MAT8034: Machine Learning

Deep Learning

Fang Kong

https://fangkongx.github.io/Teaching/MAT8034/Spring2025/index.html

Part of slide credit: ai.berkeley.edu; SJTU VE445

Outline

- History of artificial neural nets
- Perceptron
- Multilayer perceptron networks
- Activation functions
- Training: backpropagation
- Modules in modern neural networks

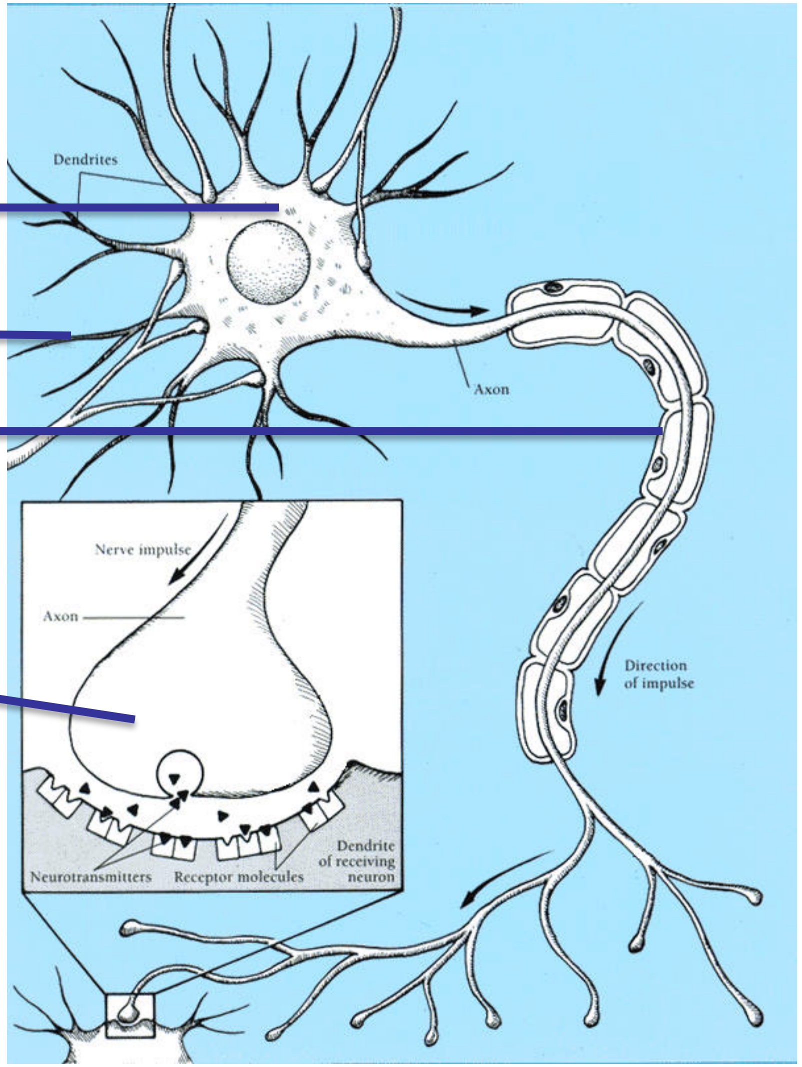
History of artificial neural nets

Brief history of artificial neural nets

- The First wave
  - 1943 McCulloch and Pitts proposed the McCulloch-Pitts neuron model
  - 1958 Rosenblatt introduced the simple single layer networks now called Perceptrons
  - 1969 Minsky and Papert's book Perceptrons demonstrated the limitation of single layer perceptrons, and almost the whole field went into hibernation
- The Second wave
  - 1986 The Back-Propagation learning algorithm for Multi-Layer Perceptrons was rediscovered and the whole field took off again
- The Third wave
  - 2006 Deep (neural networks) Learning gains popularity
  - 2012 made significant break-through in many applications

Biological neuron structure

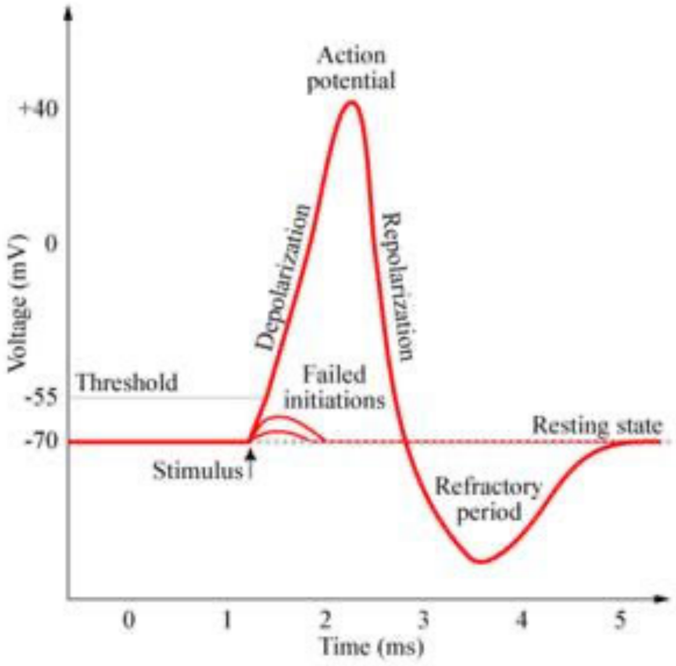
- 细胞结构
  - 细胞体
  - 树突
  - 轴突
  - 突触末梢



Slide credit: Ray Mooney

Biological neural communication

- 细胞膜间的电位表现出的电信号称为动作电位
- 电信号从细胞体中产生，沿着轴突往下传，并且导致突触末梢释放神经递质介质
- 介质通过化学扩散从突触传递到其他神经元的树突
- 神经递质可以是兴奋的或者是抑制的
- 如果从其他神经元来的神经递质是兴奋的且超过某个阈值，将会触发一个动作电位

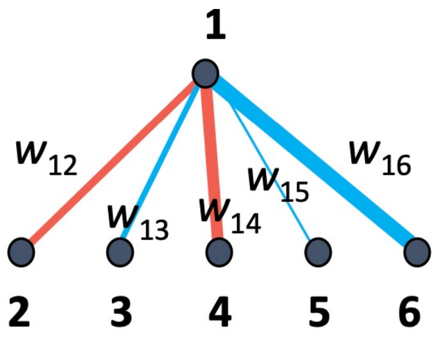


Slide credit: Ray Mooney

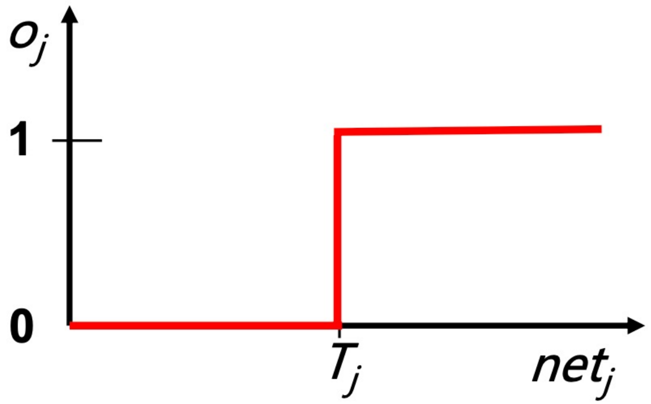
McCulloch-Pitts neuron model [1943]

- Model the network as a graph, where the units are nodes, and the synaptic connections are weighted edges from node i to node j, with the weight as $w_{j,i}$
- The input of the unit is:

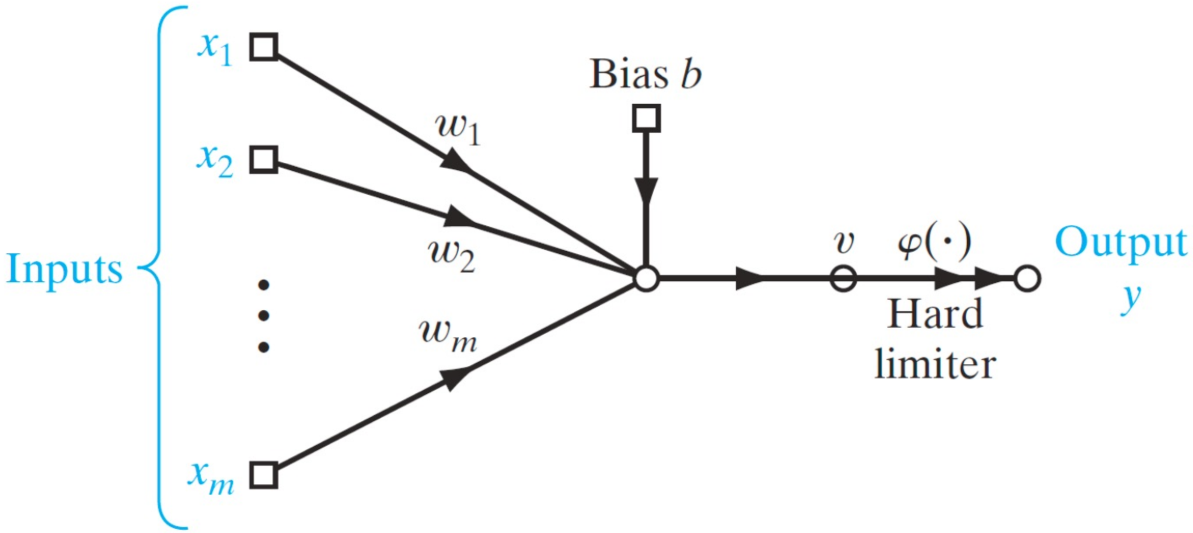
$$\text{net}_j = \sum_i w_{j,i} \cdot o_i$$



- The output of the unit is:
    - 0 if $\text{net}_j < T_j$; 1 otherwise
    - T_j is the threshold



Single-layer perception by Rosenblatt [1958]



- Rosenblatt [1958] 进一步提出感知机作为第一个在“老师”指导下进行学习的模型（即监督学习）
- 专注在如何找到合适的用于二分类任务的权重w_m
  - $y = 1$：类别1
  - $y = -1$：类别2

预测

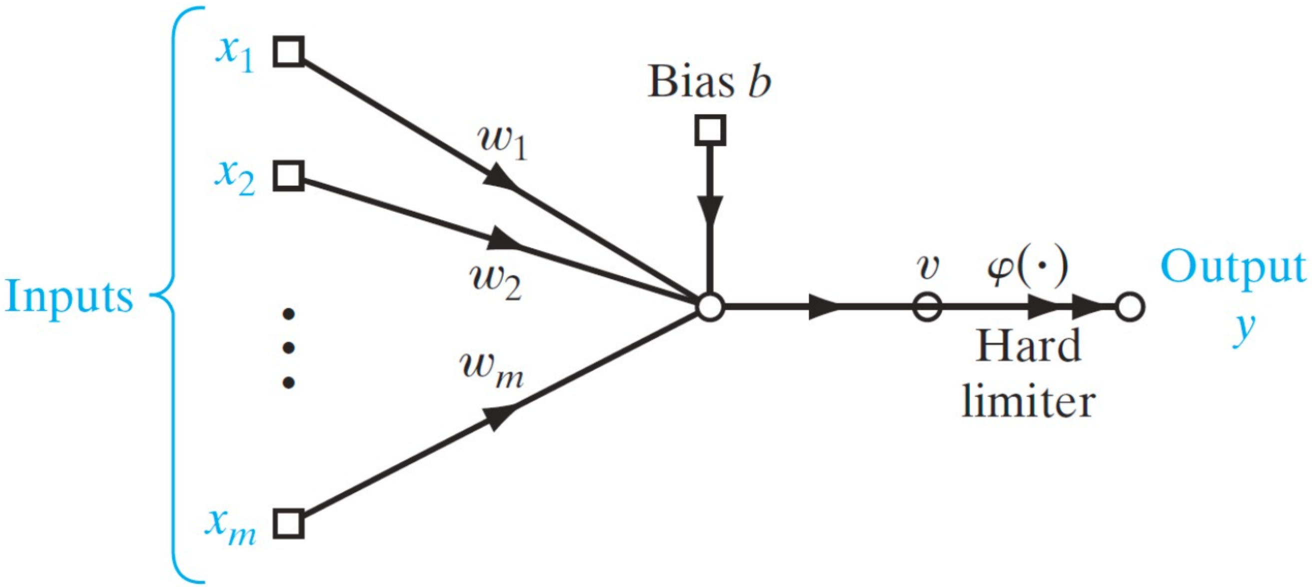
$$\hat{y} = \sigma\left(\sum_{i=1}^{m} w_i x_i + b\right)$$

