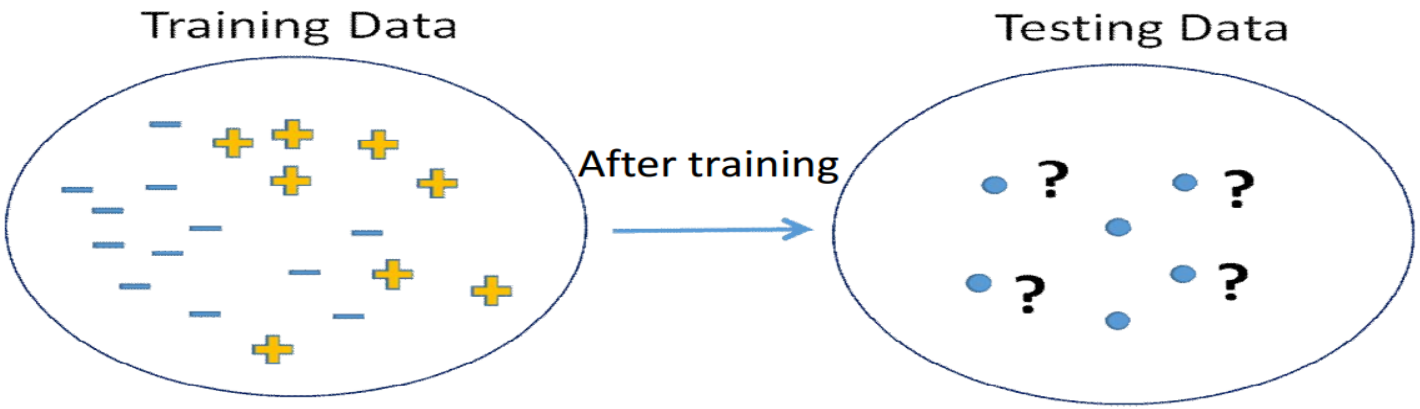
Artificial Neural Network

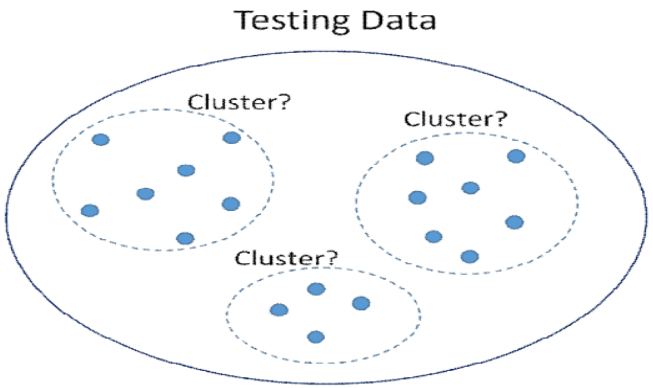
Simon Shim, Ph.D.
Professor
San Jose State University

Supervised vs unsupervised Learning

- Supervised



- Unsupervised



History

- 1943: McCulloch–Pitts “neuron”
  - Started the field
- 1962: Rosenblatt’s perceptron
  - Learned its own weight values; convergence proof
- 1969: Minsky & Papert book on perceptrons
  - Proved limitations of single-layer perceptron networks
- 1982: Hopfield and convergence in symmetric networks
  - Introduced energy-function concept
- 1986: Backpropagation of errors
  - Method for training multilayer networks
- Present: Probabilistic interpretations, Bayesian and spiking networks

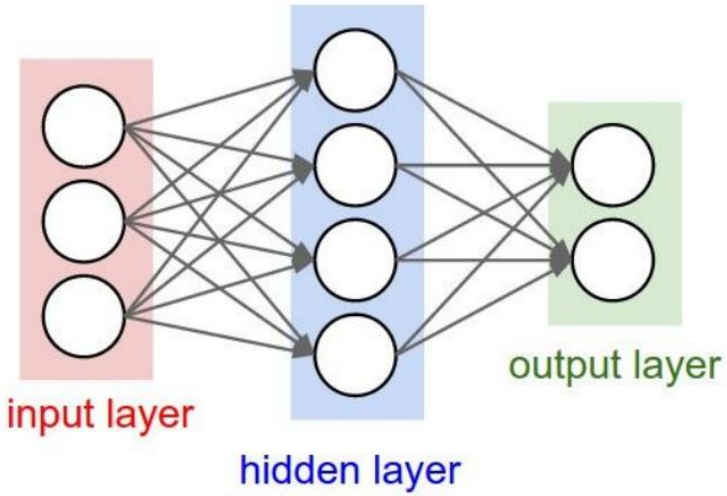
The Start of Machine Learning

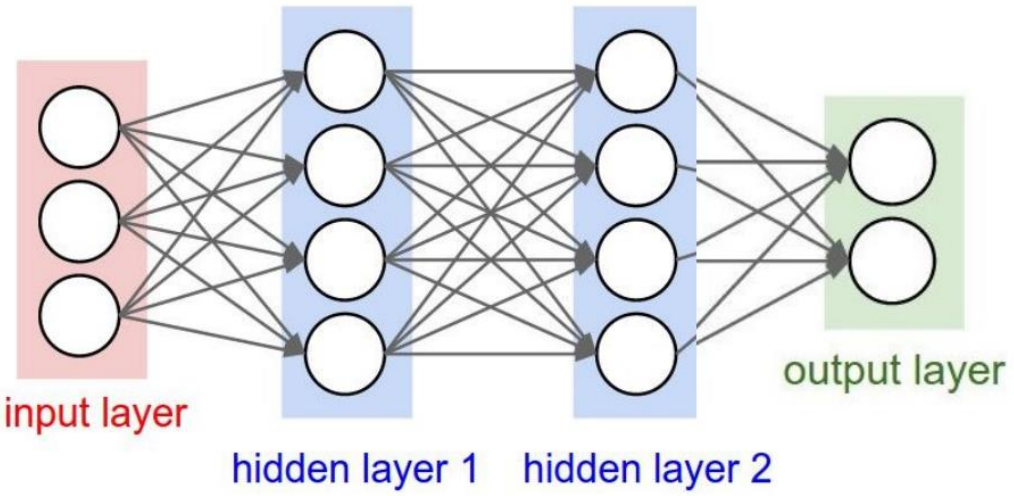
- 1957, Perceptron
  A network of neurons
    - Introduce learning
    - Why AI is not rule-based system
        - The nature of statistics
    - Introduce simulation
    - Introduce unsupervised learning
    - Optimism on Neural network application

