

Lecture 6

Neural Networks and Deep Learning

Michail G. Lagoudakis

Intelligent Systems Laboratory
School of Electrical and Computer Engineering
Technical University of Crete
Chania, Crete, Greece



Machine Learning



Machine Learning

- learn from data to make predictions (and take decisions)
- **Supervised Learning**
 - set of training data with inputs and targets
 - classification, regression, ...
- **Unsupervised Learning**
 - set of training data with inputs, but without targets
 - clustering, density estimation, dimensionality reduction, ...
- **Reinforcement Learning**
 - set of training trials of interaction with feedback by a critic
 - value function, decision policy, exploration vs. exploitation, ...
- **Learning Theory**
 - theoretical investigations: what can be learned? how fast? ...



Machine Learning Terminology

- **Dataset**

- data for training, validation, testing

- **Model**

- structure/artifact created through a training process

- **Cost/Loss/Objective function**

- measures model performance

- **Optimization Procedure**

- minimizes error over (training) data



Machine Learning Terminology ++

- **Training error:** error rate (over the training set)
- **Testing error:** generalization error (over the testing set)
(the expected value of the error on new inputs)
- **Parameters/Estimators:** a vector of estimated parameter values
(e.g. weights of a neural network) giving the "best" prediction
- **Hyperparameters:** parameter settings to control the algorithm's
behavior, learned during **validation** (**k-fold cross-validation**)
- **Bias:** expectation over data included in the estimator
- **Maximum Likelihood:** common principle for parameter estimation

$$\theta_{ML} = \arg \max p_{model}(\mathbb{X}; \theta)$$

- find parameter values to make observations most probable given the model

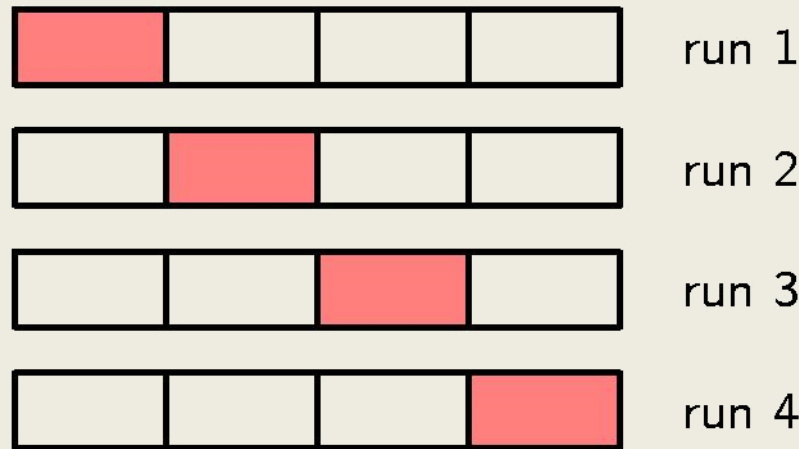


Model Selection

- **Models**
 - polynomials, mixture distributions, neural networks, ...
- **Complexity**
 - the number of (free) model parameters
- **Control**
 - selecting the order of a polynomial
 - selecting the number of distributions in the mixture
 - selecting the number of hidden units in a neural network
 - ...
- **Approach**
 - train several models on the same training set
 - look for best predictive performance on independent data



k -Fold Cross-Validation



example: $k=4$

- training on $k-1$ groups, testing on the remaining group
- averaging performance over all k runs
- all data are used for both training and testing
- for $k=N$ (number of data): leave-one-out technique
- drawback: computationally expensive, large number of runs

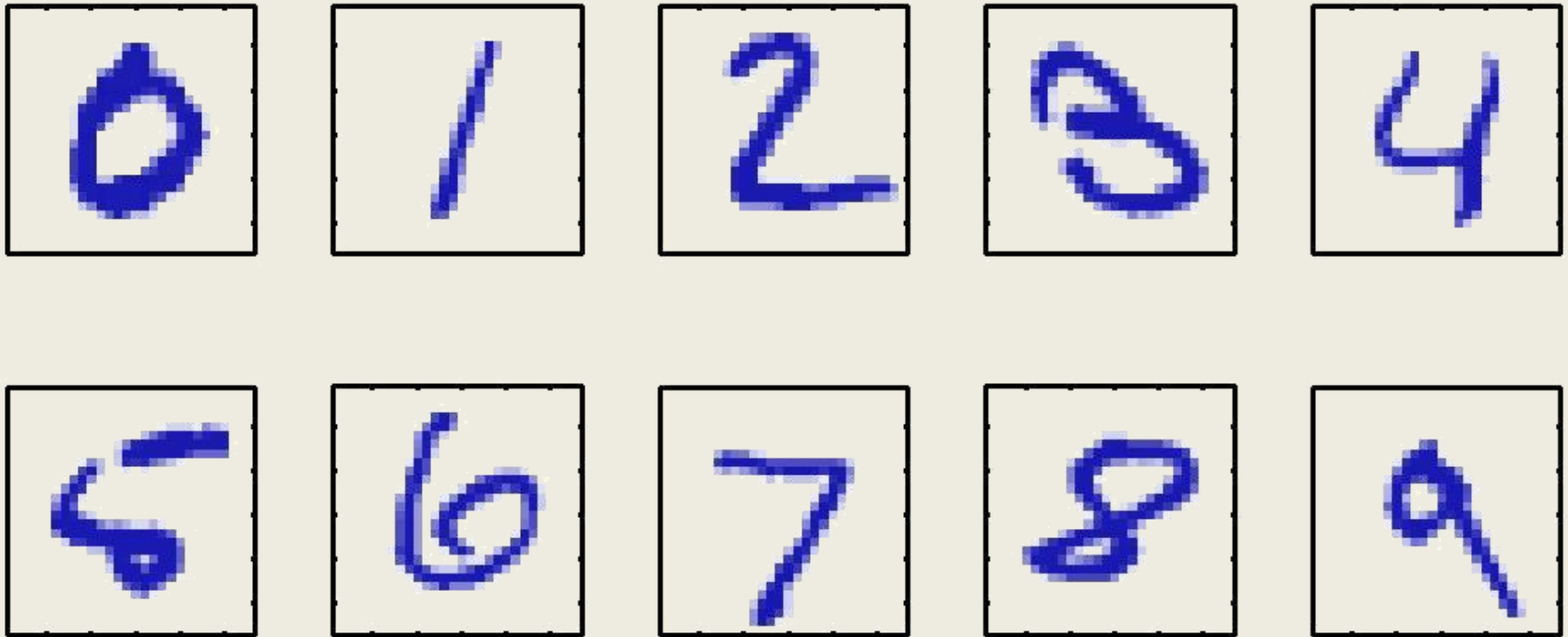


Machine Learning Examples



Hand-Written Digit Recognition

input: digital image (28x28 pixels)



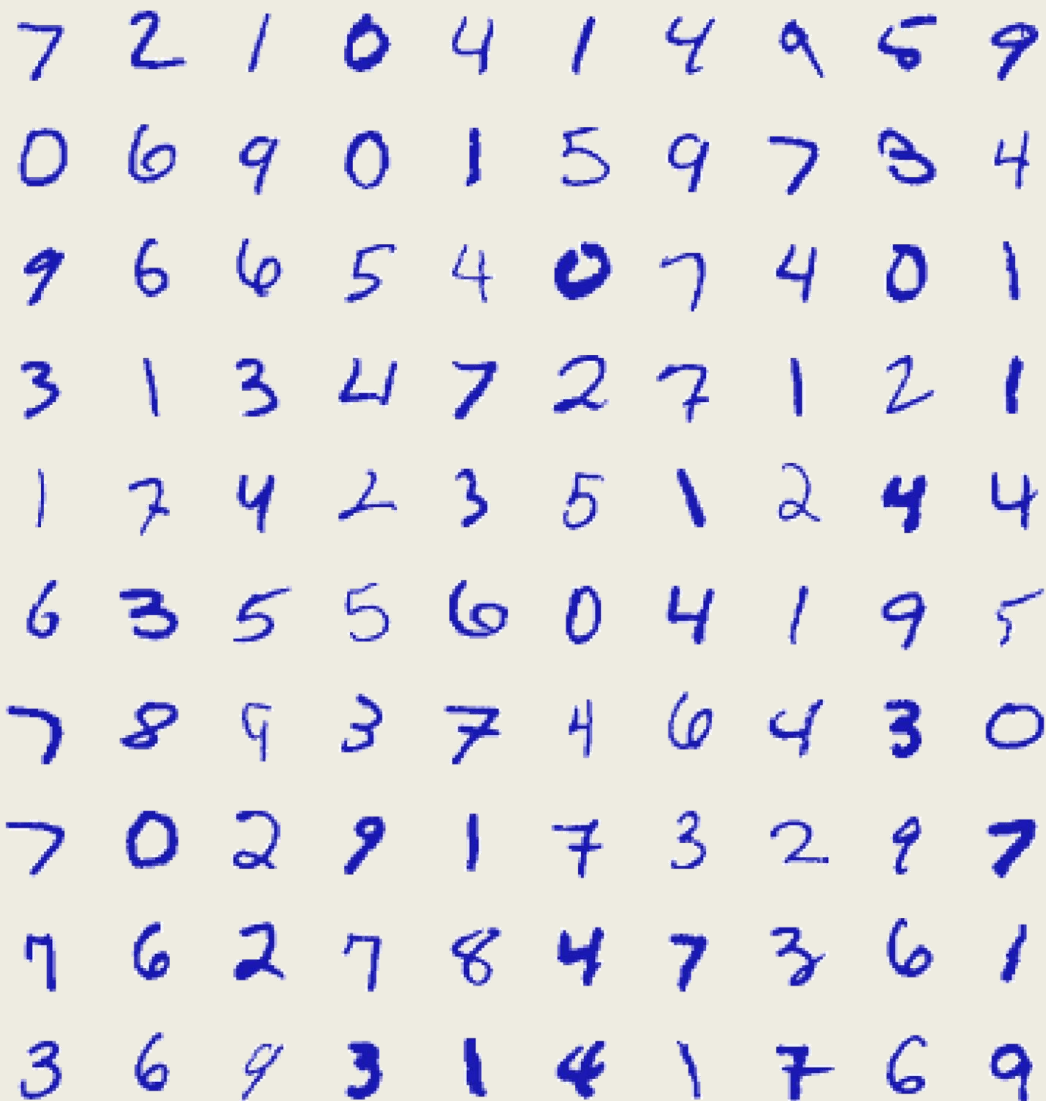
target: label 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9



Classification: Hand-Written Digit Recognition

• MINST Data Set

- originated by USPS mail
- 60000 training examples
- 10000 test examples
- normalized to 20×20
- fit to 28×28 square
- solved problem today
- scale?
- rotation?
- invariance?



Heart Abnormalities

No abnormalities



Atrial fibrillation

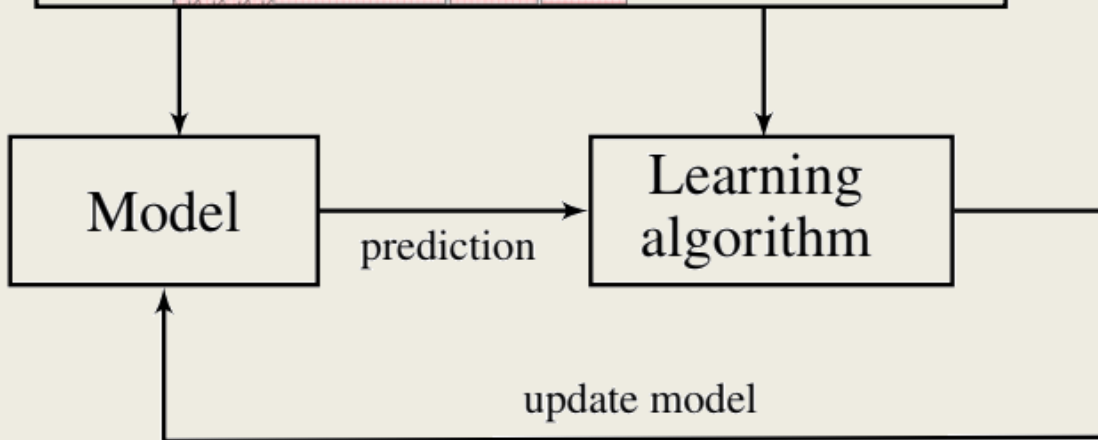


Right bundle branch block

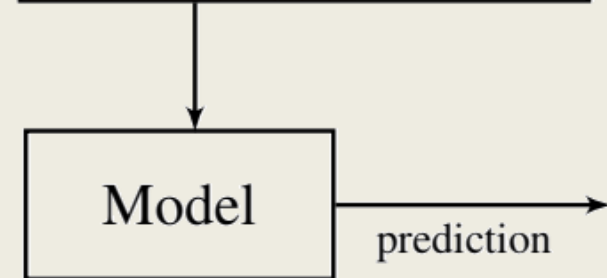
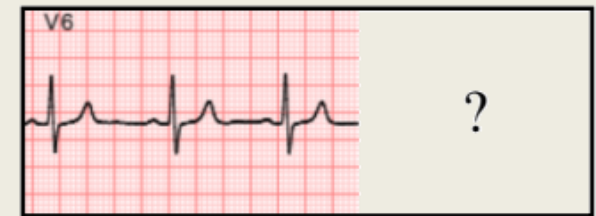


Classification: Heart Abnormalities

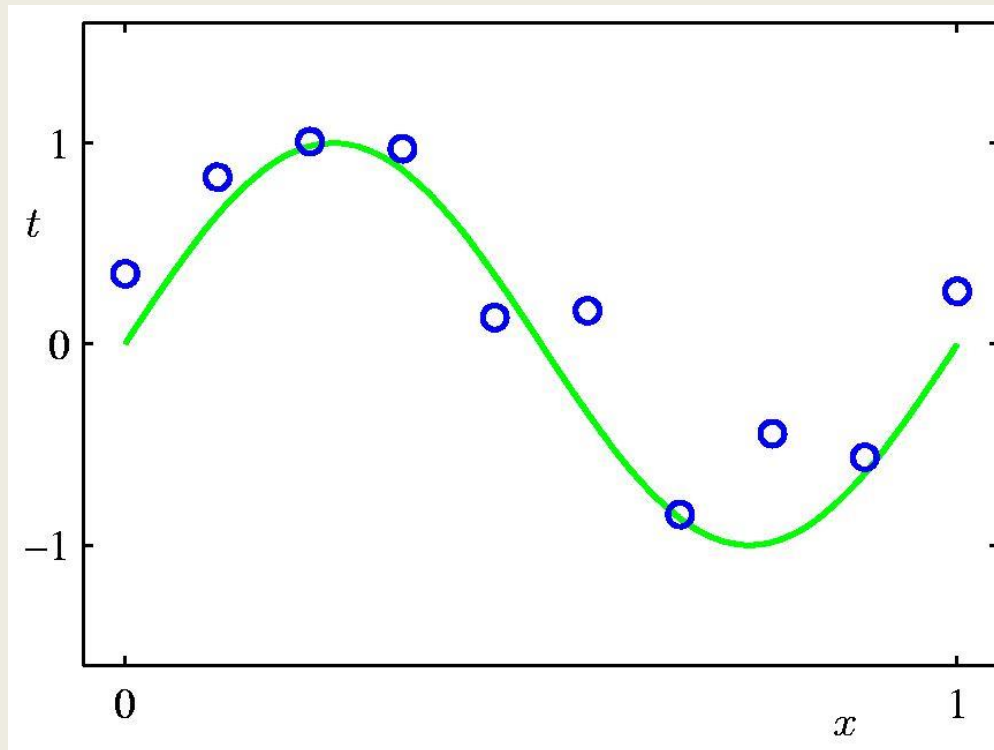
Training data



Unseen data



Regression: Polynomial Curve Fitting



$$t = \sin(2\pi x)$$

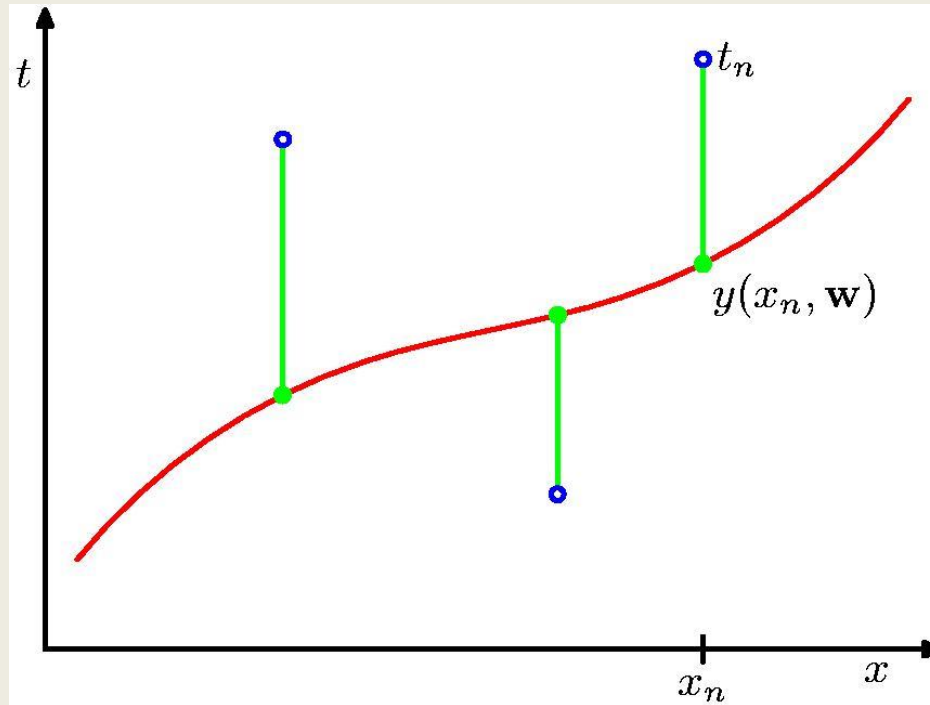
training data:
10
noisy samples
(Gaussian noise
added to targets)

model: polynomial of degree M

$$y(x, \mathbf{w}) = w_0 + w_1x + w_2x^2 + \dots + w_Mx^M = \sum_{j=0}^M w_jx^j$$