激活函数

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Slide credit: Weinan Zhang

Training perception



- 训练

$$w_i = w_i + \eta(y - \hat{y})x_i$$
$$b = b + \eta(y - \hat{y})$$

- 下列规则等价：
  - 如果输出正确，则不进行操作
  - 如果输出高了，降低正输入的权重
  - 如果输出低了，增加正输入的权重

预测

$$\hat{y} = \sigma\left(\sum_{i=1}^{m} w_i x_i + b\right)$$

激活函数

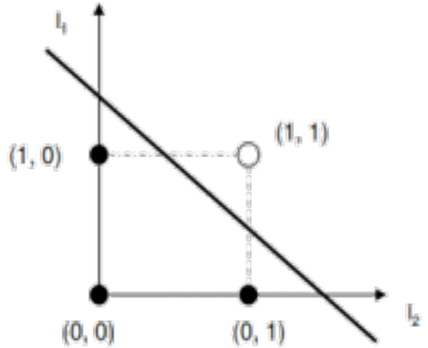
$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Slide credit: Weinan Zhang

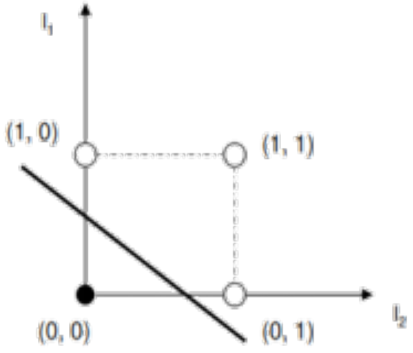
Limitation of perception

- Minsky and Papert [1969] showed that some rather elementary computations, such as XOR problem, could not be done by Rosenblatt’s one-layer perceptron
- However Rosenblatt believed the limitations could be overcome if more layers of units to be added, but no learning algorithm known to obtain the weights yet

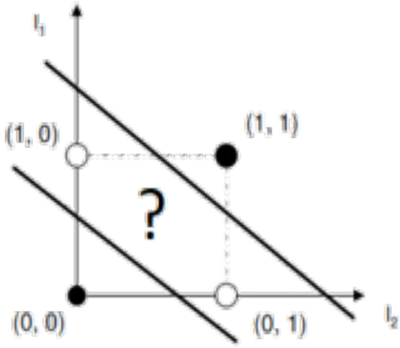
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

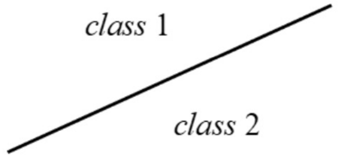


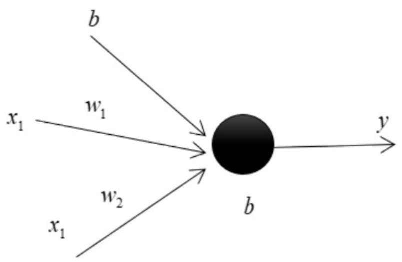
XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



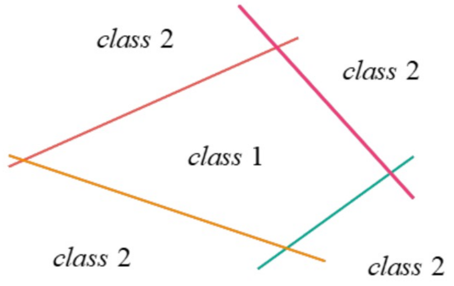
Solution: Add hidden layers

- Adding hidden layers to learn more general scenarios

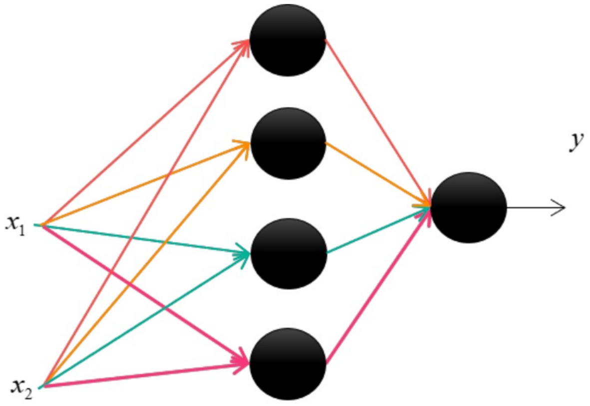




决策边界：$x_1w_1 + x_2w_2 + b = 0$



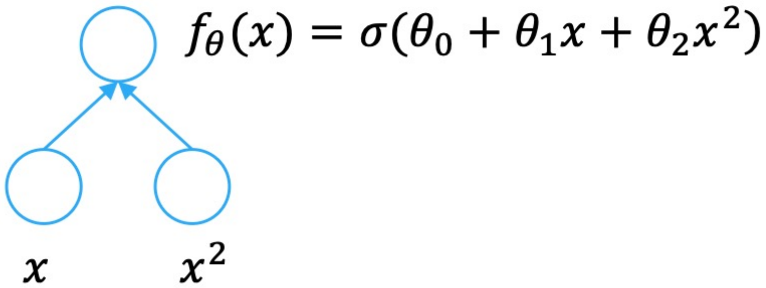
每个隐含节点负责实现凸区域的一条边界线



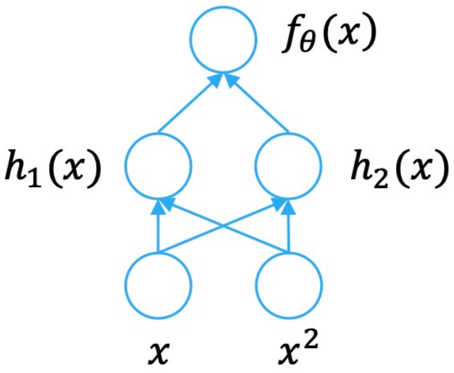
Slide credit: Weinan Zhang

Computation

- Single-layer function
  - $f_\theta(x) = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$



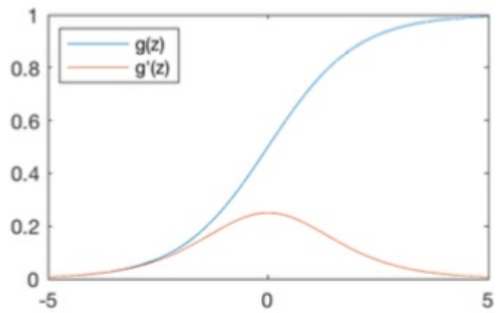
- Multi-layer function
  - $h_1(x) = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$
  - $h_2(x) = \sigma(\theta_3 + \theta_4 x_1 + \theta_5 x_2)$
  - $f_\theta(x) = \sigma(\theta_6 + \theta_7 h_1 + \theta_8 h_2)$



Non-linear activation functions

- Adding non-linearity allows the network to learn and represent complex patterns in the data
- Common non-linear activation functions

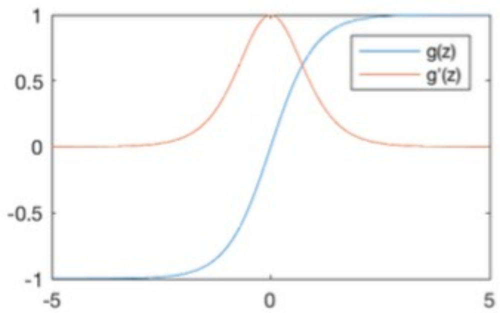
Sigmoid Function



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

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

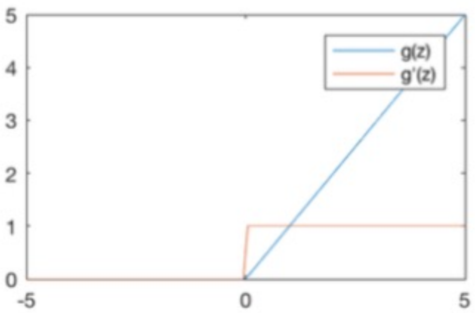
Hyperbolic Tangent



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

$$\sigma'(z) = 1 - \sigma(z)^2$$

Rectified Linear Unit (ReLU)

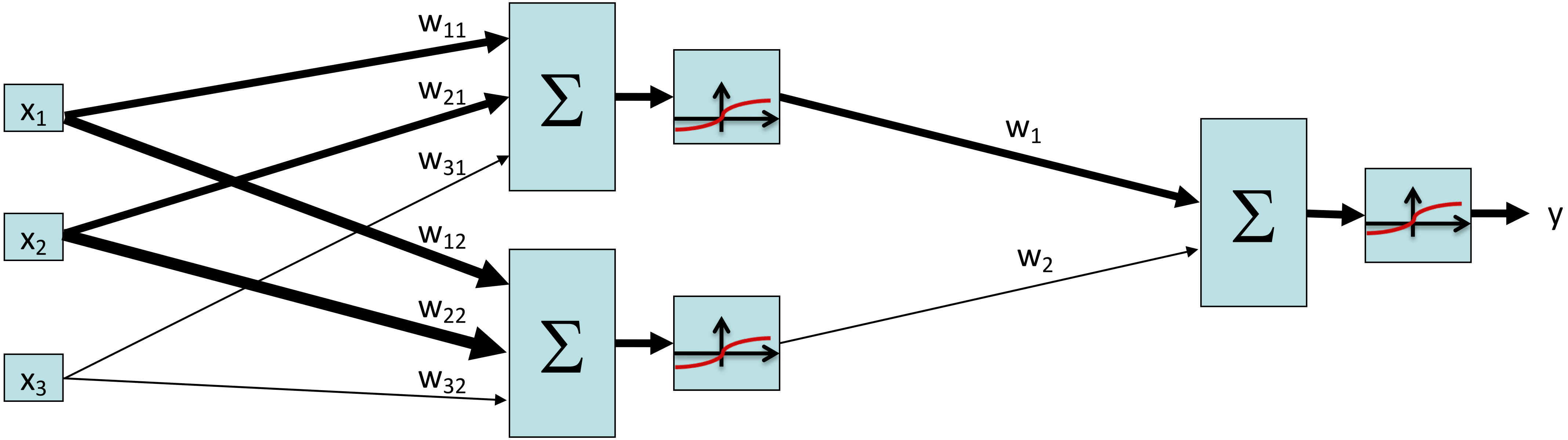


$$\sigma(z) = \max(0, z)$$

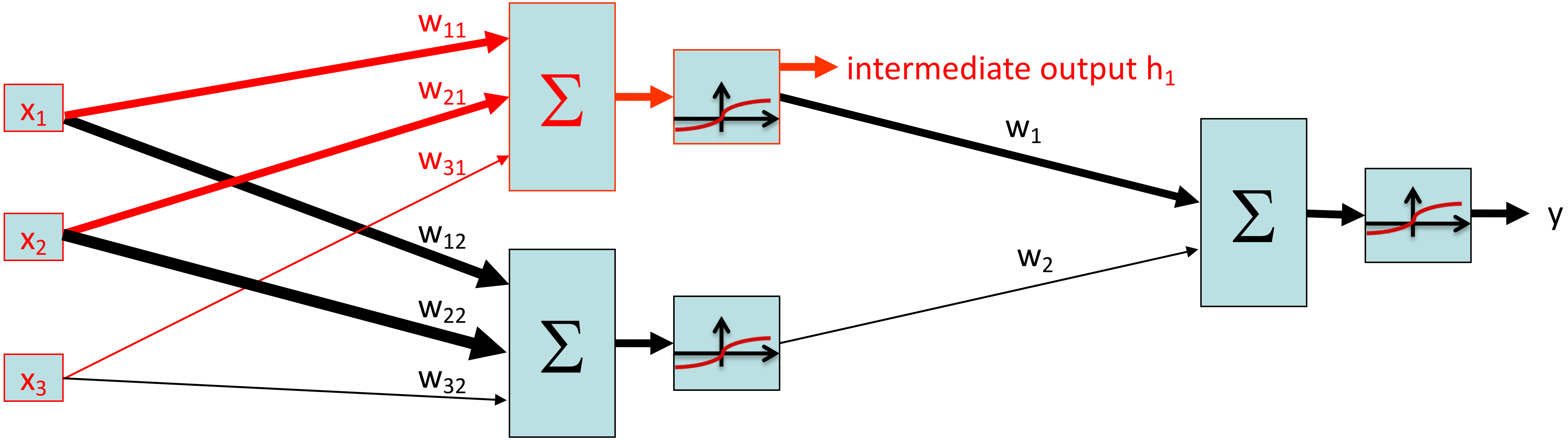
$$\sigma'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