Frank Rosenblatt

“Deep” Neural Network

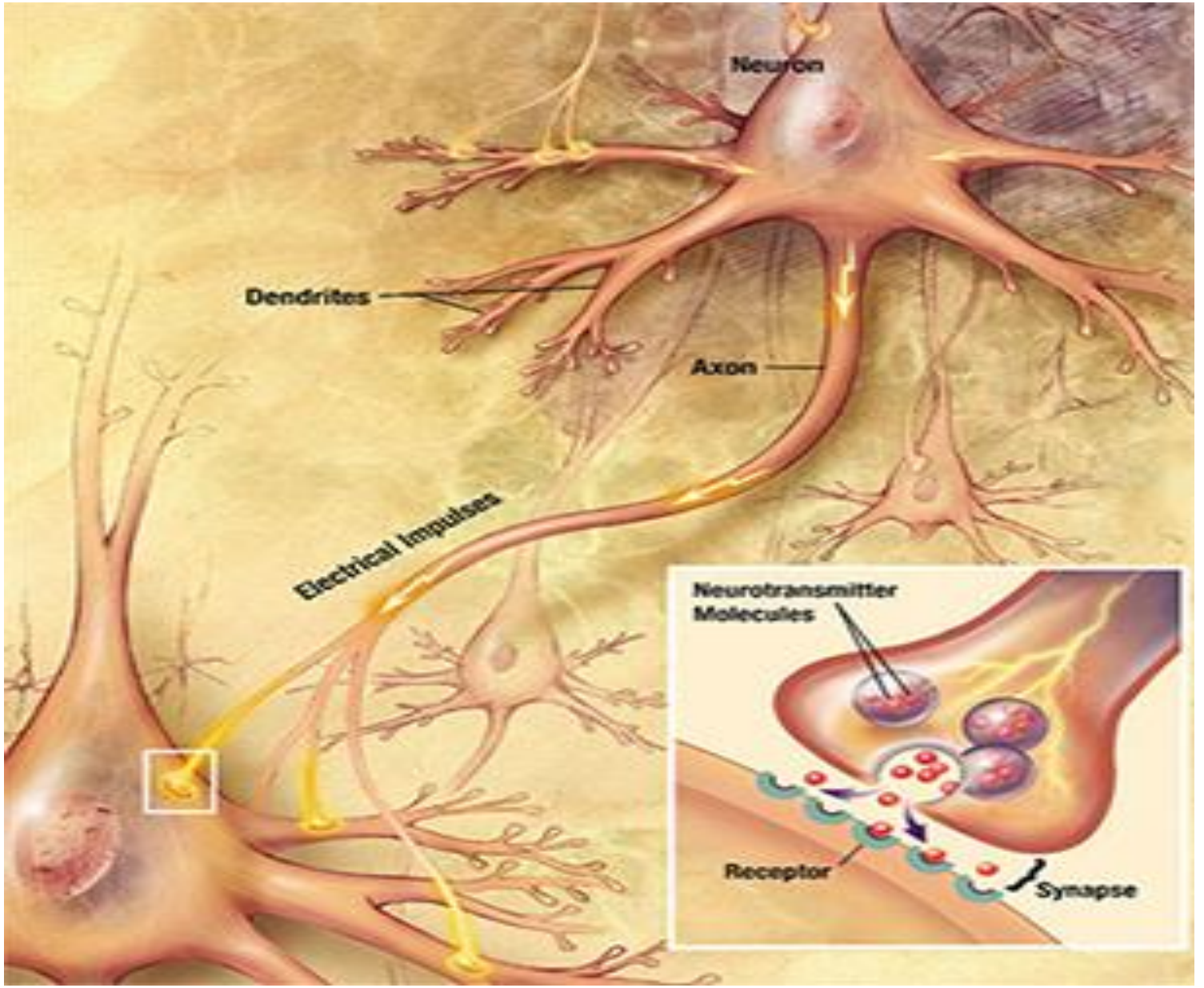
- Deep vs. Shallow



