Machine Learning Lecture 3

Neural Networks

Qian Ma
Sun Yat-sen University
Spring Semester, 2020

Acknowledgement

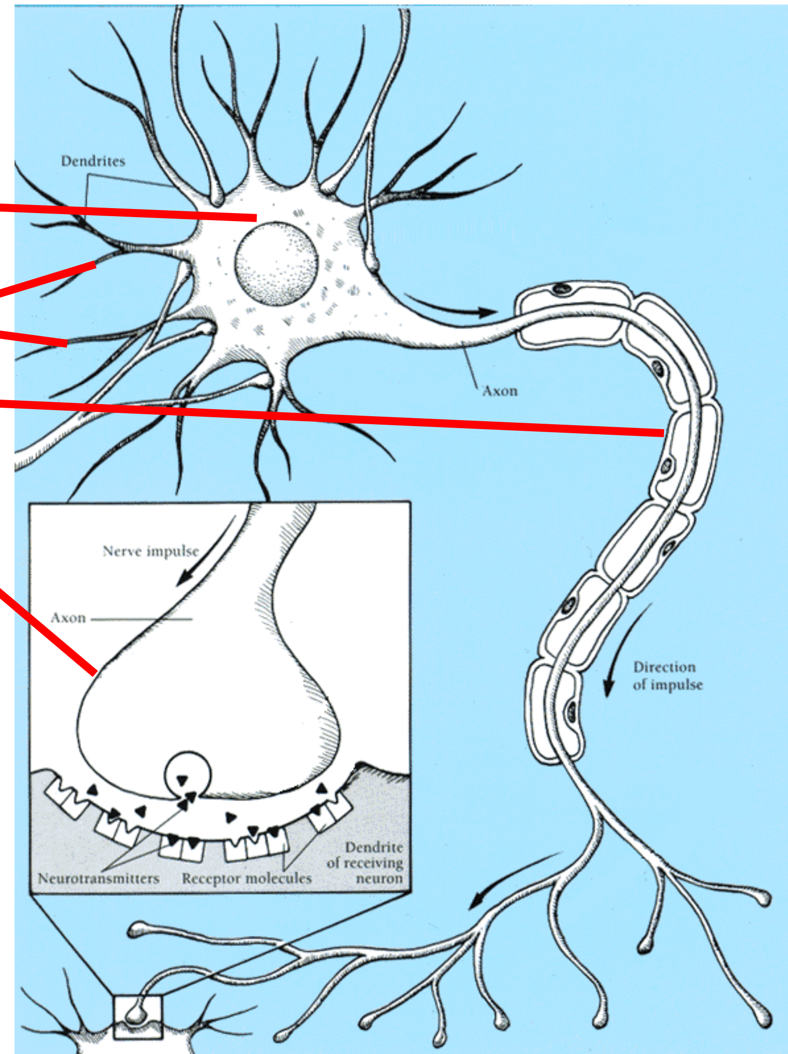
- A large part of slides in this lecture are originally from
  - Prof. Andrew Ng (Stanford University)
  - Prof. Weinan Zhang (Shanghai Jiao Tong University)

Prof. Andrew Ng
Stanford University

Prof. Weinan Zhang
Shanghai Jiao Tong University

Real Neurons

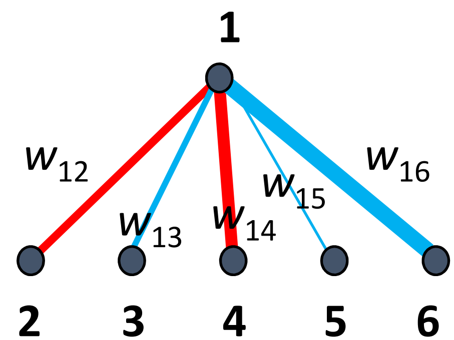
- Cell structures
  - Cell body
  - Dendrites
  - Axon
  - Synaptic terminals



Slides credit: Ray Mooney

Artificial Neuron Model

- Model network as a graph with cells as nodes and synaptic connections as weighted edges from node *i* to node *j*, w_{ji}



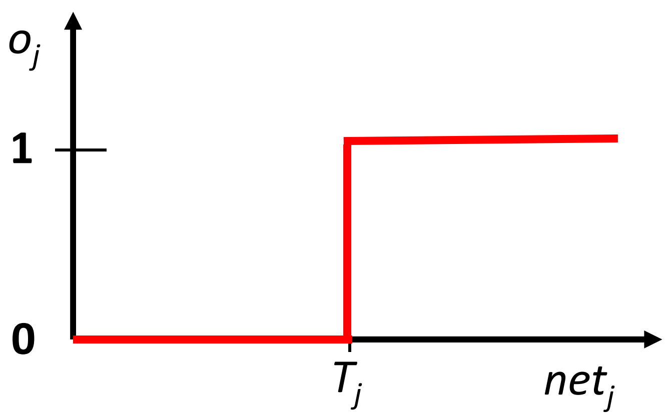
- Model net input to cell as

$$\text{net}_j = \sum_i w_{ji} o_i$$

- Cell output is

$$o_j = \begin{cases} 0 & \text{if net}_j < T_j \\ 1 & \text{if net}_j \geq T_j \end{cases}$$

(T_j is threshold for unit *j*)

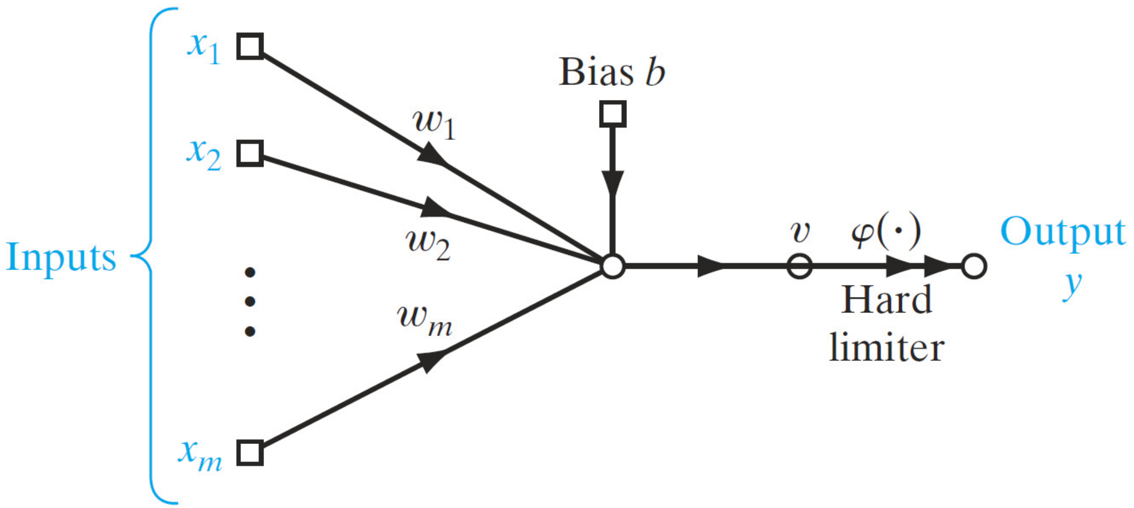


McCulloch and Pitts [1943]

Slides credit: Ray Mooney

Perceptron Model

- Rosenblatt's single layer perceptron [1958]



- Rosenblatt [1958] further proposed the *perceptron* as the first model for learning with a teacher (i.e., supervised learning)
- Focused on how to find appropriate weights w_m for two-class classification task
  - y = 1: class one
  - y = -1: class two

- Prediction

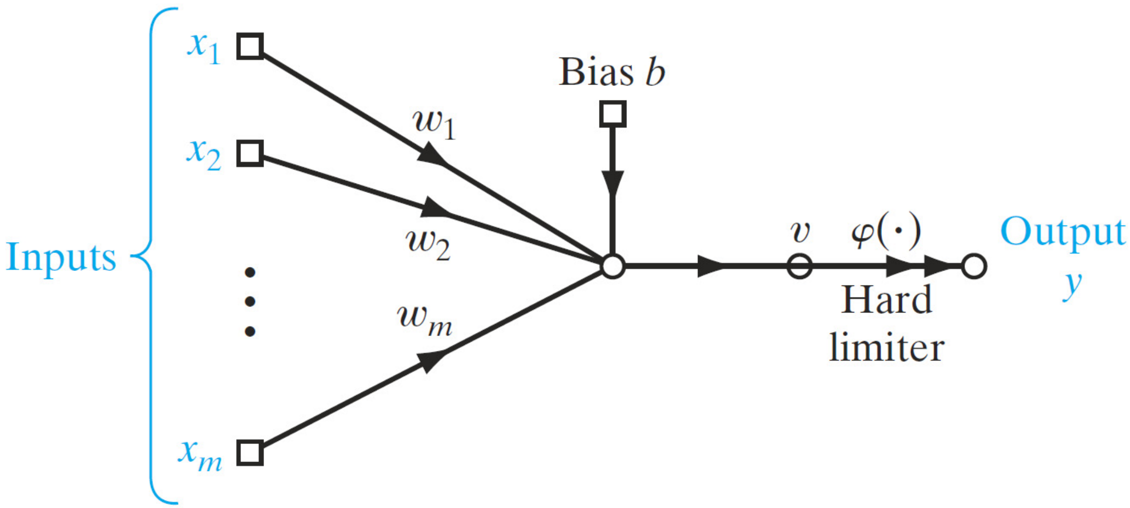
$$\hat{y} = \varphi\left(\sum_{i=1}^{m} w_i x_i + b\right)$$

- Activation function

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

Training Perceptron

- Rosenblatt's single layer perceptron [1958]



- Training

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

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

- Equivalent to rules:
  - If output is correct, do nothing
  - If output is high, lower weights on active inputs
  - If output is low, increase weights on active inputs

- Prediction

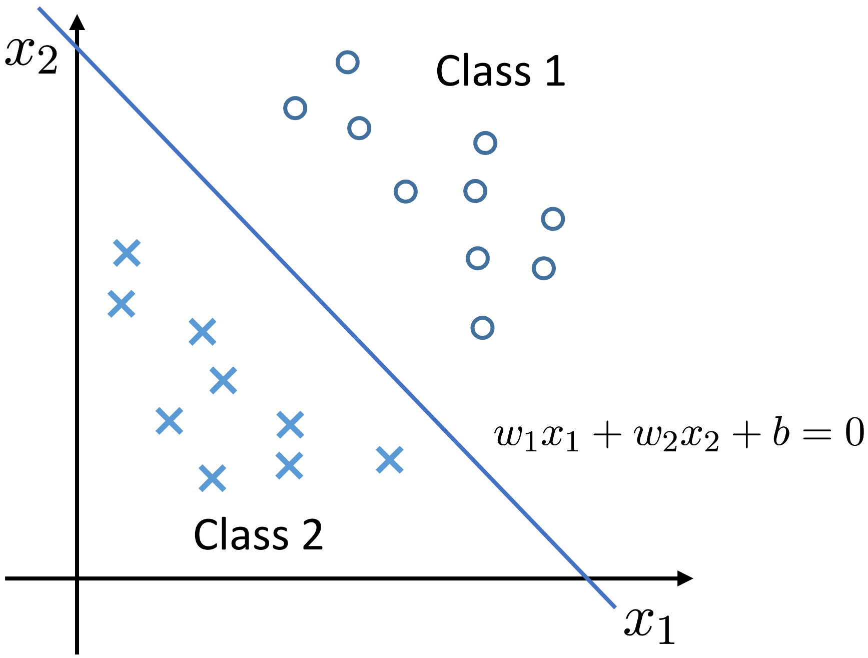
$$\hat{y} = \varphi\left(\sum_{i=1}^{m} w_i x_i + b\right)$$