[source: MIT 6.S191 introtodeeplearning.com]

2-Layer, 2-Neuron Neural Network

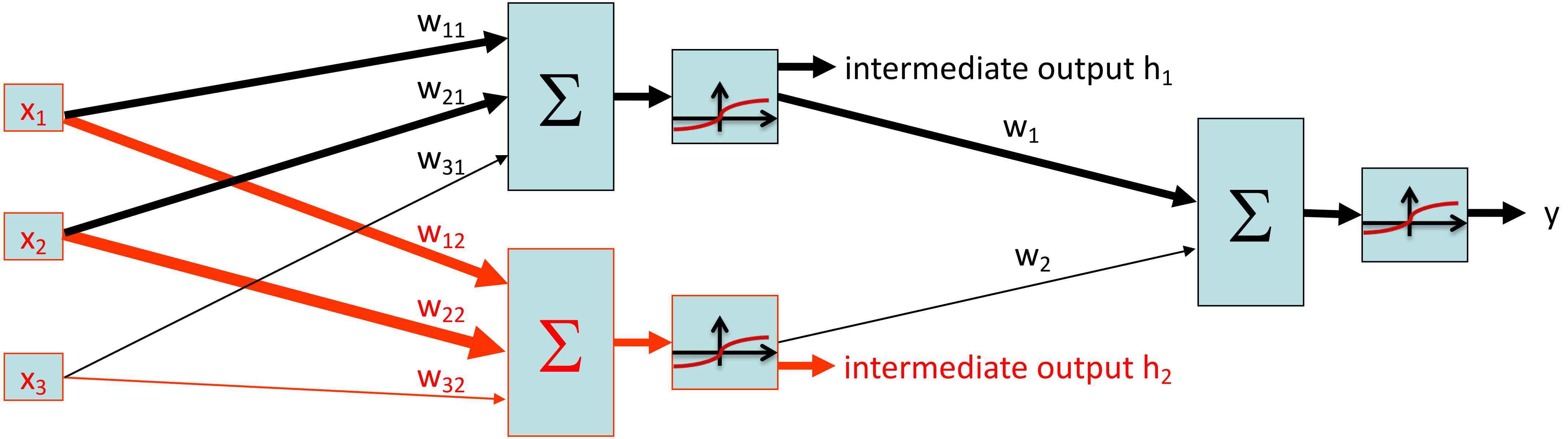


2-Layer, 2-Neuron Neural Network



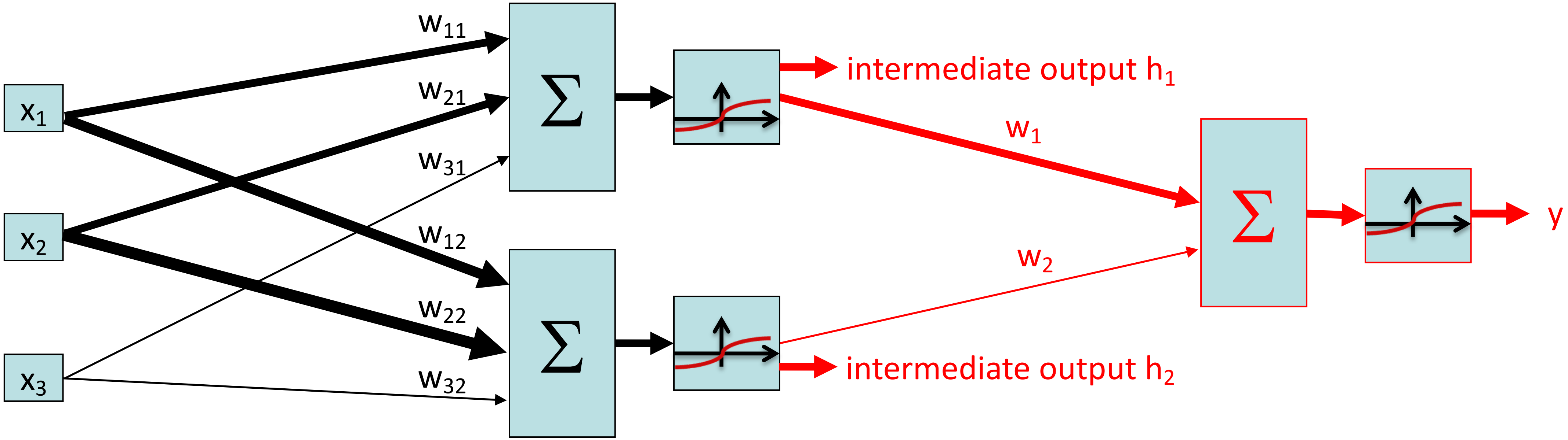
$$\text{intermediate output } h_1 = \sigma(w_{11}x_1 + w_{21}x_2 + w_{31}x_3)$$
$$= \frac{1}{1 + e^{-(w_{11}x_1 + w_{21}x_2 + w_{31}x_3)}}$$

2-Layer, 2-Neuron Neural Network



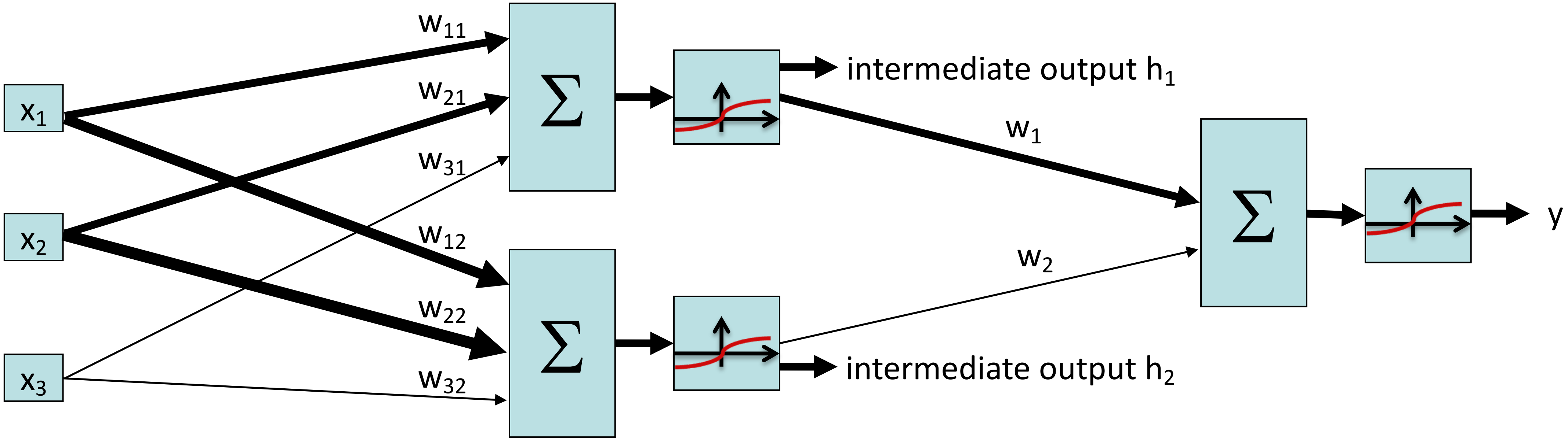
$$\text{intermediate output } h_2 = \sigma(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)$$
$$= \frac{1}{1 + e^{-(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)}}$$

2-Layer, 2-Neuron Neural Network



$$y = \sigma(w_1 h_1 + w_2 h_2)$$
$$= \frac{1}{1 + e^{-(w_1 h_1 + w_2 h_2)}}$$

2-Layer, 2-Neuron Neural Network



$$
\begin{aligned}
y &= \sigma(w_1 h_1 + w_2 h_2) \\
&= \sigma(w_1 \sigma(w_{11} x_1 + w_{21} x_2 + w_{31} x_3) + w_2 \sigma(w_{12} x_1 + w_{22} x_2 + w_{32} x_3))
\end{aligned}
$$

Vectorization

$$
\begin{aligned}
y &= \sigma(w_1 h_1 + w_2 h_2) \\
&= \sigma(w_1 \sigma(w_{11}x_1 + w_{21}x_2 + w_{31}x_3) + w_2 \sigma(w_{12}x_1 + w_{22}x_2 + w_{32}x_3))
\end{aligned}
$$

The same equation, formatted with matrices:

$$
\begin{aligned}
&\sigma\left(\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}\right) \\
= \; &\sigma(\begin{bmatrix} w_{11}x_1 + w_{21}x_2 + w_{31}x_3 & w_{12}x_1 + w_{22}x_2 + w_{32}x_3 \end{bmatrix}) \\
= \; &\begin{bmatrix} h_1 & h_2 \end{bmatrix}
\end{aligned}
$$

$$
\sigma\left(\begin{bmatrix} h_1 & h_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}\right) = \sigma(w_1 h_1 + w_2 h_2) = y
$$

The same equation, formatted more compactly by introducing variables representing each matrix:

$$
\sigma\,(x \times W_{\text{layer 1}}) = h \qquad\qquad \sigma(h \times W_{\text{layer 2}}) = y
$$

2-Layer, 2-Neuron Neural Network

$$\sigma(x \times W_{\text{layer 1}}) = h$$

Shape (1, 3).
Input feature vector.

Shape (3, 2).
Weights to be learned.

Shape (1, 2).
Outputs of layer 1,
inputs to layer 2.

$$\sigma(h \times W_{\text{layer 2}}) = y$$

Shape (1, 2).
Outputs of layer 1,
inputs to layer 2.

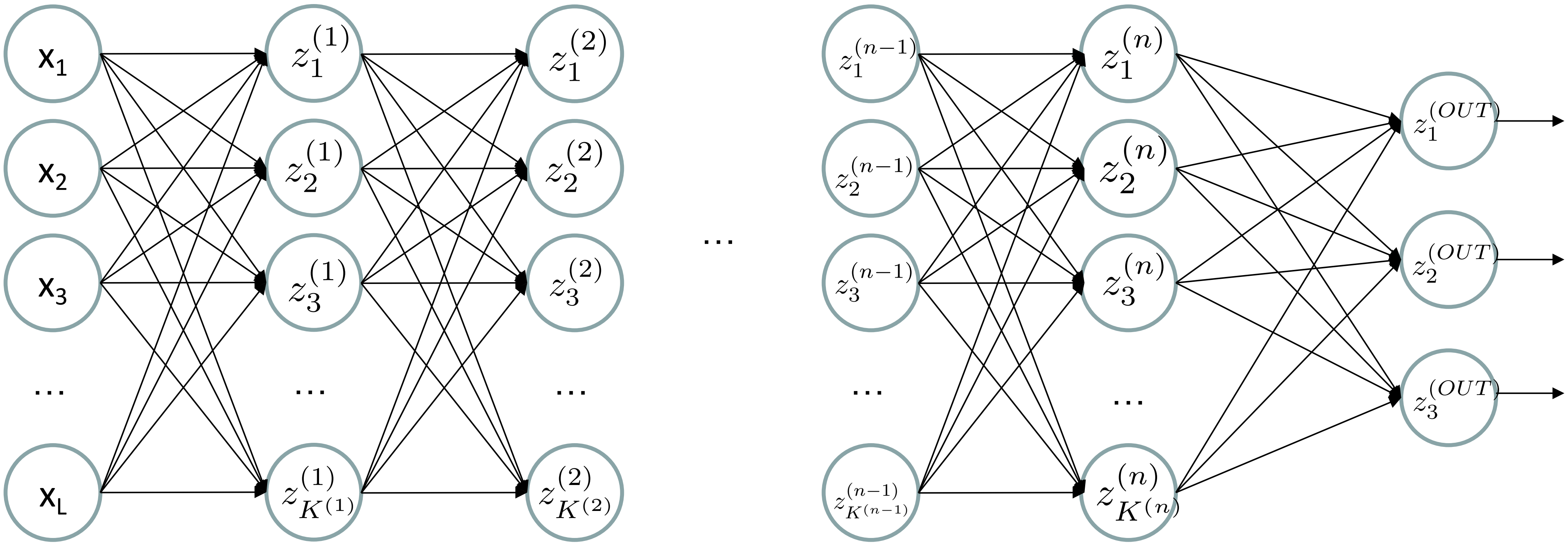
Shape (2, 1).
Weights to be learned.