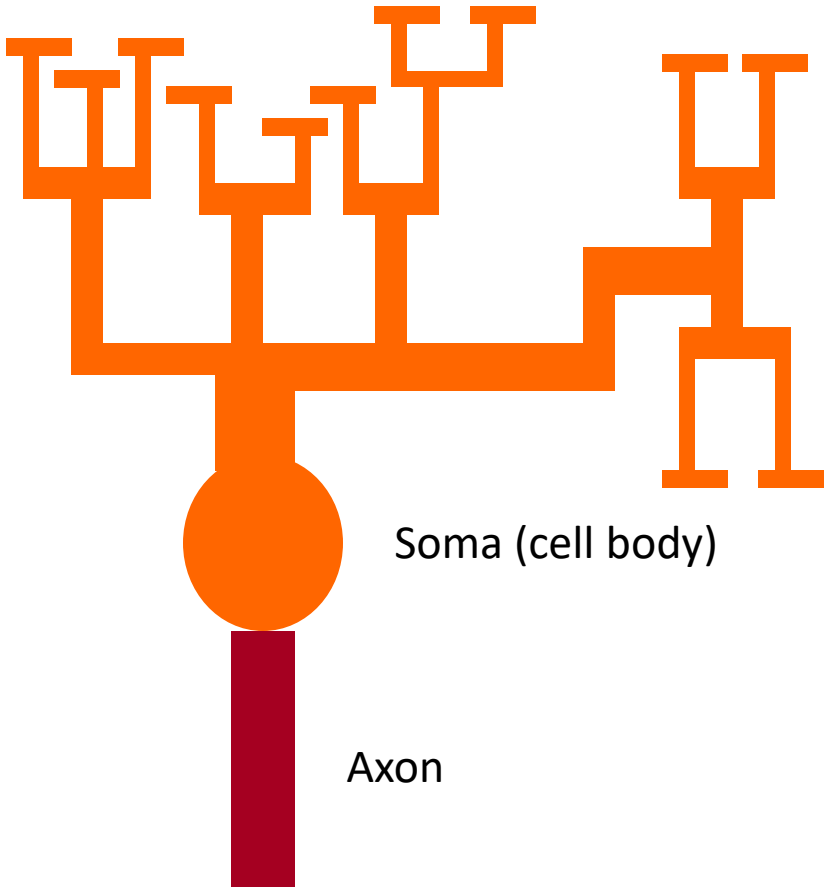


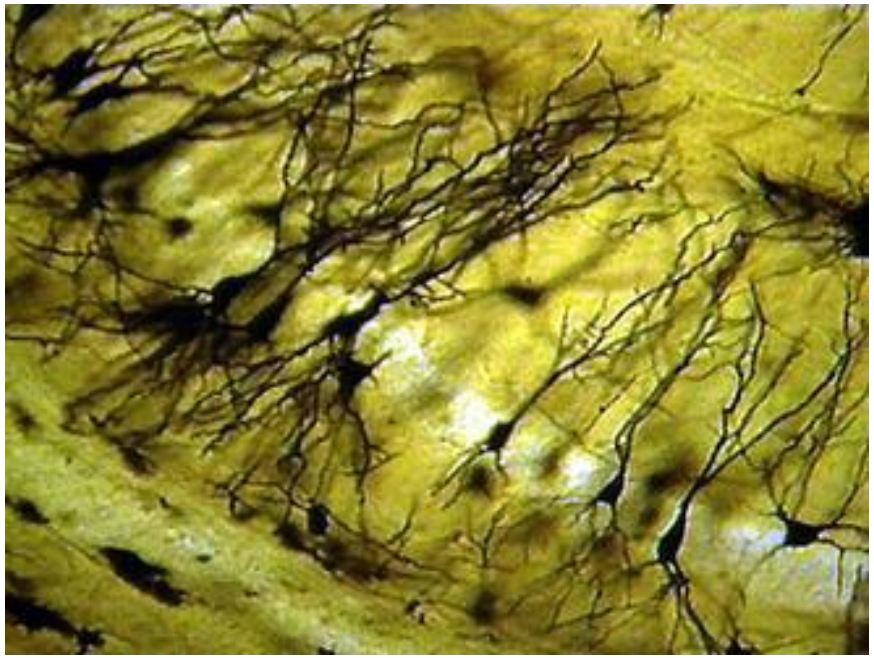
The network between biological neurons

(From: Wikipedia)