- Activation function

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

Properties of Perceptron

- Rosenblatt’s single layer perceptron [1958]

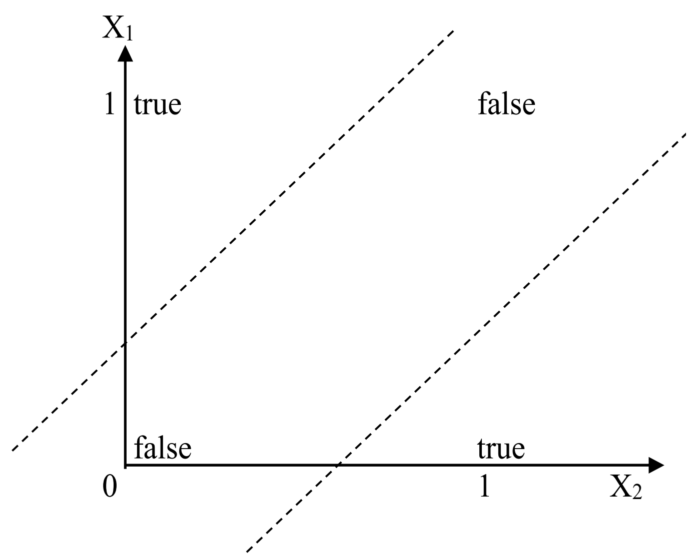


- Rosenblatt proved the convergence of a learning algorithm if two classes said to be linearly separable (i.e., patterns that lie on opposite sides of a hyperplane)
- Many people hoped that such a machine could be the basis for artificial intelligence

Properties of Perceptron

- The XOR problem

Input x		Output y
X_1	X_2	X_1 XOR X_2
0	0	0
0	1	1
1	0	1
1	1	0

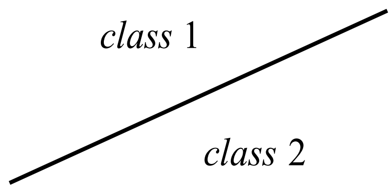


XOR is non linearly separable: These two classes (true and false) cannot be separated using a line.

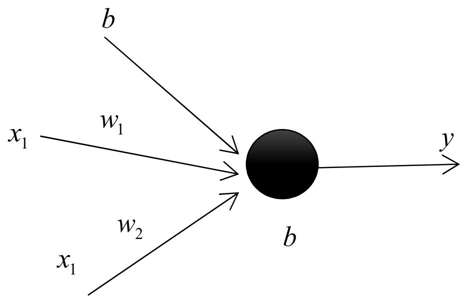
- However, Minsky and Papert [1969] showed that some rather elementary computations, such as *XOR* problem, could not be done by Rosenblatt's one-layer perceptron
- However Rosenblatt believed the limitations could be overcome if more layers of units to be added, but no learning algorithm known to obtain the weights yet
- Due to the lack of learning algorithms people left the neural network paradigm for almost 20 years

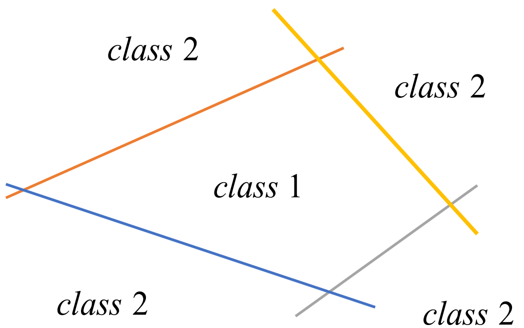
Hidden Layers and Backpropagation (1986~)

- Adding hidden layer(s) (internal presentation) allows to learn a mapping that is not constrained by linearly separable

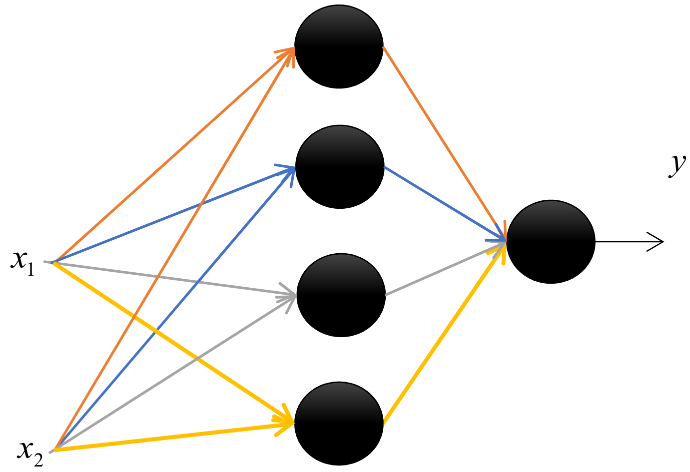


decision boundary: $x_1 w_1 + x_2 w_2 + b = 0$



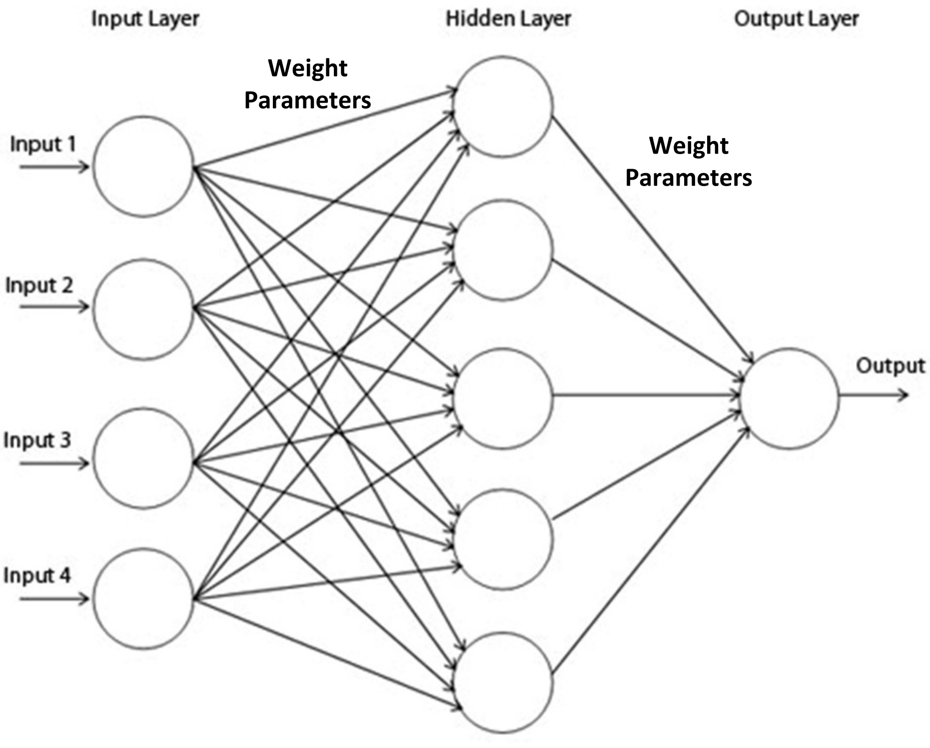


Each hidden node realizes one of the lines bounding the convex region



Hidden Layers and Backpropagation (1986~)

- Feedforward: massages move forward from the input nodes, through the hidden nodes (if any), and to the output nodes. There are no cycles or loops in the network

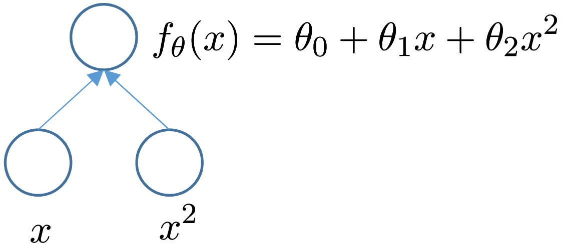


Two-layer feedforward neural network

Single / Multiple Layers of Calculation

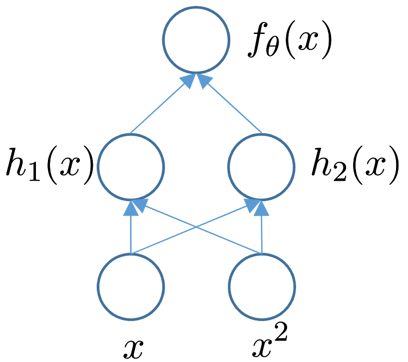
- Single layer function

$$f_\theta(x) = \sigma(\theta_0 + \theta_1 x + \theta_2 x^2)$$



- Multiple layer function

$$h_1(x) = \tanh(\theta_0 + \theta_1 x + \theta_2 x^2)$$
$$h_2(x) = \tanh(\theta_3 + \theta_4 x + \theta_5 x^2)$$
$$f_\theta(x) = f_\theta(h_1(x), h_2(x)) = \sigma(\theta_6 + \theta_7 h_1 + \theta_8 h_2)$$



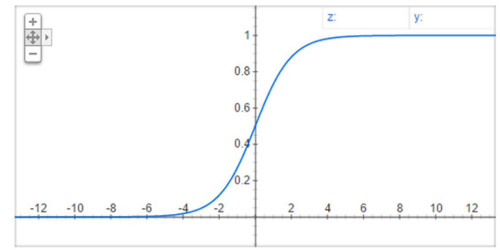
  - With non-linear activation function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Non-linear Activation Functions

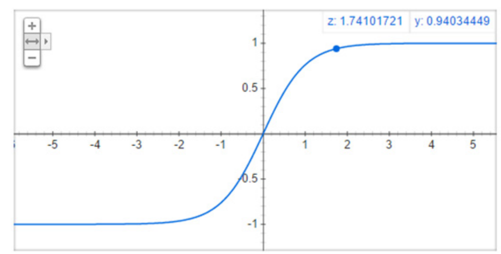
- Sigmoid

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



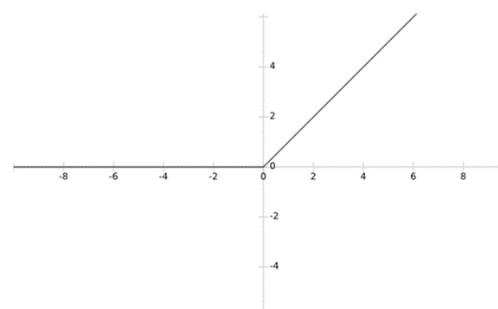
- Tanh

$$\tanh(z) = \frac{1-e^{-2z}}{1+e^{-2z}}$$



- Rectified Linear Unit (ReLU)

$$\mathrm{ReLU}(z) = \max(0, z)$$



Universal Approximation Theorem

- A feed-forward network with a single hidden layer containing a finite number of neurons (i.e., a multilayer perceptron), can approximate continuous functions