Shape (1, 1).
Output of network.

Multi-Layer Neural Network

- Input to a layer: some *dim(x)*-dimensional input vector
- Output of a layer: some *dim(y)*-dimensional output vector
  - *dim(y)* is the number of neurons in the layer (1 output per neuron)
- Process of converting input to output:
  - Multiply the (1, *dim(x)*) input vector with a (*dim(x)*, *dim(y)*) weight vector. The result has shape (1, *dim(y)*).
  - Apply some non-linear function (e.g. sigmoid) to the result. The result still has shape (1, *dim(y)*).
- Big idea: Chain layers together
  - The input could come from a previous layer's output
  - The output could be used as the input to the next layer

Deep Neural Network

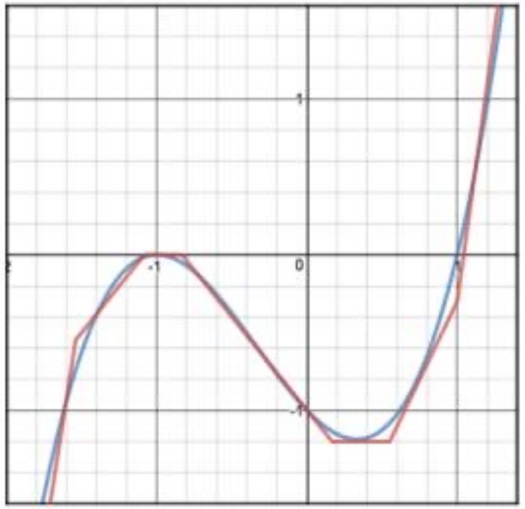


$$z_i^{(k)} = \sigma\Big(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\Big)$$

σ = nonlinear activation function

Universal approximation theorem

- Theorem (Universal Function Approximators). A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.



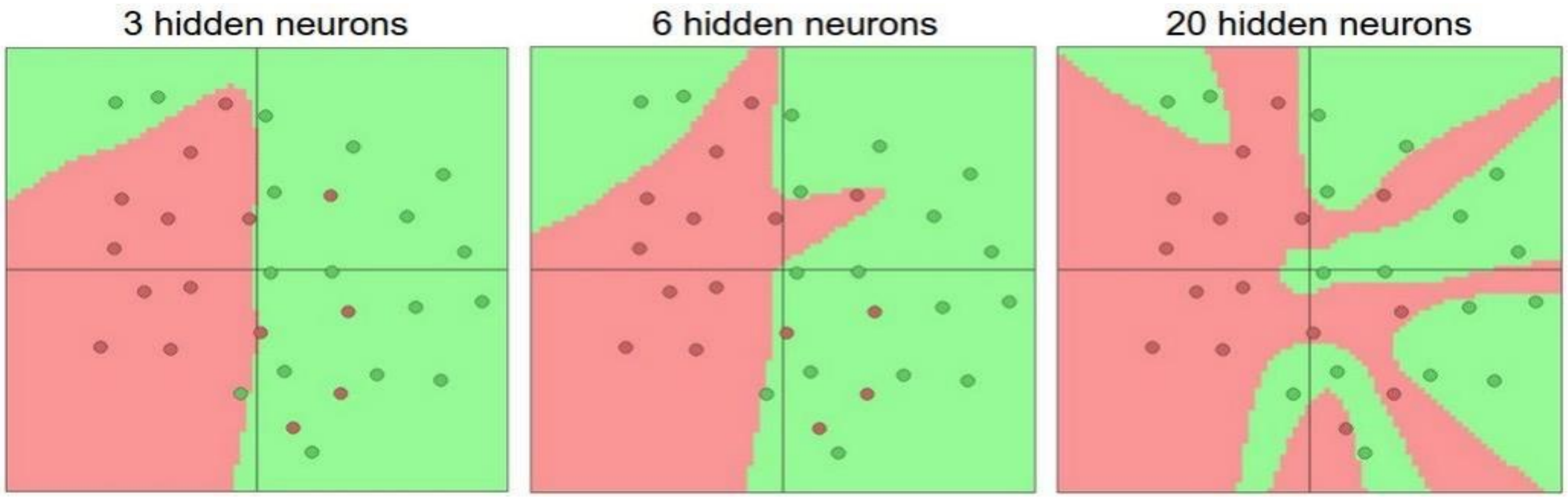
$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$\begin{aligned} Z(x) = & -n_1(x) - n_2(x) - n_3(x) \\ & + n_4(x) + n_5(x) + n_6(x) \end{aligned}$$

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." *Neural networks* 2.5 (1989): 359-366

Increasing power of approximation

- With more neurons, its approximation power increases. The decision boundary covers more details (risk of overfitting)



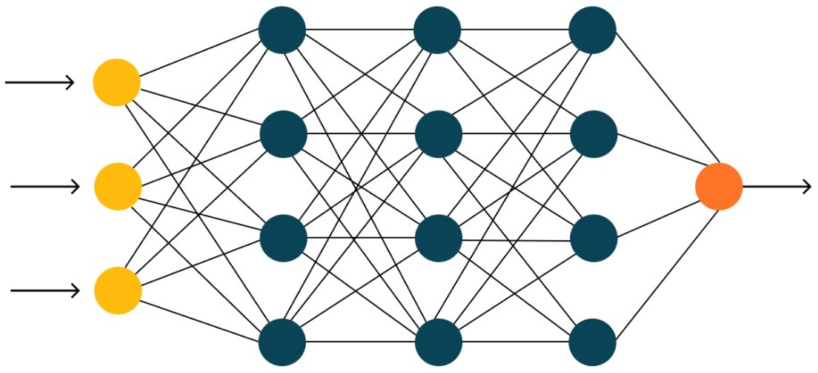
  - Usually in applications, we use more layers with structures to approximate complex functions instead of one hidden layer with many neurons

http://playground.tensorflow.org/

Connection to the kernel methods

Connection to the kernel methods

- Kernel methods
  - Design the non-linear feature map function
  - The performance significantly depends on the choice of feature map
  - Feature engineering: process of choosing the feature maps
- Neural network
  - Automatically learn the right feature map
  - Requires often less feature engineering



Training: Backpropagation

Review: Derivatives and Gradients

- What is the derivative of the function $g(x) = x^2 + 3$?

$$\frac{dg}{dx} = 2x$$

- What is the derivative of g(x) at x=5?

$$\frac{dg}{dx}\Big|_{x=5} = 10$$

Review: Derivatives and Gradients

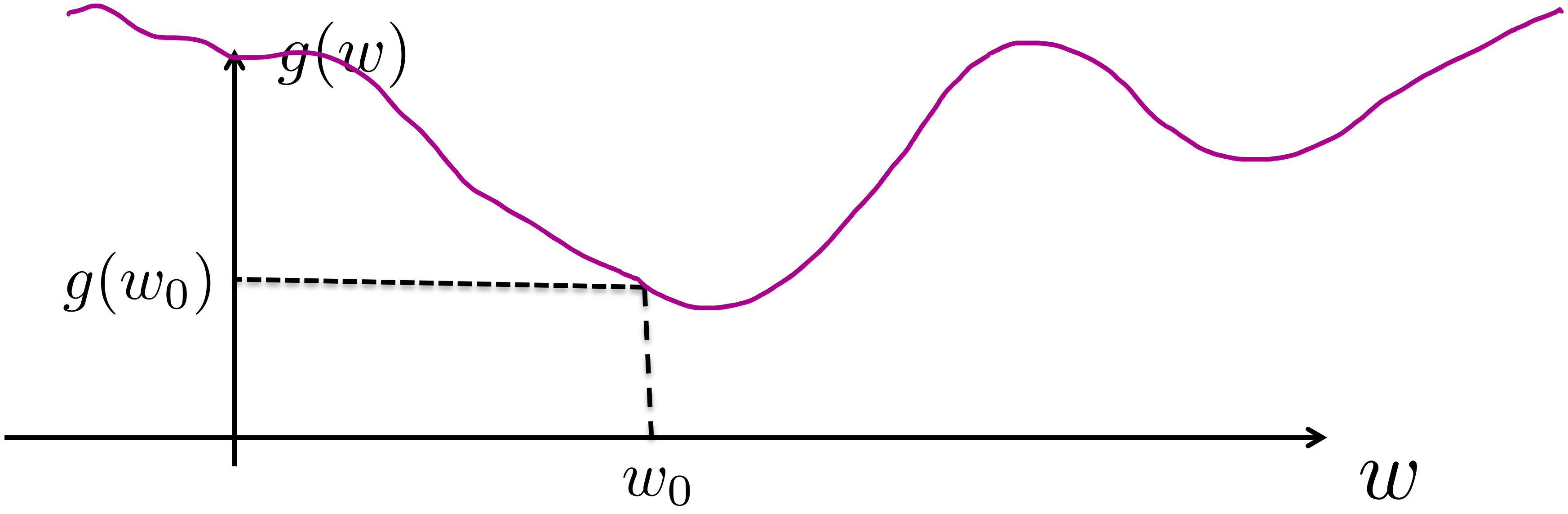
- What is the gradient of the function $g(x, y) = x^2 y$?
  - Recall: Gradient is a vector of partial derivatives with respect to each variable

$$\nabla g = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 2xy \\ \\ x^2 \end{bmatrix}$$

- What is the derivative of g(x, y) at x=0.5, y=0.5?

$$\nabla g|_{x=0.5, y=0.5} = \begin{bmatrix} 2(0.5)(0.5) \\ (0.5^2) \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.25 \end{bmatrix}$$

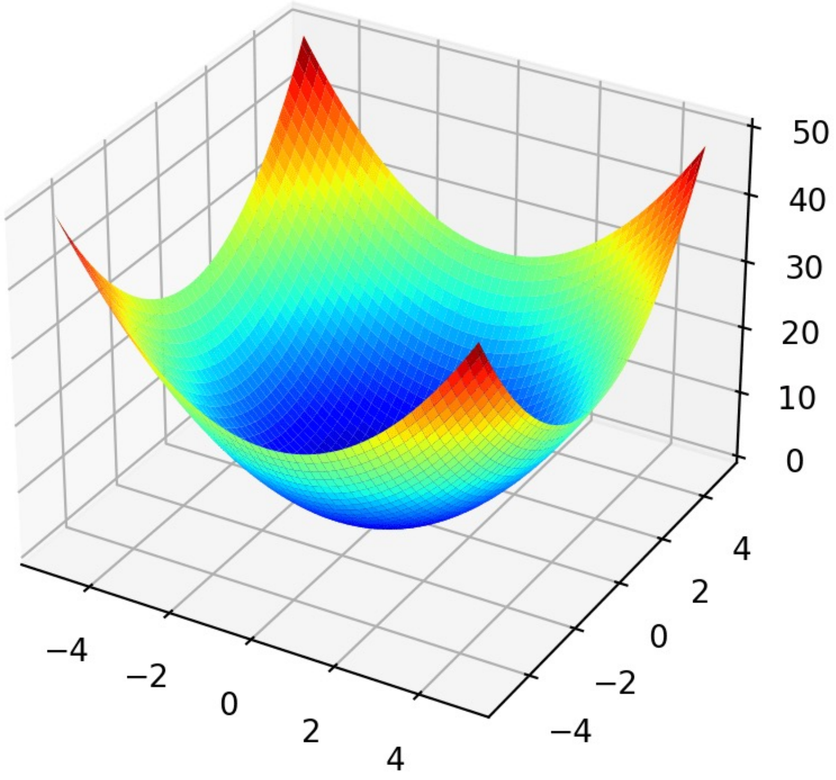
1-D Optimization



- Could evaluate $g(w_0 + h)$ and $g(w_0 - h)$
  - Then step in best direction

- Or, evaluate derivative: $$\frac{\partial g(w_0)}{\partial w} = \lim_{h \to 0} \frac{g(w_0 + h) - g(w_0 - h)}{2h}$$
  - Tells which direction to step into