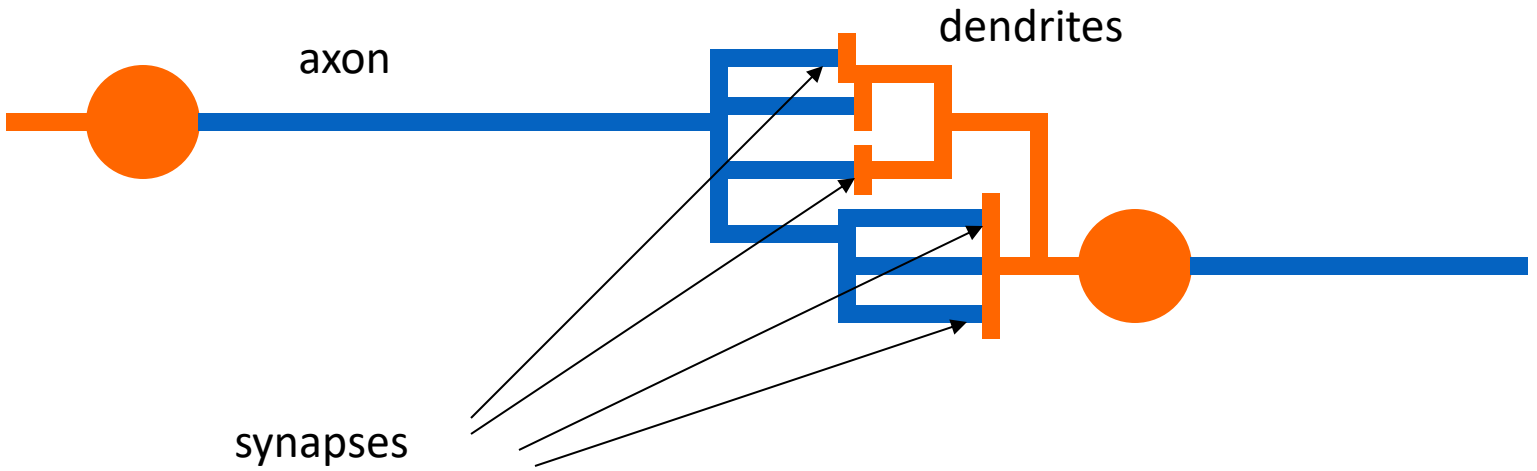
Biological inspiration





Source: Peter Andras

Biological inspiration

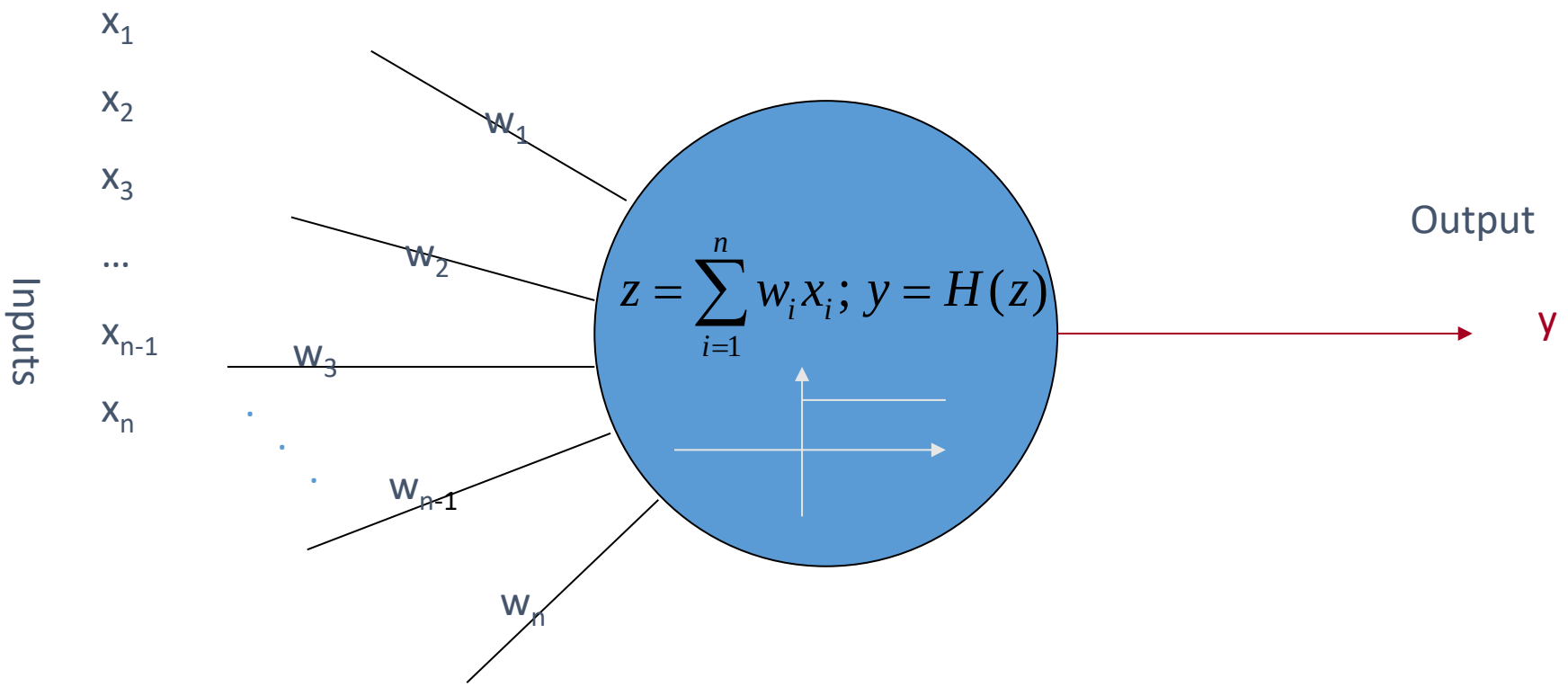


The information transmission happens at the synapses.

Source: Peter Andras

Artificial neurons

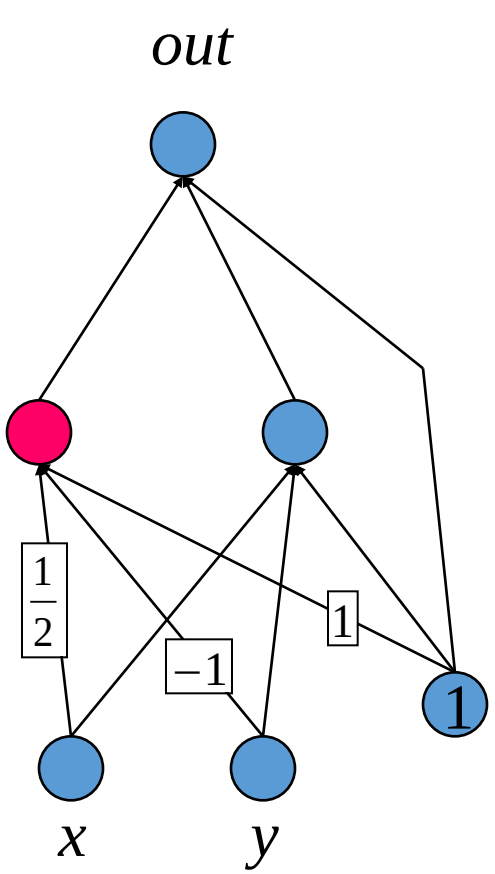
Neurons work by processing information. They receive and provide information in form of spikes.