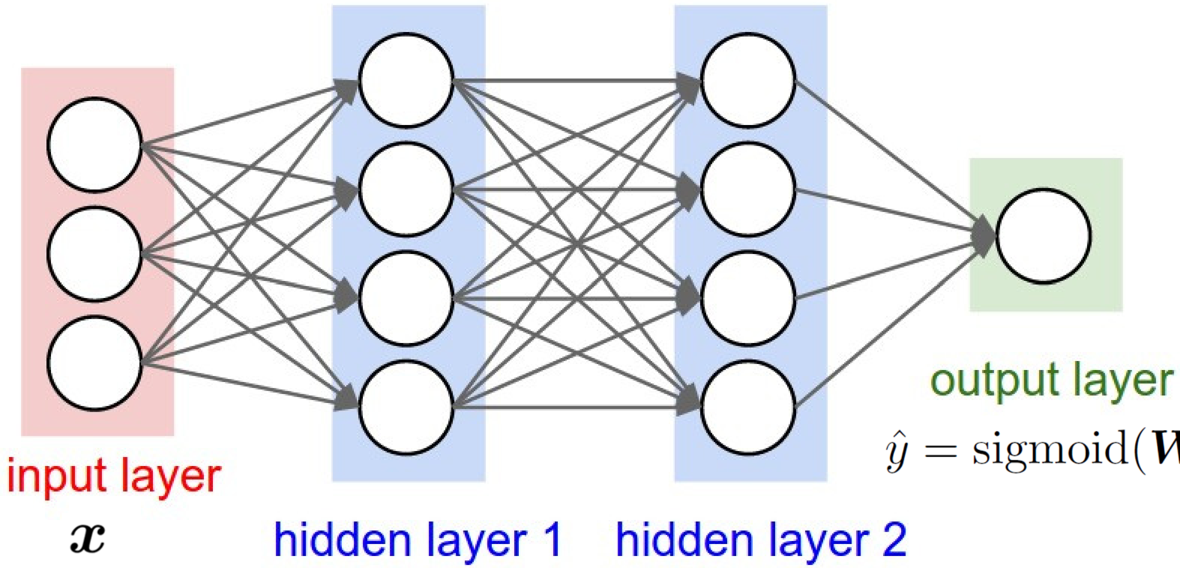
  - on compact subsets of $\mathbb{R}^n$

  - under mild assumptions on the activation function
    - Such as Sigmoid, Tanh and ReLU

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

Universal Approximation

- Multi-layer perceptron approximate any continuous functions on compact subset of $\mathbb{R}^n$



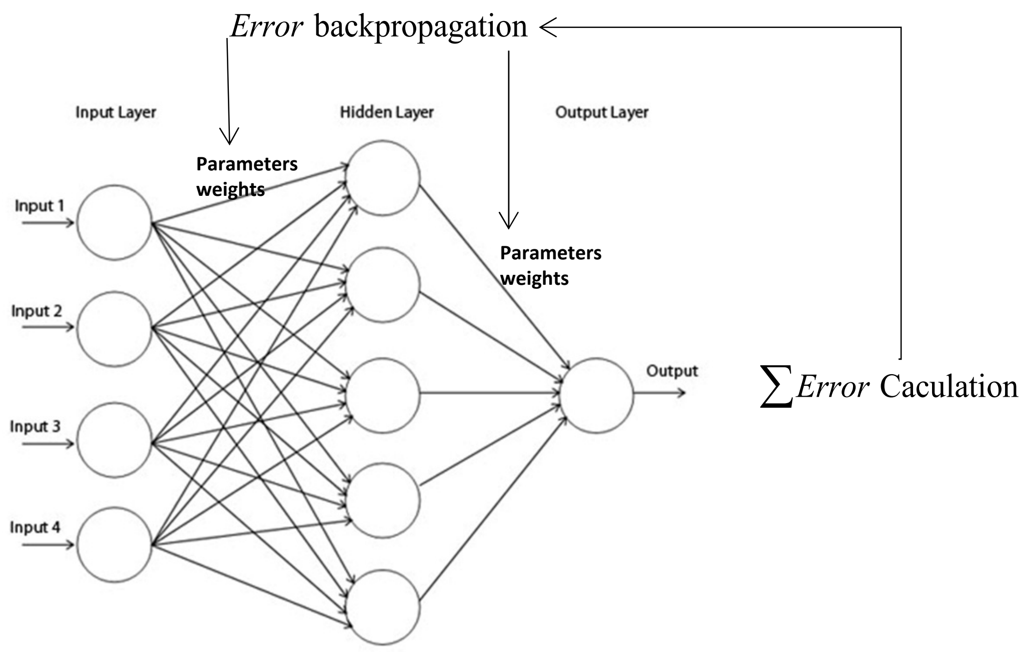
$$\hat{y} = \mathrm{sigmoid}(\boldsymbol{W}_3 \boldsymbol{l}_2 + b_3)$$

$$\boldsymbol{l}_1 = \tanh(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) \quad \boldsymbol{l}_2 = \tanh(\boldsymbol{W}_2 \boldsymbol{l}_1 + \boldsymbol{b}_2)$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

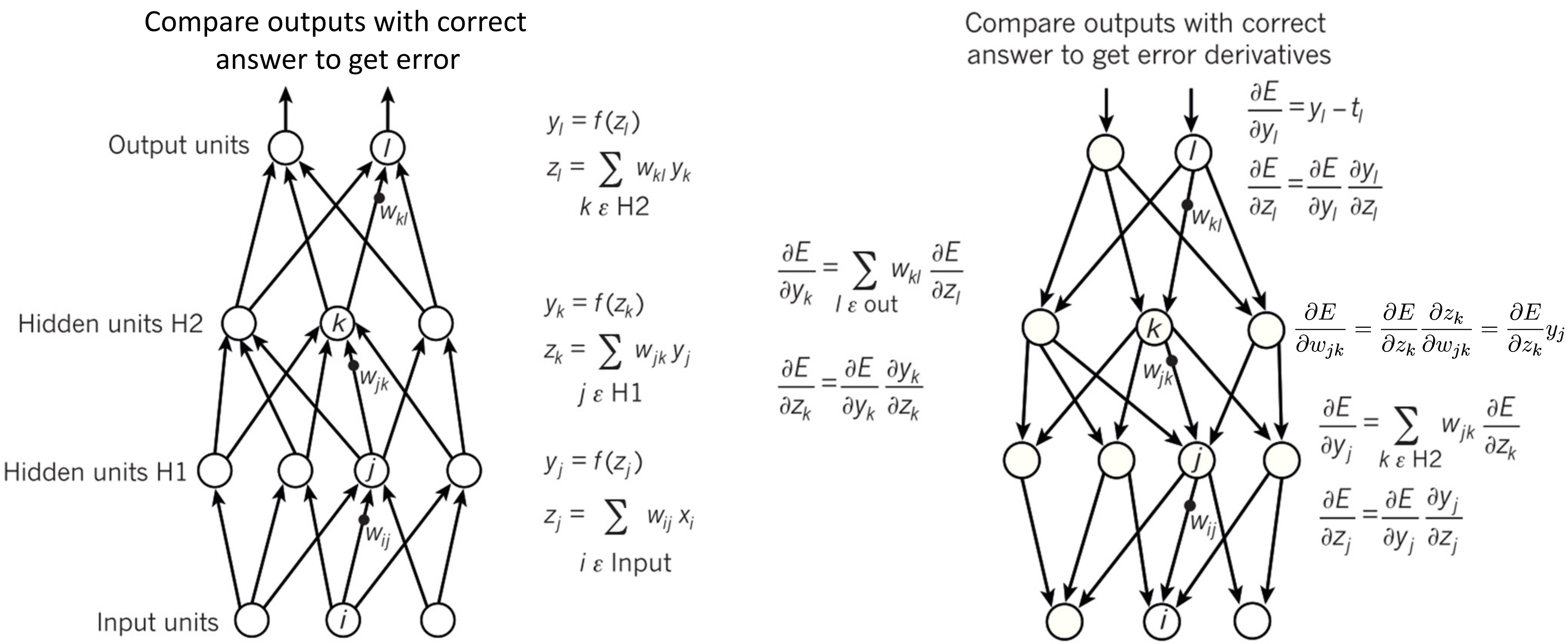
Hidden Layers and Backpropagation (1986~)

- One of the efficient algorithms for multi-layer neural networks is the *Backpropagation* algorithm
- It was re-introduced in 1986 and Neural Networks regained the popularity



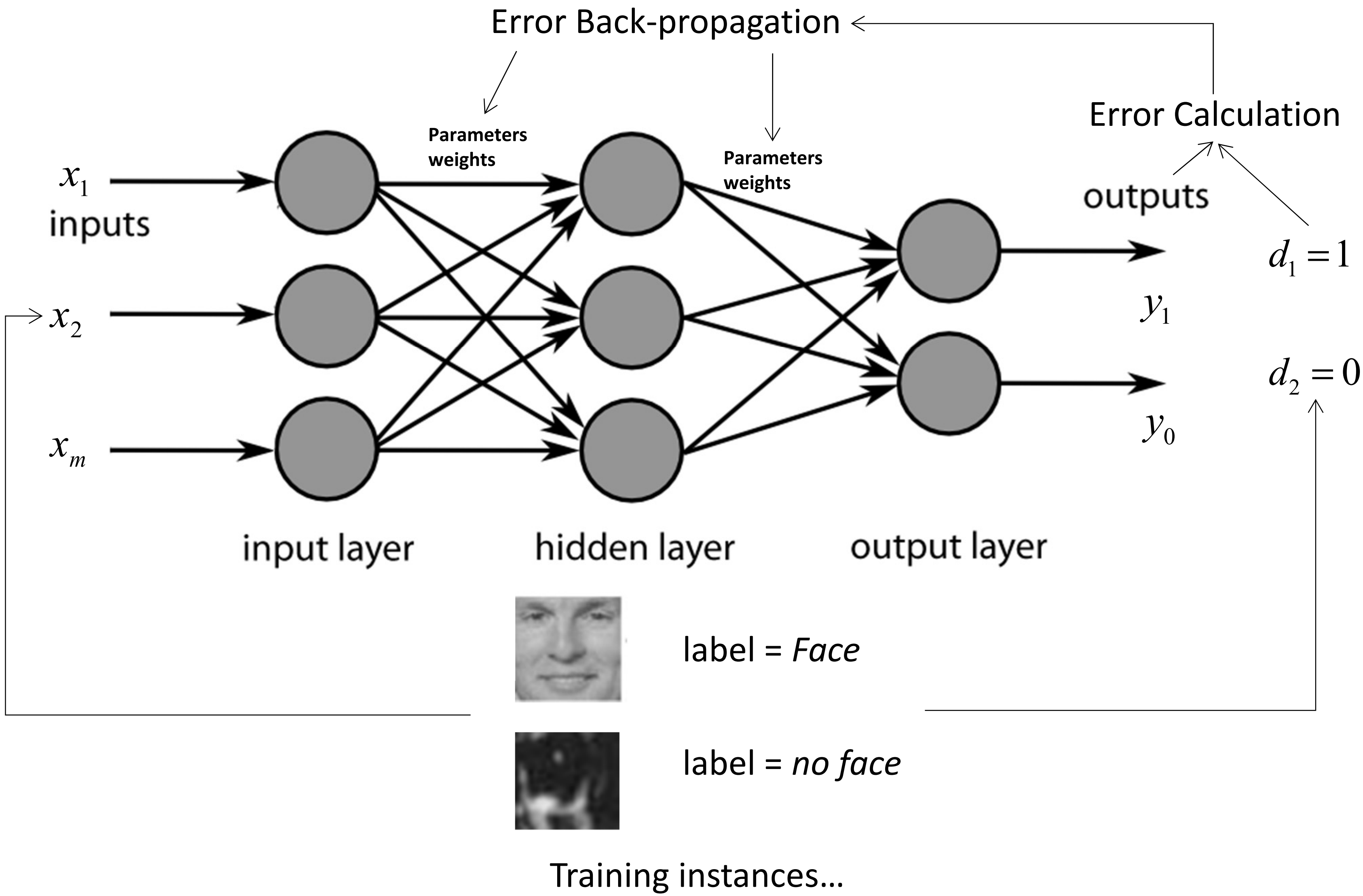
Note: *backpropagation* appears to be found by Werbos [1974]; and then independently rediscovered around 1985 by Rumelhart, Hinton, and Williams [1986] and by Parker [1985]