2-D Optimization



https://machinelearningmastery.com/2d-test-functions-for-function-optimization/

Gradient descent

- Perform update in downhill direction for each coordinate
- The steeper the slope (i.e. the higher the derivative) the bigger the step for that coordinate
- E.g., consider: $g(w_1, w_2)$

  - Updates:

$$w_1 \leftarrow w_1 - \alpha * \frac{\partial g}{\partial w_1}(w_1, w_2)$$

$$w_2 \leftarrow w_2 - \alpha * \frac{\partial g}{\partial w_2}(w_1, w_2)$$

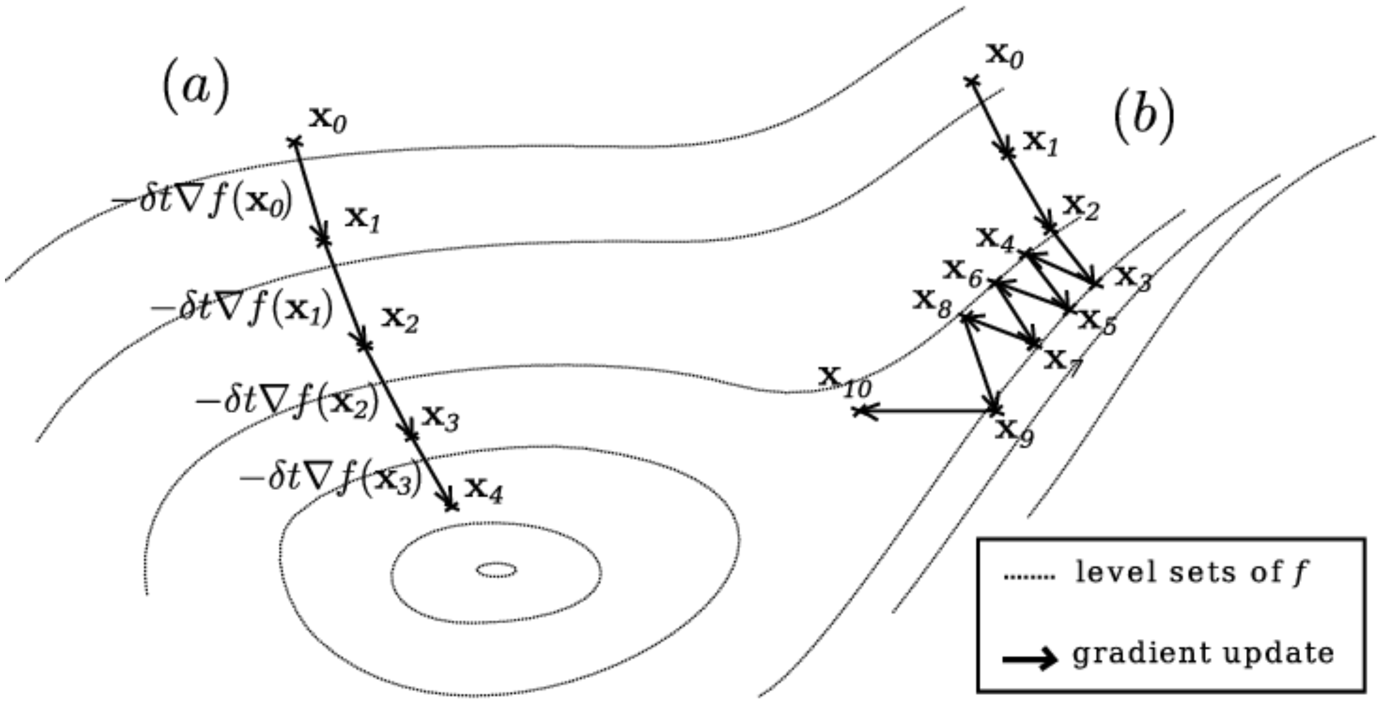
  - Updates in vector notation:

$$w \leftarrow w - \alpha * \nabla_w g(w)$$

with: $\nabla_w g(w) = \begin{bmatrix} \frac{\partial g}{\partial w_1}(w) \\ \frac{\partial g}{\partial w_2}(w) \end{bmatrix}$ **= gradient**

Gradient descent

- Idea:
  - Start somewhere
  - Repeat: Take a step in the gradient direction



https://ludovicarnold.com/teaching/optimization/gradient-descent/

Gradient in n dimensions

$$\nabla g = \begin{bmatrix} \frac{\partial g}{\partial w_1} \\ \frac{\partial g}{\partial w_2} \\ \cdots \\ \frac{\partial g}{\partial w_n} \end{bmatrix}$$

Optimization procedure: Gradient descent

- `init` w
- `for iter = 1, 2, …`

$$w \leftarrow w - \alpha * \nabla g(w)$$

- α : learning rate --- tweaking parameter that needs to be chosen carefully

Training Neural Networks

- Step 1: For each input in the training (sub)set *x*, predict a classification *y* using the current weights

$$\sigma(x \times W_{\text{layer 1}}) = h \qquad \sigma(h \times W_{\text{layer 2}}) = y$$

- Step 2: Compare predictions with the true *y* values, using a **loss function**
  - Higher value of loss function = bad model
  - Lower value of loss function = good model
  - Example: **zero-one loss**: count the number of misclassified inputs
  - Example: **log loss** (derived from maximum likelihood)
  - Example: **sum of squared errors** (more on this soon)
- Step 3: Use numerical method (e.g. gradient descent) to minimize loss
  - Loss is a function of the weights. Optimization goal: find weights that minimize loss

Optimization Procedure: Gradient Descent

- `init` w
- `for iter = 1, 2, …`

$$w \leftarrow w - \alpha \nabla \text{Loss}(w)$$

- α : learning rate --- tweaking parameter that needs to be chosen carefully

Computing Gradients

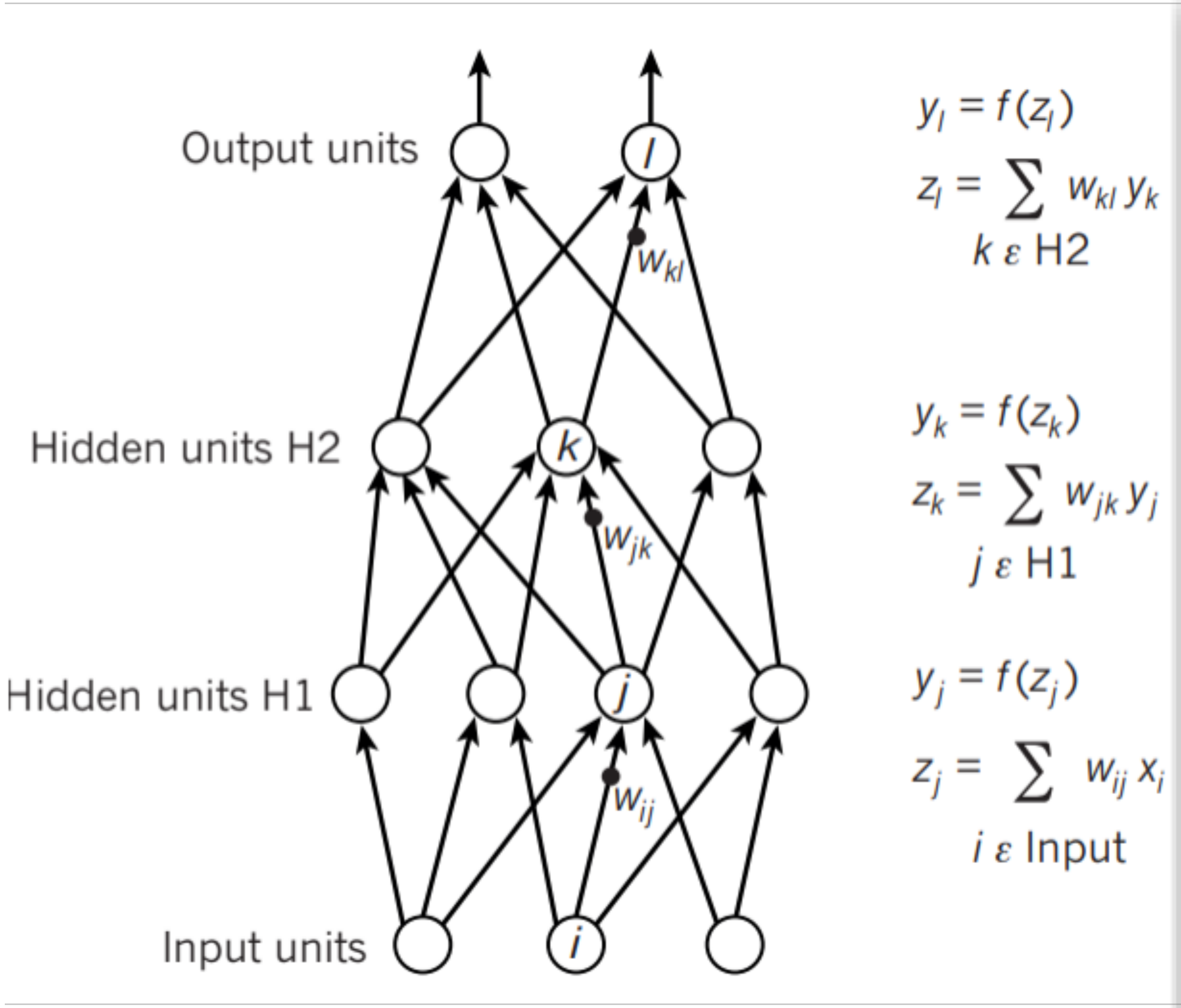
- How do we compute gradients of these loss functions?
  - Repeated application of the chain rule:

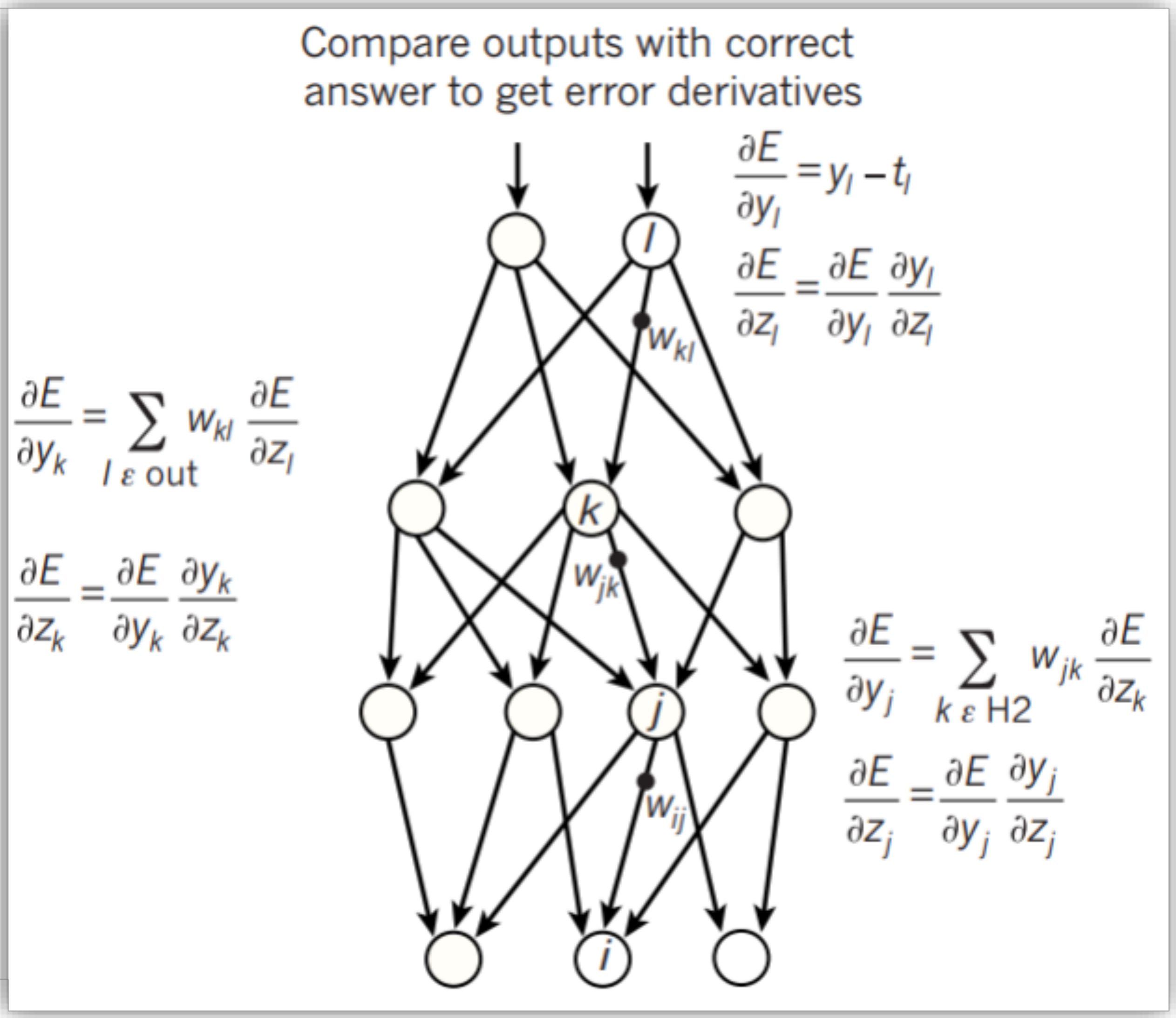
If $$f(x) = g(h(x))$$

Then $$f'(x) = g'(h(x))h'(x)$$

→ **Derivatives can be computed by following well-defined procedures**

Feed forward vs. Backpropagation

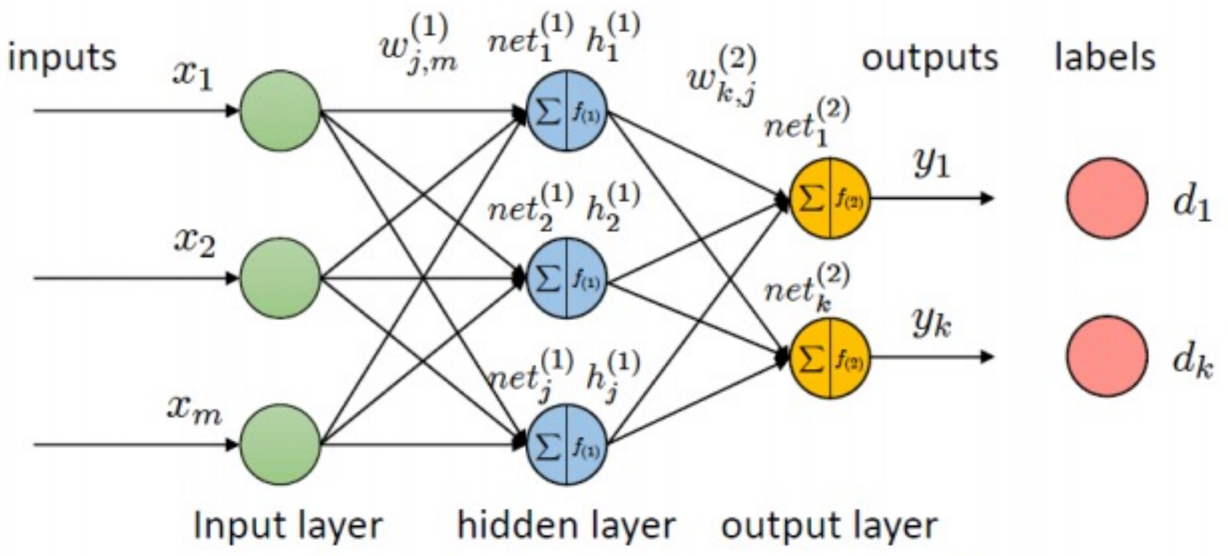




Backpropagation - demo

- Backpropagation demo

Make a prediction



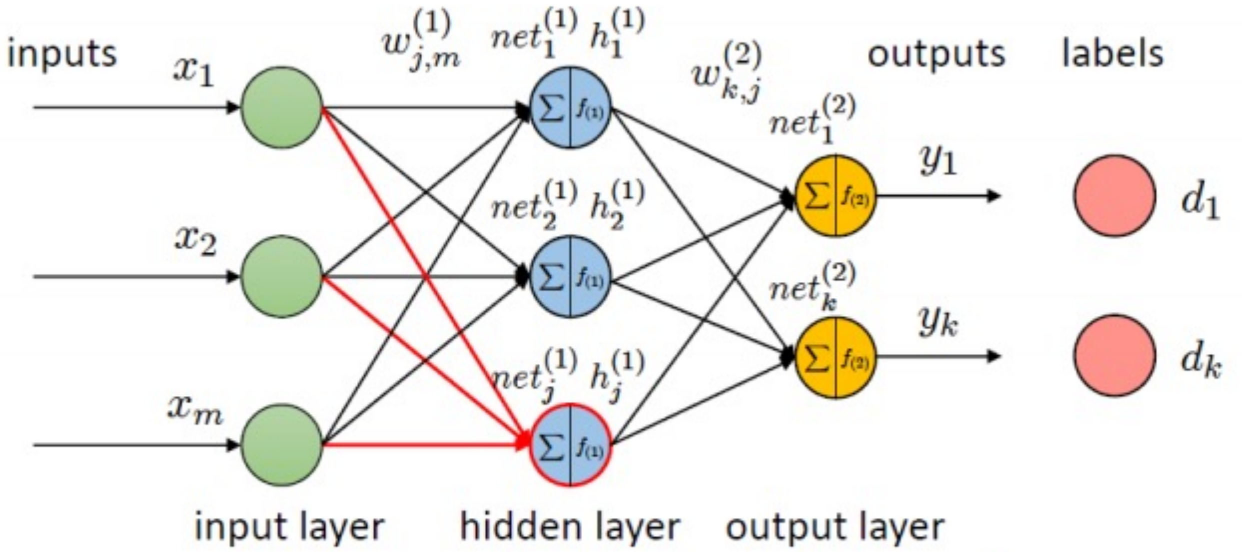
Two-layer feedforward neural network

Feed-forward prediction:

$$x = (x_1, \ldots, x_m) \xrightarrow{h^{(1)}_j = f_{(1)}(net^{(1)}_j) = f_{(1)}(\sum_m w^{(1)}_{j,m} x_m)} h^{(1)}_j \xrightarrow{y_k = f_{(2)}(net^{(2)}_k) = f_{(2)}(\sum_j w^{(1)}_{k,j} h^{(1)}_j)} y_k$$

where $net^{(1)}_j = \sum_m w^{(1)}_{j,m} x_m$ $\qquad$ $net^{(2)}_k = \sum_j w^{(2)}_{k,j} h^{(1)}_j$

Make a prediction (cont.)



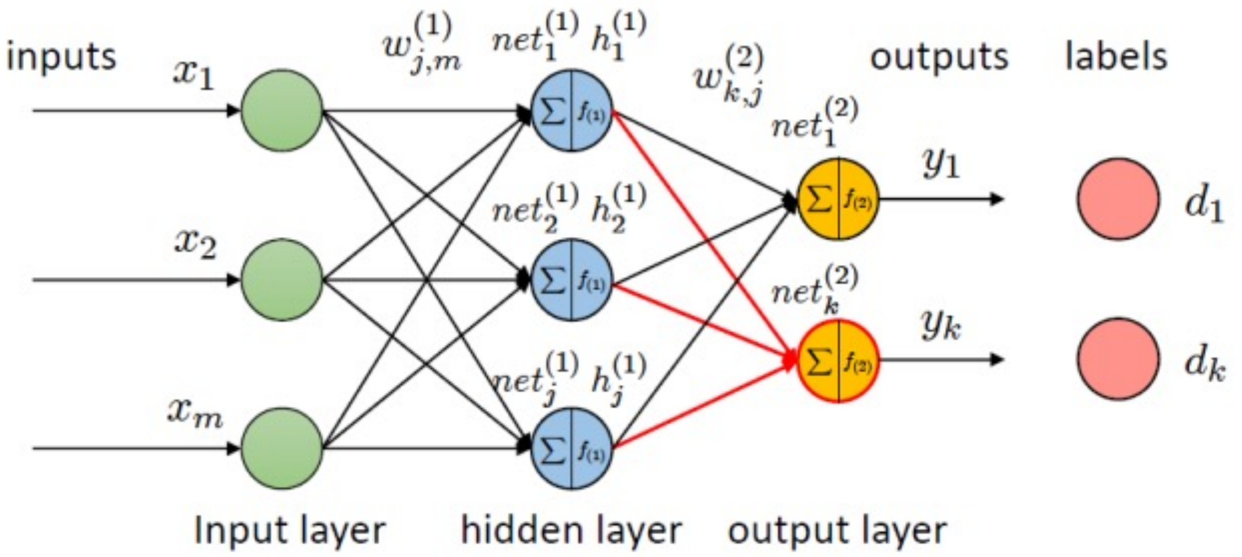
Two-layer feedforward neural network

Feed-forward prediction:

$$x = (x_1, \ldots, x_m) \xrightarrow{h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}(\sum_m w_{j,m}^{(1)} x_m)} h_j^{(1)} \xrightarrow{y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}(\sum_j w_{k,j}^{(1)} h_j^{(1)})} y_k$$

where $net_j^{(1)} = \sum_m w_{j,m}^{(1)} x_m$ $\qquad net_k^{(2)} = \sum_j w_{k,j}^{(2)} h_j^{(1)}$

Make a prediction (cont.)