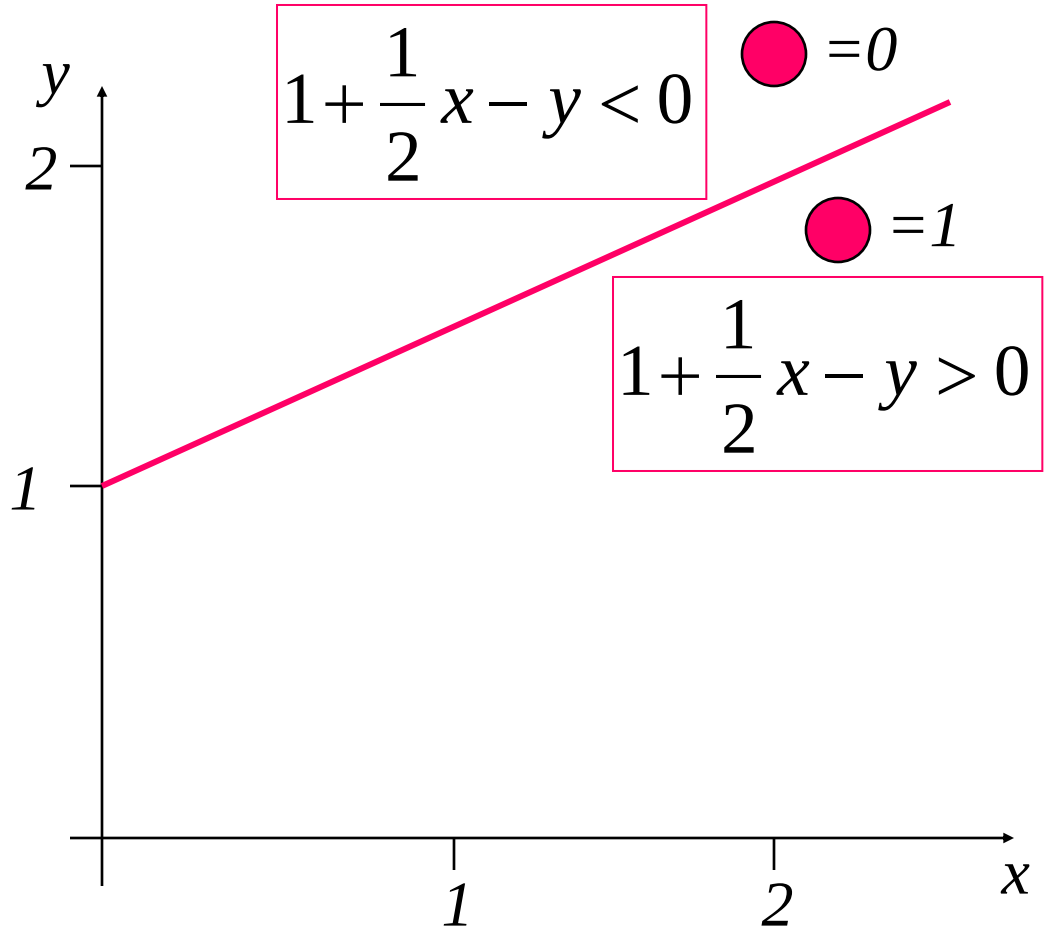


The McCullogh-Pitts model

Source: Peter Andras

Example: Perceptrons as Constraint Satisfaction Networks





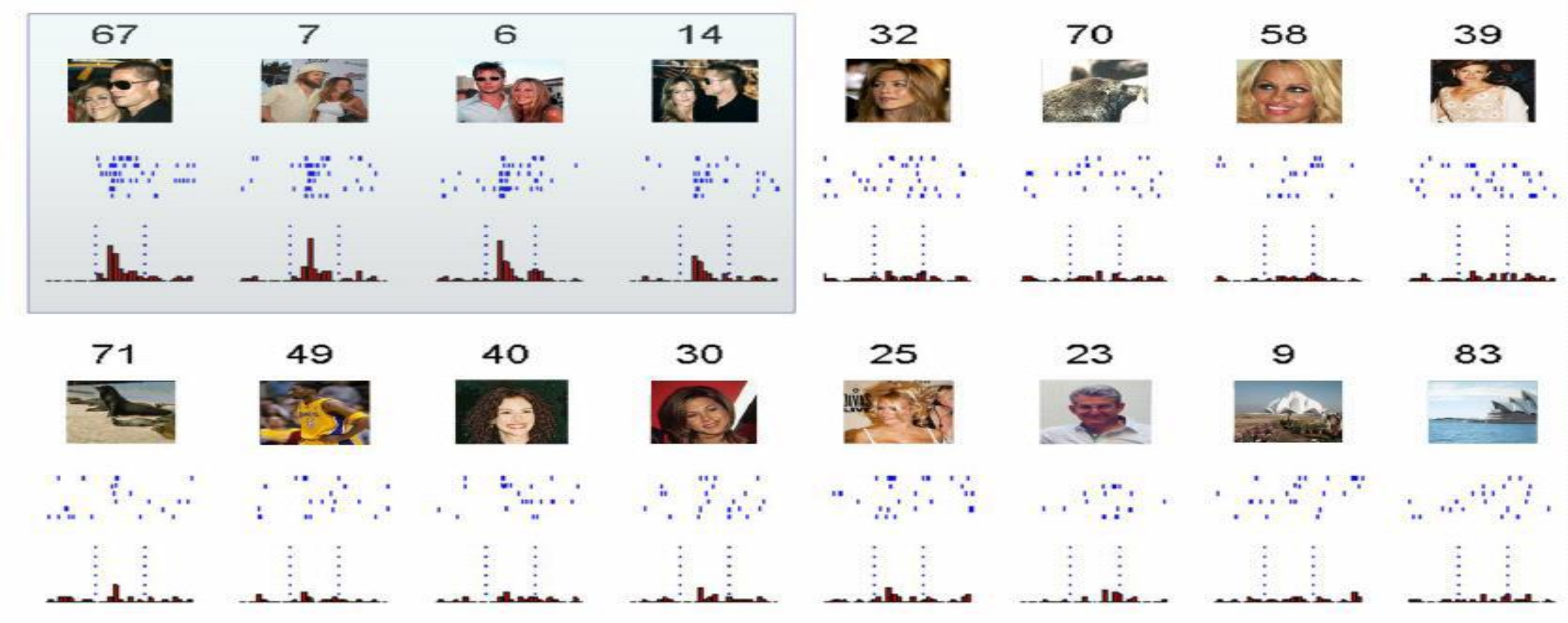
Artificial neurons

The McCullogh-Pitts model:

- spikes are interpreted as spike rates;
- synaptic strength are translated as synaptic weights;
- excitation means positive product between the incoming spike rate and the corresponding synaptic weight;
- inhibition means negative product between the incoming spike rate and the corresponding synaptic weight;

Source: Peter Andras

Stimulus



QuianQuiroga, Reddy, Kreiman, Koch and Fried, *Nature* (2005)