Learning NN by Back-Propagation

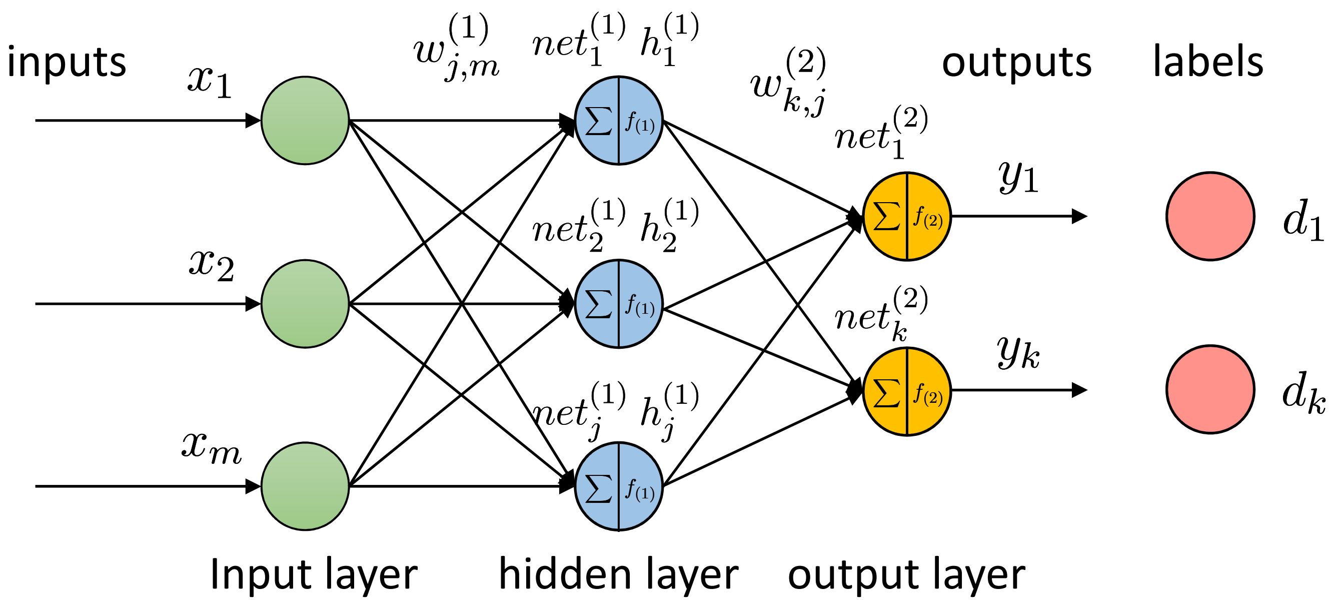


[LeCun, Bengio and Hinton. Deep Learning. Nature 2015.]

Learning NN by Back-Propagation



Make a Prediction



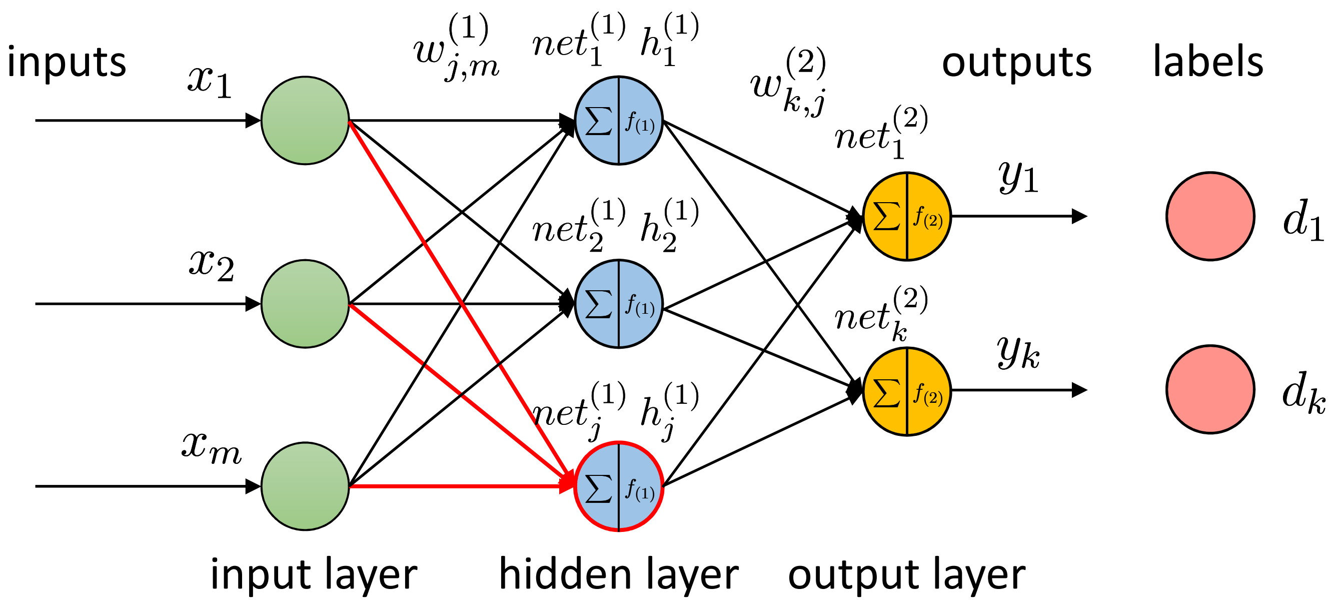
Two-layer feedforward neural network

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



Two-layer feedforward neural network

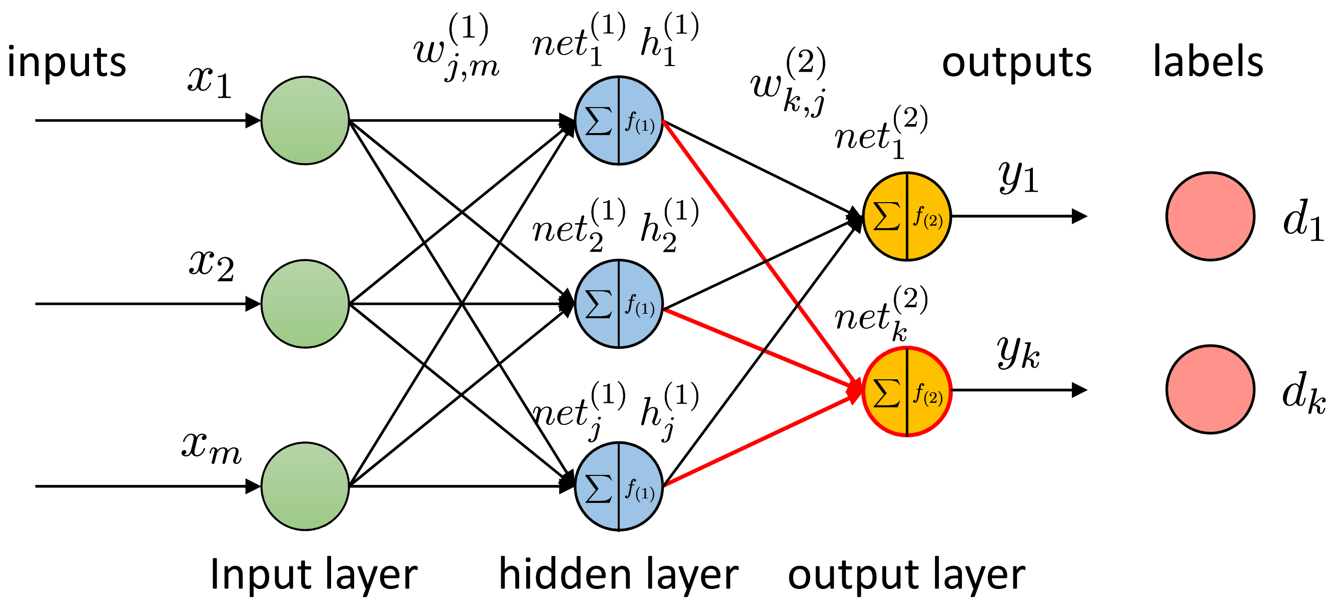
Feed-forward prediction:

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

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

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



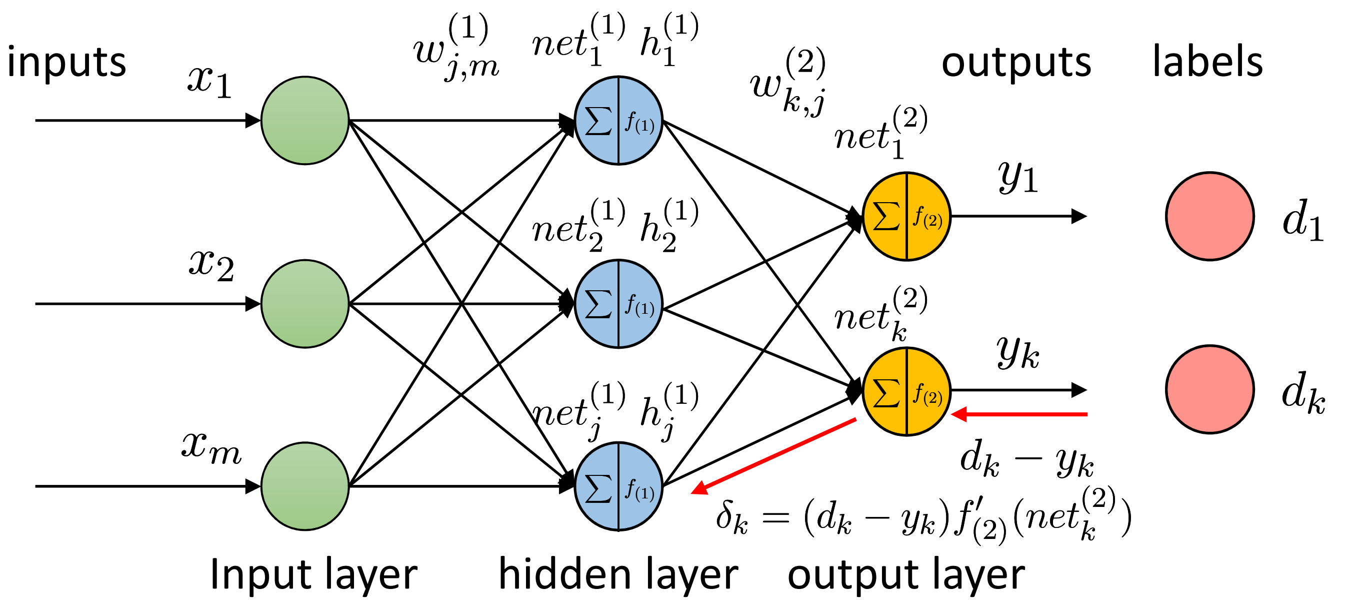
Two-layer feedforward neural network

Feed-forward prediction:

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

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

When Backprop/Learn Parameters



Two-layer feedforward neural network

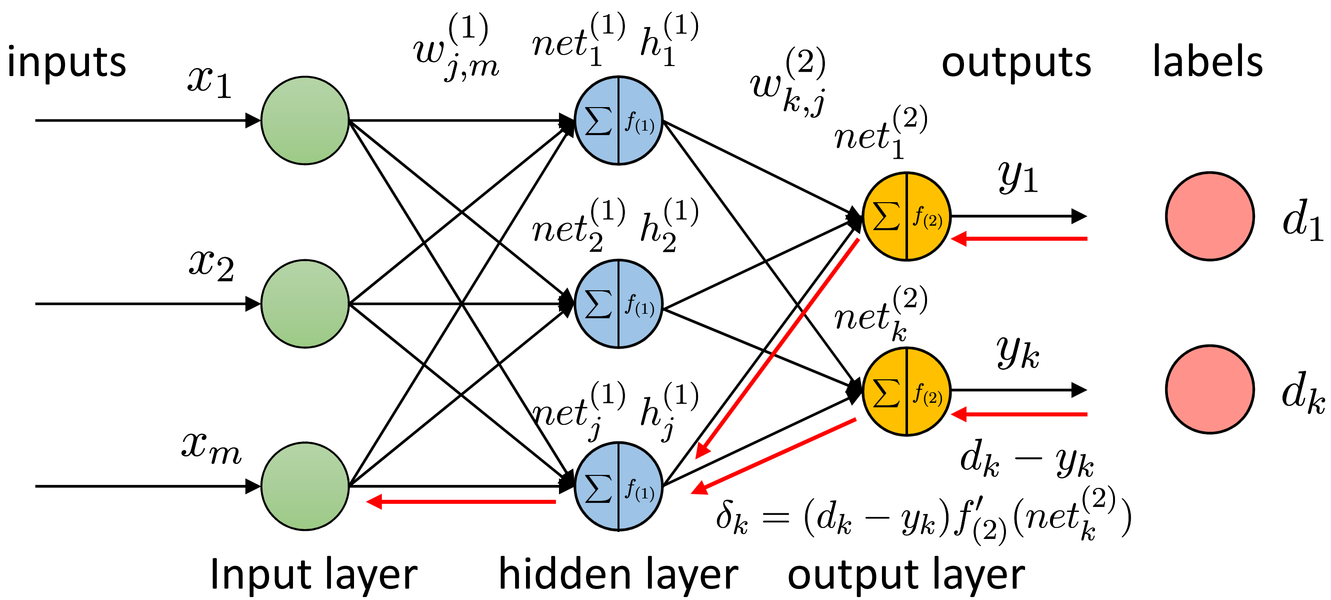
Notations: $net^{(1)}_j = \sum_m w^{(1)}_{j,m} x_m$ $\qquad net^{(2)}_k = \sum_j w^{(2)}_{k,j} h_j$

Backprop to learn the parameters

$$\boxed{w^{(2)}_{k,j} = w^{(2)}_{k,j} + \Delta w^{(2)}_{k,j}} \xleftarrow{\Delta w^{(2)}_{k,j} = \eta Error_k Output_j = \eta \delta_k h^{(1)}_j} \qquad E(W) = \frac{1}{2}\sum_k (y_k - d_k)^2$$

$$\Delta w^{(2)}_{k,j} = -\eta \frac{\partial E(W)}{\partial w^{(2)}_{k,j}} = -\eta (y_k - d_k) \frac{\partial y_k}{\partial net^{(2)}_k} \frac{\partial net^{(2)}_k}{\partial w^{(2)}_{k,j}} = \eta (d_k - y_k) f'_{(2)}(net^{(2)}_k) h^{(1)}_j = \eta \delta_k h^{(1)}_j$$

When Backprop/Learn Parameters



Two-layer feedforward neural network

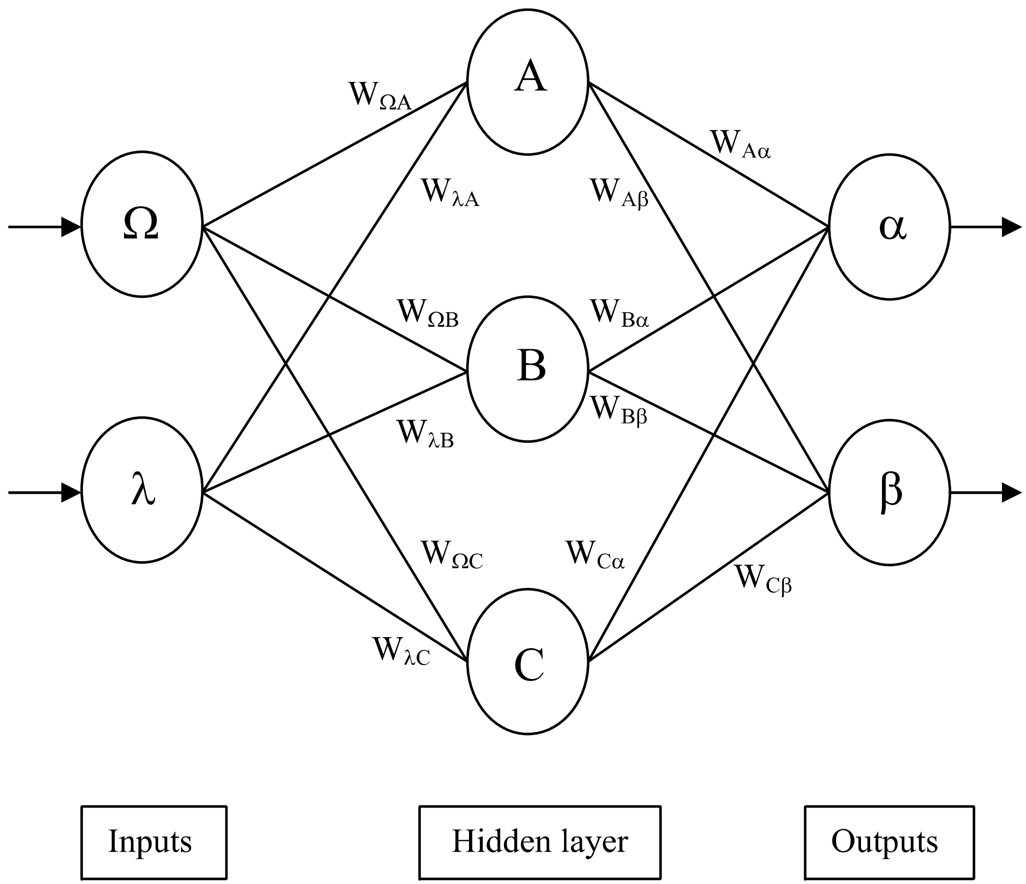
Notations: $net^{(1)}_j = \sum_m w^{(1)}_{j,m} x_m$ $\qquad net^{(2)}_k = \sum_j w^{(2)}_{k,j} h_j$

Backprop to learn the parameters

$$\boxed{w^{(1)}_{j,m} = w^{(1)}_{j,m} + \Delta w^{(1)}_{j,m}} \xleftarrow{\Delta w^{(2)}_{k,j} = \eta Error_j Output_m = \eta \delta_j x_m} \quad E(W) = \frac{1}{2}\sum_k (y_k - d_k)^2$$

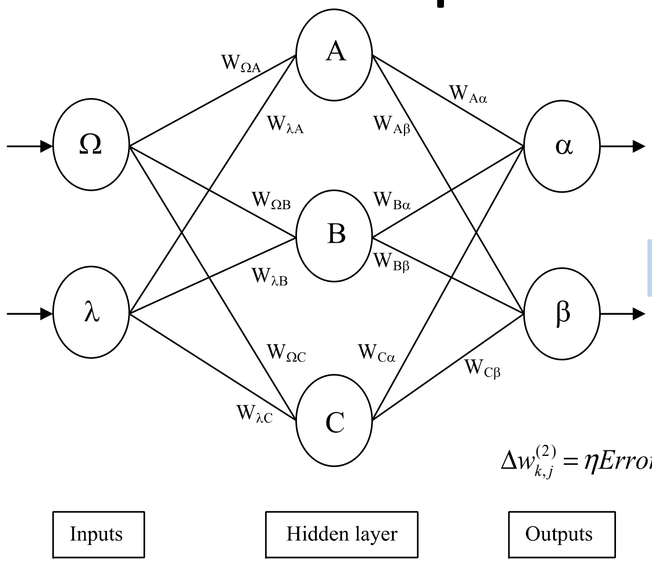
$$\Delta w^{(1)}_{j,m} = -\eta \frac{\partial E(W)}{\partial w^{(1)}_{j,m}} = -\eta \frac{\partial E(W)}{\partial h^{(1)}_j} \frac{\partial h^{(1)}_j}{\partial w^{(1)}_{j,m}} = \eta \sum_k (d_k - y_k) f'_{(2)}(net^{(2)}_k) w^{(2)}_{k,j} x_m f'_{(1)}(net^{(1)}_j) = \eta \delta_j x_m$$

An example for Backprop



https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf

An example for Backprop



1. Calculate errors of output neurons

$$\delta_k = (d_k - y_k)\ f_{(2)}'(net_k^{(2)})$$

$$\delta_\alpha = out_\alpha\ (1 - out_\alpha)\ (Target_\alpha - out_\alpha)$$
$$\delta_\beta = out_\beta\ (1 - out_\beta)\ (Target_\beta - out_\beta)$$