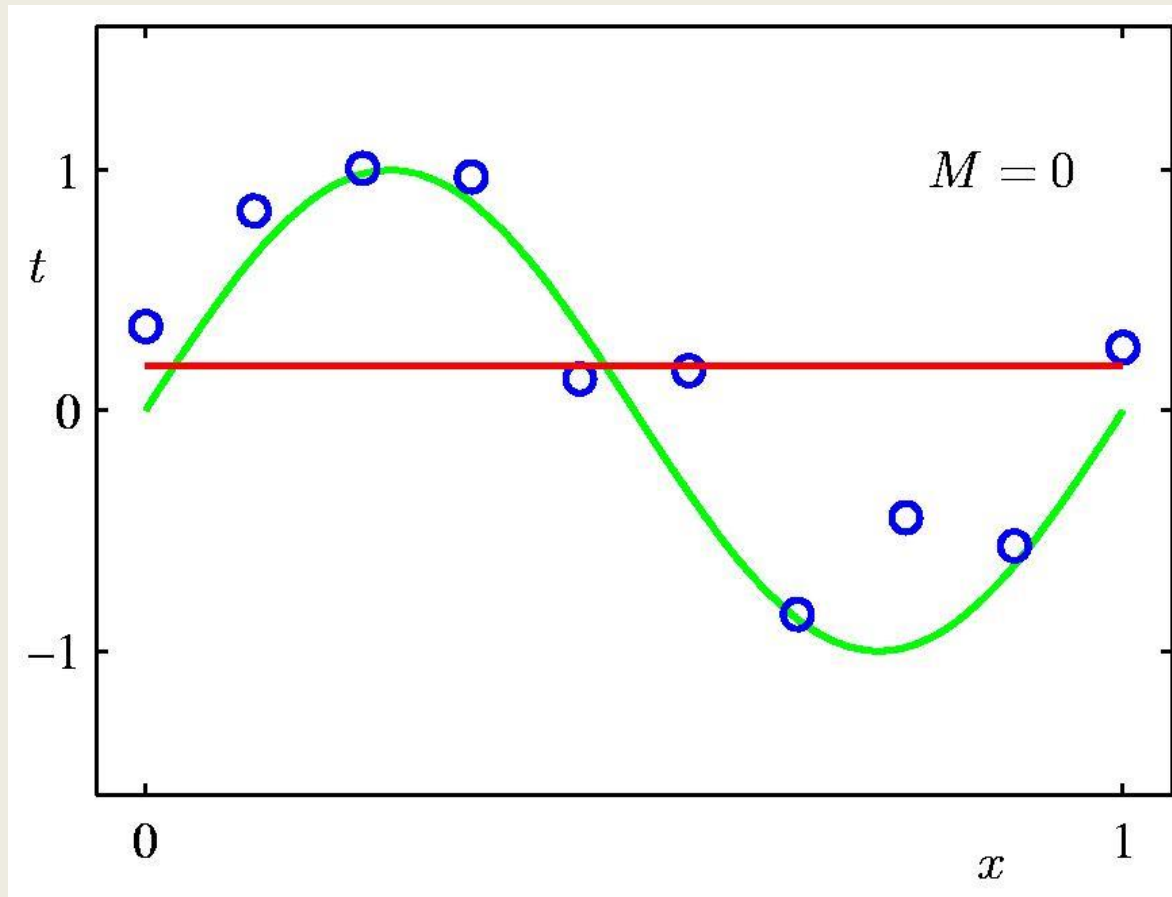
Sum-of-Squares Error Function



$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \{y(x_n, \mathbf{w}) - t_n\}^2$$



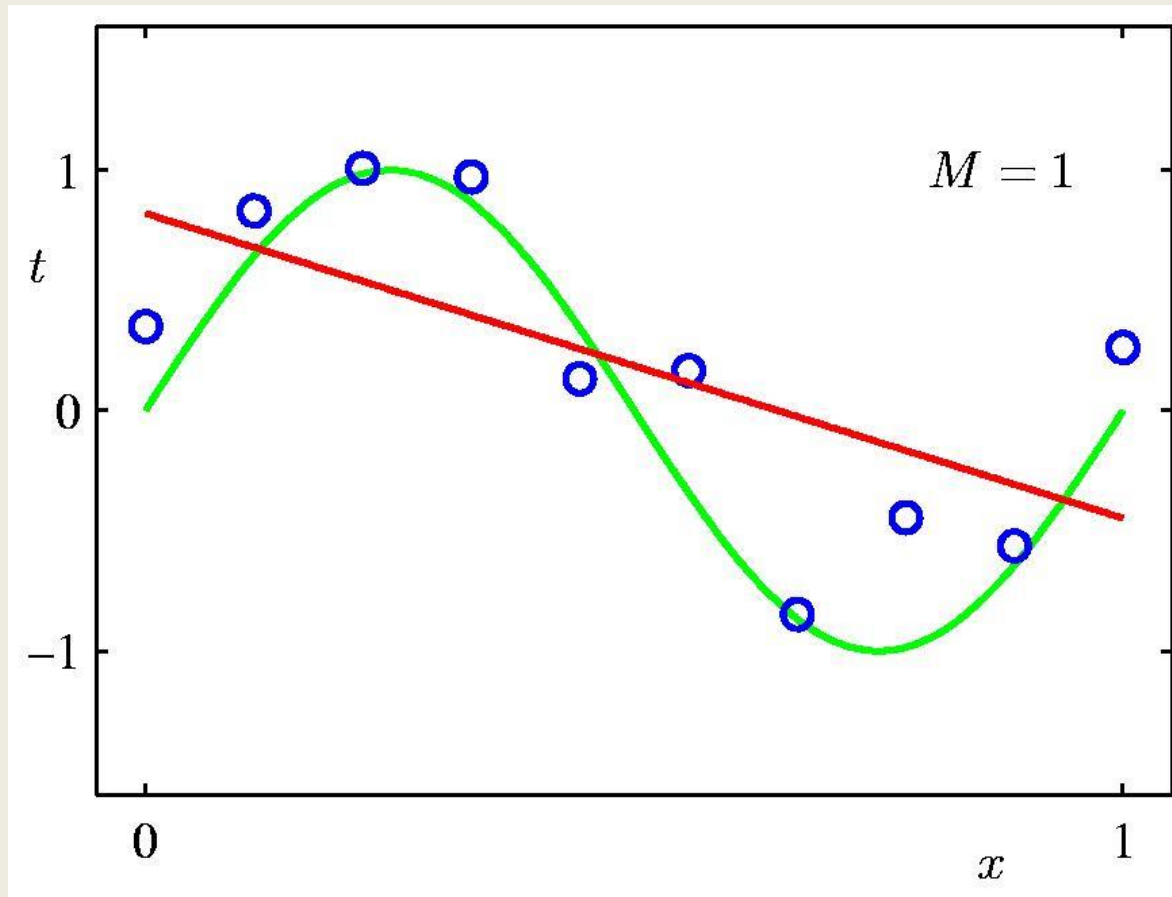
Model: 0th Order Polynomial



$$y(x, \mathbf{w}) = w_0 = \sum_{j=0}^0 w_j x^j$$



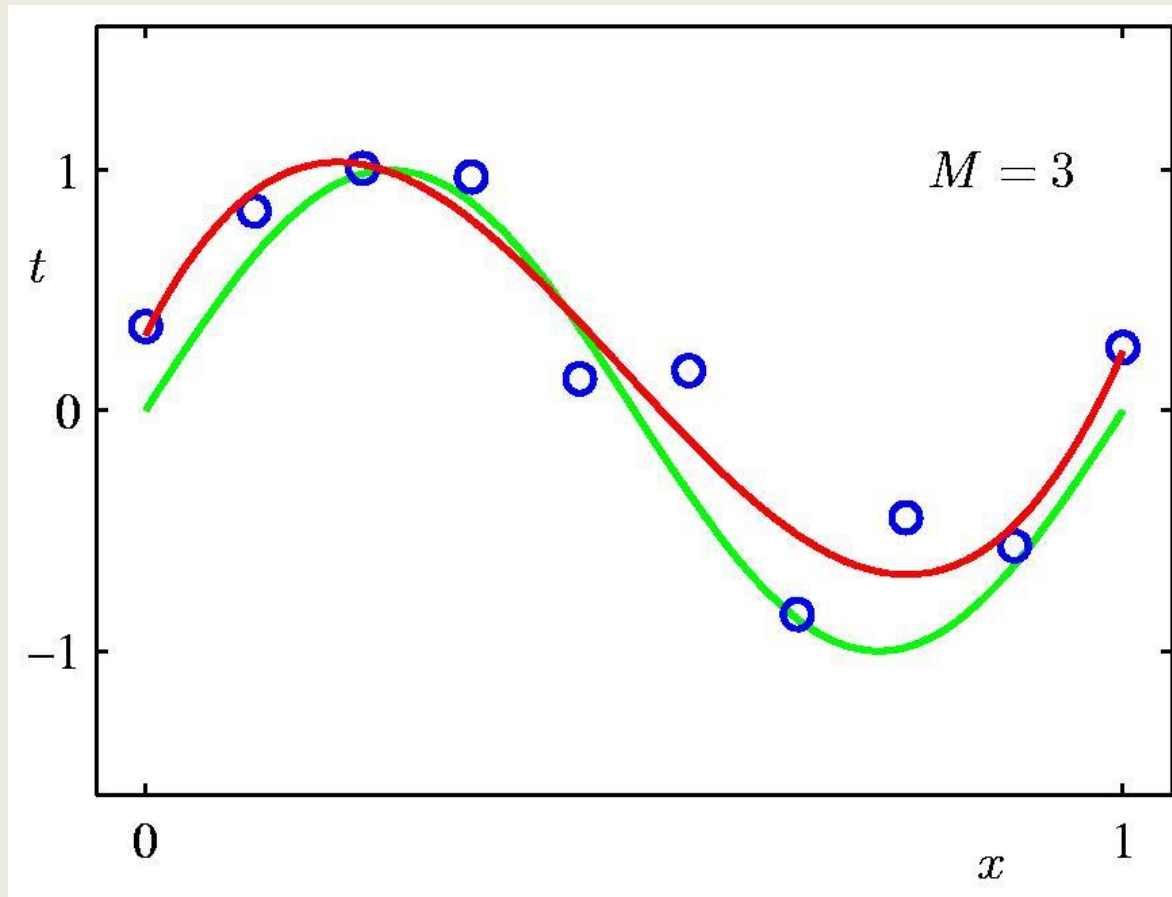
Model: 1st Order Polynomial



$$y(x, \mathbf{w}) = w_0 + w_1 x = \sum_{j=0}^1 w_j x^j$$



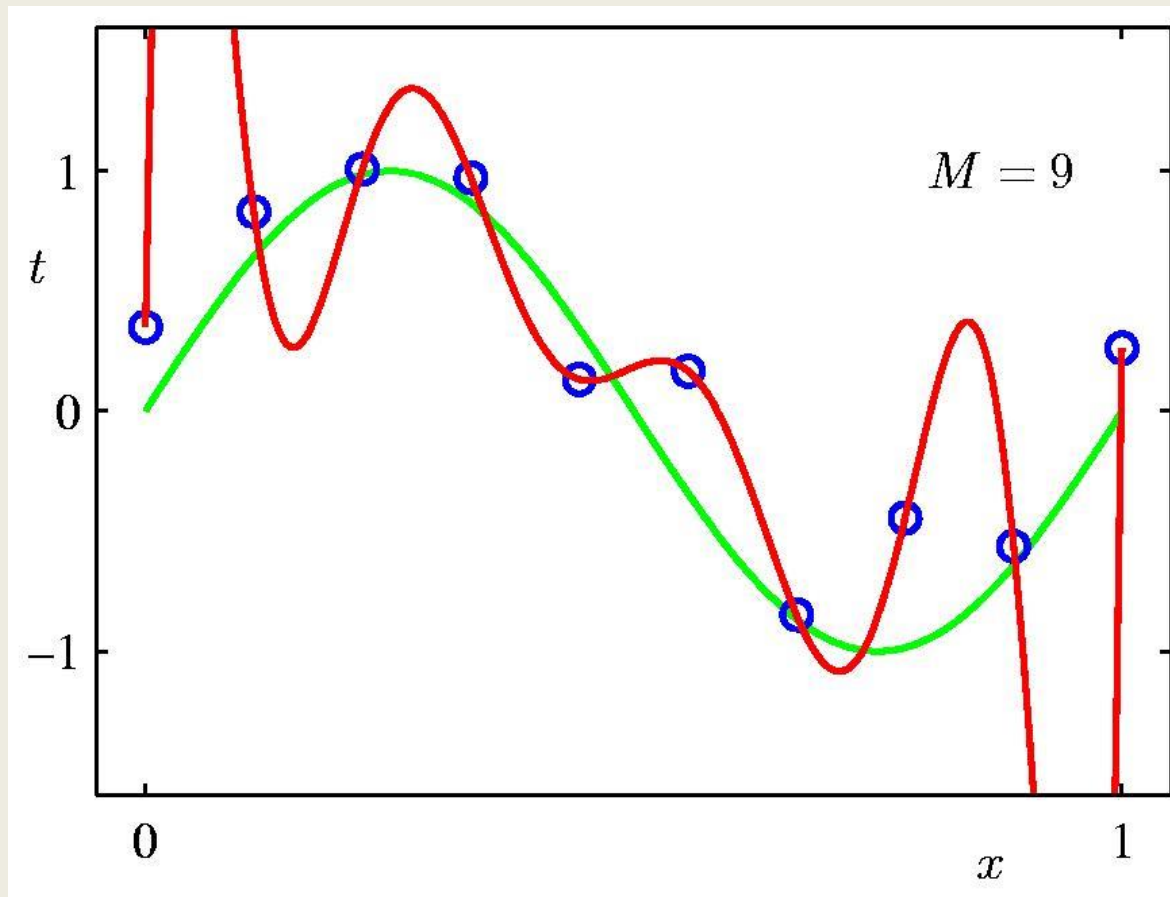
Model: 3rd Order Polynomial



$$y(x, \mathbf{w}) = w_0 + w_1x + w_2x^2 + w_3x^3 = \sum_{j=0}^3 w_j x^j$$



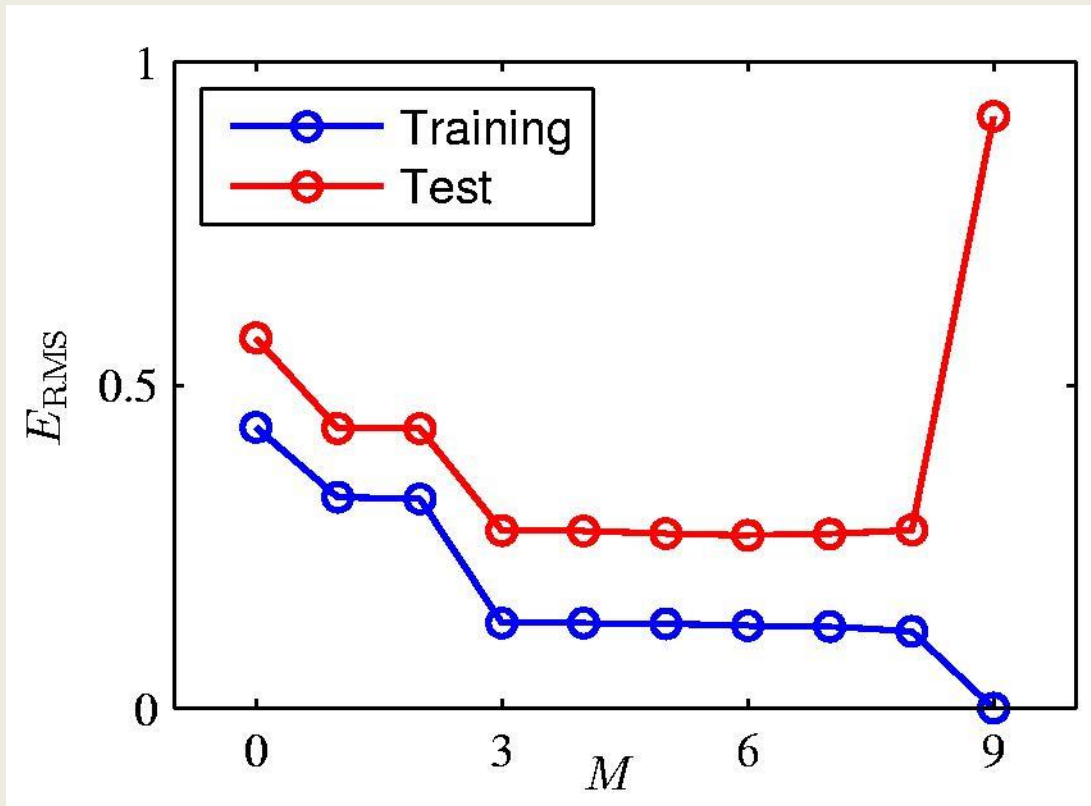
Model: 9th Order Polynomial



$$y(x, \mathbf{w}) = w_0 + w_1x + w_2x^2 + \dots + w_9x^9 = \sum_{j=0}^9 w_j x^j$$



Problem: Over-Fitting



|training| = 10 examples

|test|=100 examples

Root-Mean-Square (RMS) Error: $E_{\text{RMS}} = \sqrt{2E(\mathbf{w}^*)/N}$

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \{y(x_n, \mathbf{w}) - t_n\}^2$$



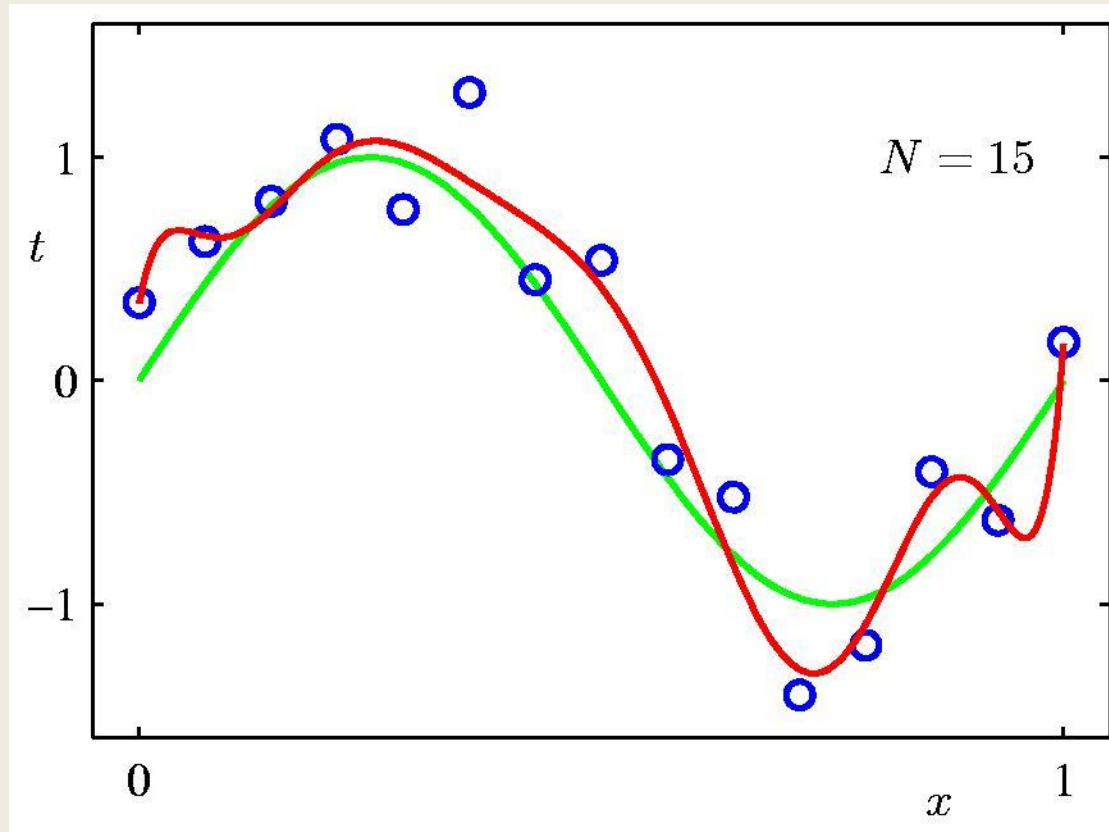
Learned Values of Polynomial Coefficients

	$M = 0$	$M = 1$	$M = 3$	$M = 9$
w_0^*	0.19	0.82	0.31	0.35
w_1^*		-1.27	7.99	232.37
w_2^*			-25.43	-5321.83
w_3^*			17.37	48568.31
w_4^*				-231639.30
w_5^*				640042.26
w_6^*				-1061800.52
w_7^*				1042400.18
w_8^*				-557682.99
w_9^*				125201.43



Increasing the Data Set Size: $N=15$

9th Order Polynomial

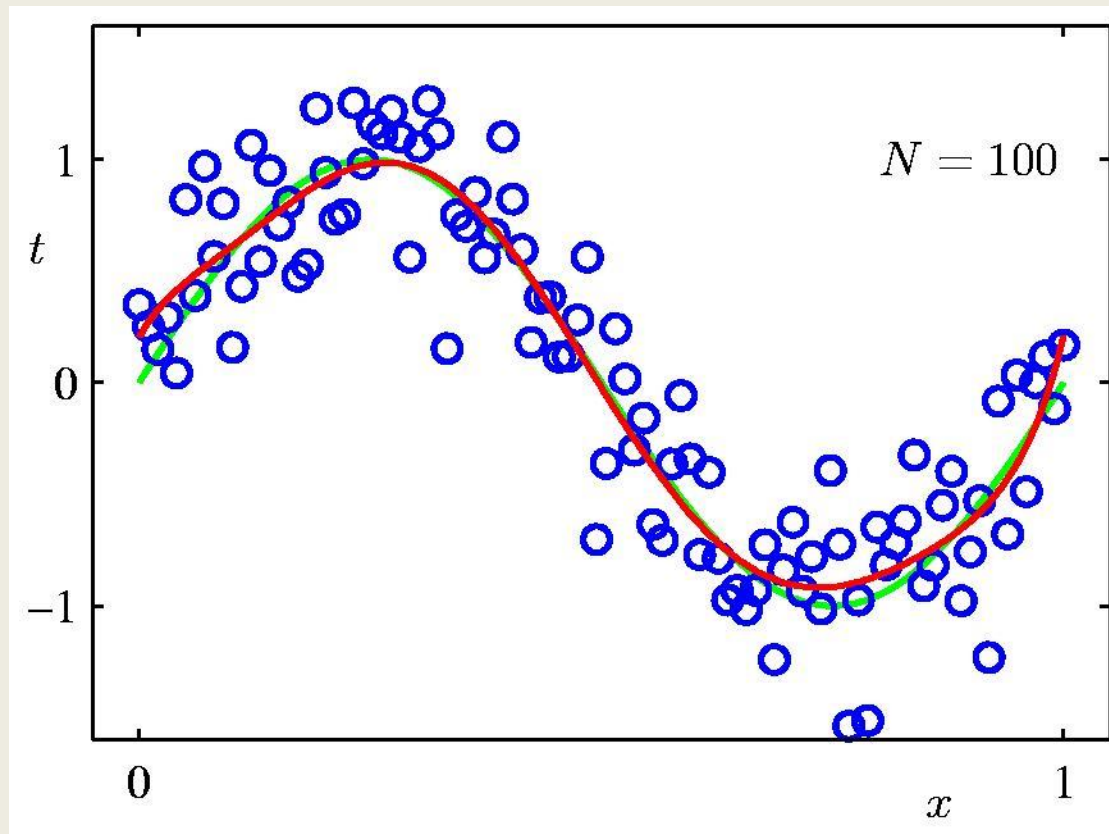


$$y(x, \mathbf{w}) = w_0 + w_1x + w_2x^2 + \dots + w_9x^9 = \sum_{j=0}^9 w_j x^j$$



Increasing the Data Set Size: $N=100$

9th Order Polynomial



$$y(x, \mathbf{w}) = w_0 + w_1x + w_2x^2 + \dots + w_9x^9 = \sum_{j=0}^9 w_j x^j$$



Real-Time Image Segmentation



Segmented Image (cars, signs, road, ...)

