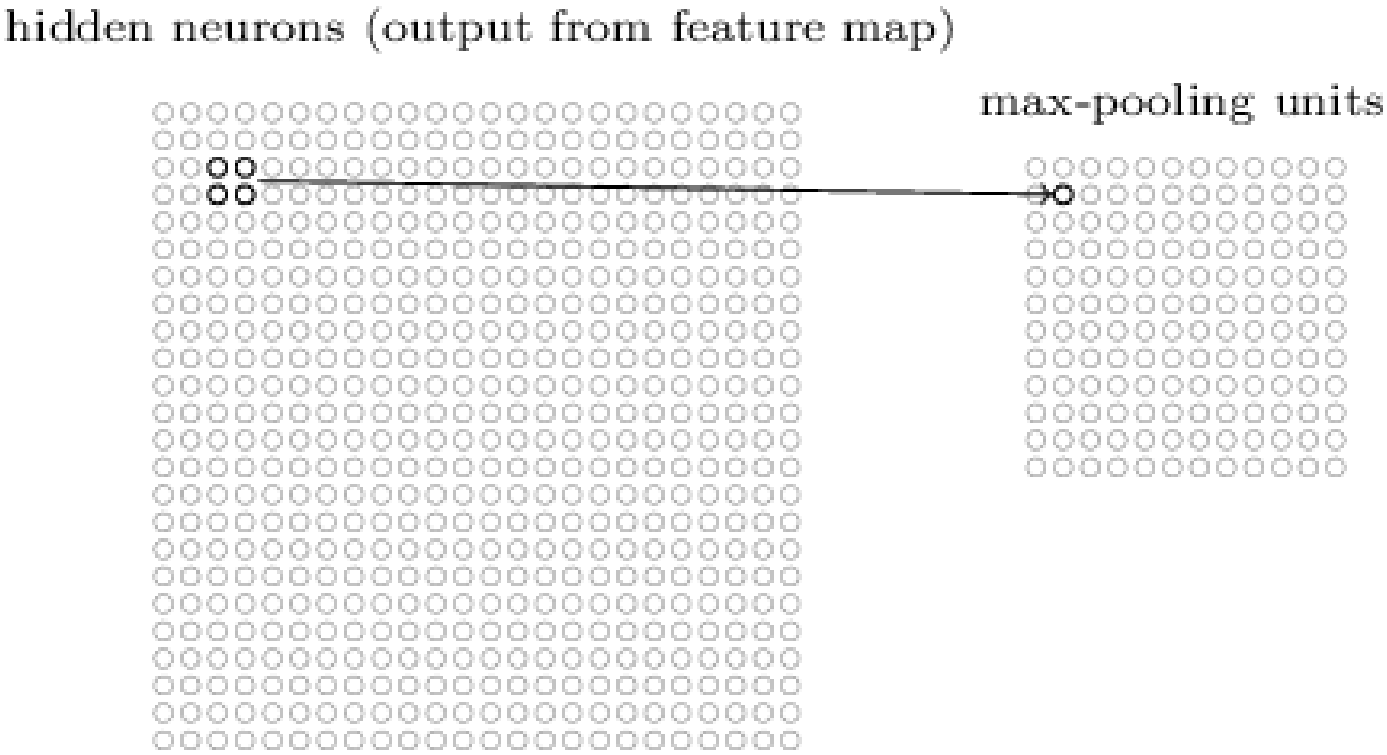
- Example



Max-Pooling

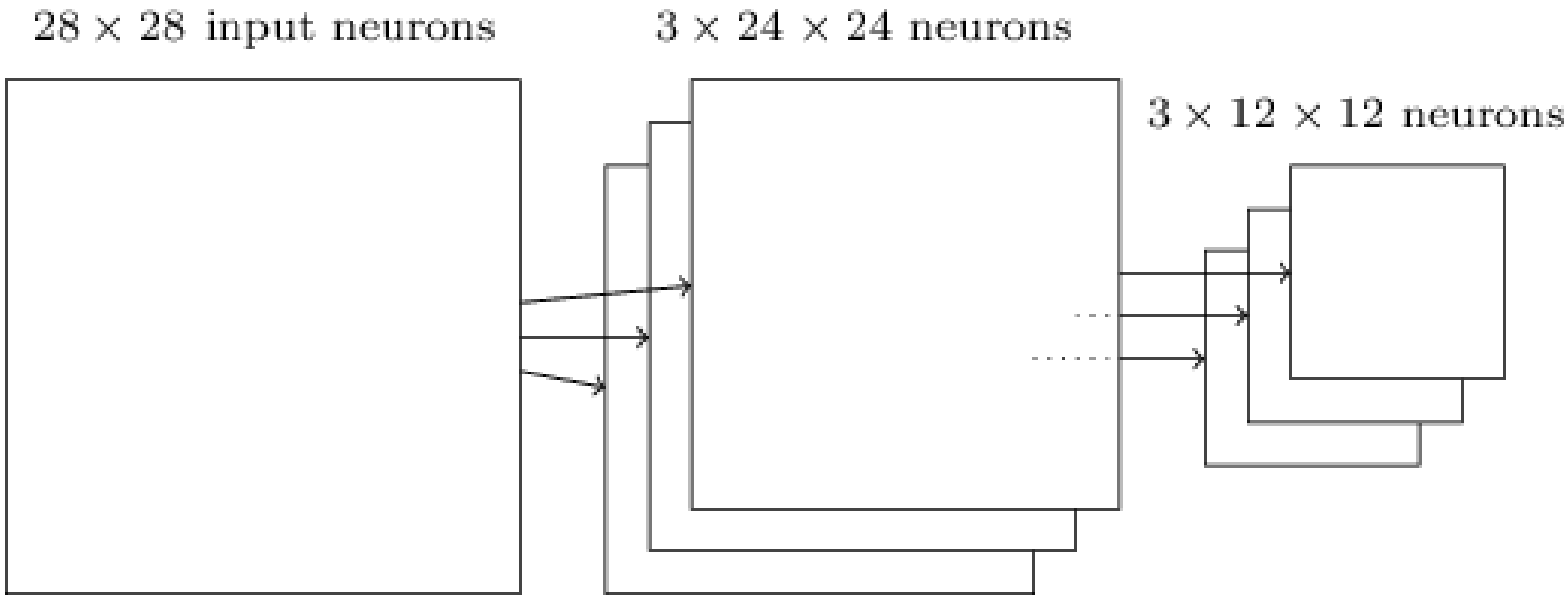
- It is a form of **non-linear down-sampling**
- Max-pooling partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value.
- Max-pooling is useful in vision for two reasons:
  - By eliminating non-maximal values, it reduces computation for upper layers.
  - It provides a form of translation invariance.
- Since it provides additional robustness to position, max-pooling is a "smart" way of reducing the dimensionality of intermediate representations.