2. Change output layer weights

$$\Delta w_{k,j}^{(2)} = \eta Error_k Output_j = \eta \delta_k h_j^{(1)}$$

$$W^+_{A\alpha} = W_{A\alpha} + \eta\delta_\alpha\, out_A \qquad W^+_{A\beta} = W_{A\beta} + \eta\delta_\beta\, out_A$$
$$W^+_{B\alpha} = W_{B\alpha} + \eta\delta_\alpha\, out_B \qquad W^+_{B\beta} = W_{B\beta} + \eta\delta_\beta\, out_B$$
$$W^+_{C\alpha} = W_{C\alpha} + \eta\delta_\alpha\, out_C \qquad W^+_{C\beta} = W_{C\beta} + \eta\delta_\beta\, out_C$$

3. Calculate (back-propagate) hidden layer errors

$$\delta_j = f_{(1)}'(net_j^{(1)}) \sum_k \delta_k w_{k,j}^{(2)}$$

$$\delta_A = out_A\ (1 - out_A)\ (\delta_\alpha W_{A\alpha} + \delta_\beta W_{A\beta})$$
$$\delta_B = out_B\ (1 - out_B)\ (\delta_\alpha W_{B\alpha} + \delta_\beta W_{B\beta})$$
$$\delta_C = out_C\ (1 - out_C)\ (\delta_\alpha W_{C\alpha} + \delta_\beta W_{C\beta})$$

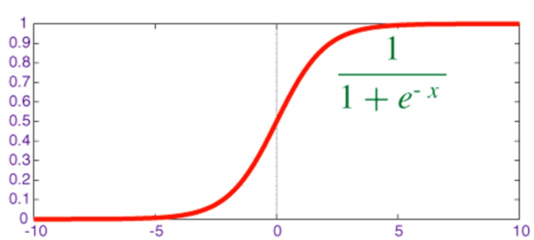
4. Change hidden layer weights

$$\Delta w_{j,m}^{(1)} = \eta Error_j Output_m = \eta \delta_j x_m$$

$$W^+_{\lambda A} = W_{\lambda A} + \eta\delta_A\, in_\lambda \qquad W^+_{\Omega A} = W^+_{\Omega A} + \eta\delta_A\, in_\Omega$$
$$W^+_{\lambda B} = W_{\lambda B} + \eta\delta_B\, in_\lambda \qquad W^+_{\Omega B} = W^+_{\Omega B} + \eta\delta_B\, in_\Omega$$
$$W^+_{\lambda C} = W_{\lambda C} + \eta\delta_C\, in_\lambda \qquad W^+_{\Omega C} = W^+_{\Omega C} + \eta\delta_C\, in_\Omega$$

Consider sigmoid activation function

$$f_{Sigmoid}(x) = \frac{1}{1+e^{-x}}$$



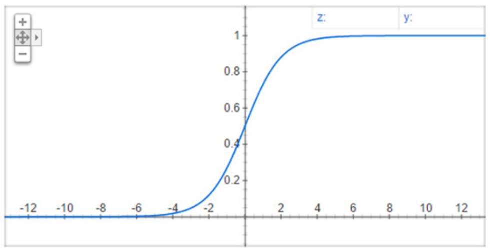
$$f'_{Sigmoid}(x) = f_{Sigmoid}(x)(1 - f_{Sigmoid}(x))$$

https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf

Non-linear Activation Functions

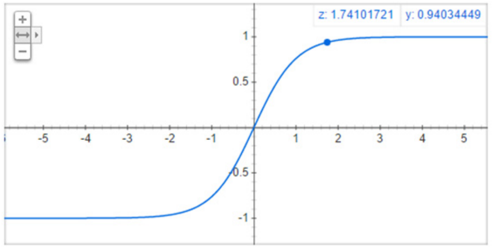
- Sigmoid

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



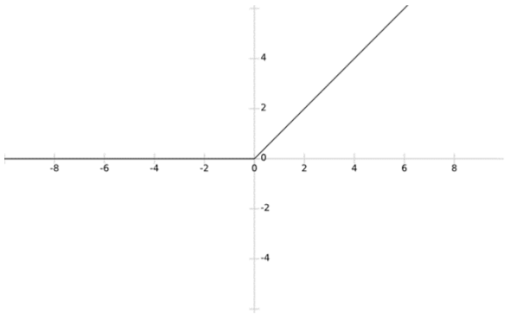
- Tanh

$$\tanh(z) = \frac{1 - e^{-2z}}{1 + e^{-2z}}$$

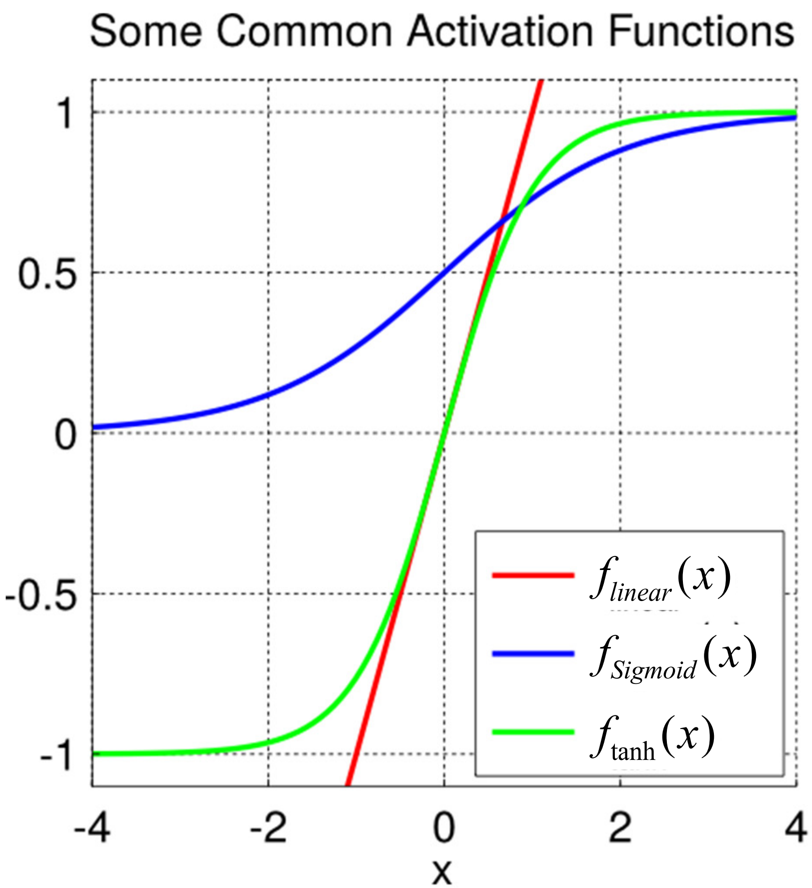


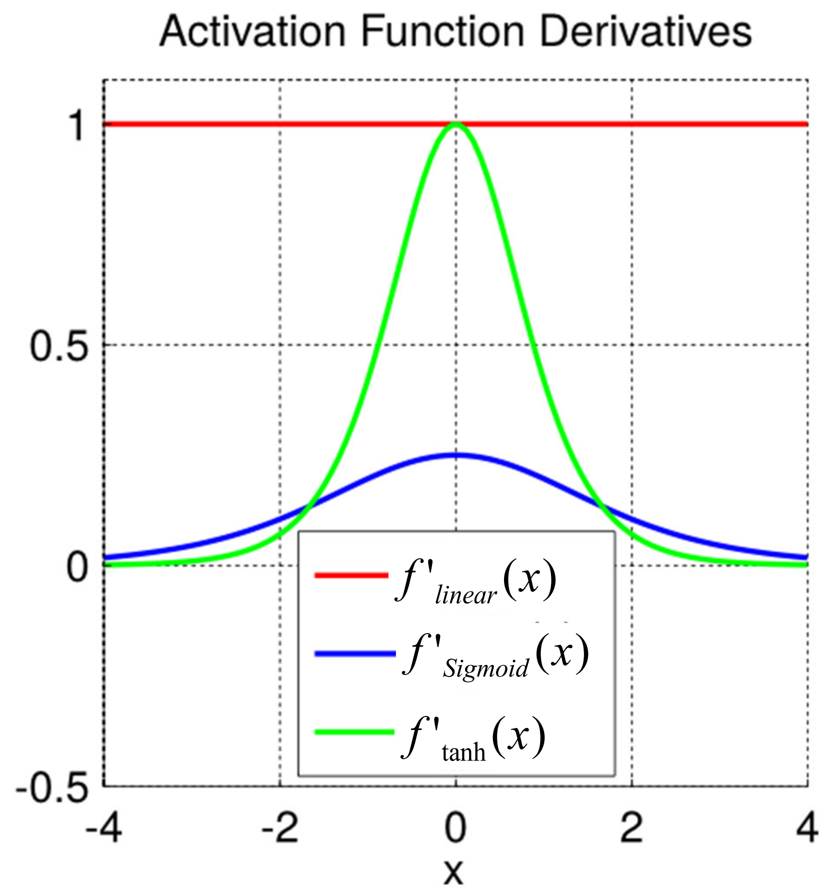
- Rectified Linear Unit (ReLU)

$$\mathrm{ReLU}(z) = \max(0, z)$$



Active functions





https://theclevermachine.wordpress.com/tag/tanh-function/

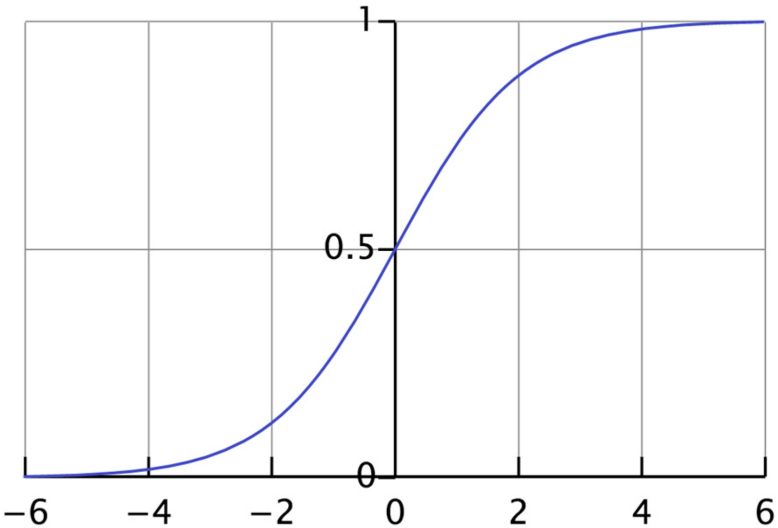
Activation functions

- **Logistic Sigmoid:**

$$f_{Sigmoid}(x) = \frac{1}{1+e^{-x}}$$

Its derivative:

$$f'_{Sigmoid}(x) = f_{Sigmoid}(x)(1 - f_{Sigmoid}(x))$$

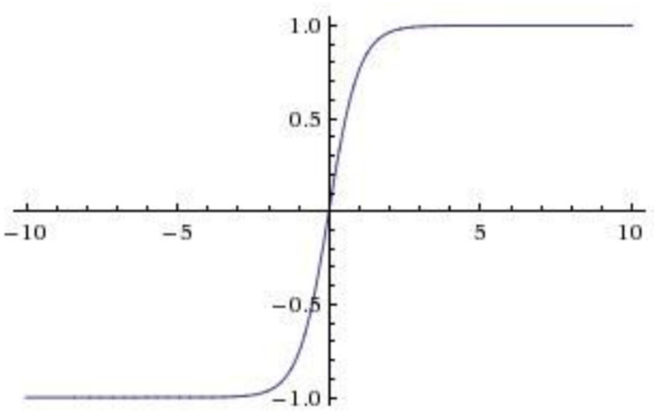


- Output range [0,1]
- Motivated by biological neurons and can be interpreted as the probability of an artificial neuron “firing” given its inputs
- However, saturated neurons make gradients vanished **(why?)**