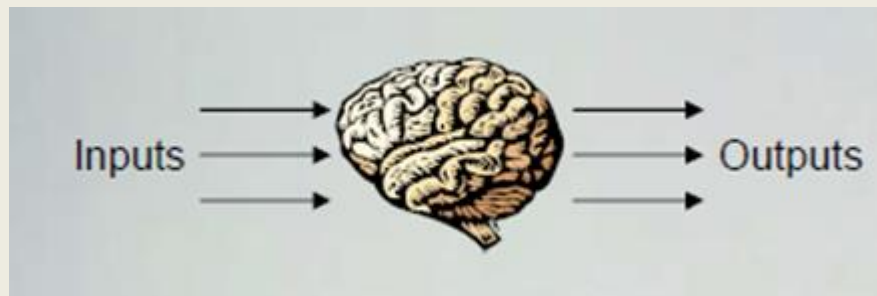


Artificial Neural Networks

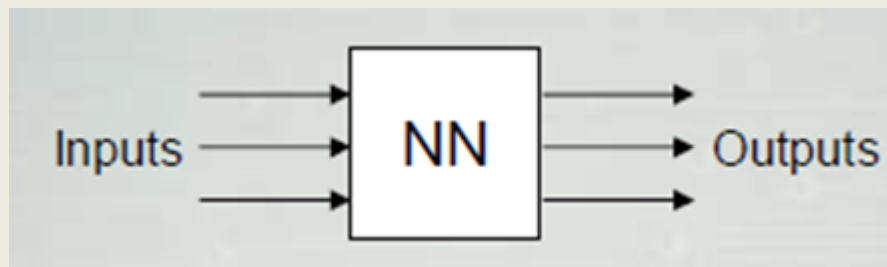


Artificial Neural Networks (ANNs)

- ANNs are a simplified model of human brain

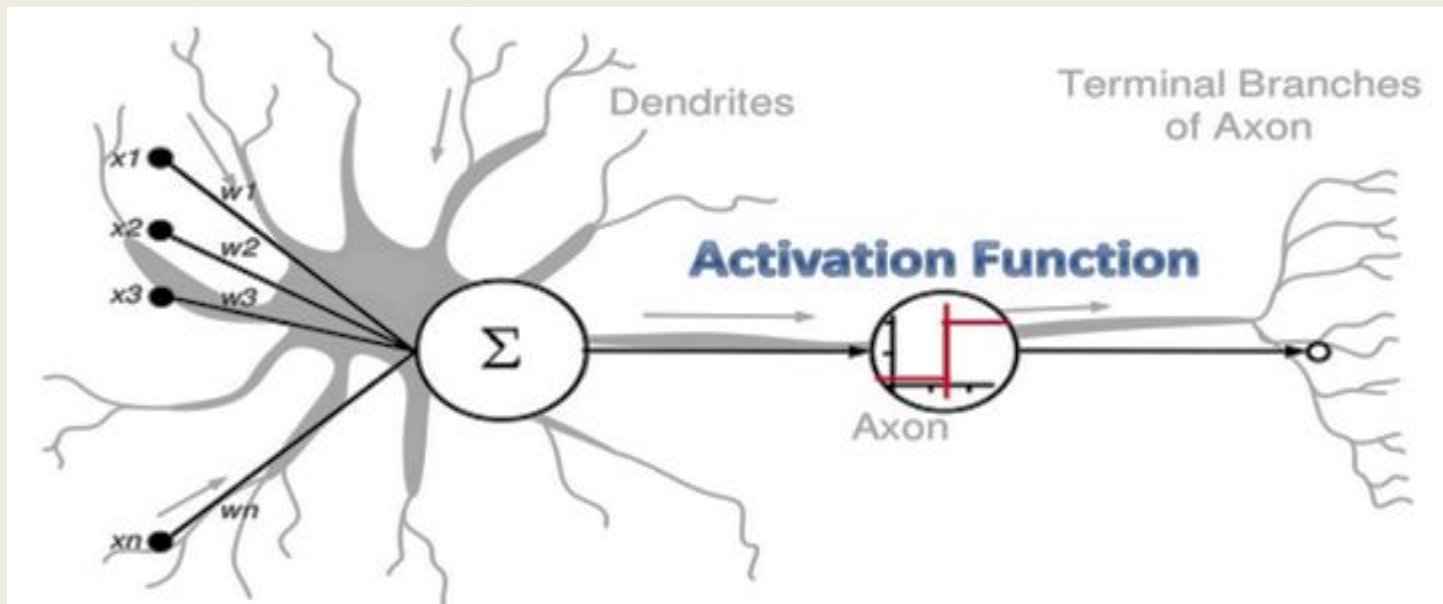


- A function which transforms input data to output data based on training over many correct examples



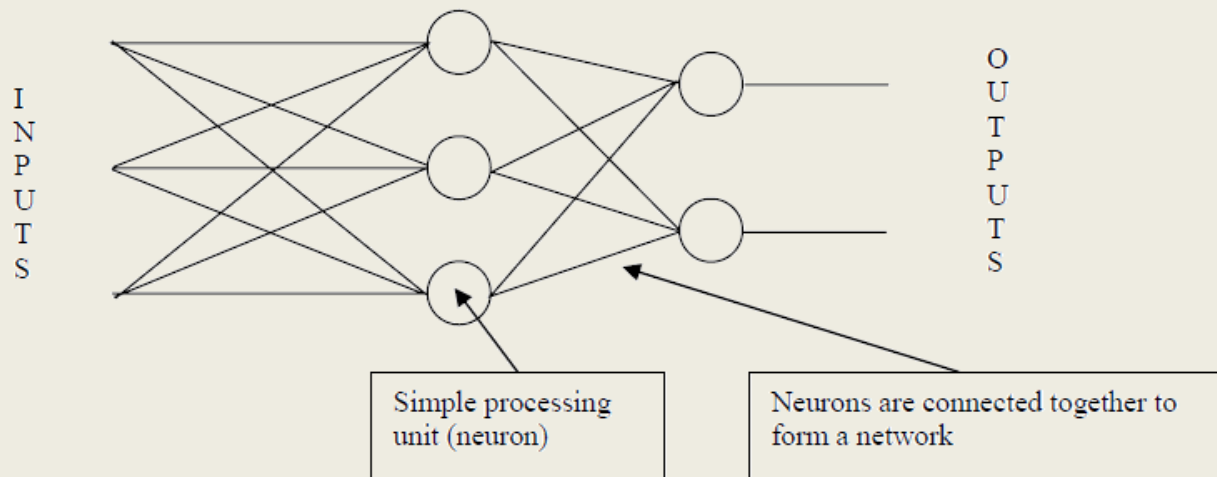
From Biology to ANNs

- Each neuron has 10^4 synapses, each with a synaptic weight
- Weights between neurons can be positive or negative
- Input: sum of the weighted input signals (linear combination)
- Depending on the total input signal the neuron is activated (stimulated) or not (inhibited): output activation function

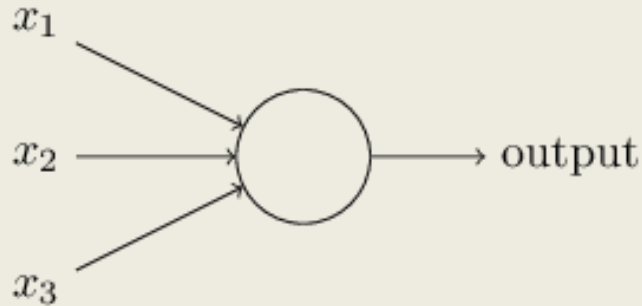


From Biology to ANNs ++

- In 1943, Warren McCulloch and Walter Pitts suggested the computational model of a neuron.
- Network: collection of connected processing units (neurons)
- Each processing unit is able to execute a simple and specific mathematical operation.
- Although individual neurons are simple, the entire neural network exhibits great processing potential.
- Flexible, non-linear systems, which exhibit “artificial intelligence”



Perceptrons



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

“bias” parameter $b \equiv -\text{threshold}$
i.e., how likely is the neuron to “fire”

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \quad w \cdot x \equiv \sum_j w_j x_j$$

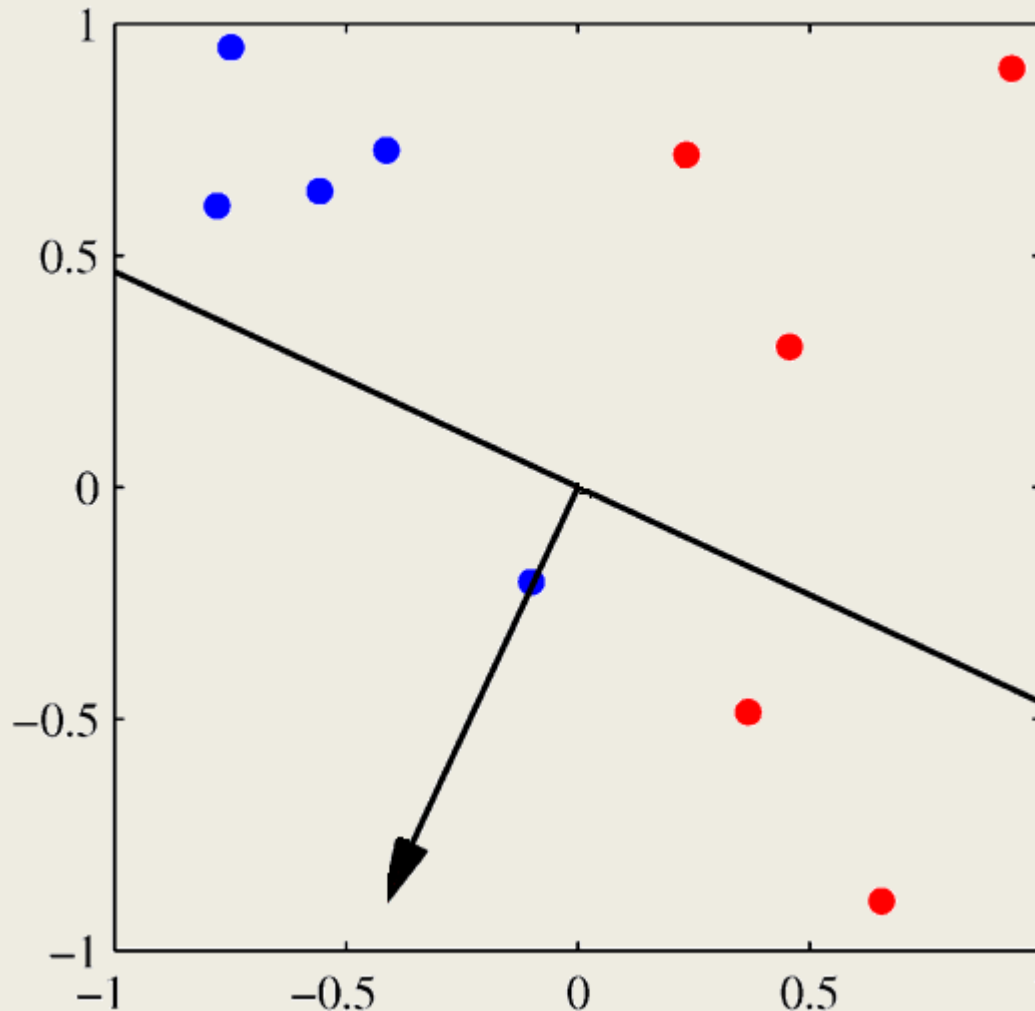
Perceptron learning

- iteratively go through the misclassified training data points
- update weights using the perceptron rule to reduce the error
- converges to a clear decision boundary, if data are separable



Perceptron Iterative Learning Example (0)

black arrow
(weight vector)
indicates the
classification
region of the
red class; the
other is **blue**

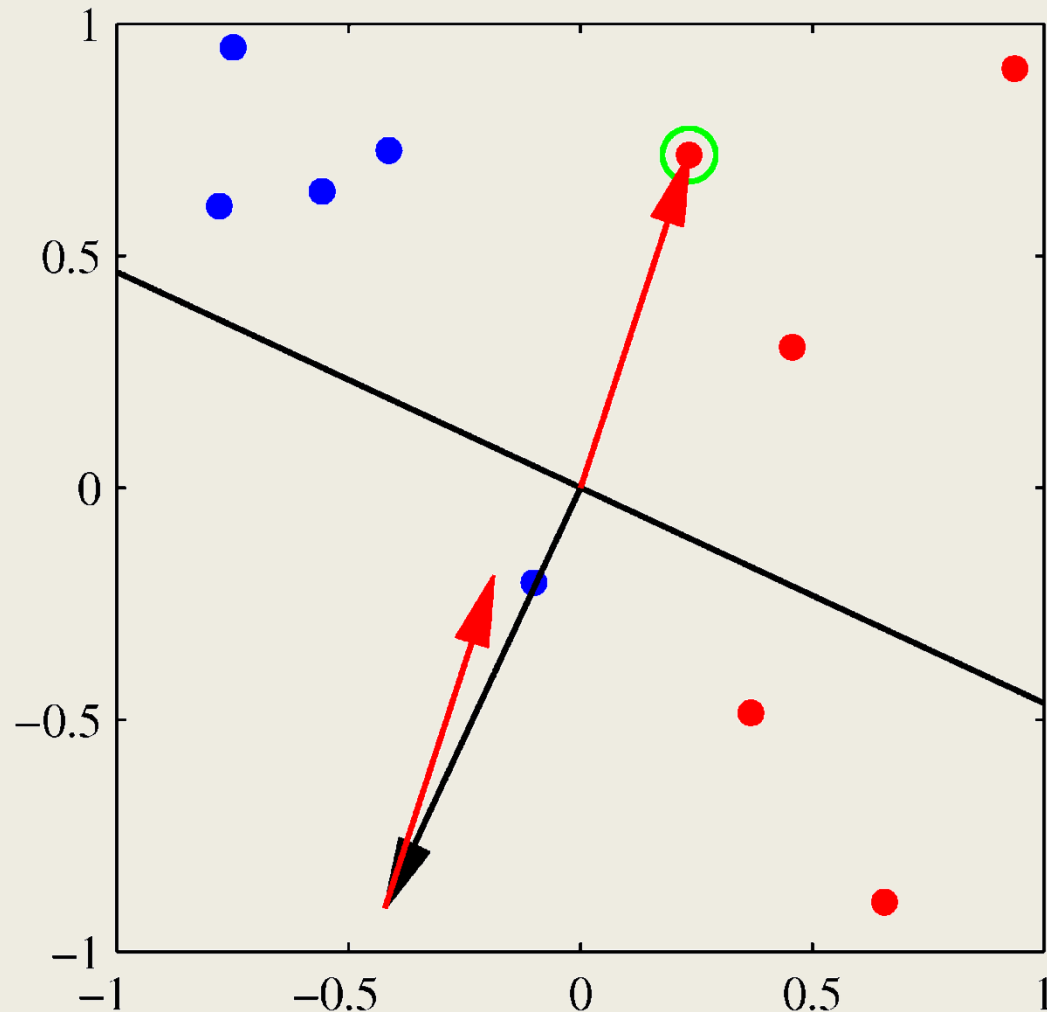


4 misclassified
data points
(3 red, 1 blue)



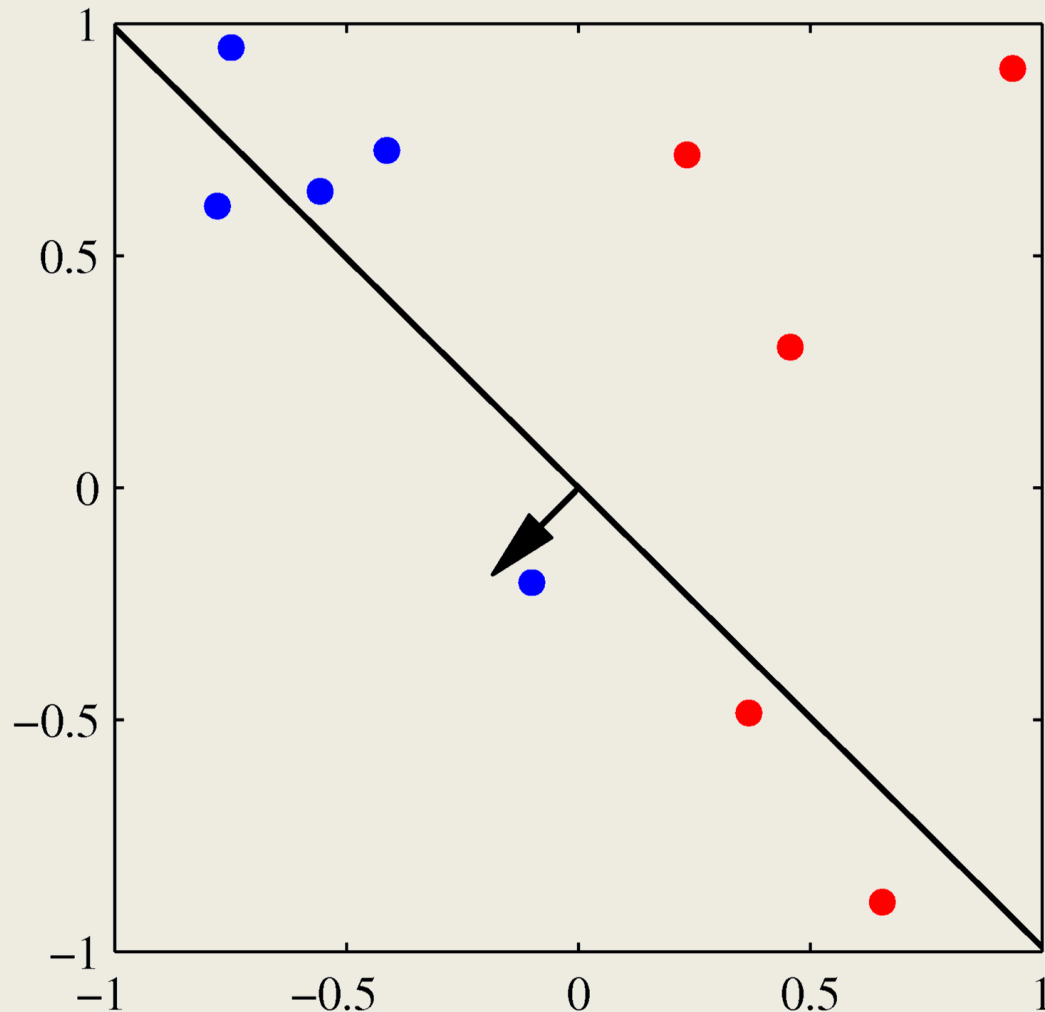
Perceptron Iterative Learning Example (1)

pick some
misclassified
data point,
compute the
error vector
(red arrow),
update the
weight vector