Learning in biological systems

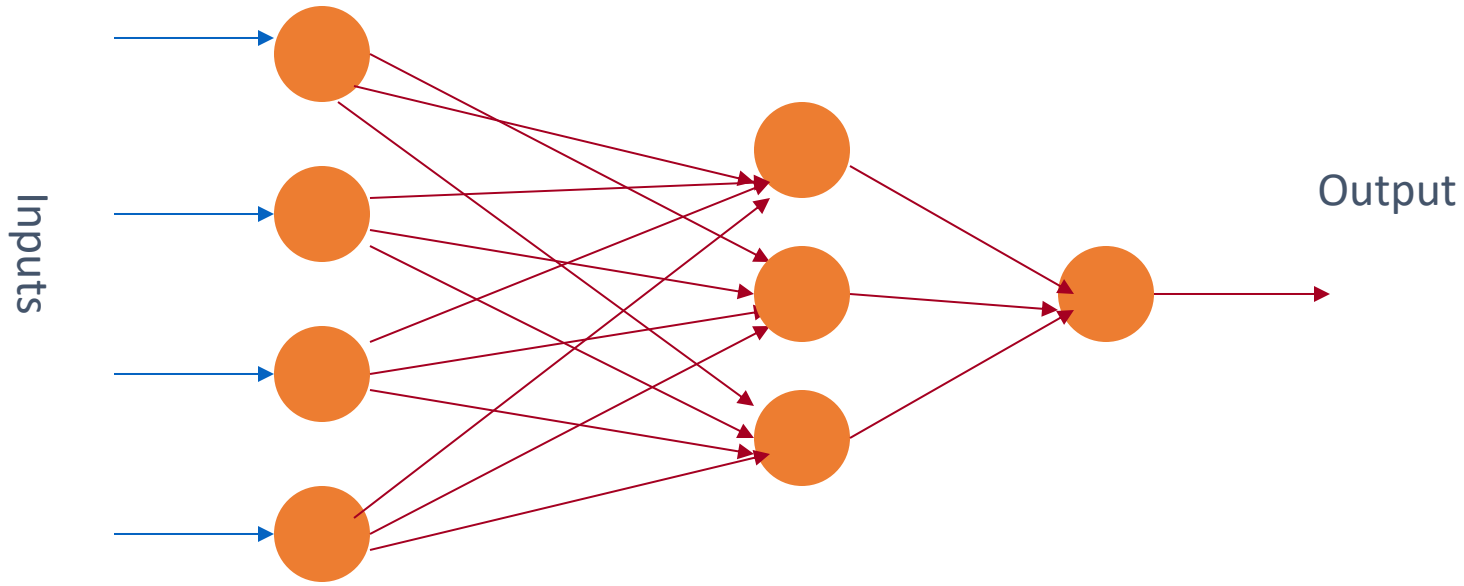
Learning = learning by adaptation

The young animal learns that the green fruits are sour, while the yellowish/reddish ones are sweet. The learning happens by adapting the fruit picking behavior.

At the neural level the learning happens by changing of the synaptic strengths, eliminating some synapses, and building new ones.

Source: Peter Andras

Neural network mathematics



$$y_1^1 = f(x_1, w_1^1)$$
$$y_2^1 = f(x_2, w_2^1)$$
$$y_3^1 = f(x_3, w_3^1)$$
$$y_4^1 = f(x_4, w_4^1)$$

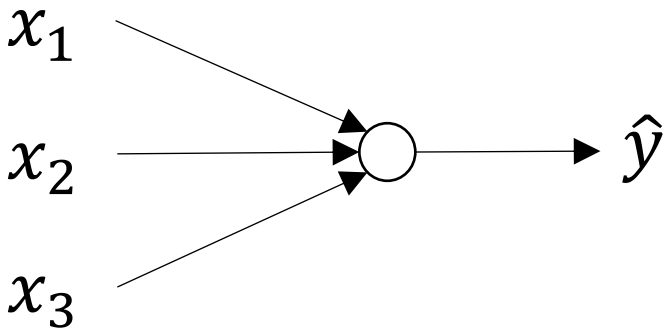
$$y^1 = \begin{pmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \\ y_4^1 \end{pmatrix}$$

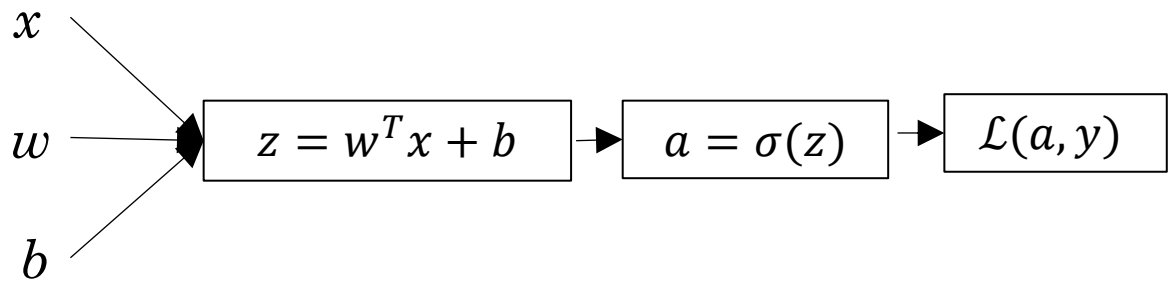
$$y_1^2 = f(y^1, w_1^2)$$
$$y_2^2 = f(y^1, w_2^2)$$
$$y_3^2 = f(y^1, w_3^2)$$

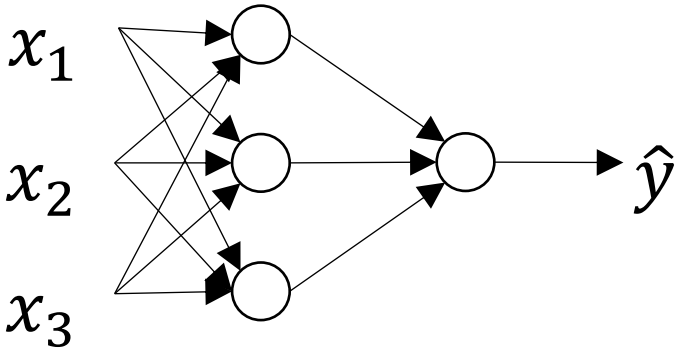
$$y^2 = \begin{pmatrix} y_3^2 \\ y_3^2 \\ y_3^2 \end{pmatrix}$$