Activation functions

- Tanh function

$$f_{\tanh}(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



Its gradient:

$$f_{\tanh}(x) = 1 - f_{\tanh}(x)^2$$

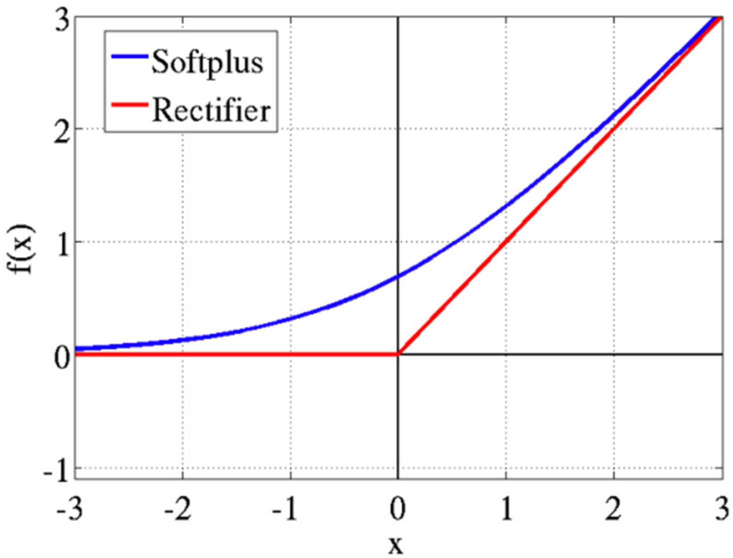
- Output range [-1,1]
- Thus strongly negative inputs to the tanh will map to negative outputs.
- Only zero-valued inputs are mapped to near-zero outputs
- These properties make the network less likely to get “stuck” during training

https://theclevermachine.wordpress.com/tag/tanh-function/

Active Functions

- ReLU (rectified linear unit)

$$f_{\mathrm{ReLU}}(x) = \max(0, x)$$



http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40811.pdf

- The derivative:

$$f_{\mathrm{ReLU}}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

- Another version is Noise ReLU:

$$f_{\mathrm{NoisyReLU}}(x) = \max(0, x + N(0, \delta(x)))$$

- ReLU can be approximated by softplus function

$$f_{\mathrm{Softplus}}(x) = \log(1 + e^{x})$$

- ReLU gradient doesn't vanish as we increase x
- It can be used to model positive number
- It is fast as no need for computing the exponential function
- It eliminates the necessity to have a "*pretraining*" phase

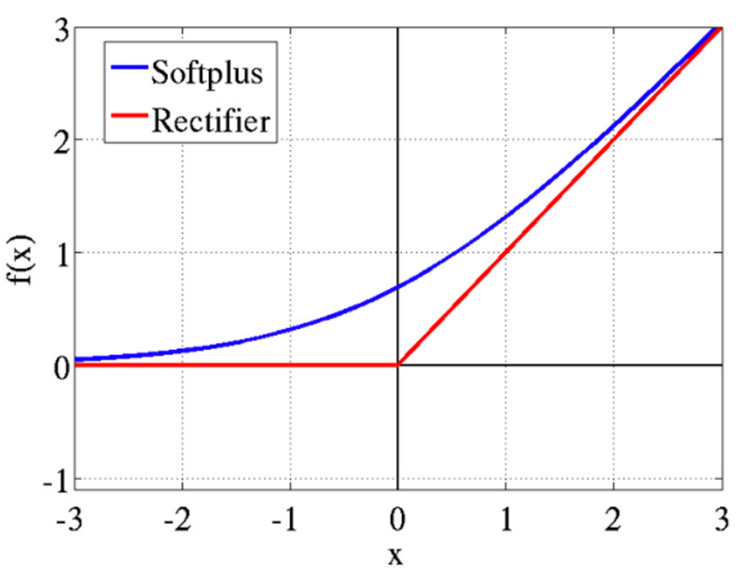
Active Functions

- ReLU (rectified linear unit)

$$f_{\mathrm{ReLU}}(x) = \max(0, x)$$

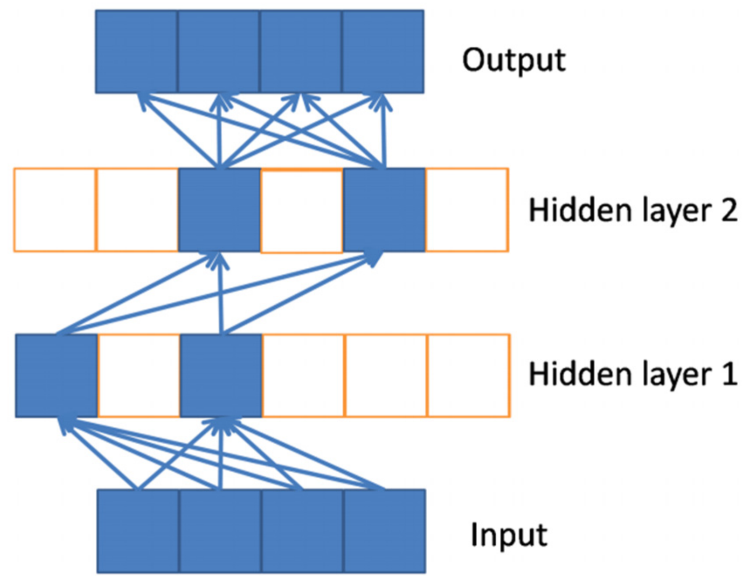
ReLU can be approximated by softplus function

$$f_{\mathrm{Softplus}}(x) = \log(1 + e^x)$$



Additional active functions:
Leaky ReLU, Exponential LU, Maxout etc

- The only non-linearity comes from the path selection with individual neurons being active or not
- It allows sparse representations:
  - for a given input only a subset of neurons are active



Sparse propagation of activations and gradients

http://www.jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf

Deep Learning

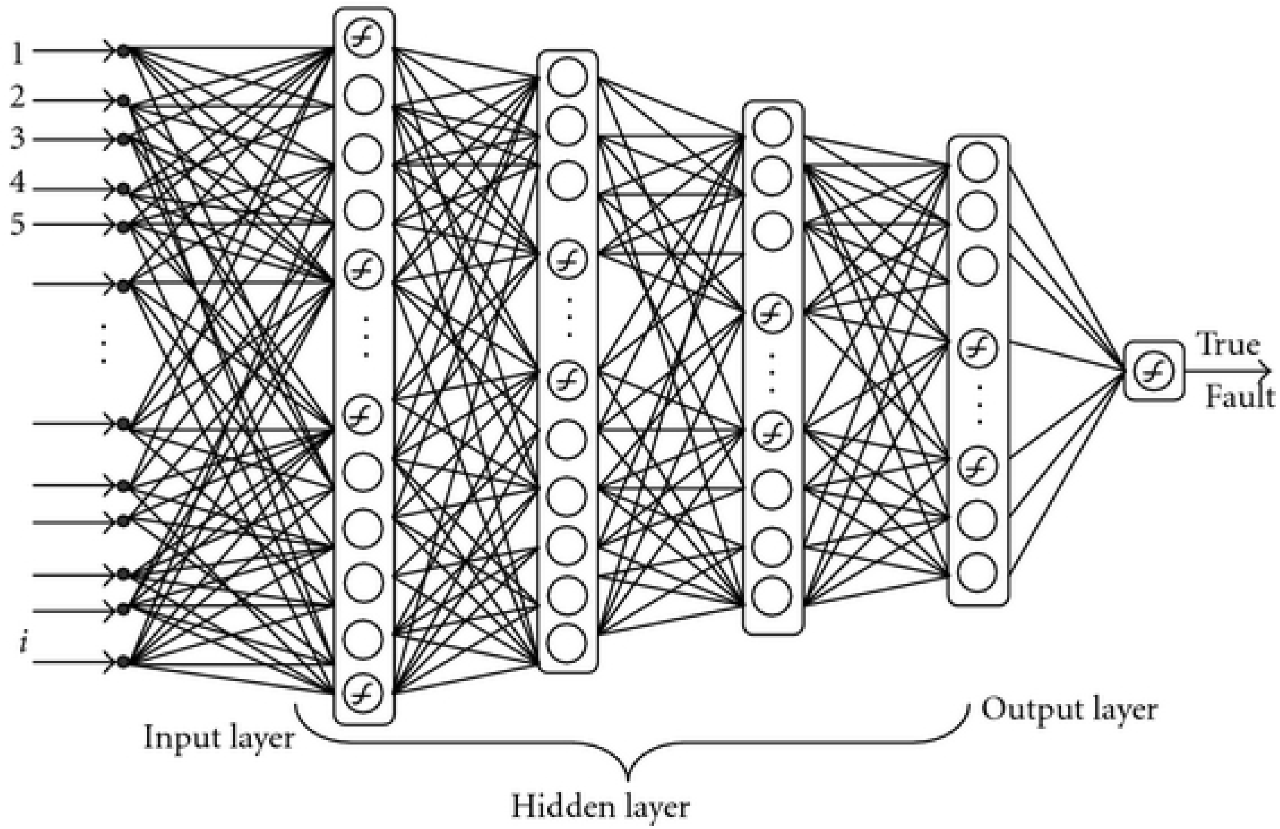
What is Deep Learning

- Deep learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level.

- Mostly implemented via neural networks

[LeCun, Bengio and Hinton. Deep Learning. Nature 2015.]