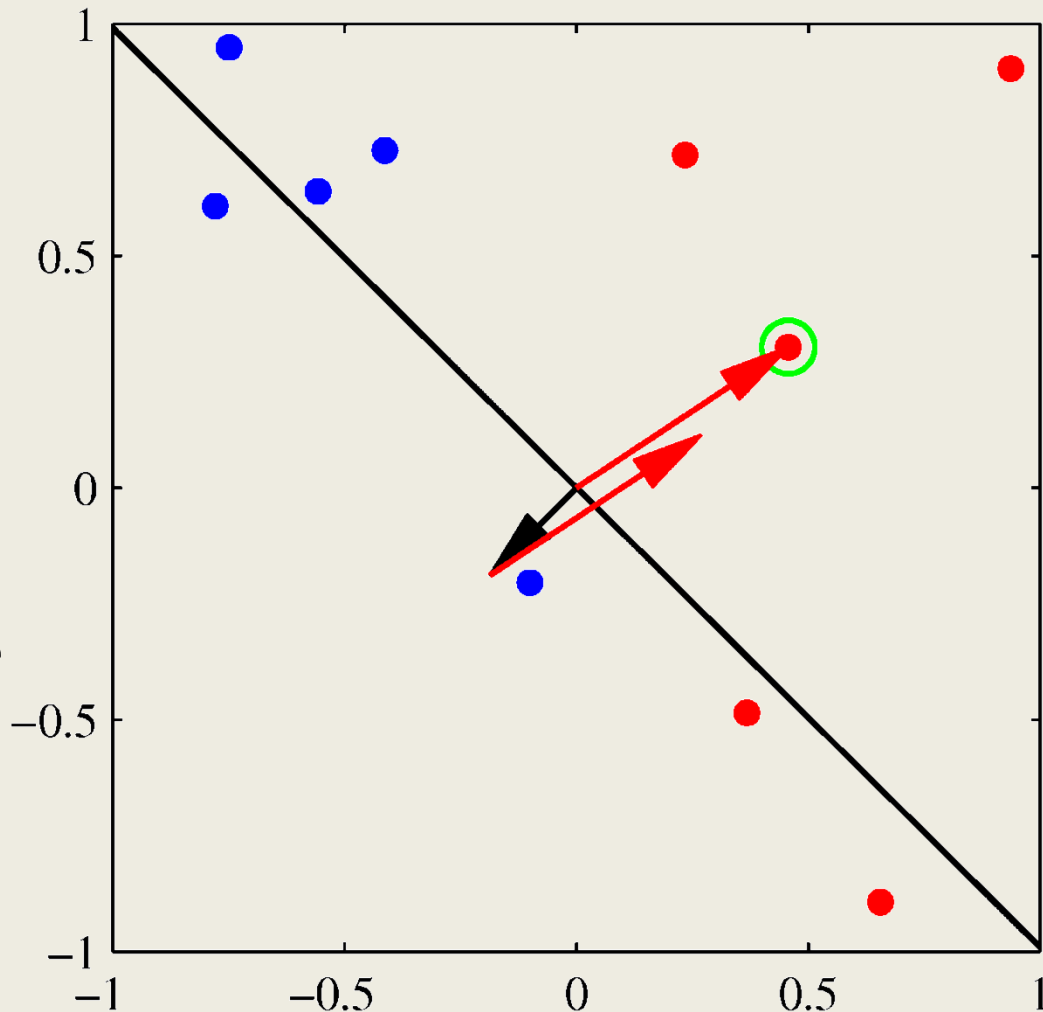
Perceptron Iterative Learning Example (2)

black arrow
(weight vector)
indicates the
classification
region of the
red class; the
other is **blue**

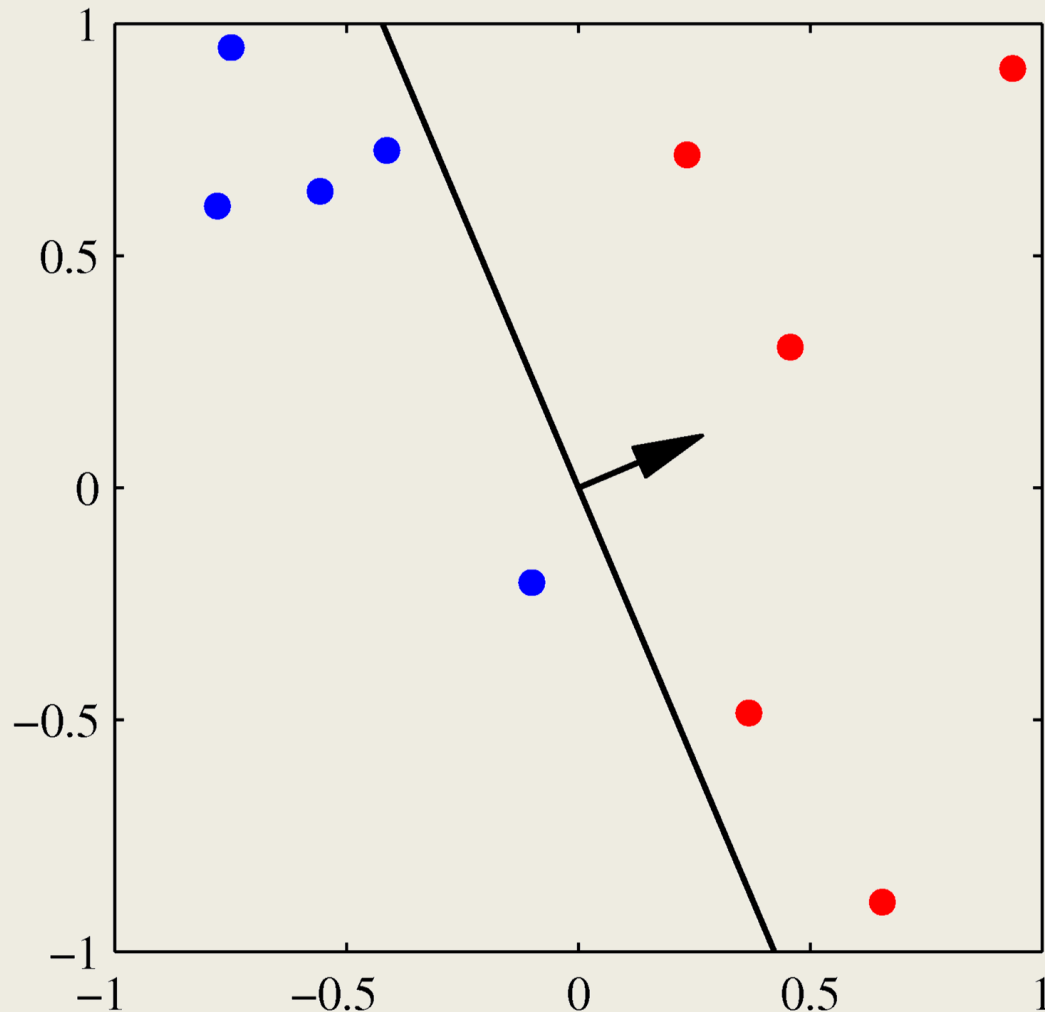


Perceptron Iterative Learning Example (3)

pick another
misclassified
data point,
compute the
error vector,
and update the
weight vector



Perceptron Iterative Learning Example (4)



success!!!

terminate the
iterative
learning
procedure

no
misclassified
data points

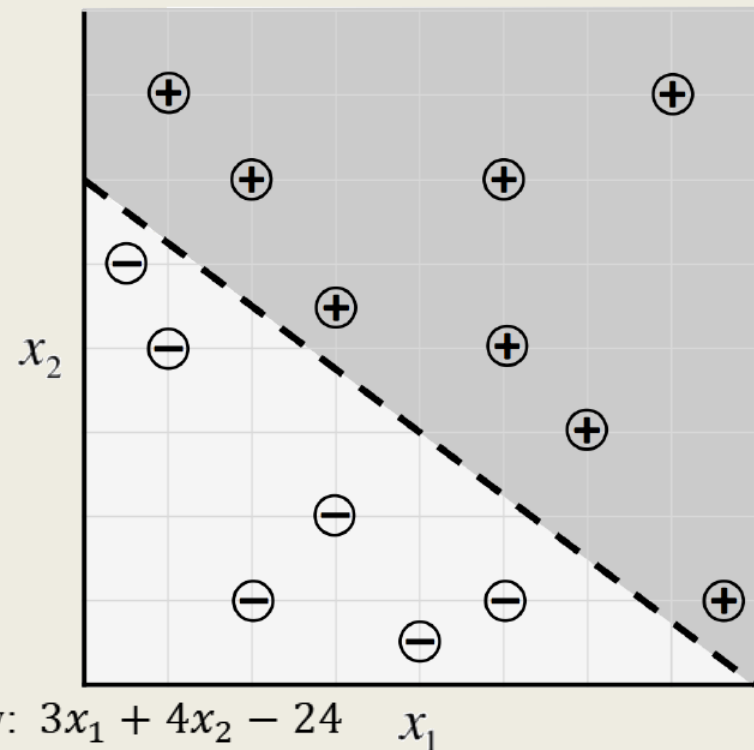
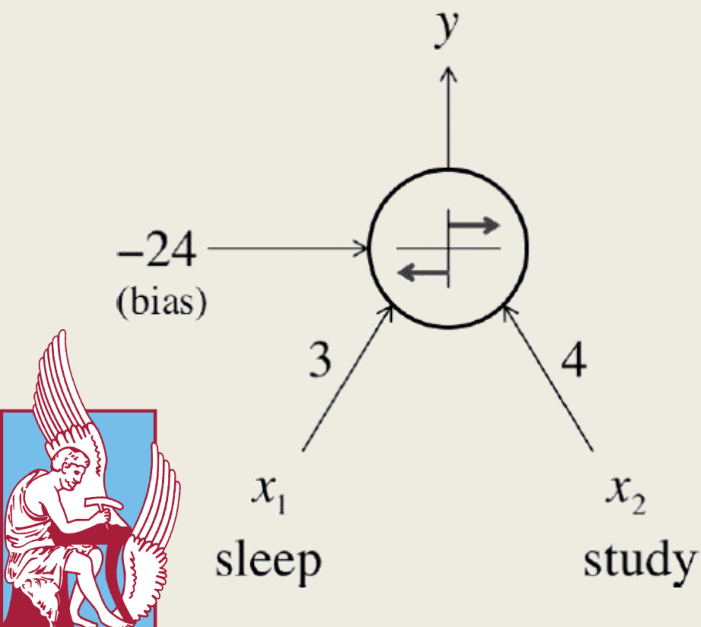


Linear Perceptron Example: Exam Performance

- Predict exam performance (above-below average) based on $\mathbf{x} = [x_1, x_2]$
 - x_1 = number of hours of sleep
 - x_2 = number of hours of study
- Define the linear classifier with parameters $\boldsymbol{\theta}$:

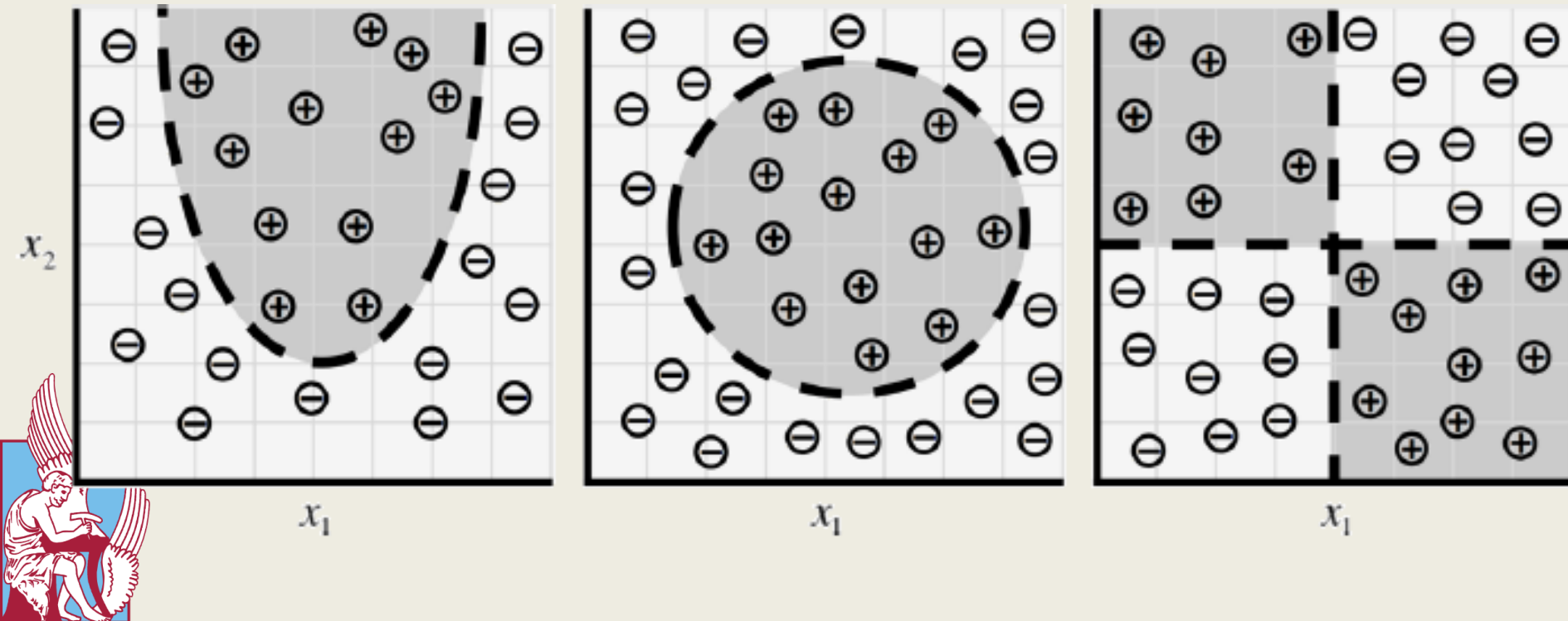
$$h(\mathbf{x}, \boldsymbol{\theta}) = \begin{cases} -1, & \mathbf{x}^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 < 0 \\ 1, & \mathbf{x}^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 \geq 0 \end{cases}$$

Possible solution: $\boldsymbol{\theta} = [-24, 3, 4]^T$

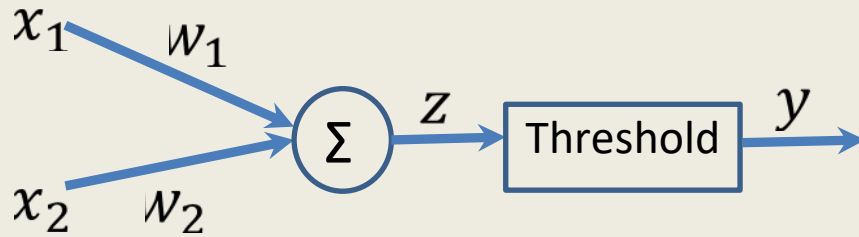


Linear Perceptron Limitations

- perceptrons cannot handle non-separable data
- perceptrons can handle only binary classification problems
- perceptrons can only learn linear decision boundaries
- many data distributions cannot be separated linearly



Neuron's Resilience to Noise



Threshold neuron
with weights (0,1)
and threshold 0.5

x_1	x_2	Z	Output
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	1

x_1	x_2	Z	Output
0	0	0	0
0	1	0.8	1
1	0	0.2	0
1	1	1	1

Threshold neuron with
noisy weights (0.2,0.8)
and a threshold of 0.5

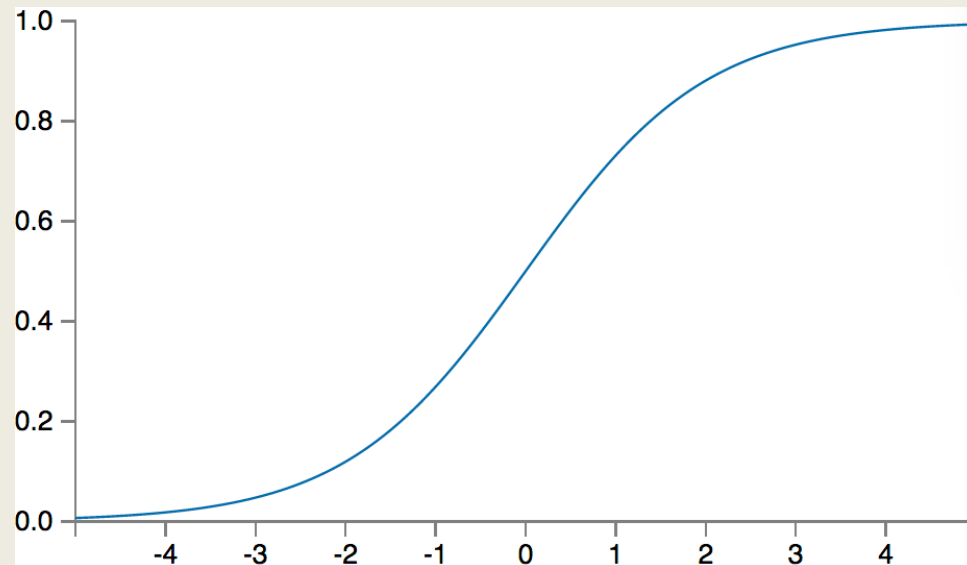
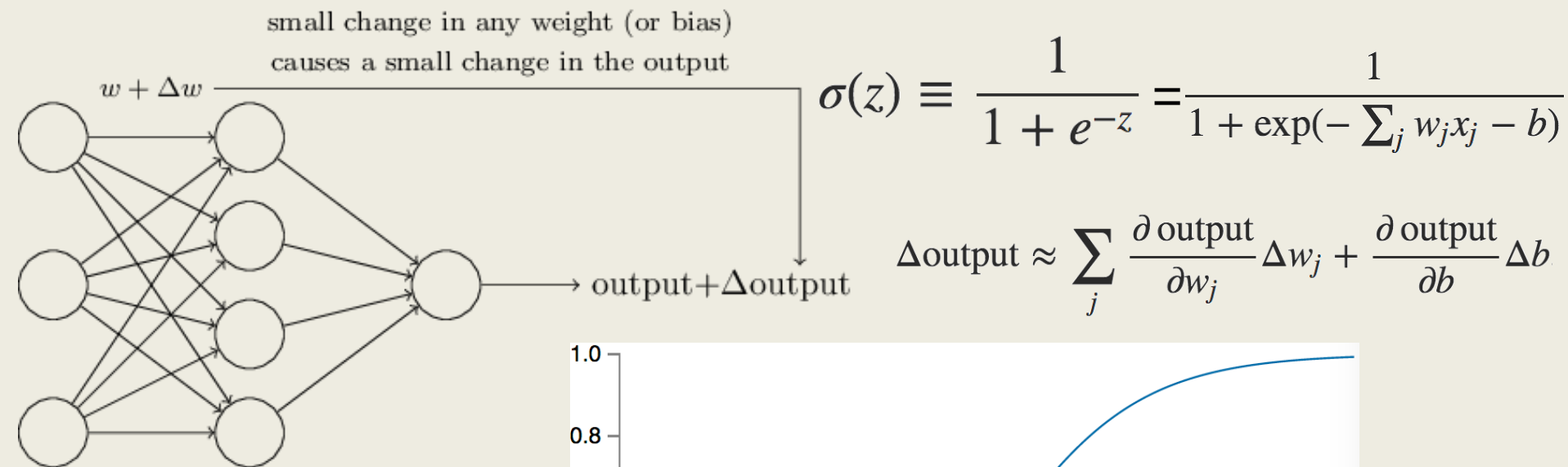
x_1	x_2	Z	Output
0.2	0.2	0.2	0
0.2	0.8	0.8	1
0.8	0.2	0.2	0
0.8	0.8	0.8	1