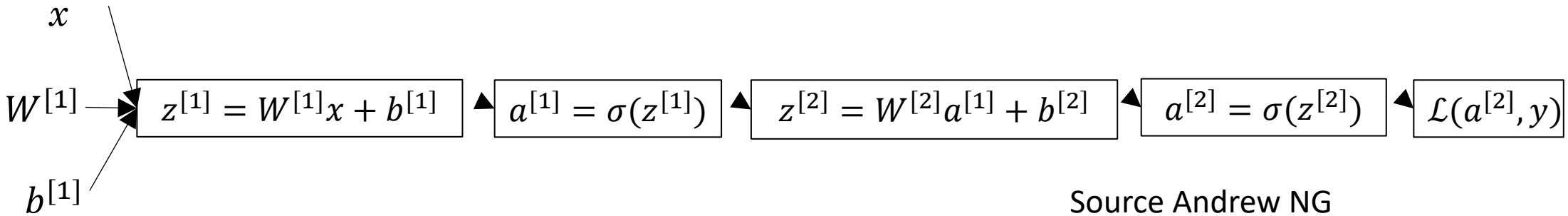
$$y_{Out} = f(y^2, w_1^3)$$

Neural Network









Source Andrew NG

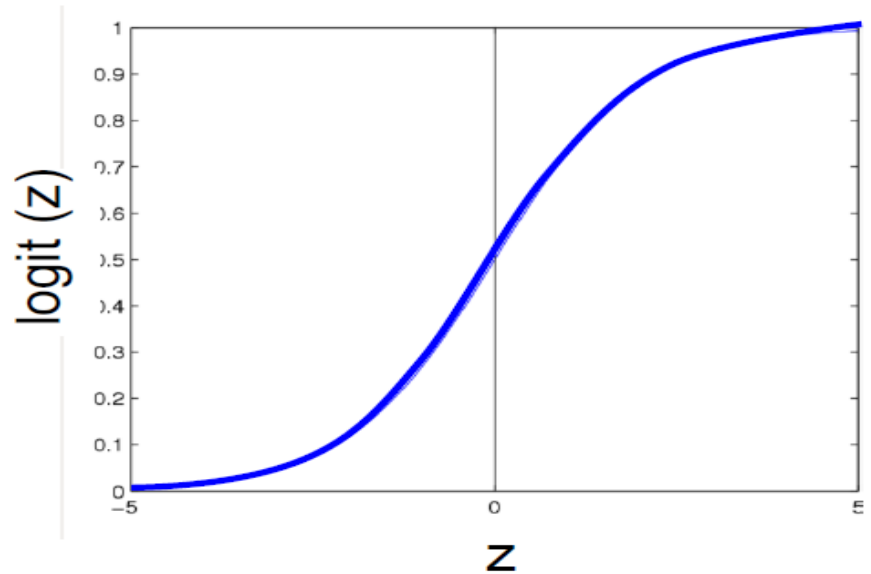
Logistic Regression

Assumes the following functional form for P(Y|X):

$$P(Y=1|X) = \frac{1}{1+\exp(-(w_0+\sum_i w_i X_i))}$$

Logistic function applied to a linear function of the data

Logistic function (or Sigmoid): $\frac{1}{1+exp(-z)}$

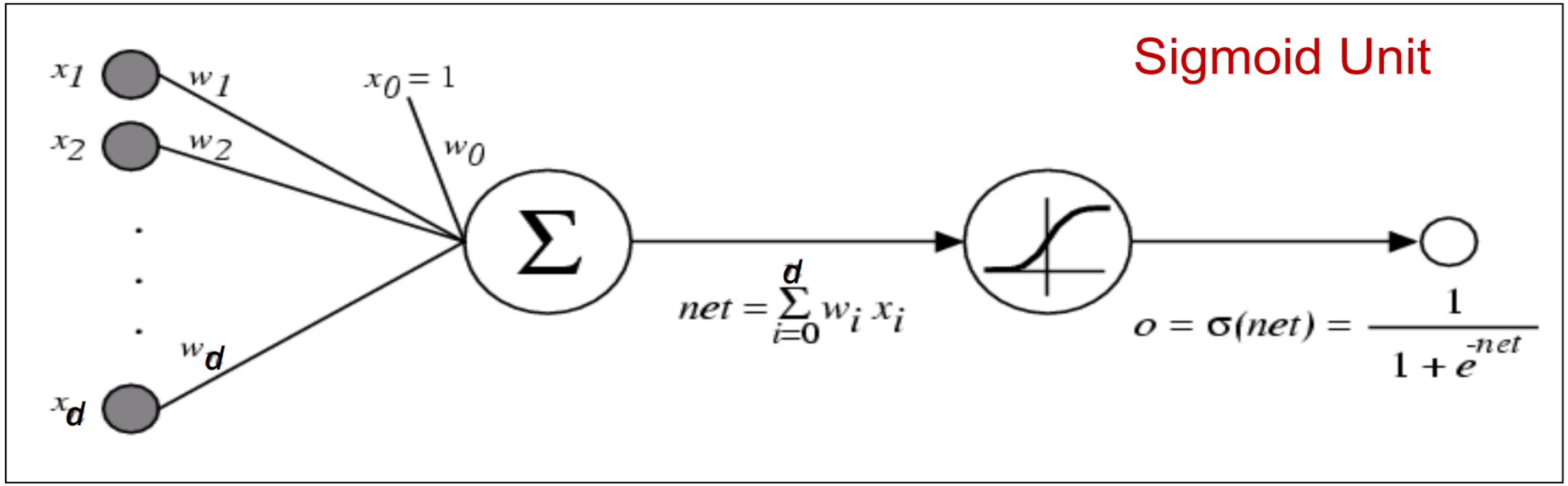


Source: Aarti Singh

Features can be discrete or continuous!