Max-Pooling



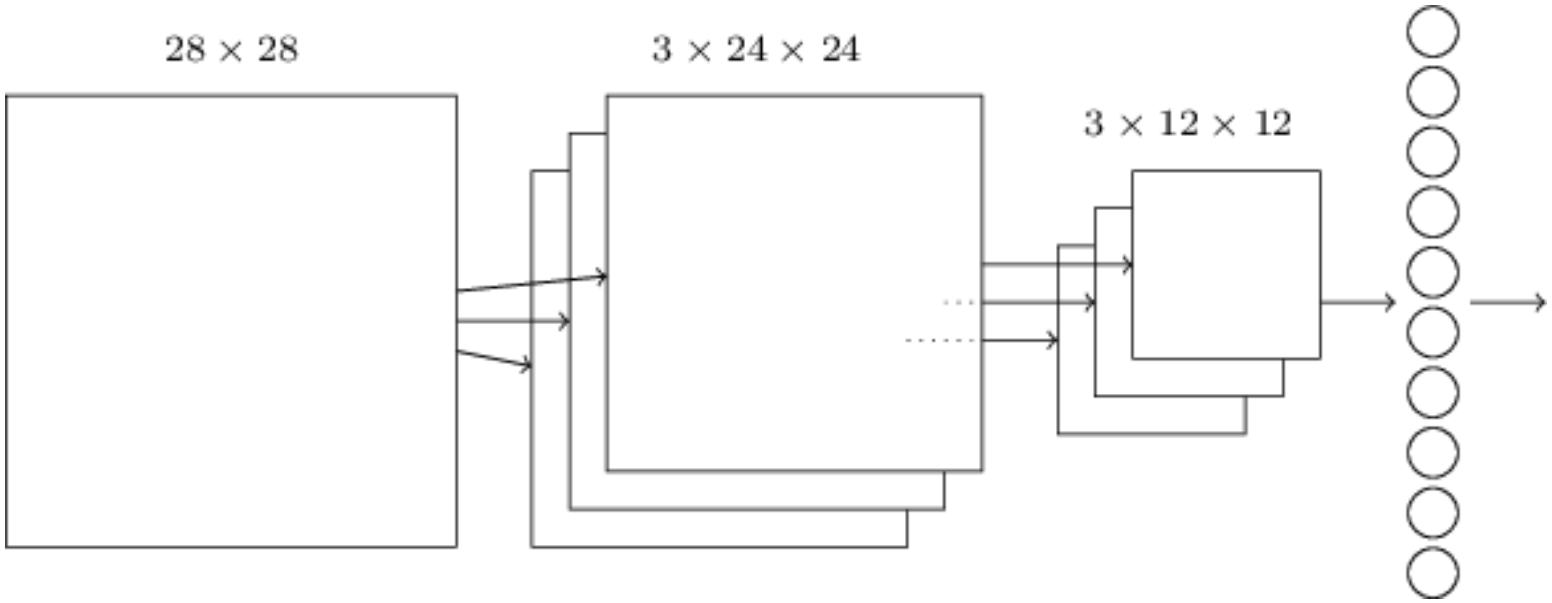
Convolution + Max-Pooling

- combined convolutional and max-pooling layers for three feature maps



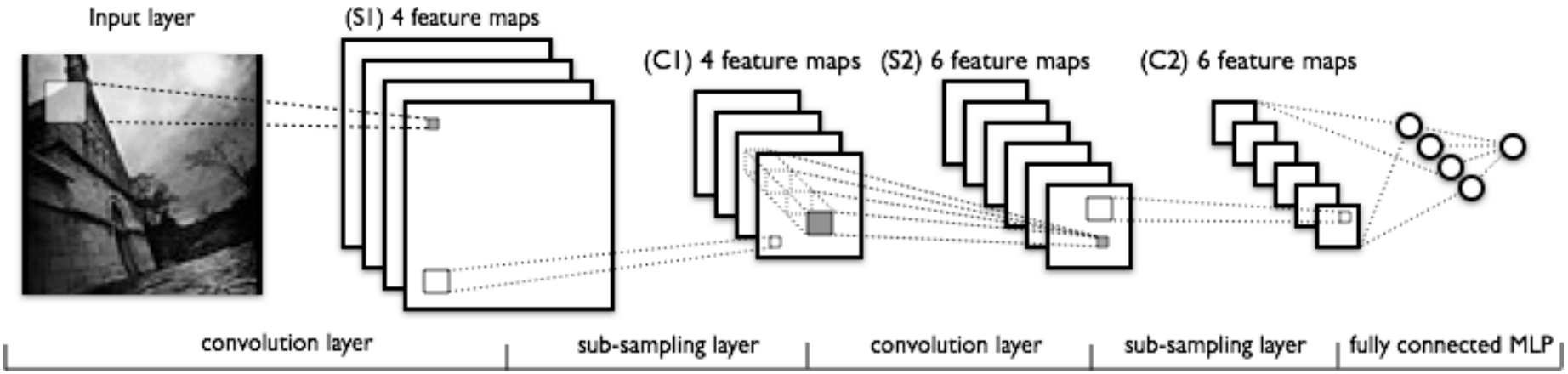
CNN: Putting all together

- Simple CNN example



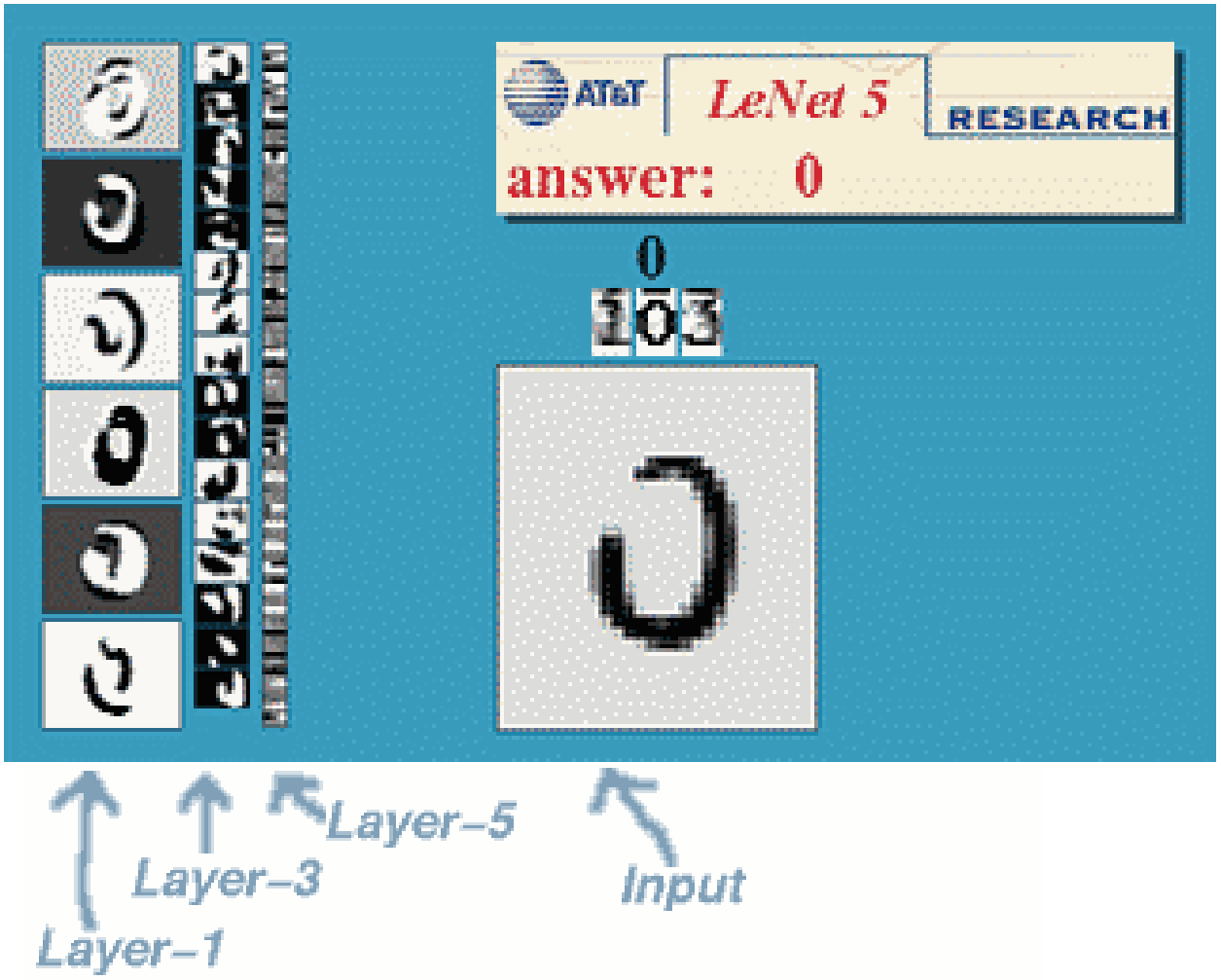
Example of A Full CNN Model: LeNet

- Sparse, convolutional layers and max-pooling are at the heart of the LeNet family of models.



The lower-layers are composed to alternating convolution and max-pooling layers. The upper-layers however are fully-connected and correspond to a traditional MLP (hidden layer + logistic regression). The input to the first fully-connected layer is the set of all features maps at the layer below.