Deep Neural Network (DNN)



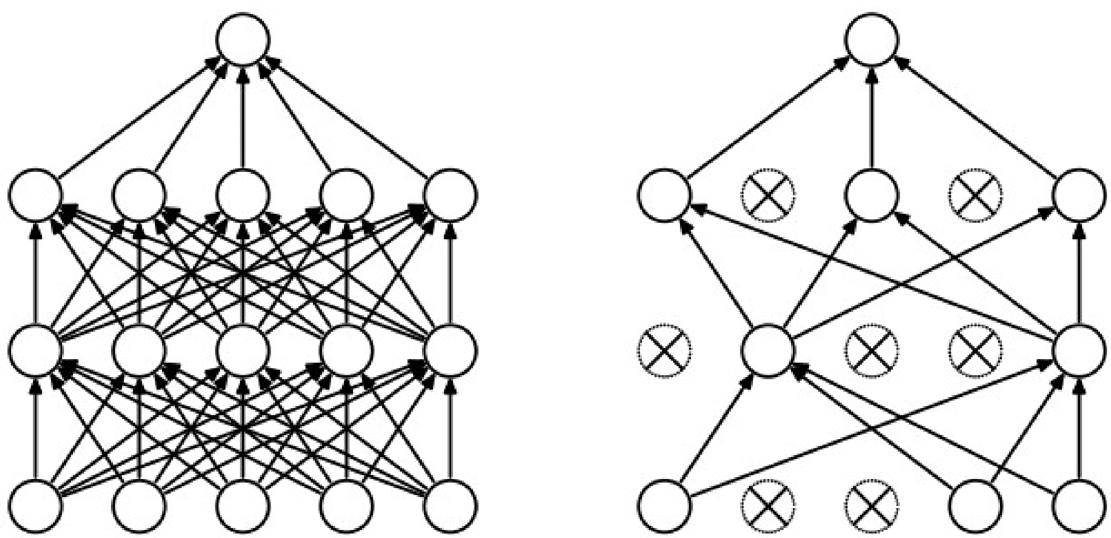
- Multi-layer perceptron with many hidden layers

Difficulty of Training Deep Nets

- Lack of big data
  - Now we have a lot of big data
- Lack of computational resources
  - Now we have GPUs and HPCs
- Easy to get into a (bad) local minimum
  - Now we use pre-training techniques & various optimization algorithms
- Gradient vanishing
  - Now we use ReLU
- Regularization
  - Now we use Dropout

Dropout

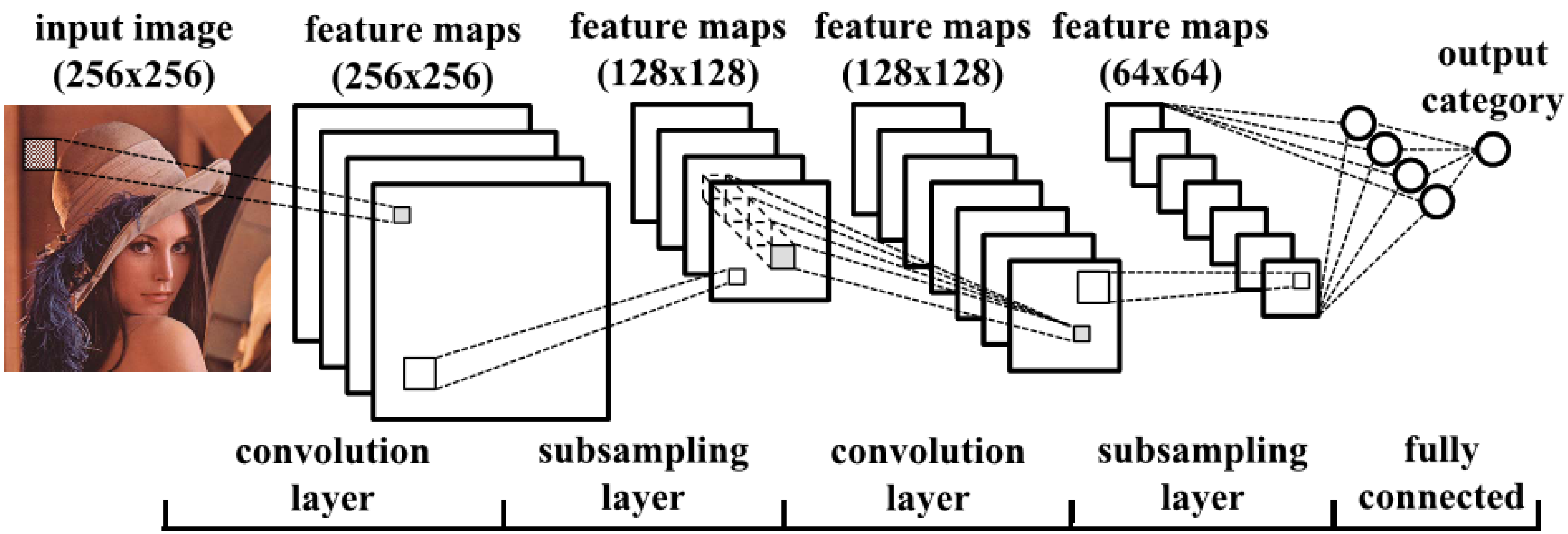
- Dropout randomly ‘drops’ units from a layer on each training step, creating ‘sub-architectures’ within the model.
- It can be viewed as a type of sampling of a smaller network within a larger network
- Prevent neural networks from overfitting



(a) Standard Neural Net (b) After applying dropout.

Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.

Convolutional Neural Network (CNN)



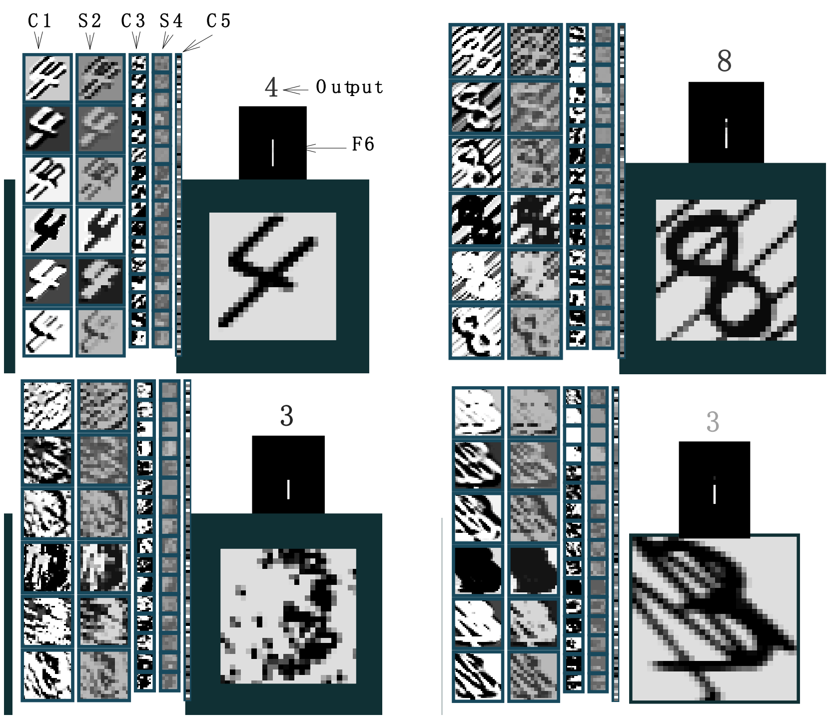
[Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11) 1998]

Use Case: Digits Recognition

- MNIST (handwritten digits) Dataset:

  http://yann.lecun.com/exdb/mnist/

  - 60k training and 10k test examples
- Test error rate 0.95%



Total only 82 errors from LeNet-5. correct answer left and right is the machine answer.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998

Recurrent Neural Network (RNN)

- To model sequential data
  - Text
  - Time series
- Trained by Back-Propagation Through Time (BPTT)

x : input vector, o : output vector,

s : hidden state vector,

$\mathbf{U}$: layer 1 param. matrix,

$\mathbf{V}$: layer 2 param. matrix,

f: tanh or ReLU

$$o = f(s\mathbf{V})$$

$$s = f(x\mathbf{U})$$

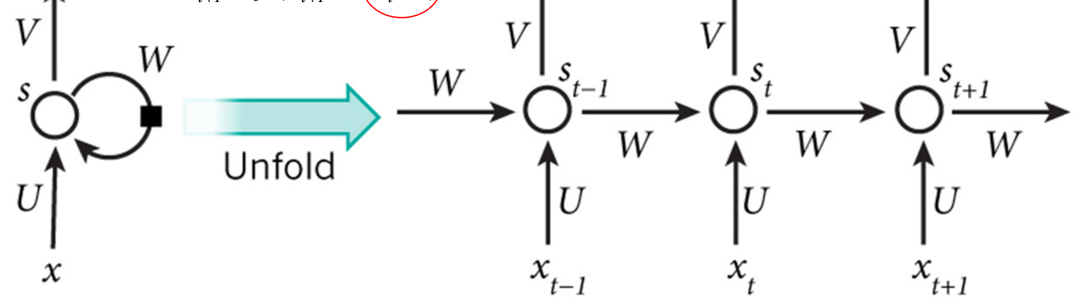
Two-layer feedforward network

Add time-dependency of the hidden state *s*

W : State transition param. matrix

$$o_{t+1} = f(s_{t+1}\mathbf{V})$$

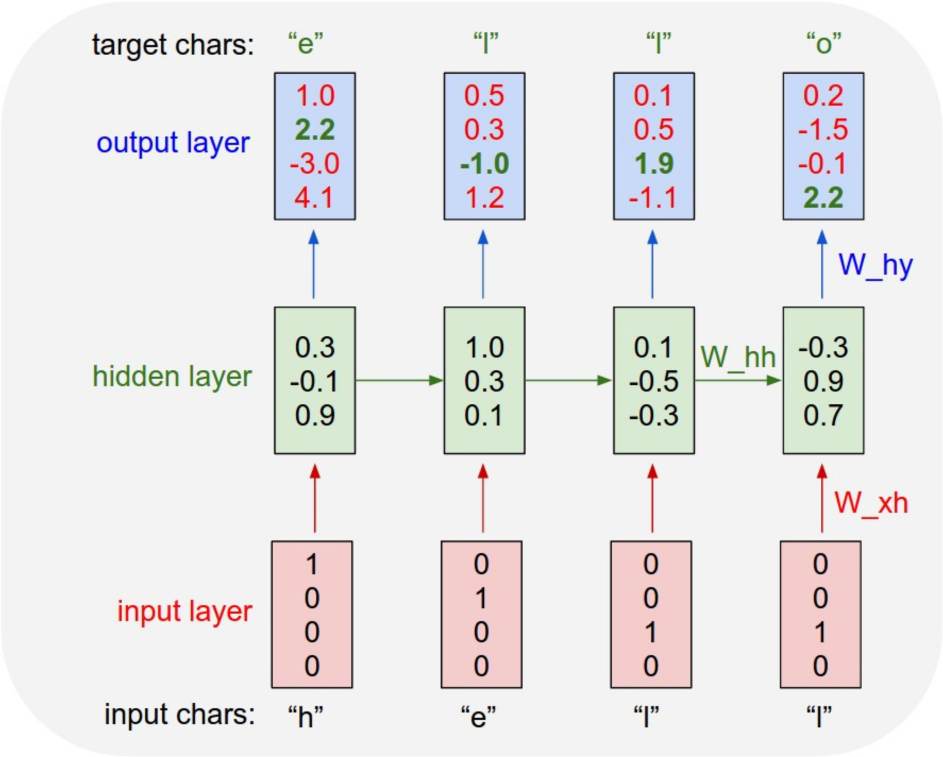
$$s_{t+1} = f(x_{t+1}\mathbf{U} + s_t\mathbf{W})$$



[http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/]

Use Case: Language Model

- Word-level or even character-level language model
  - Given previous words/characters, predict the next



[http://karpathy.github.io/2015/05/21/rnn-effectiveness/]

Summary

- Universal Approximation: two-layer neural networks can approximate any functions

- Backpropagation is the most important training scheme for multi-layer neural networks so far

- Deep learning, i.e. deep architecture of NN trained with big data, works incredibly well

- Neural works built with other machine learning models achieve further success