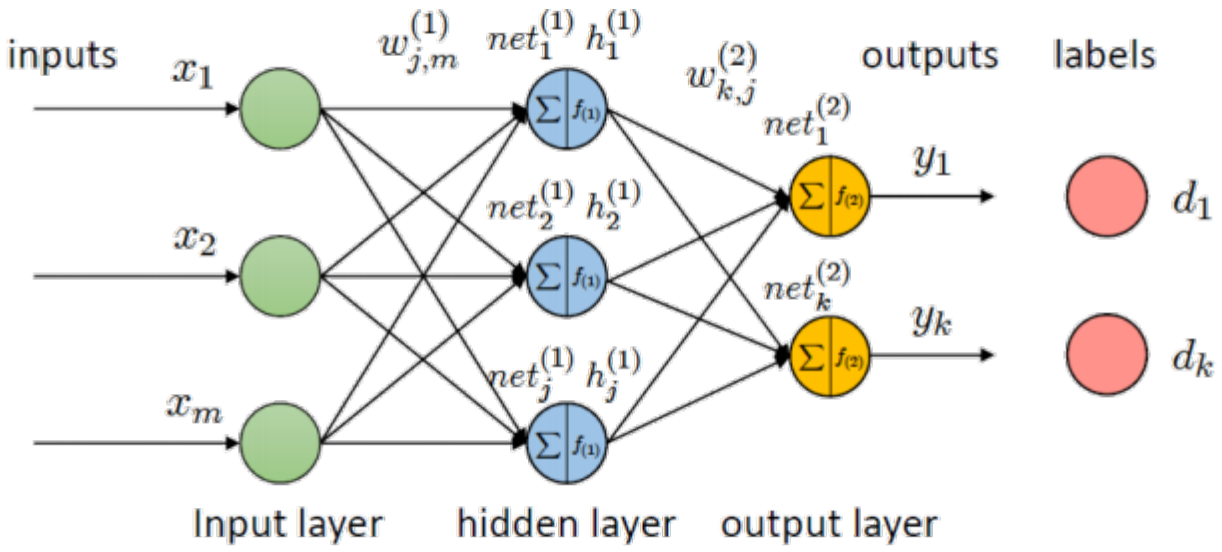
Two-layer feedforward neural network

Feed-forward prediction:

$$x = (x_1, \ldots, x_m) \xrightarrow{h^{(1)}_j = f_{(1)}(net^{(1)}_j) = f_{(1)}(\sum_m w^{(1)}_{j,m} x_m)} h^{(1)}_j \xrightarrow{y_k = f_{(2)}(net^{(2)}_k) = f_{(2)}(\sum_j w^{(1)}_{k,j} h^{(1)}_j)} y_k$$

where $\quad net^{(1)}_j = \sum_m w^{(1)}_{j,m} x_m \qquad net^{(2)}_k = \sum_j w^{(2)}_{k,j} h^{(1)}_j$

Backpropagation



Two-layer feedforward neural network

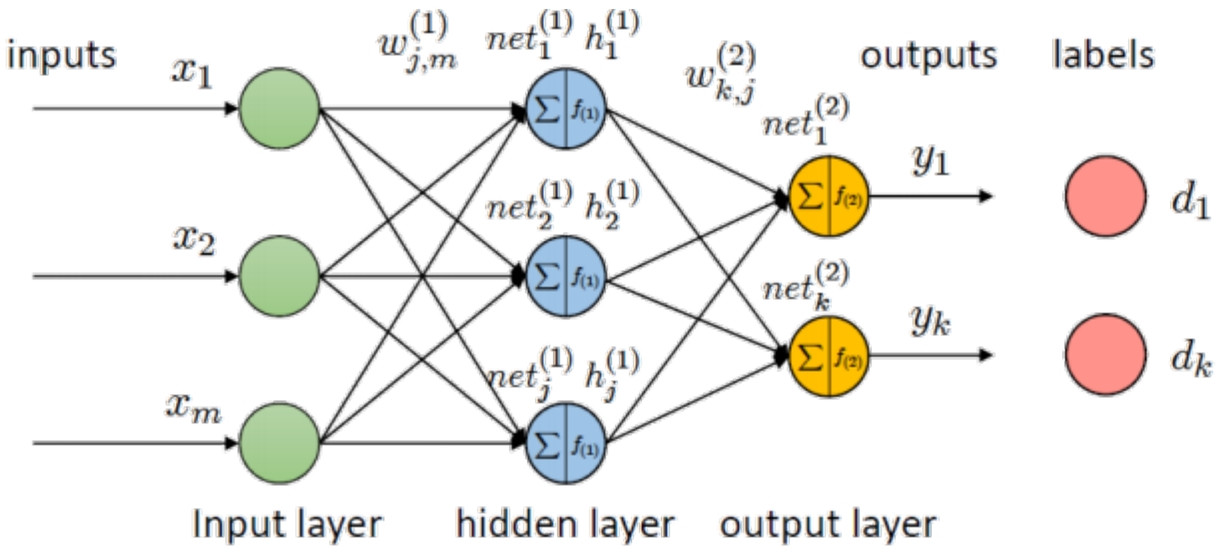
- Assume all the activation functions are sigmoid
- Error function $E = \frac{1}{2}\sum_k (y_k - d_k)^2$
- $\frac{\partial E}{\partial y_k} = y_k - d_k$
- $\frac{\partial y_k}{\partial w_{k,j}^{(2)}} = f'_{(2)}(net_k^{(2)})h_j^{(1)} = y_k(1-y_k)h_j^{(1)}$
- $\Rightarrow \frac{\partial E}{\partial w_{k,j}^{(2)}} = (y_k - d_k)y_k(1-y_k)h_j^{(1)}$
- $\Rightarrow w_{k,j}^{(2)} \leftarrow w_{k,j}^{(2)} - \eta(y_k - d_k)y_k(1-y_k)h_j^{(1)}$

Feed-forward prediction:

$$x = (x_1, \dots, x_m) \xrightarrow{h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}(\sum_m w_{j,m}^{(1)} x_m)} \boxed{h_j^{(1)} \xrightarrow{y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}(\sum_j w_{k,j}^{(2)} h_j^{(1)})} y_k}$$

where $\quad net_j^{(1)} = \sum_m w_{j,m}^{(1)} x_m \qquad \boxed{net_k^{(2)} = \sum_j w_{k,j}^{(2)} h_j^{(1)}}$

Backpropagation (cont.)



Two-layer feedforward neural network

- Error function $E = \frac{1}{2}\sum_k (y_k - d_k)^2$
- $\frac{\partial E}{\partial y_k} = y_k - d_k$
- $\frac{\partial y_k}{\partial h_j^{(1)}} = y_k(1-y_k)w_{k,j}^{(2)}$
- $\frac{\partial h_j^{(1)}}{\partial w_{j,m}^{(1)}} = f'_{(1)}\left(net_j^{(1)}\right)x_m = h_j^{(1)}\left(1-h_j^{(1)}\right)x_m$
- $\Rightarrow \frac{\partial E}{\partial w_{j,m}^{(1)}} = h_j^{(1)}\left(1-h_j^{(1)}\right)\sum_k w_{k,j}^{(2)}(y_k - d_k)y_k(1-y_k)x_m$
- $\Rightarrow w_{j,m}^{(1)} \leftarrow w_{j,m}^{(1)} - \eta h_j^{(1)}\left(1-h_j^{(1)}\right)\sum_k w_{k,j}^{(2)}(y_k - d_k)y_k(1-y_k)x_m$

Feed-forward prediction:

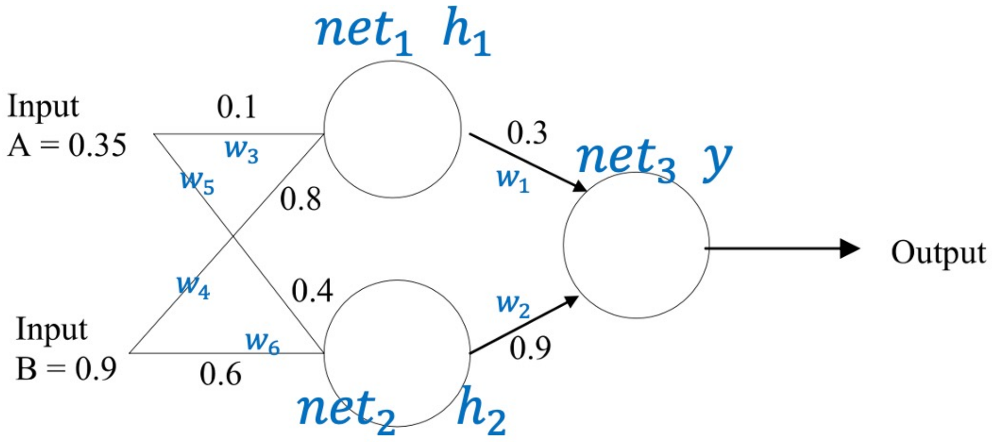
$$\boxed{h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}\Big(\sum_m w_{j,m}^{(1)}x_m\Big)} \qquad y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}\Big(\sum_j w_{k,j}^{(1)}h_j^{(1)}\Big)$$

$$x = (x_1,\ldots,x_m) \longrightarrow h_j^{(1)} \longrightarrow y_k$$

where $\quad \boxed{net_j^{(1)} = \sum_m w_{j,m}^{(1)}x_m} \qquad net_k^{(2)} = \sum_j w_{k,j}^{(2)}h_j^{(1)}$

Backpropagation - example

- Consider the simple network below:



- Assume that the neurons have sigmoid activation function
  - Perform a forward pass on the network and find the predicted output
  - Perform a reverse pass (training) once (target = 0.5) with $\eta = 1$
  - Perform a further forward pass and comment on the result

Backpropagation – example (cont.)

(i)

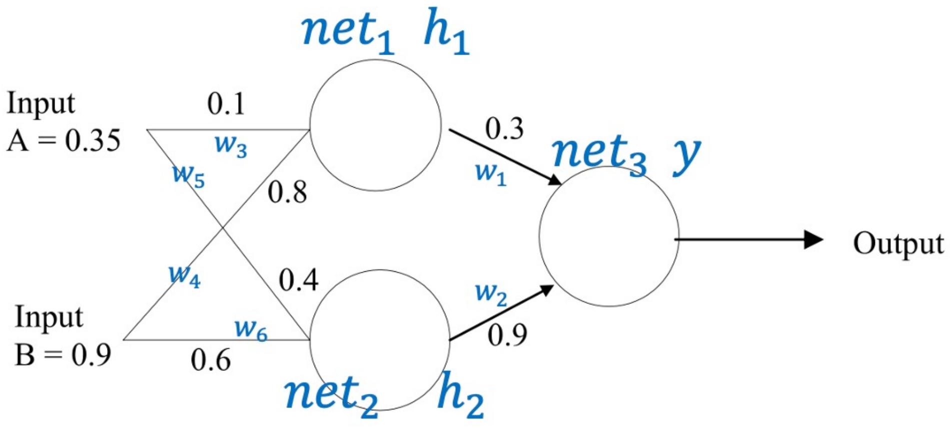
Input to top neuron = $(0.35 \times 0.1) + (0.9 \times 0.8) = 0.755$. Out = 0.68.
Input to bottom neuron = $(0.9 \times 0.6) + (0.35 \times 0.4) = 0.68$. Out = 0.6637.
Input to final neuron = $(0.3 \times 0.68) + (0.9 \times 0.6637) = 0.80133$. Out = 0.69.

(ii) New weights for output layer:

$$\begin{aligned} w1^+ &= w1 - (y-t)y(1-y)h_1 \\ &= 0.3 - (0.69 - 0.5) \times 0.69 \times (1 - 0.69) \times 0.68) = 0.272392. \\ w2^+ &= w2 - (y-t)y(1-y)h_2 \\ &= 0.9 - (0.69 - 0.5) \times 0.69 \times (1 - 0.69) \times 0.6637) = 0.87305. \end{aligned}$$

New weights for hidden layer:

$$\begin{aligned} w3^+ &= w3 - (y-t)y(1-y)w_1 h_1(1-h_1)A \\ &= ? \\ w4^+ &= ? \\ w5^+ &= ? \\ w6^+ &= ? \end{aligned}$$



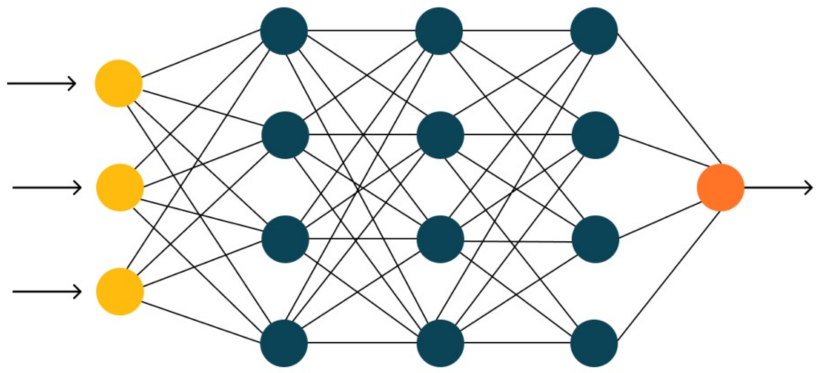
Fun Neural Net Demo Site

- Demo-site:
  - http://playground.tensorflow.org/

Modules in modern neural networks

Multi-layer perceptron (MLP)

- Denote the matrix multiplication operation with (W, b) as
  - $\mathrm{MM}_{W,b}(x) = Wx + b$