Digit Recognition



Recent Progress in CNN

- Training CNN is still computationally expensive for large-scale data sets
- Take advantages of Parallel Computation
- Speed up CNN training using graphics processing unit (GPU) technology
- For example, CNN for ImageNet contest
  - Have to train on 1.2 million images in ImageNet LSVRC-2010 contest into 1000 different classes
- Other breakthroughs in training algorithms
  - Dropout: setting to zero the output of each hidden neuron with probability 0.5
  - Able to avoid overfitting

CNN with 2-GPU for ImageNet

- 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax

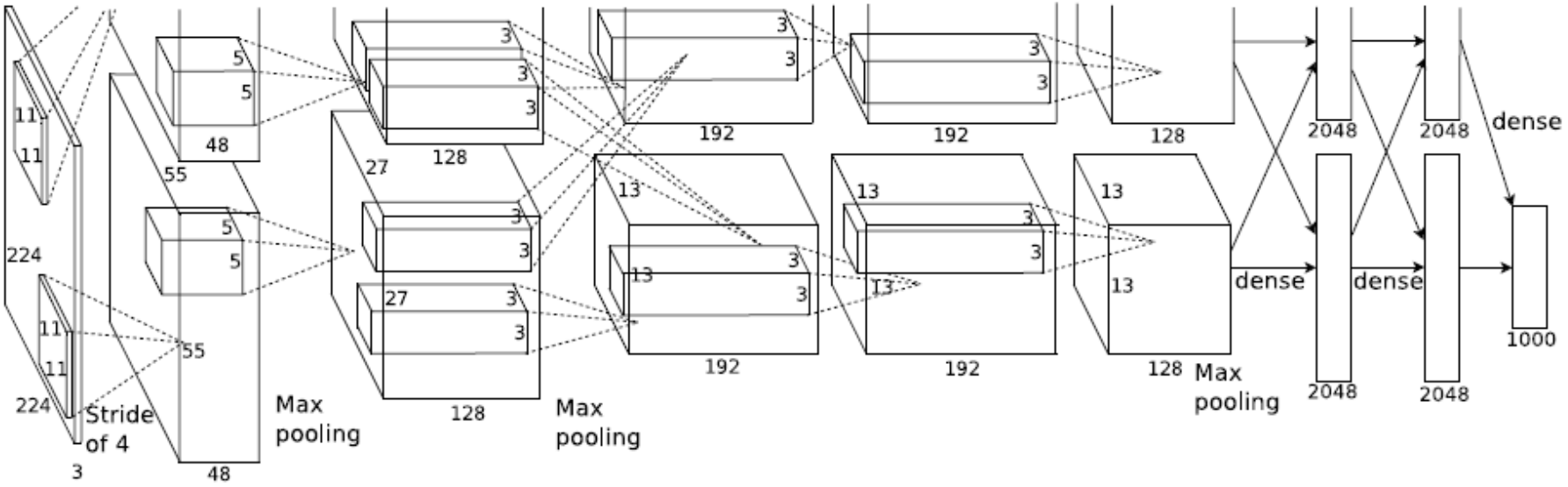


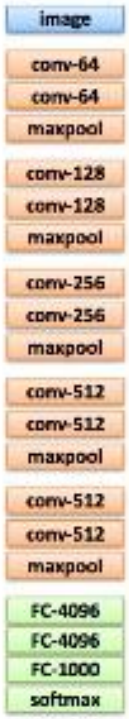
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

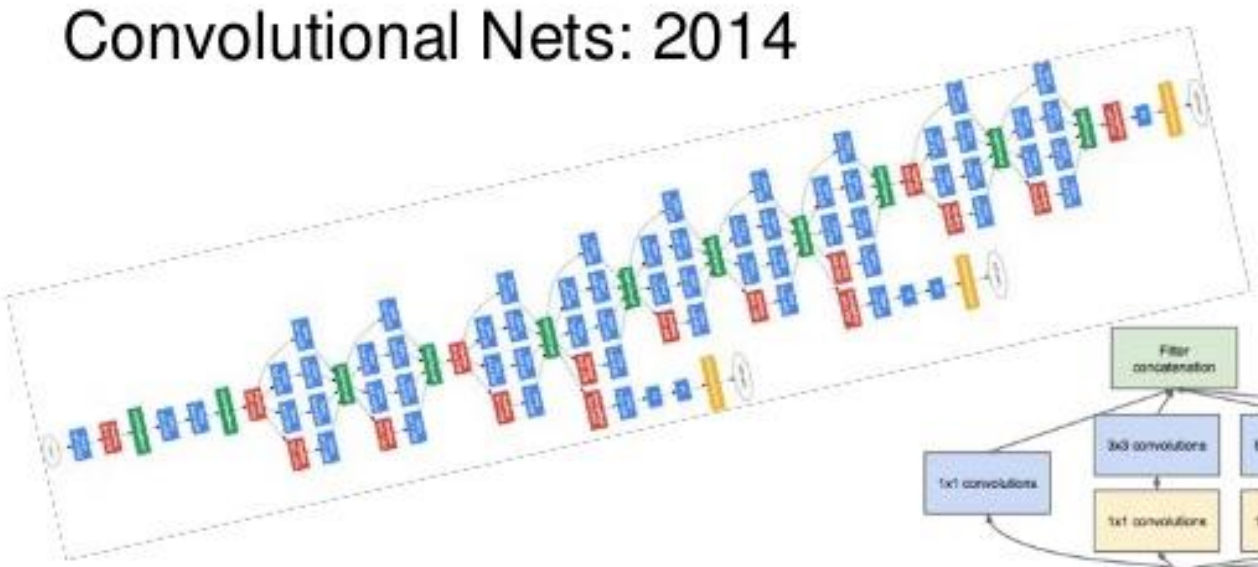
ImageNet benchmark: 1000 Categories for Visual Object Recognition

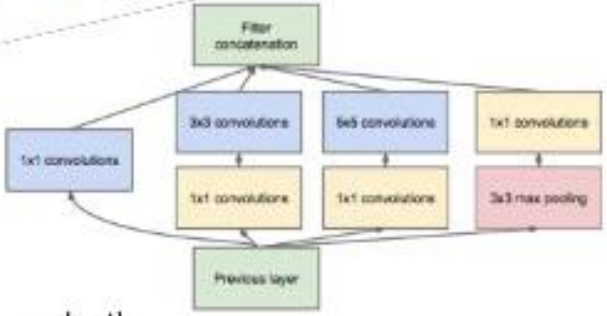
More Deep CNN Models

VGG
(19 layers)

GoogeLeNet







ILSVRC14 Winners: **~6.6% Top-5 error**

- GoogLeNet: composition of multi-scale dimension-reduced modules (pictured)
- VGG: 16 layers of 3x3 convolution interleaved with max pooling + 3 fully-connected layers

+ depth
+ data
+ dimensionality reduction

CNN Results on ImageNet

Model	Top-1	Top-5
Sparse coding [2]	*47.1%*	*28.2%*
SIFT + FVs [24]	*45.7%*	*25.7%*
CNN	**37.5%**	**17.0%**

Table 1: Comparison of results on ILSVRC-2010 test set. In *italics* are best results achieved by others.