Threshold neuron with weights (0,1) and a
threshold of 0.5 when the input signal is noisy



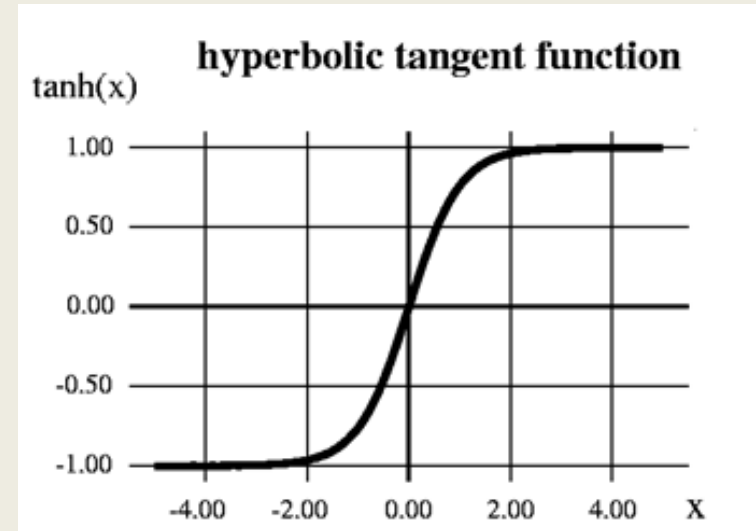
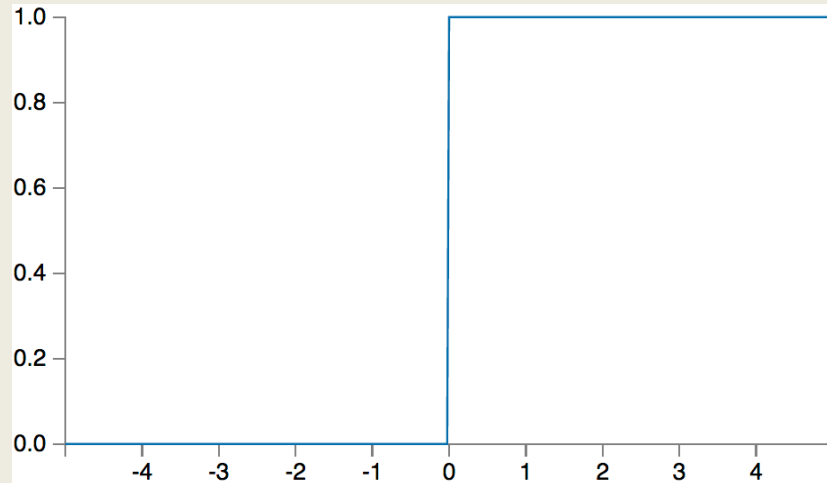
Sigmoid Activation Function

- Use *logistic* or other *sigmoid functions* for smooth behavior!



More Non-Linear Activation Functions

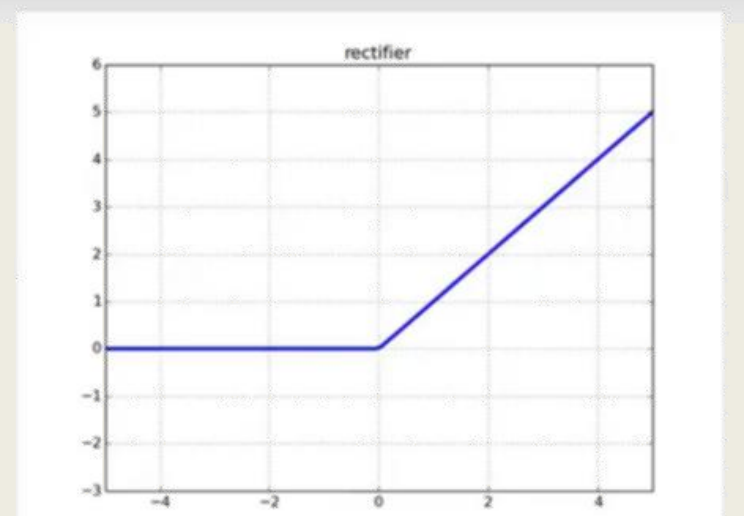
- Step/Threshold function



- Rectified Linear Unit (ReLU)

ReLU

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$



SoftMax Neurons

- Activation Function:

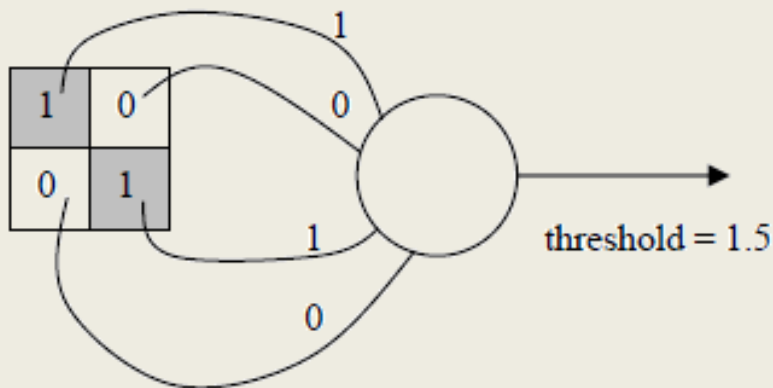
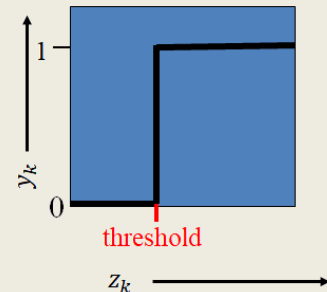
$$y_l(k) = \frac{e^{z_l(k)}}{\sum_{i=1}^{d_l} e^{z_l(i)}}$$

- d_l number of neurons
- Usually used at the neurons of the output layer
- Converts a vector of real numbers to a vector of posterior probabilities
 - Output values are between 0 and 1 and sum to unit
 - The output of a neuron depends the outputs of all other neurons in its layer
 - It is a measure of uncertainty of the output
 - Strong prediction → single entry in the output vector close to 1
 - Weak prediction → multiple entries almost equally likely



Recognizing Simple Patterns

- Image with 4 grey scaled pixels (from 0=white to 1=black)
 - $z = (1 \times 1) + (0 \times 0) + (1 \times 1) + (0 \times 0) = 2$
 - Define a threshold value $\theta = 1.5$ to emulate the action potential
 - $z = 2 > 1.5 \Rightarrow \text{Output} = 1$
 - Even if some white pixels had values a bit more than 0 the neuron could recognize the pattern correctly



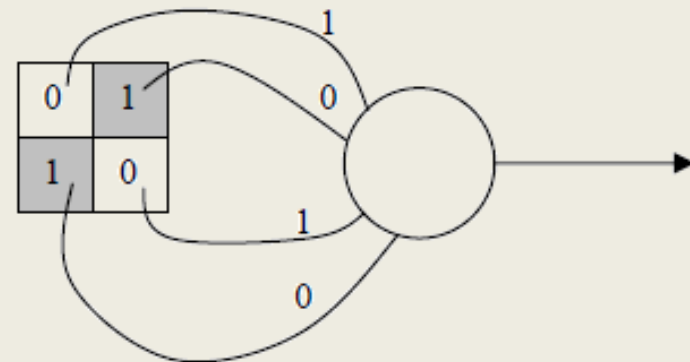
Output = 1 if pattern
is recognised

Output = 0 if pattern
is not recognised

- Rotate the same image (pattern):

- $z = (0 \times 1) + (1 \times 0) + (0 \times 1) + (1 \times 0) = 0 < 1.5$

- Fail to recognize



Recognizing Simple Patterns

- Can it recognize the noisy image? →

0.7	0.1
0.2	0.9

– $z = (0.7 \times 1) + (0.9 \times 1) = 1.6 > 1.5 \rightarrow$ marginally successful

- Use the logistic function as activation:

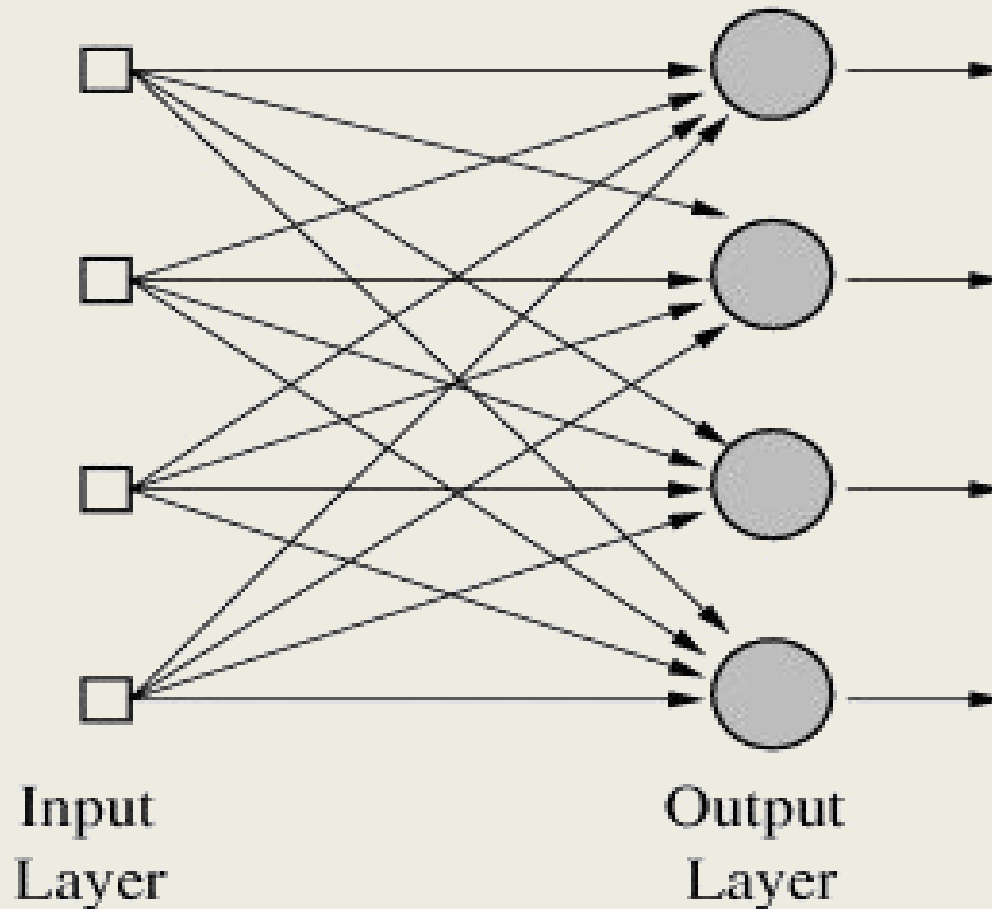
– $\text{output} = \frac{1}{1+e^{-1.6}} = 0.8320$ (close to max=1)

$$y_k = \frac{1}{1 + \exp(-z_k)}$$

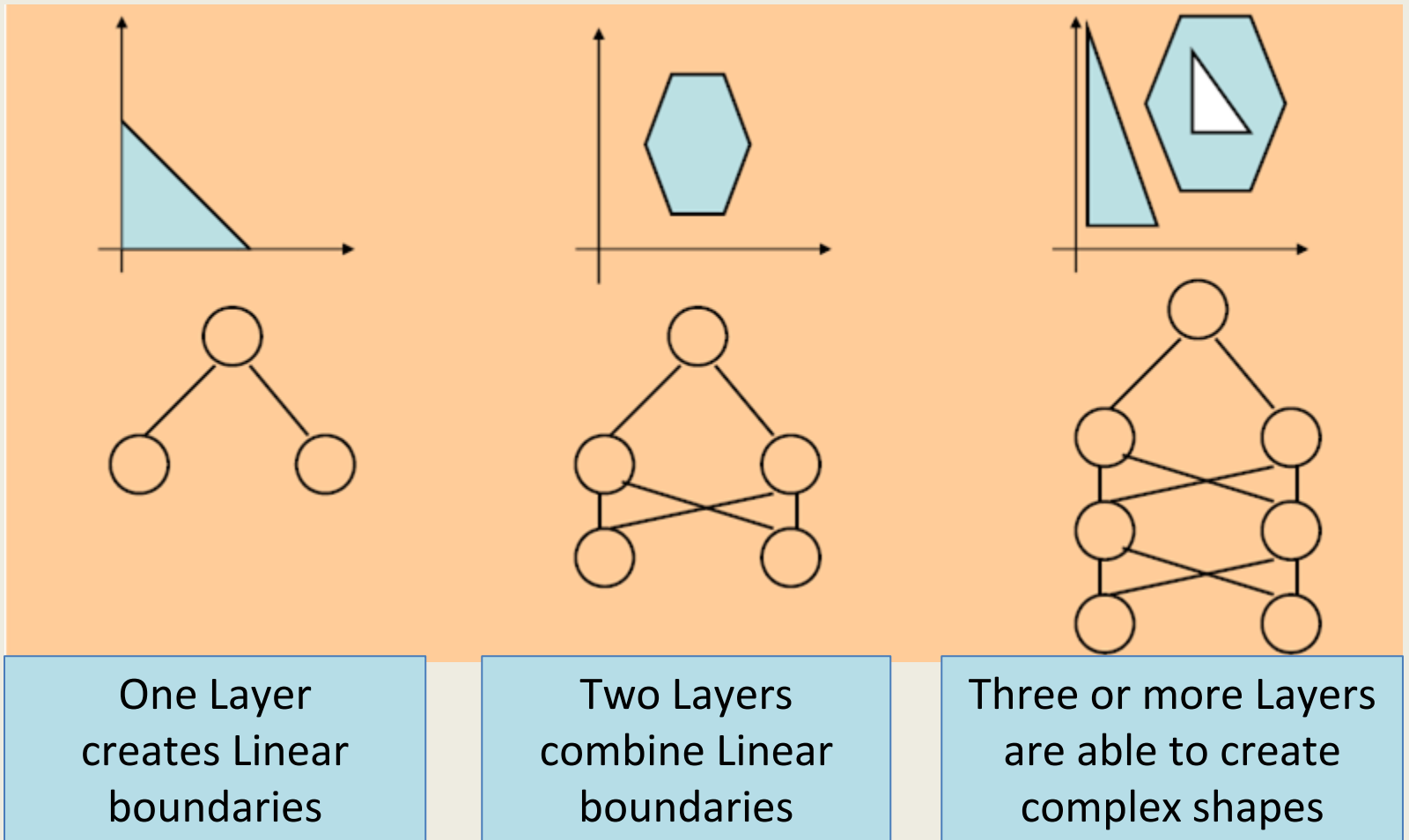


Neural Network Structure

- Neurons are organized in layers
 - simplest form: 1 Input layer and 1 output layer

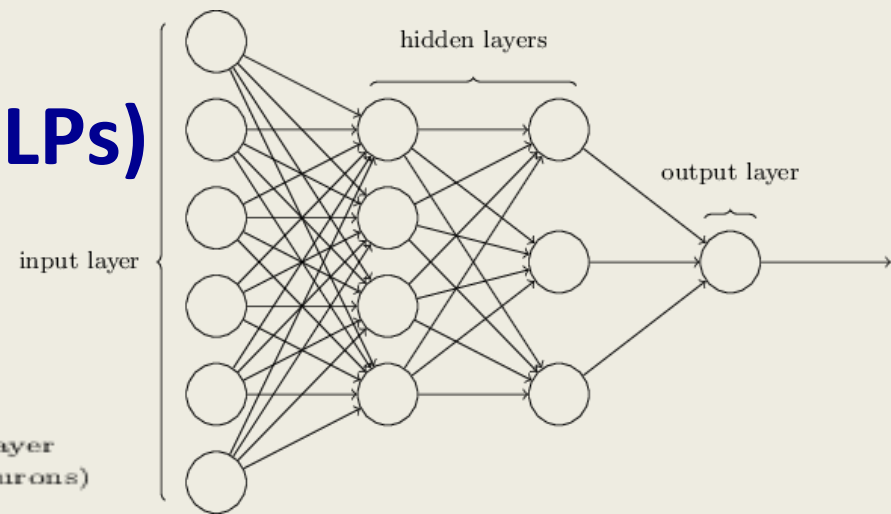


Why do we need hidden layers?

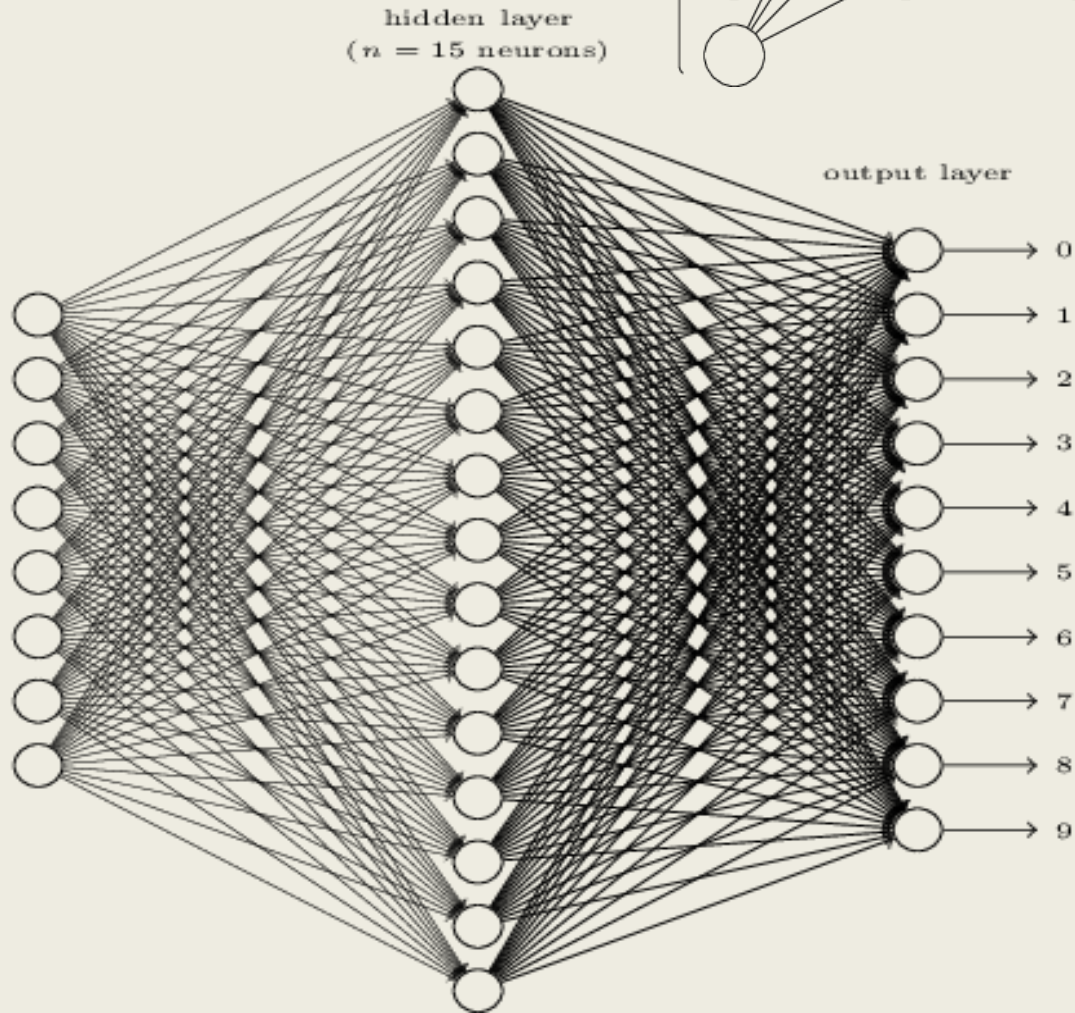


Multi-Layer Perceptrons (MLPs)

- Terminology confusing! Neurons in MLPs use the sigmoid function for their output.



input layer
(784 neurons)



Learning Process of Neural Networks

- The NN structure is determined first (inference model)
- how many layers? how many units per layer? activation functions?...