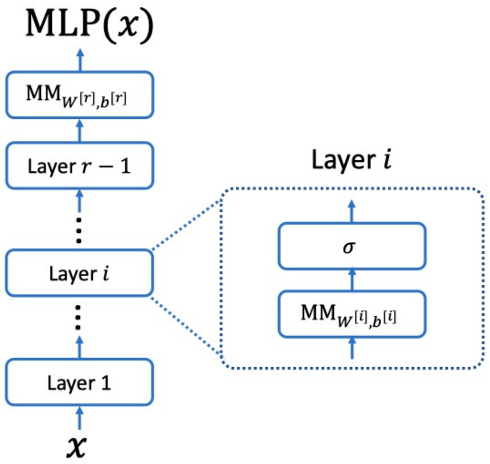


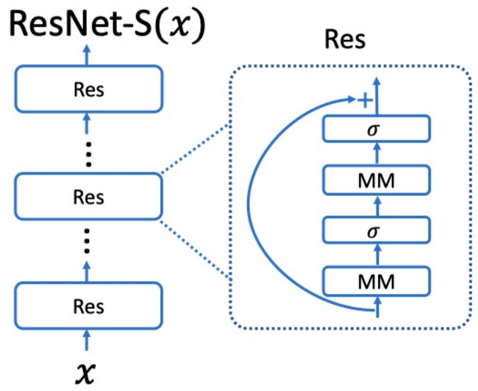
- Denote σ as the activation function
- Denote $W^{[r]}, b^{[r]}$ as the weight/bias of the r-th layer
- Then the MLP can be represented as

$$\mathrm{MLP}(x) = \mathrm{MM}_{W^{[r]},b^{[r]}}(\sigma(\mathrm{MM}_{W^{[r-1]},b^{[r-1]}}(\sigma(\cdots \mathrm{MM}_{W^{[1]},b^{[1]}}(x)))))$$

Residual connections

- An important network structure in CV: ResNet





- Residual connections

$$\mathrm{Res}(z) = z + \sigma(\mathrm{MM}(\sigma(\mathrm{MM}(z))))$$

- A much simplified ResNet (not the classic one)
  - Composition of many residual blocks followed by a matrix multiplication

$$\text{ResNet-S}(x) = \mathrm{MM}(\mathrm{Res}(\mathrm{Res}(\cdots \mathrm{Res}(x))))$$

Classic ResNet uses convolution layers instead of vanilla matrix multiplication, and adds batch normalization between convolutions and activations.

Residual connections (cont'd)

- Advantages of residual connections
  - Enable identity mapping, Improve the ability of model expression
  - Mitigate gradient disappearance, Ease training of deep networks

- Applications
  - Computer Vision (ResNet)
  - Natural Language Processing (Transformer encoder/decoder block)
  - Reinforcement Learning (policy/value networks)

Layer normalization

- Maps a vector to a more normalized vector
- A sub-module of the layer normalization $\longrightarrow$ $\text{LN-S}(z) = \begin{bmatrix} \frac{z_1 - \hat{\mu}}{\hat{\sigma}} \\ \frac{z_2 - \hat{\mu}}{\hat{\sigma}} \\ \vdots \\ \frac{z_m - \hat{\mu}}{\hat{\sigma}} \end{bmatrix}$
  - $\hat{\mu} = \frac{\sum_{i=1}^{m} z_i}{m}$ is the empirical mean of the vector
  - $\hat{\sigma} = \sqrt{\frac{\sum_{i=1}^{m} (z_i - \hat{\mu}^2)}{m}}$ is the empirical standard deviation
  - Intuition: normalized to having empirical mean zero and empirical standard deviation 1

Layer normalization (cont'd)

- More general mean and variance

$$\mathrm{LN}(z) = \beta + \gamma \cdot \mathrm{LN\text{-}S}(z) = \begin{bmatrix} \beta + \gamma\left(\frac{z_1 - \hat{\mu}}{\hat{\sigma}}\right) \\ \beta + \gamma\left(\frac{z_2 - \hat{\mu}}{\hat{\sigma}}\right) \\ \vdots \\ \beta + \gamma\left(\frac{z_m - \hat{\mu}}{\hat{\sigma}}\right) \end{bmatrix}$$

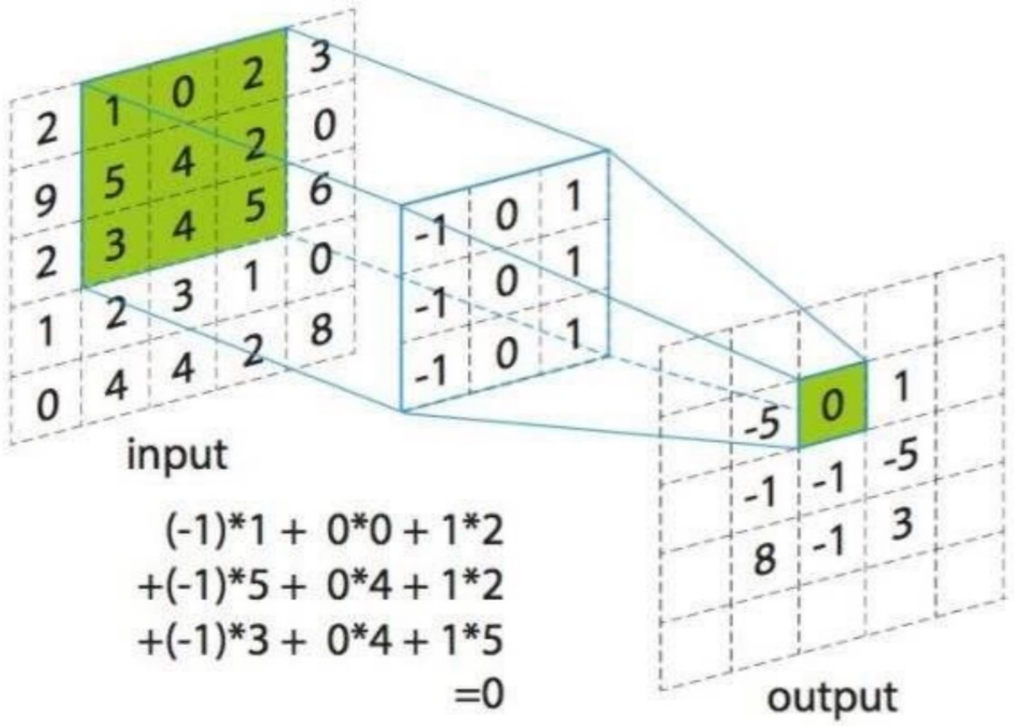
  - β, γ are learnable parameters
- Properties: Scaling-invariant

$$\mathrm{LN}(\mathrm{MM}_{\alpha W, \alpha b}(z)) = \mathrm{LN}(\mathrm{MM}_{W,b}(z)), \forall \alpha > 0.$$

- Applications
  - Transformer / BERT / GPT / RL policy networks

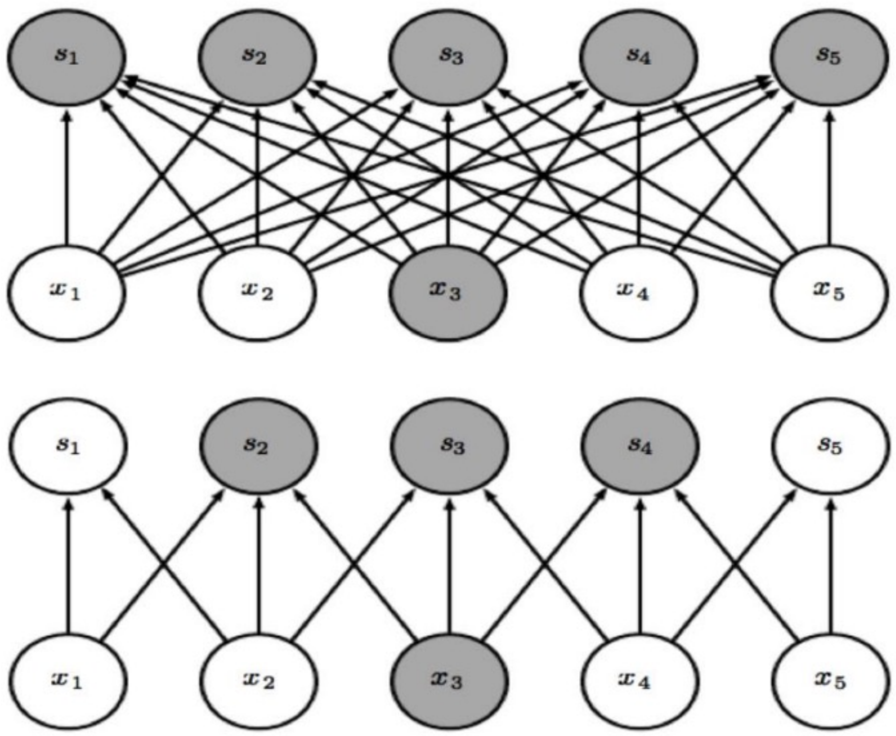
Convolutional layers

- Intuition
  - Given an input matrix (e.g. an image)
  - Use a small matrix (called filter or kernel) to screening the input at every position of the input matrix
  - Put the convolution results at corresponding positions



Convolutional layers (cont'd)

- Advantage
  - Sparse connections
  - Weight sharing



MLP
Edges: 5*5
Parameters: 5*5

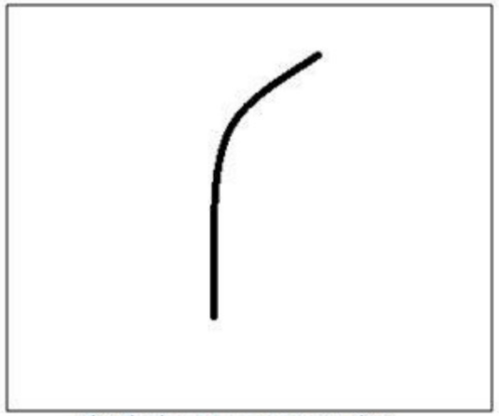
Convolution
Edges: 3*3+2*2
Parameters: 3

Interpretation of convolution

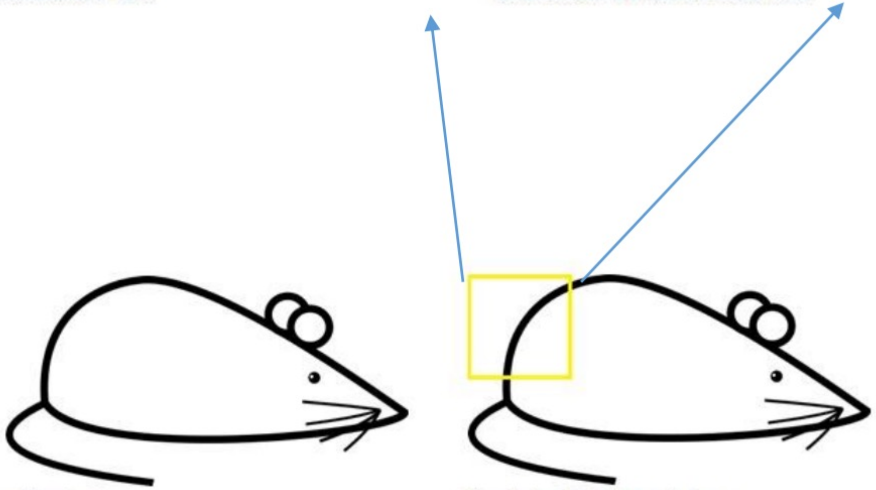
- Convolution can be used to find an area with particular patterns
- Example
  - The filter in the left represents the edge in the right, which is the back of a mouse

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter



Original image

Visualization of the filter on the image

Interpretation of convolution (cont’d)

- When the filter moves to the back of the mouse, the convolution operation will generate a very large value

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

- Otherwise, it generates a very small value

Visualization of the filter on the image

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = 0

Summary

- History of artificial neural nets
- Perceptron
- Multilayer perceptron networks
- Activation functions
- Training: backpropagation
- Modules in modern neural networks