Model	Top-1 (val)	Top-5 (val)	Top-5 (test)
SIFT + FVs [7]	—	—	*26.2%*
1 CNN	40.7%	18.2%	—
5 CNNs	38.1%	16.4%	**16.4%**
1 CNN*	39.0%	16.6%	—
7 CNNs*	36.7%	15.4%	**15.3%**

Table 2: Comparison of error rates on ILSVRC-2012 validation and test sets. In *italics* are best results achieved by others. Models with an asterisk* were "pre-trained" to classify the entire ImageNet 2011 Fall release. See Section 6 for details.

	team	top-5 (**test**)
in competition ILSVRC 14	MSRA, SPP-nets [11]	8.06
	VGG [25]	7.32
	GoogLeNet [29]	6.66
post-competition	VGG [25] (arXiv v5)	6.8
	Baidu [32]	5.98
	MSRA, PReLU-nets	**4.94**

Human Error:

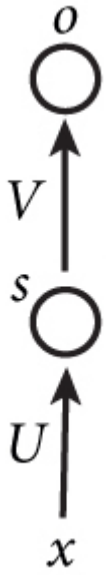
5.1% @ top-5

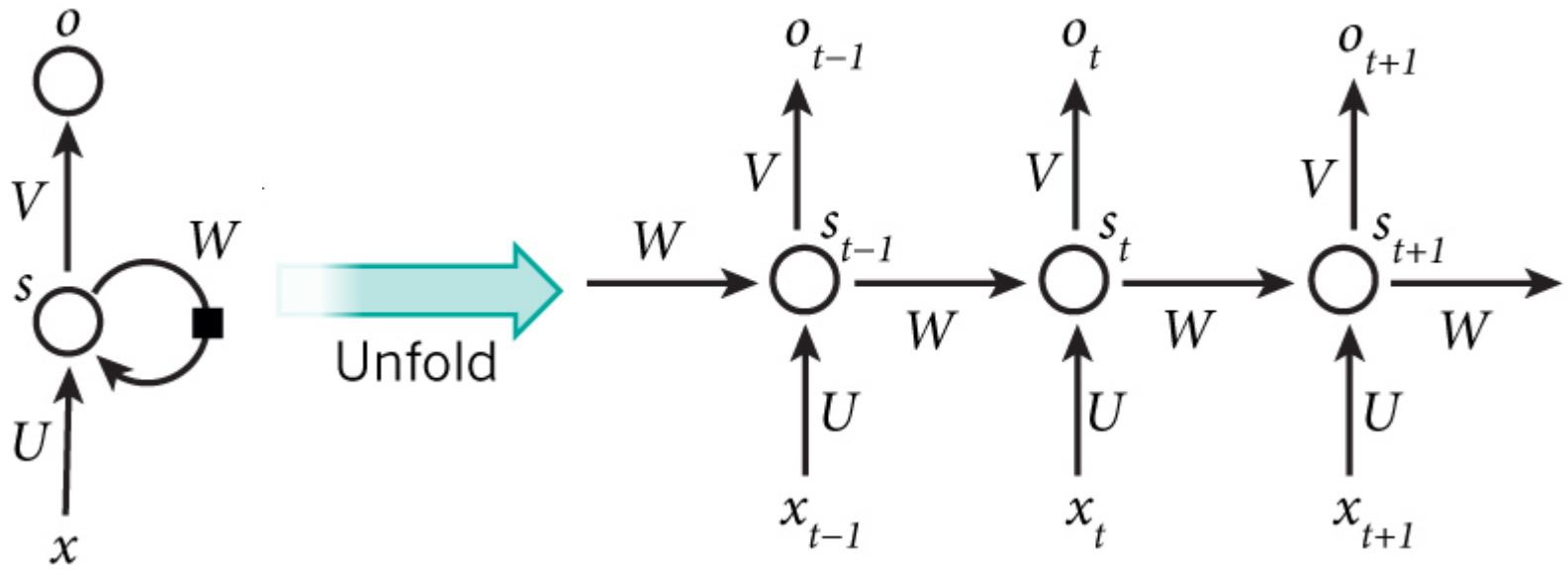
(Russakovsky et al. 2014)

Note: Results by MSRA on 6 Feb 2015

One More Thing

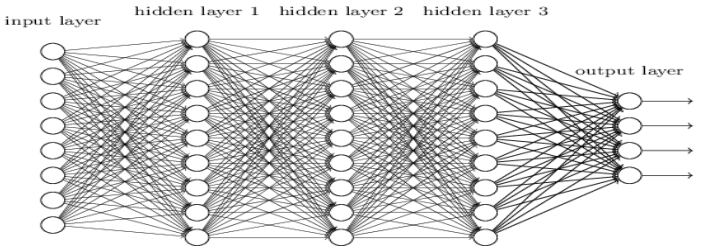
- Feedforward neural networks
  - DNN, CNN, etc
- Recurrent neural networks (RNN)