- **Learning process**

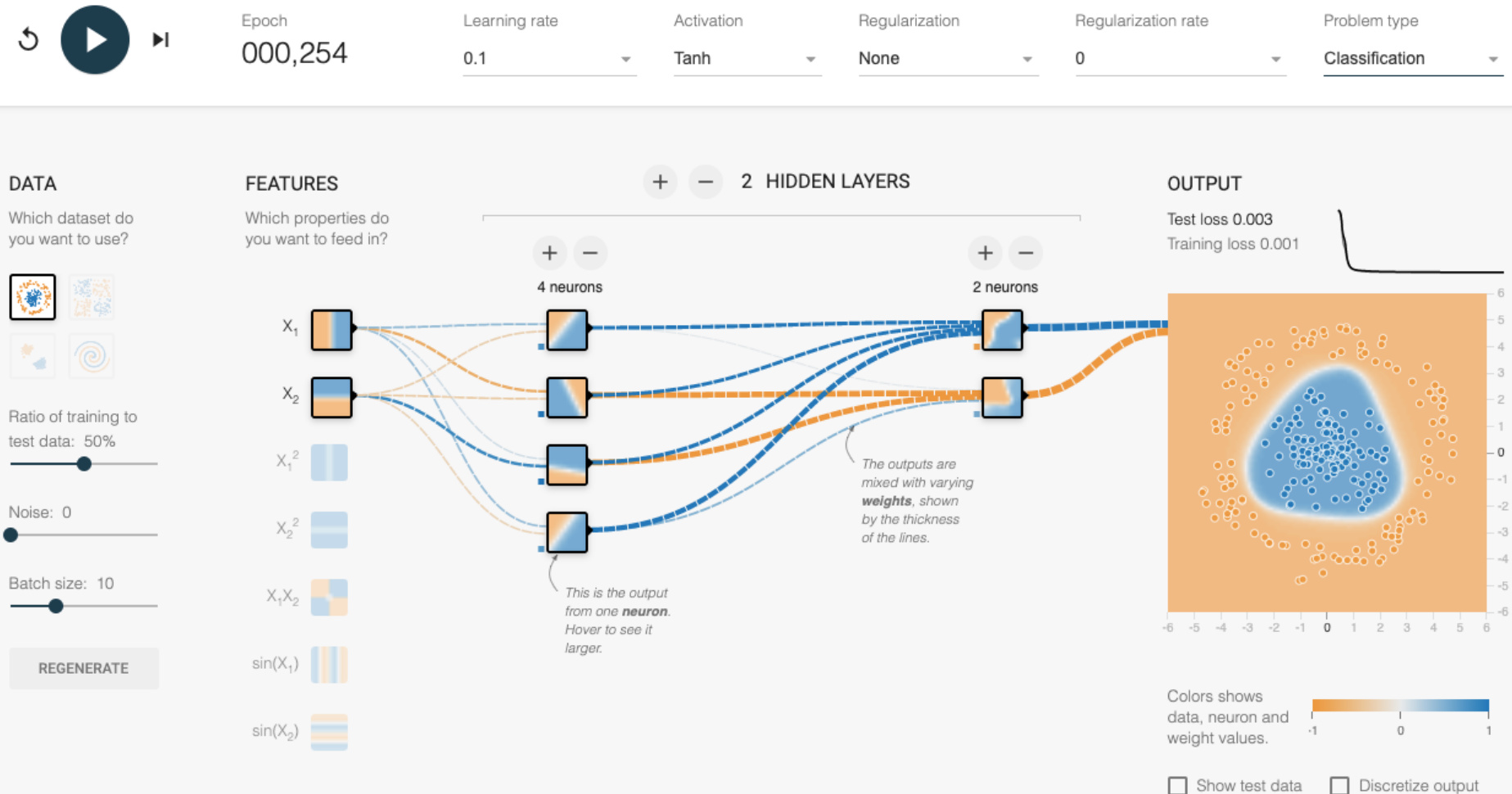
- The NN learns a specific task by adjusting its parameters (weights) from the correctly labeled training examples (supervised learning)

- **Adjusting the parameters**

1. randomly initialize the weights
2. for some input, obtain a predicted output from the network
3. if the output matches the real label, do not change the weights
4. if the output is larger than the real label,
adjust the weights that contribute to large output values
5. if the output is smaller than the real label,
adjust the weights that contribute to small output values
6. repeat the previous steps, until the error converges to a minimum



<http://playground.tensorflow.org>



Backpropagation



How to learn?

Use gradient descent on the cost function

The key idea

- adjust a model's weights in response to the error it produces, until you can't reduce the error any more

How do the errors vary, as the weights are adjusted?

- calculate how a change in *weight* affects a change in *Error*
- first, calculate how a change in *activation* affects a change in *Error*,
- and then, how a change in *weight* affects a change in *activation*

$$\frac{dError}{dweight} = \frac{dError}{dactivation} * \frac{dactivation}{dweight}$$



How to learn?

Use gradient descent on the cost function

- **use a smooth cost function** → helps figure out how to make small changes in weights and biases to get an improvement in the cost

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

- **perform stochastic gradient descent** to guarantee reduction of cost

$X_i \leftarrow$ training inputs as **a mini-batch**
(small set of m training inputs)

A training epoch

Train over

a randomly chosen mini-batch,
then on another,
until you exhaust all mini-batches

[repeat]

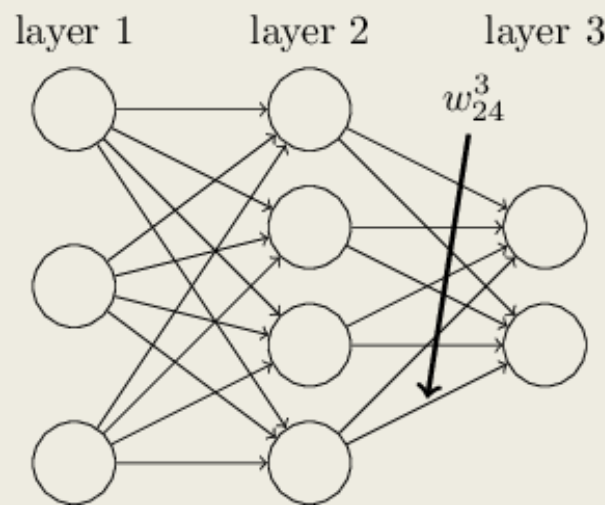
$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$



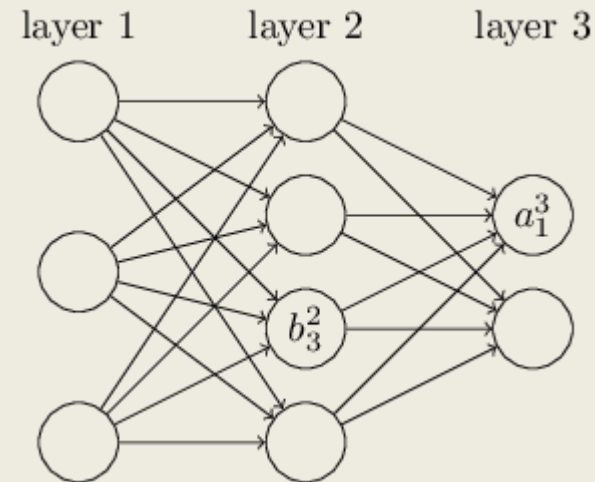
Computing the gradient of the cost function: The Backpropagation Algorithm

- But, how to compute the gradient of the cost function?
- Use the backpropagation algorithm!



w_{jk}^l is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right)$$



$$a^l = \sigma(w^l a^{l-1} + b^l)$$

how do activations in one layer relate to those in the previous layer? apply the weight matrix to the activations, add the bias vector, and apply σ

$$a^l = \sigma(z^l) \quad \text{with the weighted input } z^l \quad z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$



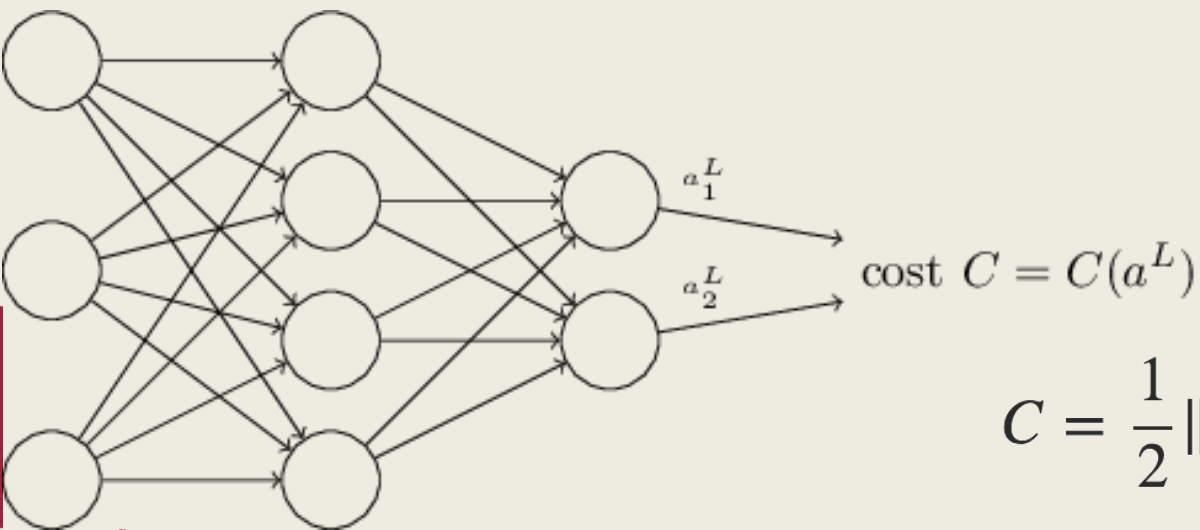
Let's define the cost function

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

L : number of layers; n : total number of training examples x ; $y(x)$: desired output;
 $a^L(x)$: vector of activations that are output from the network, when x is input

We take $C = \frac{1}{n} \sum_x C_x$ (required for averaging over all training examples)

We also take C to be a quadratic function of the output activations



$$C = \frac{1}{2} \|y - a^L\|^2 = \frac{1}{2} \sum_j (y_j - a_j^L)^2$$

Computing the gradient of the cost function

the error δ_j^l of neuron j in layer l $\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$

$$\frac{d\text{Error}}{d\text{weight}} = \frac{d\text{Error}}{d\text{activation}} * \frac{d\text{activation}}{d\text{weight}}$$

the error in the output layer

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

$s \odot t$ denotes the *elementwise* product of the two vectors

$$(s \odot t)_j = s_j t_j \quad \begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

For the quadratic cost function

$$C = \frac{1}{2} \sum_j (y_j - a_j^L)^2$$

we can easily compute the gradient with respect to the activation

$$\partial C / \partial a_j^L = (a_j^L - y_j)$$



Computing the gradient of the cost function

The equation for the error δ^l in terms of the error δ^{l+1} in the next layer

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

propagates the error backward (from layer $l+1$ to layer l),
first assessing it on l layer's output, then to l 's weighted input via σ .

The rate of change of the cost with respect to any **bias** in the network

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

The rate of change of the cost with respect to any **weight** in the network

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$



The Backpropagation Algorithm

1. **Input x :** Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \dots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.
3. **Output error δ^L :** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error:** For each $l = L - 1, L - 2, \dots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.



The Backpropagation Algorithm

In other words:

- We compute the error vectors δ' backward, layer by layer, starting from the final layer.
- The backward propagation is a consequence of the fact that the cost is a function of outputs from the network.

Now, we can combine backpropagation with stochastic gradient descent for adjusting weights and biases, in which we compute the gradient for **many** training examples.

In practice, we also need:

- an outer loop generating mini-batches of training examples
- another outer loop stepping through multiple epochs of training



Gradient descent with backpropagation for a mini-batch of m training examples



1. Input a set of training examples

2. For each training example x : Set the corresponding input activation $a^{x,1}$, and perform the following steps:

- **Feedforward:** For each $l = 2, 3, \dots, L$ compute

$$z^{x,l} = w^l a^{x,l-1} + b^l \text{ and } a^{x,l} = \sigma(z^{x,l}).$$

- **Output error $\delta^{x,L}$:** Compute the vector

$$\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L}).$$

- **Backpropagate the error:** For each

$$l = L - 1, L - 2, \dots, 2 \text{ compute}$$

$$\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l}).$$

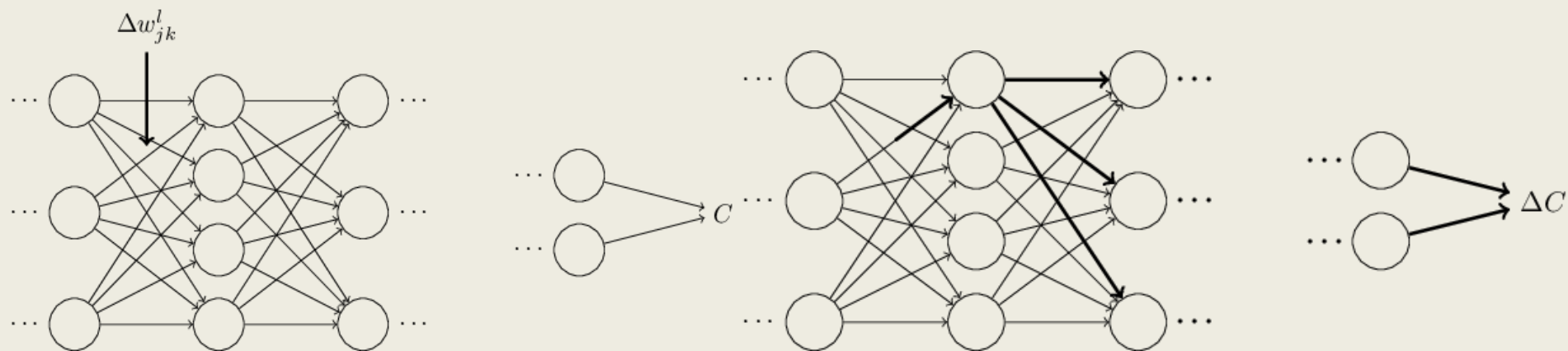
3. Gradient descent: For each $l = L, L - 1, \dots, 2$ update the weights according to the rule $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and the biases according to the rule $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

Backpropagation: More intuitions

- Every edge between two neurons in the network is associated with a **rate factor**, which is just **the partial derivative of one neuron's activation with respect to the other neuron's activation**.
- The rate factor for a forward path in the network is just the product of the rate factors along the entire path.
- The total rate of change of C with respect to a weight in the network, is just the sum of the rate factors of all paths from the initial weight to the final cost.
- The backpropagation algorithm provides a way of computing this sum over the rate factor for all these paths.



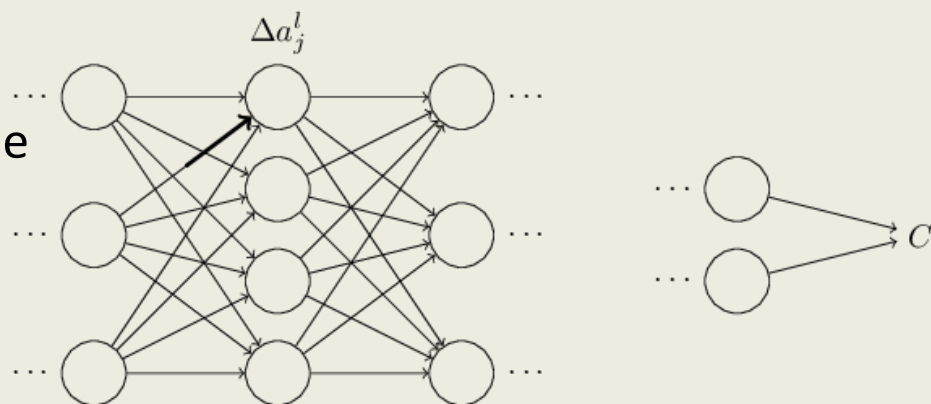
Backpropagation: More intuitions



Given the change, overall: $\Delta C \approx \frac{\partial C}{\partial w_{jk}^l} \Delta w_{jk}^l$

First, the change causes a small change in the activation of the j th neuron in the l th layer:

$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

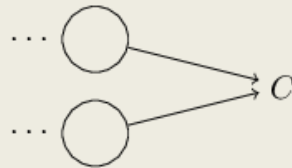
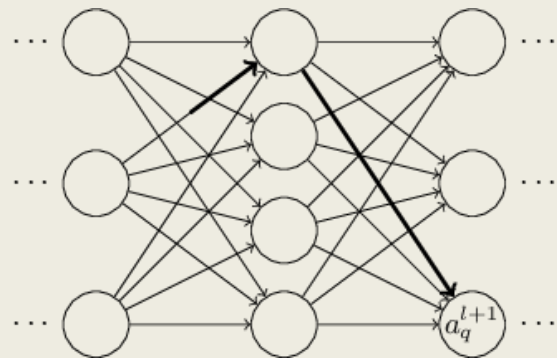


This change in turn, will cause changes in *all* the activations in the next layer. Let's concentrate on one of them...



$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

Backpropagation: More intuitions



Imagining a path
all the way to C:

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

Since the change propagates across all possible paths to affect C:

$$\Delta C \approx \sum_{mnp \dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$



Backpropagation: More intuitions

$$\Delta C \approx \sum_{mnp\dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l \quad \Delta C \approx \frac{\partial C}{\partial w_{jk}^l} \Delta w_{jk}^l$$

By combining the two equations above, it follows that:

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_{mnp\dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l}$$

- We are computing the rate of change of C with respect to a weight in the network.
- This is just the sum of the rate factors of all paths from that weight to the final cost.
- The backpropagation algorithm computes the sum over the rate factor for all these paths.

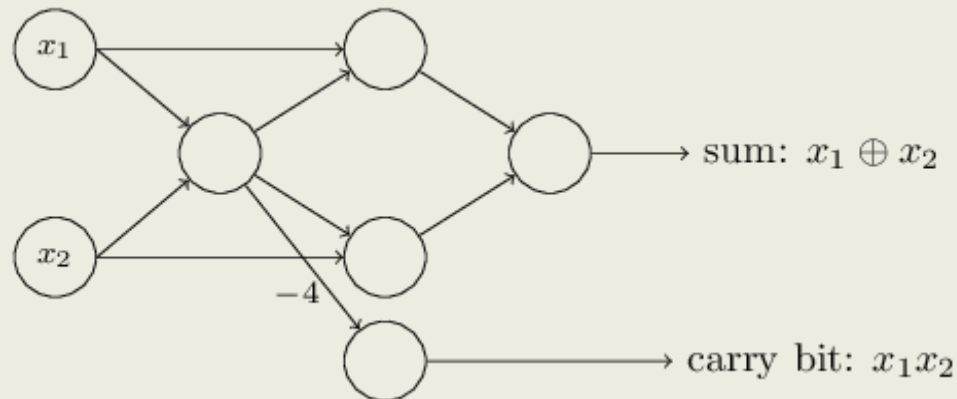
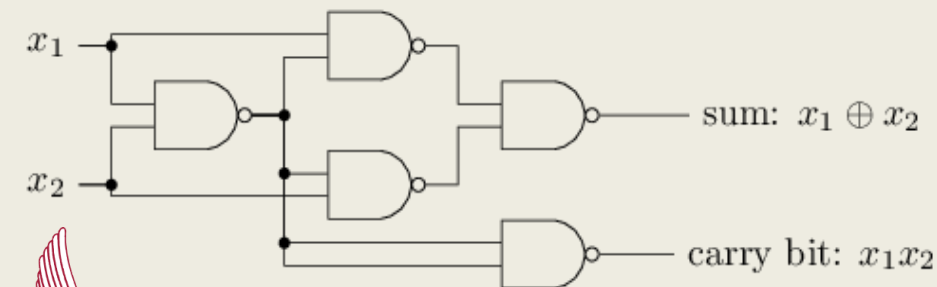


Deep Considerations ...



The Universal Approximation Theorem

- “A feed-forward network with a single hidden layer containing a finite number of neurons, can approximate continuous functions on compact subsets of R^n , under mild assumptions on the activation function.”
 - shown, e.g., for sigmoid functions (Cybenko, 1989)
- **Intuition:** a network of perceptrons can simulate a circuit containing NAND gates. Since any Boolean function can be implemented using NAND gates, it follows that perceptrons are universal computers!



other weights: -2; biases= 3



The Universal Approximation Theorem

- **Kolmogorov Theorem**

- *A neural network with at least 3 layers of sufficiently large number of neurons is able to learn any mathematical function!*

- **Thus, in theory**

- Simple networks can represent essentially any function!
- All we need is networks of perceptrons, likely MLPs.
- But, these may require a huge number of neurons!
- Also, no recipe on how exactly to structure the network.
- Also, no recipe on how to actually do the learning/computations.
- In a sense, we probably just confirm that a network of perceptrons is just a new type of NAND gate!



- **Practice+new theory shows that deep networks are good**

Why Deep NN?

- First ... what does “deep” mean?
- Well, **deep** means **more than 3** (input+hidden+output) layers
- So, strictly speaking, it is not “a new thing”!
- Why “**3**”? With depth 3 we have a universal approximator.
- Recent theoretical results show the power of “deep”.
- Deep architectures can represent a function with exponentially fewer units than a NN with 3 layers!



Deep Learning and Deep Neural Networks

■ Why now (i.e., after 2010)?

- more powerful CPUs + appearance of GPUs + big data
- high-quality software (tensorflow, PyTorch, MXNet, ...)
- new theoretical results that suggest their power

• A Spiral Effect

- Deep NNs do well →
- more and more people now get interested →
- people get to learn how to tweak Deep NNs →
- Deep NNs get even better in more domains →
- [repeat]



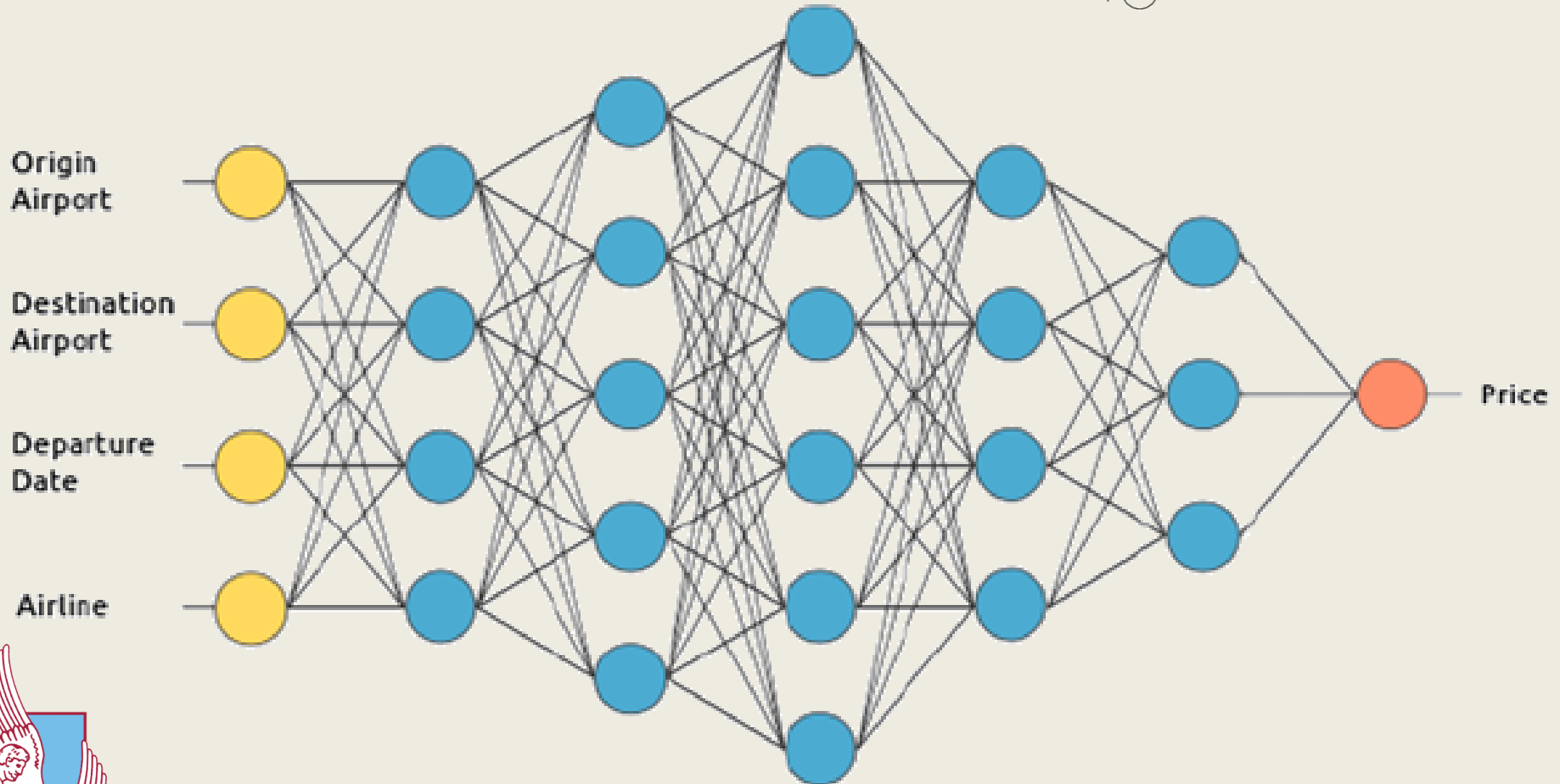
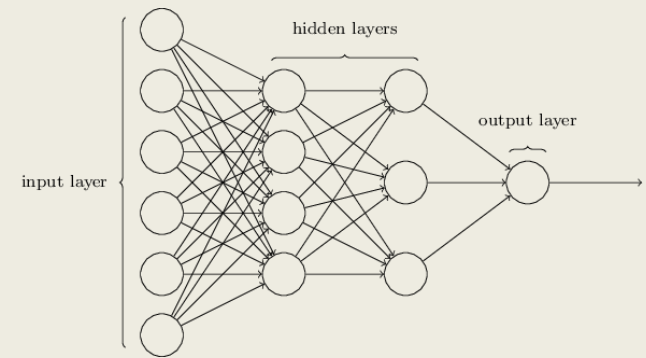
Deep Neural Networks in a Nutshell

input layer + a number of hidden layers + output layer

- Each unit's input: a **weighted sum of outputs** of units from previous layers (weights are on links between layers)
- Units' outputs: **nonlinear transformation** of weighted inputs by **activation functions** (tanh, logistic, rectified linear unit, ...)
- Compute **error derivatives** and **backpropagate gradients** from the output layer towards the input layer.
- **Update weights** from gradients to optimize loss function.
- **Repeat** until convergence.

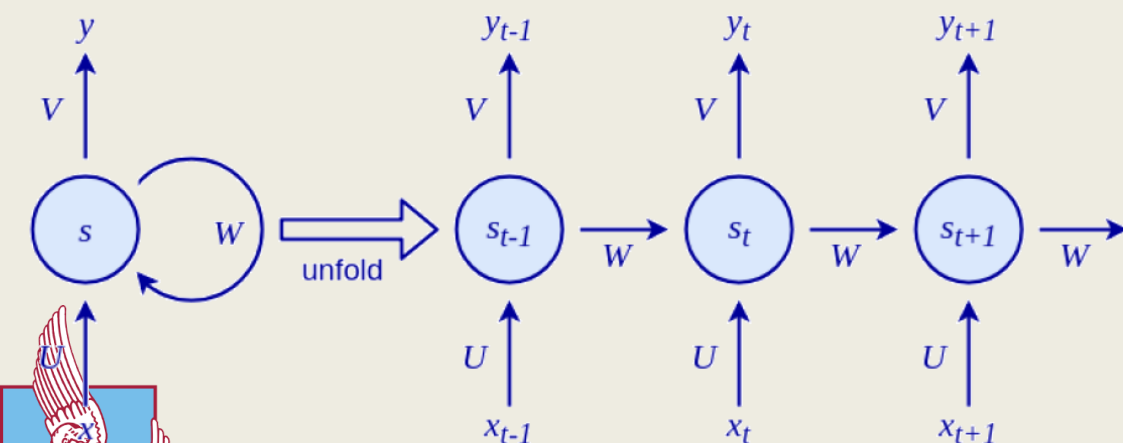


Deep Feedforward NNs



Recurrent Neural Networks (RNNs)

- In feed-forward NNs we assume that all inputs (and outputs) are independent of each other → Could be a bad idea
- RNNs have at least one feedback node $s_t = f(s_{t-1}, x_t)$
 - Can process sequential data with variable length
 - Consider time dependencies via the recurrence relation
- They might exhibit complicated dynamics and is difficult to train
- They act similarly to the biological model
- Increasing interest on RNN architectures

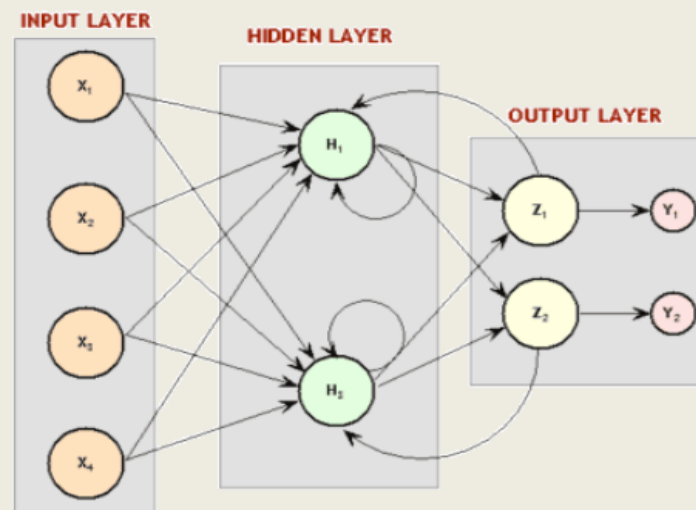


Recurrence relation:

$$s_t = f(W \cdot s_{t-1} + U \cdot x_t)$$

Final Output:

$$y_t = V \cdot s_t$$

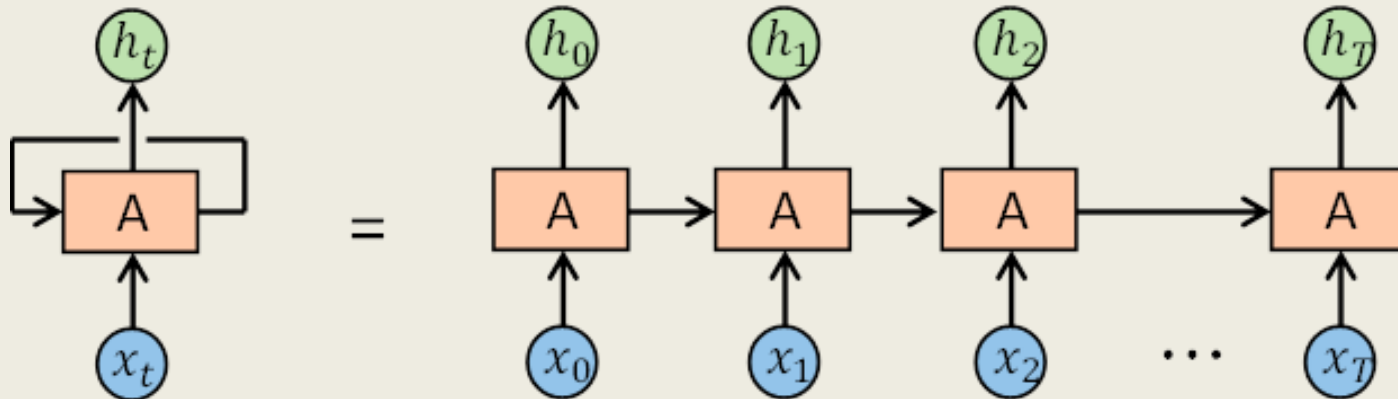


$$\{\widehat{TD}_{1(t+1)}, \widehat{TD}_{2(t+1)}, \dots, \widehat{TD}_{N(t+1)}\} \quad \{\widehat{TD}_{1(T+1)}, \widehat{TD}_{2(T+1)}, \dots, \widehat{TD}_{N(T+1)}\}$$

Prediction for t+1

Prediction for T+1

Recurrent NNs



- **Idea:** capture sequential dependencies among inputs, add “memory” for calculations
- layer A (at t) looks at input x_t , outputs h_t ; loop allows sequence info to propagate
- Loop $\equiv$ multiple copies of the same network
- At each t , a hidden state C_t (memory) is calculated given x_t and the previous state C_{t-1}
- The weights are *the same at all steps* – since same task at each step, different inputs
- **Problem:** “vanishing gradient”, *same as in feed-forward networks*

Example

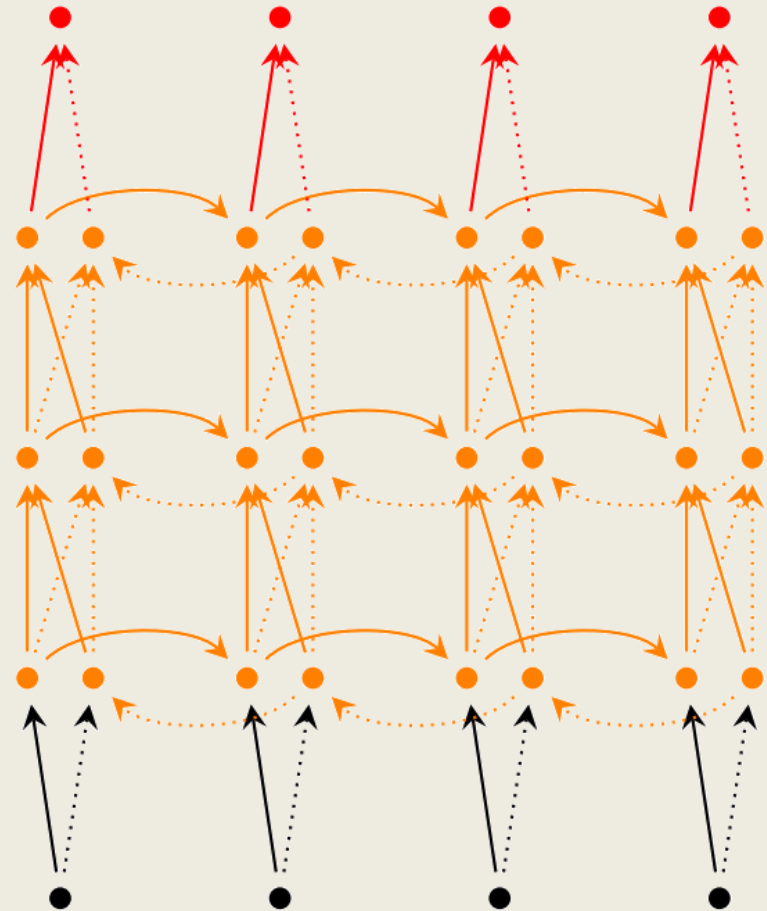
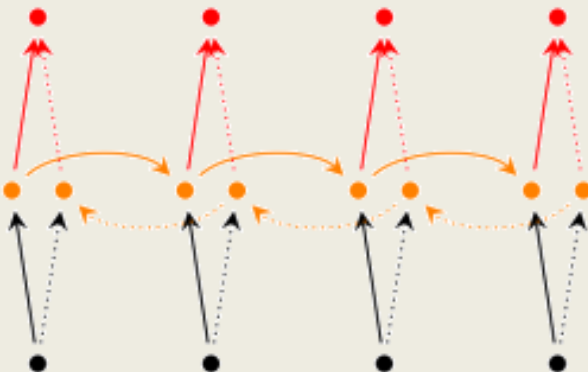
- Trip demand prediction problem
- x_t = at t : trip demands at N zones, hour, day of week, precipitation, temperature

$$\{TD_{1t}, TD_{2t}, \dots, TD_{Nt}, H_t, DW_t, P_t, T_t\}$$

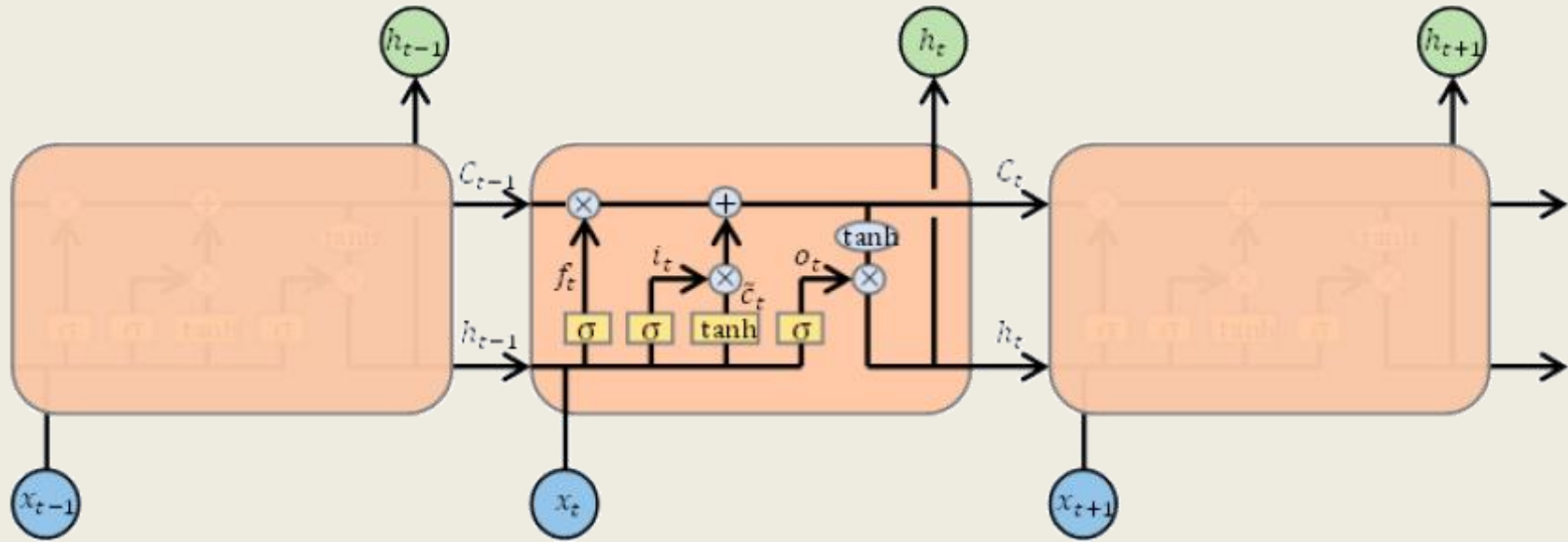
RNNs made deeper

- RNNs are already deep, but perhaps not deep enough!
- Deep bidirectional RNNs →