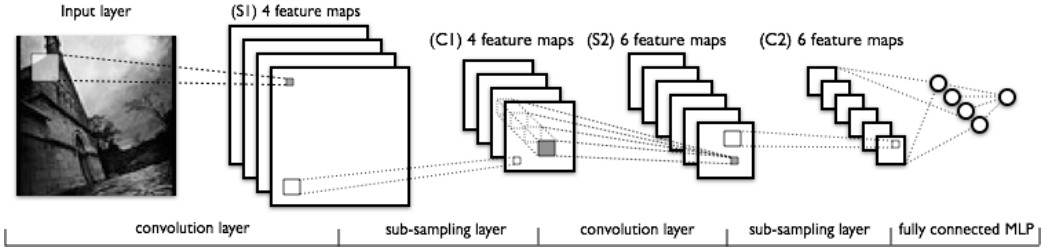


A Family of Deep Neural Networks

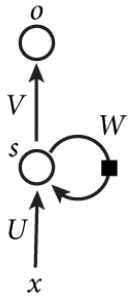
- Deep Neural Networks (DNN)
  - Generic data classification
  - Generic machine learning tasks

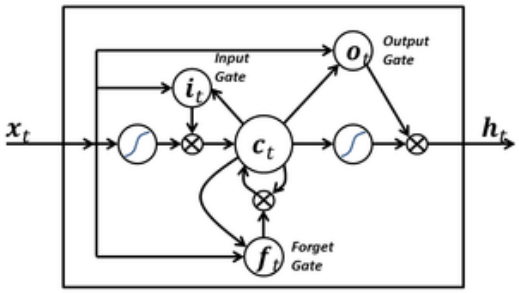


- Convolutional Neural Networks (CNN)
  - Image recognition
  - Object detection
  - Pattern recognition



- Recurrent Neural Networks (RNN), e.g., Long Short Term Memory (LSTM)
  - Speech recognition
  - Language modeling
  - Machine translation





Summary

- Basics of Neural Networks
- Perceptrons
- Stochastic Gradient Descent
- Multi-Layer Networks (Multi-Layer Perceptron)
- Backpropagation
- Convolutional Neural Networks (LeNet)

Appendix

- Derivation of Error Gradient
- Autoencoders: Learning the hidden representation

Derivation of Error Gradient

- x_{ji} : the i-th input to unit j

$$net_j = \sum_i w_{ji} x_{ji}$$

- Weight with the i-th input to unit j

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial E}{\partial w_{ji}}$$

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} = \frac{\partial E}{\partial net_j} x_{ji}$$

Derivation of Error Gradient

- **Case 1: Output unit weights**

$$
\begin{aligned}
\frac{\partial E}{\partial net_j} &= \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j} \\
&= \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2 o_j(1-o_j) \\
&= \frac{1}{2}2(t_j - o_j)\frac{\partial (t_j - o_j)}{\partial o_j} o_j(1-o_j) \\
&= -(t_j - o_j)o_j(1-o_j)
\end{aligned}
$$

Derivation of Error Gradient

- **Case 2: Hidden unit Weights**

$$\delta_k = -\frac{\partial E}{\partial net_k}$$

$$\begin{aligned}
\frac{\partial E}{\partial net_j} &= \sum_{k \in Output(j)} \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Output(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Output(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Output(j)} -\delta_k w_{kj} o_j (1 - o_j)
\end{aligned}$$

Derivation of Error Gradient

- Summary

2. For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. For each hidden unit h

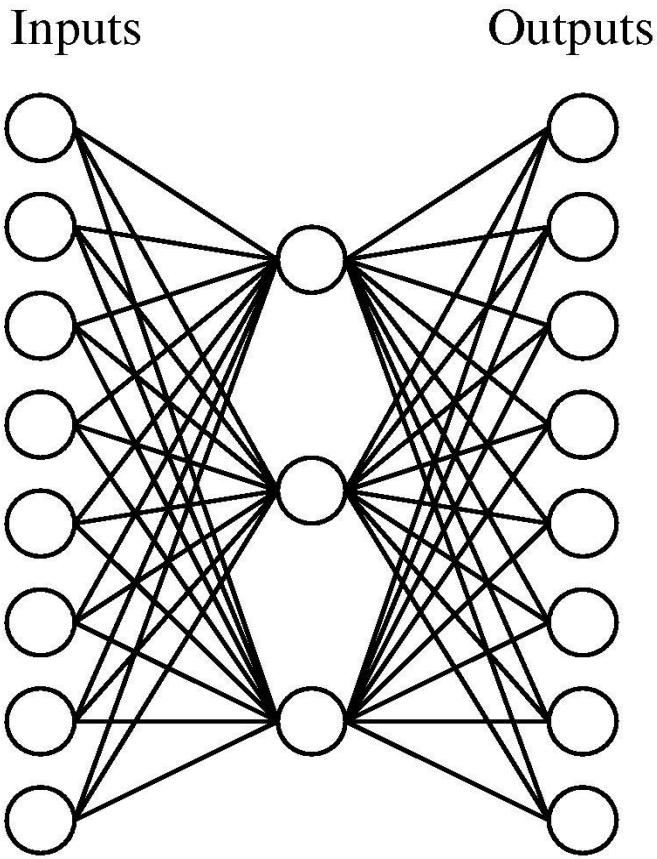
$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

4. Update each network weight $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

Learning Hidden Layer Representations



A target function:

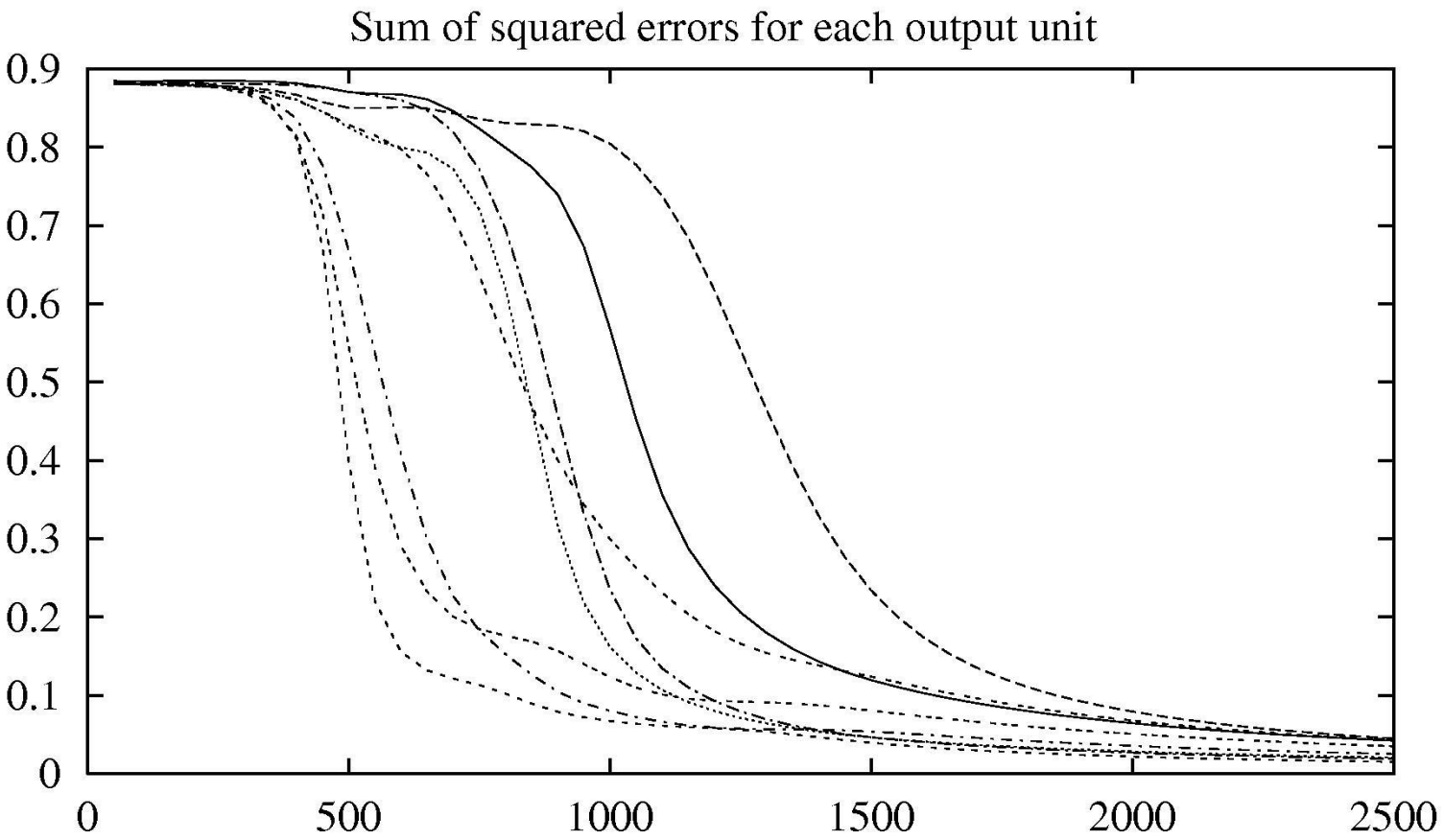
Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned?

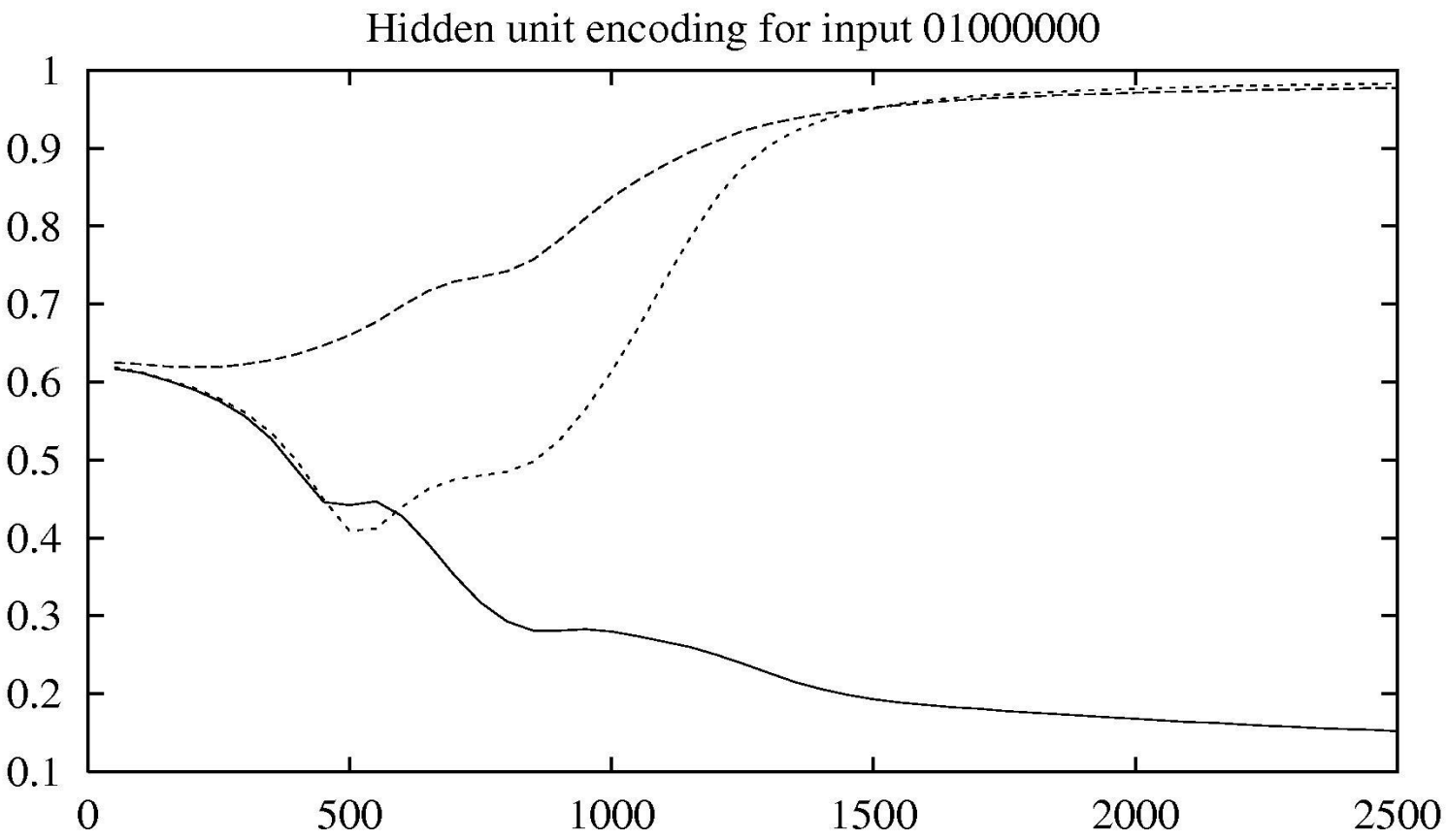
Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