Bidirectional RNN
(captures dependencies on future elements)



Long Short-Term Memories (LSTMs)



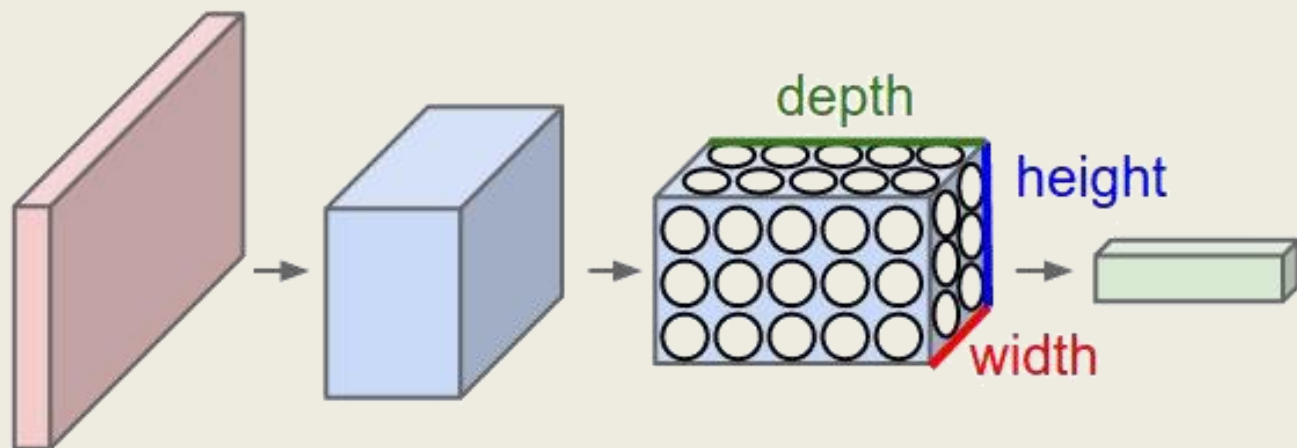
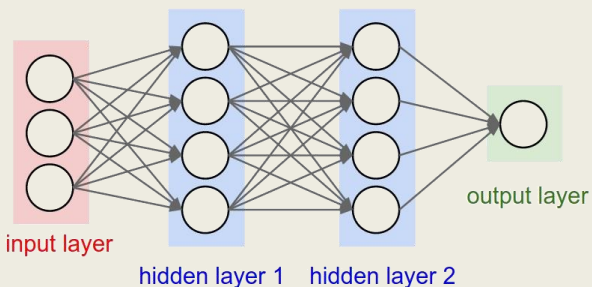
- **Idea:** capture sequential dependencies among inputs, add “memory” for calculations, and **overcome the vanishing gradient problem (their short-term memory can last long) ...how?**
- **Four layers in each repeating module**
 - A logistic “forget gate layer” decides the extent to which a value in C_{t-1} remains in C_t
 - A logistic “input gate layer” decides the extent to which input values will be updated
 - A tanh layer outputs an array of new candidate values to add to the forget layer output
 - A logistic “output gate layer” decides the extent to which values are used in the output



Excellent results for NLP and speech recognition tasks!

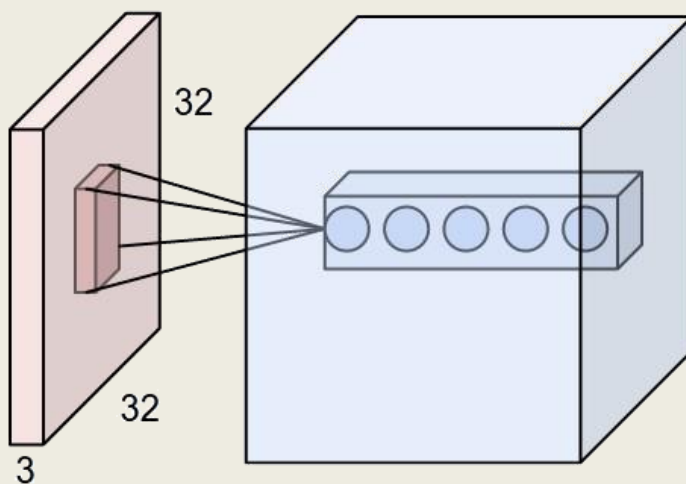
Convolutional NNs

A 3D neural architecture!



Biologically-inspired:

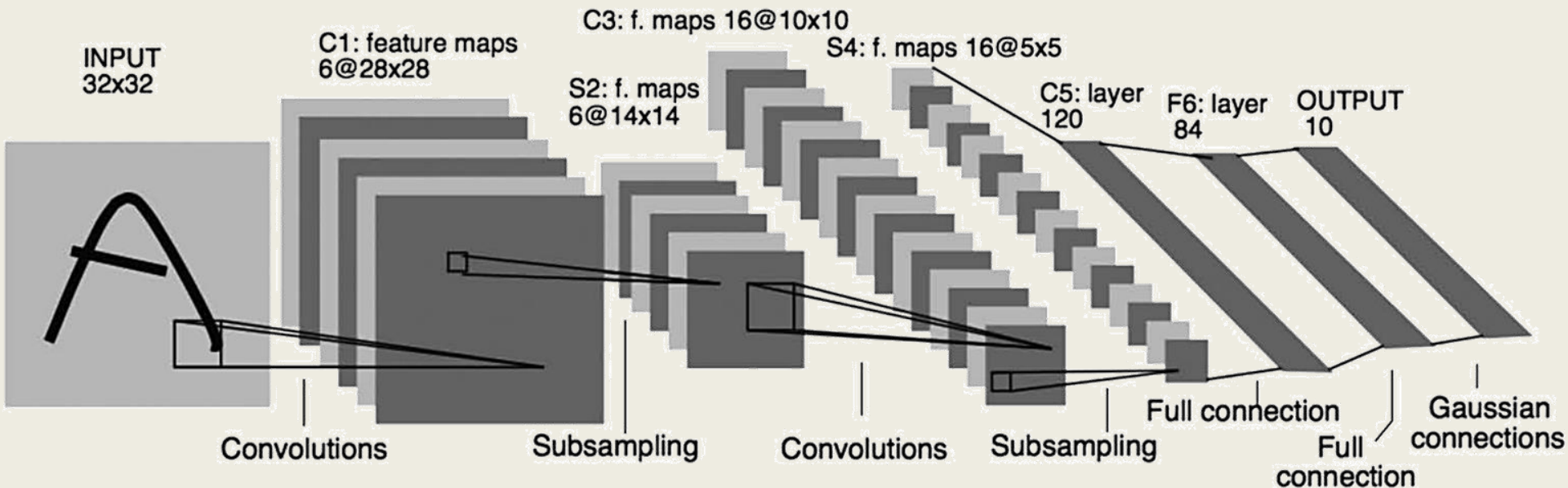
- resemble the architecture of the visual cortex
- they use "receptive fields"



One key common technique used:
depth-slicing+parameter sharing

Obviously good for image/video recognition,
but also for NLP and recommender systems!

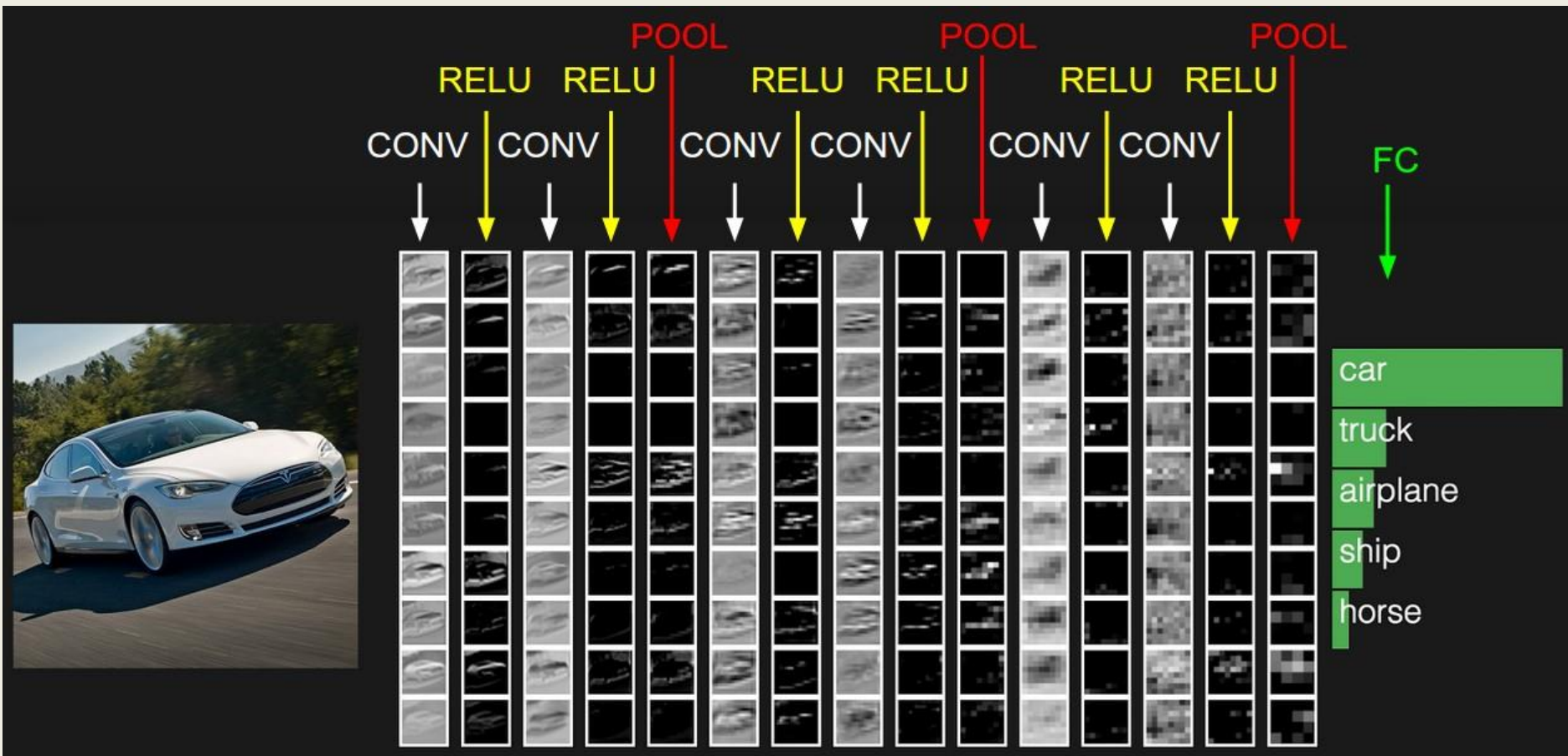
CNNs: an example Conv-Net architecture



<http://incompleteideas.net/book/the-book-2nd.html>

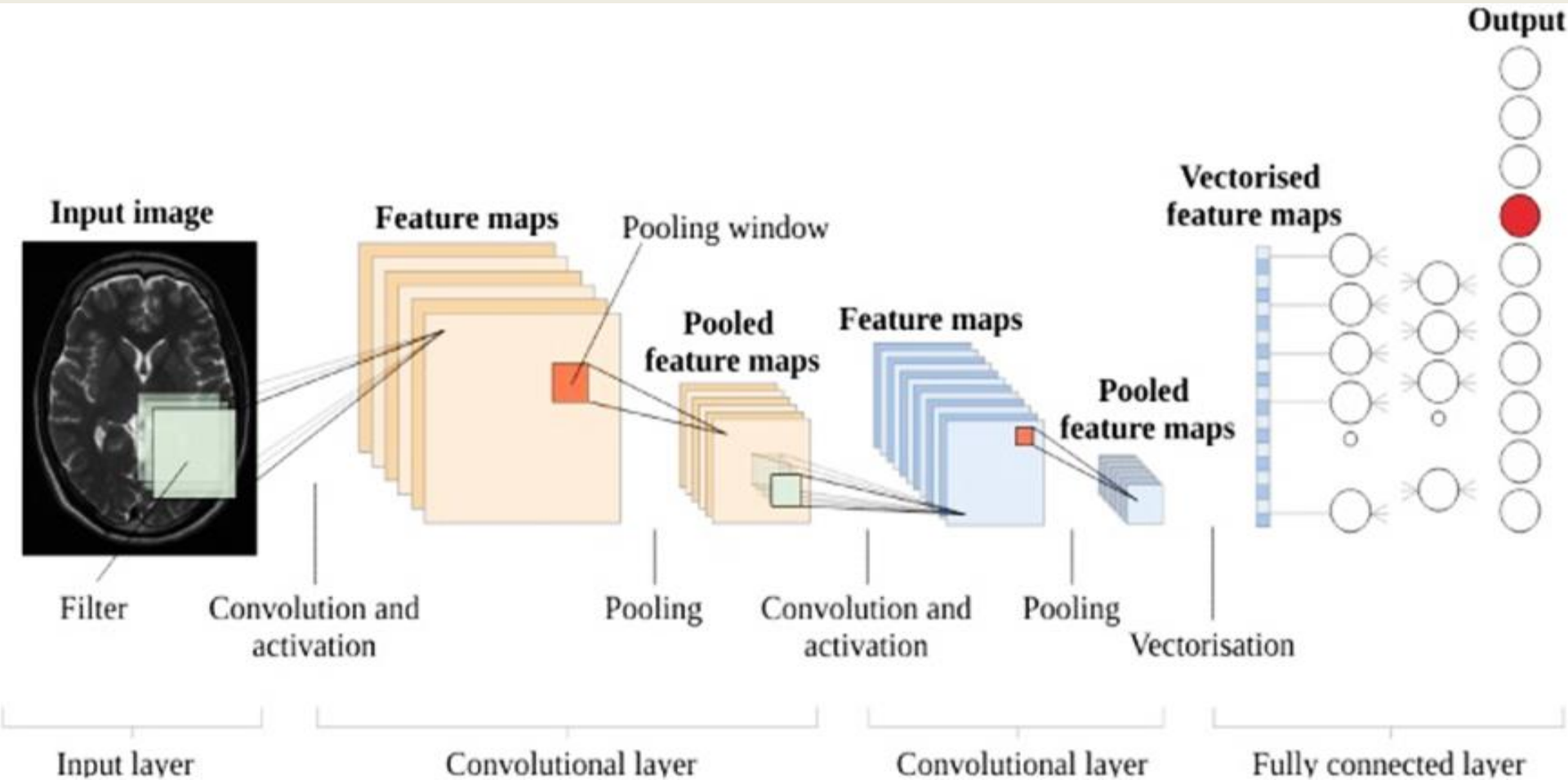


CNN use in yet another Conv-Net architecture

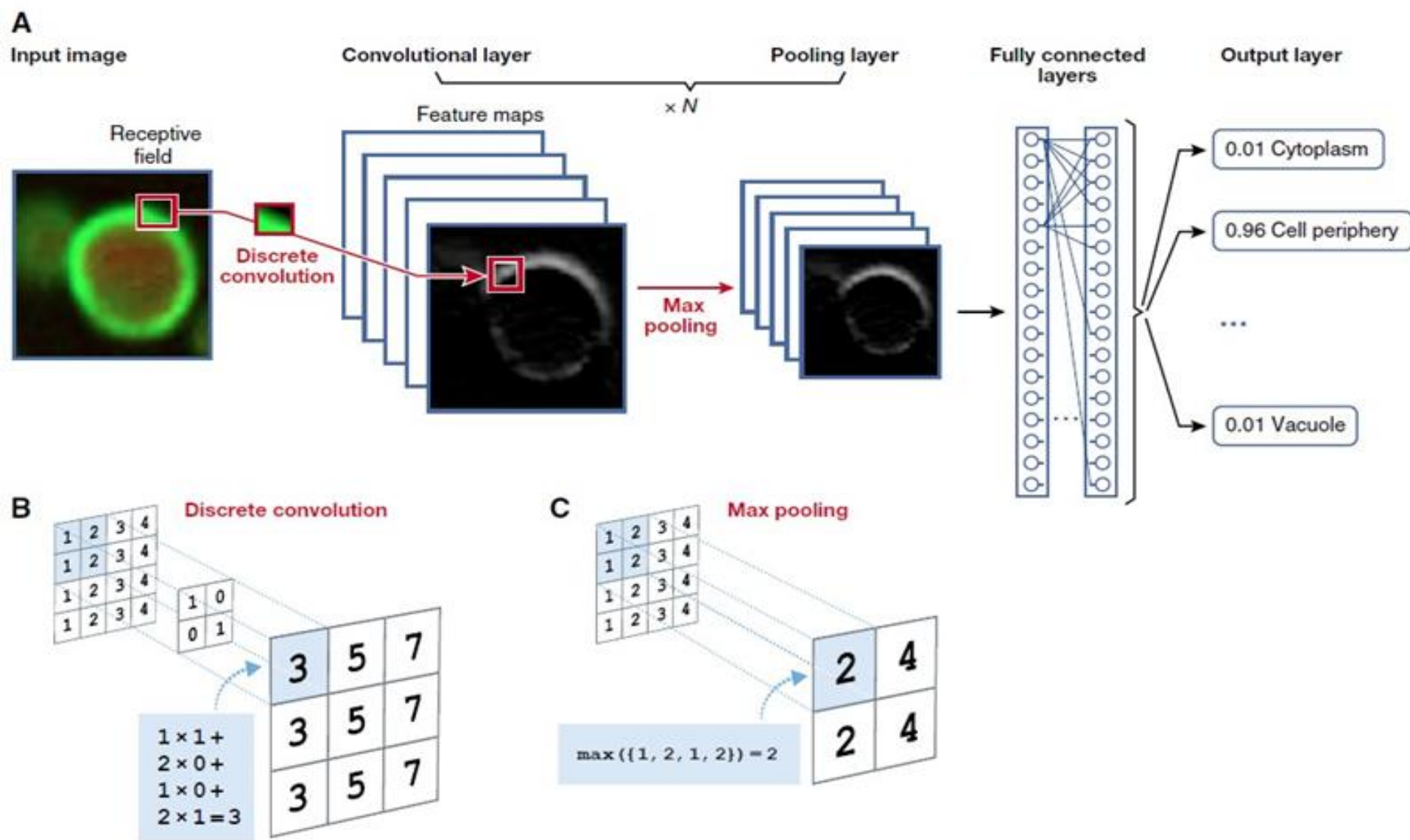


<http://cs231n.github.io/convolutional-networks/>

Typical CNN



Typical CNN Diagram



CNNs for Medical Imaging

- **BrainNet**: developed based on TensorFlow, aims to train deep neural networks to segment GM (Gray Matter) and WM (White Matter) from brain MR images
- **LiviaNet**: developed based on Theano, aims to train 3-D fully convolutional neural networks by using MR images to segment sub-cortical brain structures
- **DIGITS**: developed to rapidly train accurate deep neural networks for image segmentation, classification, and tissue detection tasks (e.g. Alzheimer's disease detection)
- **DeepMedic**: developed based on Theano, aims to train multi-scale 3-D convolutional neural networks for brain lesion segmentation from MR images (winner of the ISLES 2015 competition)



Main References

- Michael Nielsen (2015): **Neural Networks and Deep Learning** *Free online book.* neuralnetworksanddeeplearning.com
- Ian Goodfellow and Yoshua Bengio and Aaron Courville (2016): **Deep Learning** *Free online book.* www.deeplearningbook.org

A few other good reads

- Delalleau, O. and Bengio, Y. (2011). *Shallow vs. deep sum-product networks*. In Neural Information Processing Systems (NeurIPS)
- Hinton, G., Deng, L., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. (2012). *Deep neural networks for acoustic modeling in speech recognition*. IEEE Signal Processing Magazine, 29 (6), 82–97
- Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). *Maxout networks*. in Intl Conf on ML (ICML)