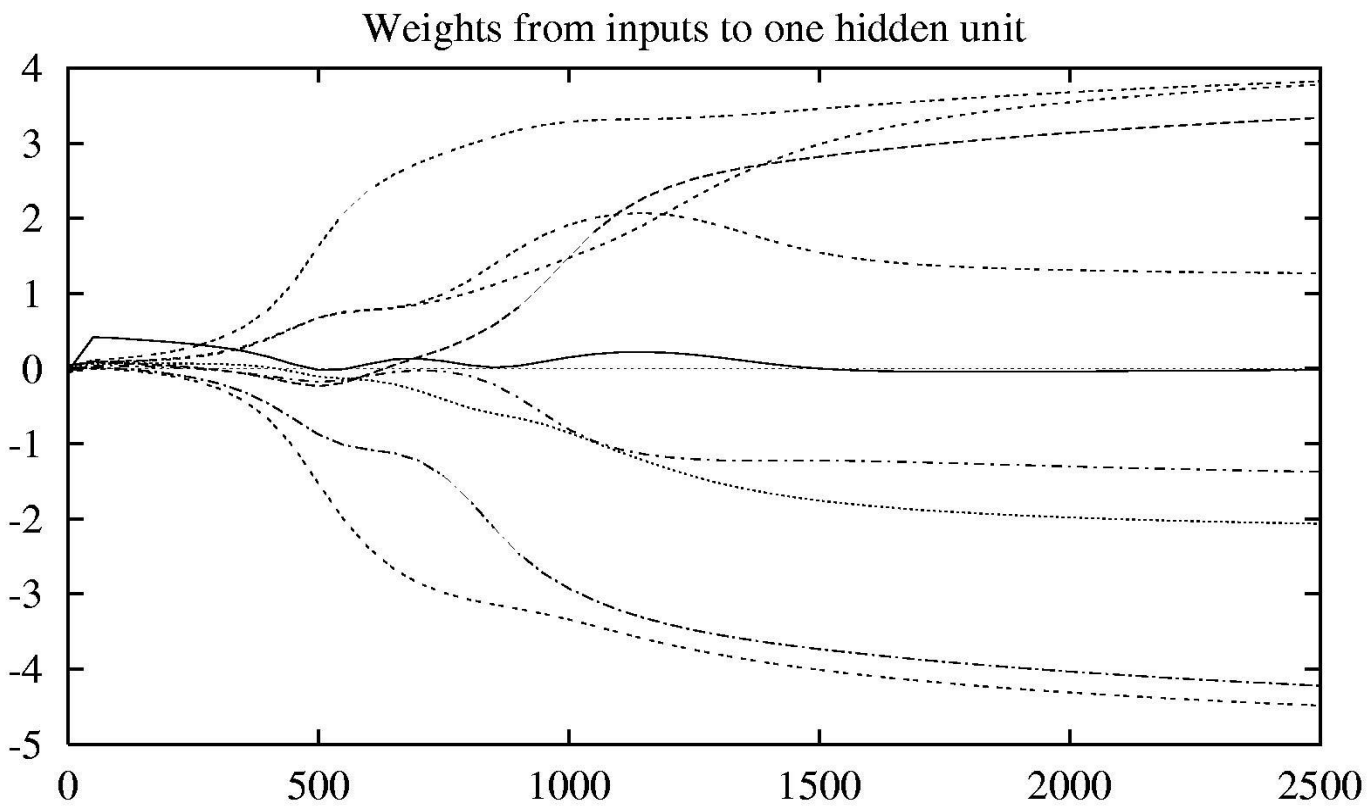
Training


Sum of squared errors for each output unit
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500

Training


Hidden unit encoding for input 01000000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
500
1000
1500
2000
2500

Training


Weights from inputs to one hidden unit
4
3
2
1
0
-1
-2
-3
-4
-5
0
500
1000
1500
2000
2500