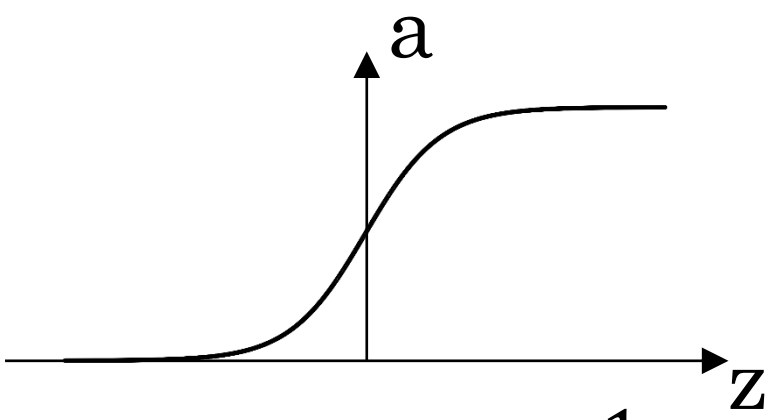
Logistic Regression as a Graph

$$\text{Output, } o(\mathbf{x}) = \sigma(w_0 + \sum_i w_i X_i) = \frac{1}{1 + \exp(-(w_0 + \sum_i w_i X_i))}$$

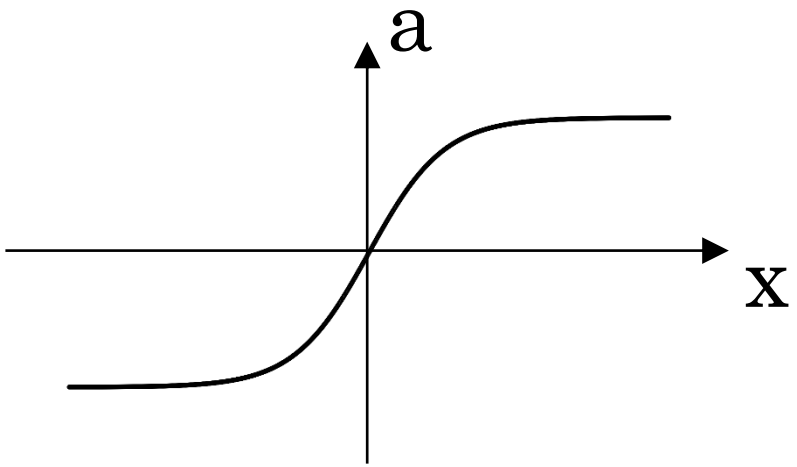


Source: Aarti Singh

Activation Functions

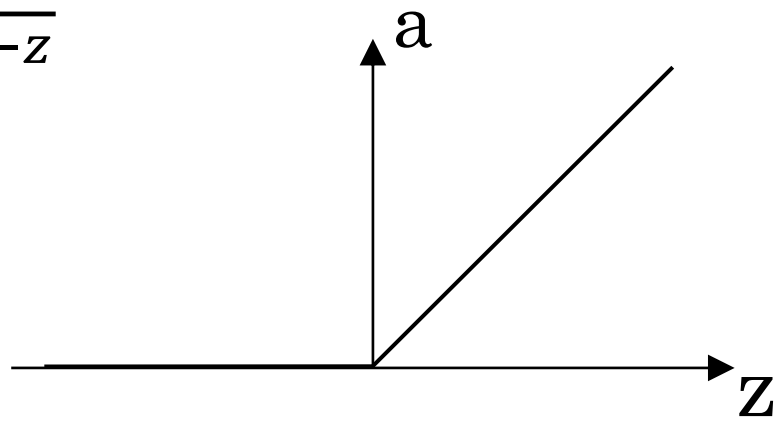


$$\text{sigmoid: } a = \frac{1}{1 + e^{-z}}$$



$$g(z) = \tanh(z)$$

$$y = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

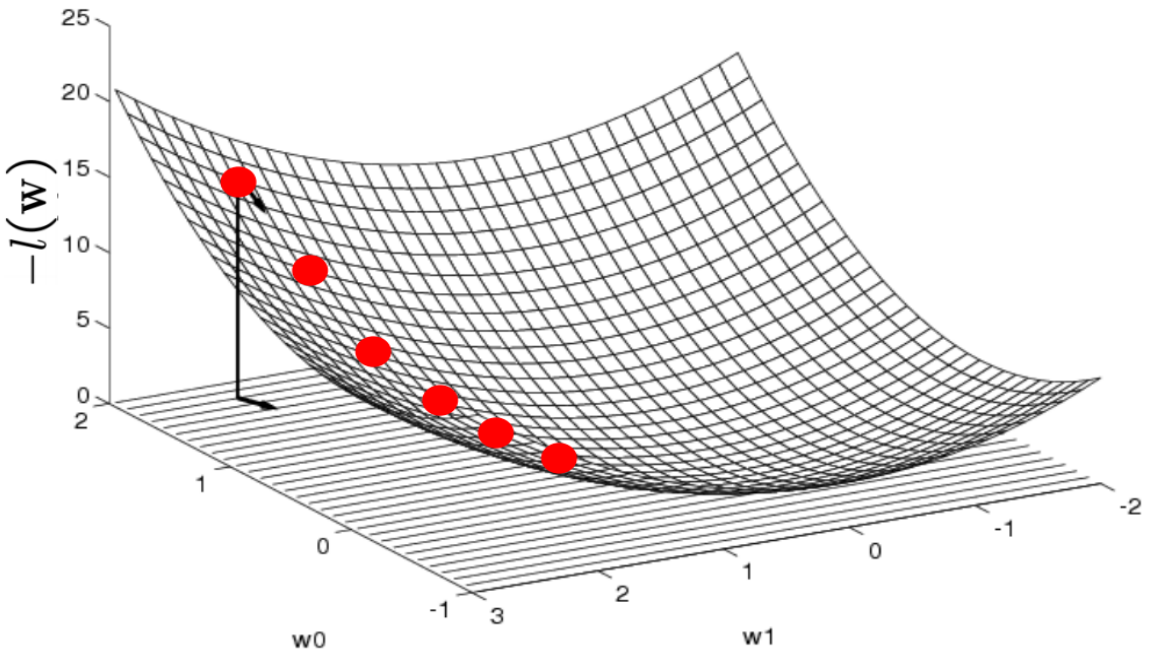


ReLU

Optimizing concave/convex function

- Conditional likelihood for Logistic Regression is concave
- Maximum of a concave function = minimum of a convex function

Gradient Ascent (concave)/ Gradient Descent (convex)



Gradient:

$$\nabla_{\mathbf{w}} l(\mathbf{w}) = [\frac{\partial l(\mathbf{w})}{\partial w_0}, \ldots, \frac{\partial l(\mathbf{w})}{\partial w_d}]'$$

Update rule: **Learning rate, η>0**

$$\Delta \mathbf{w} = \eta \nabla_{\mathbf{w}} l(\mathbf{w})$$

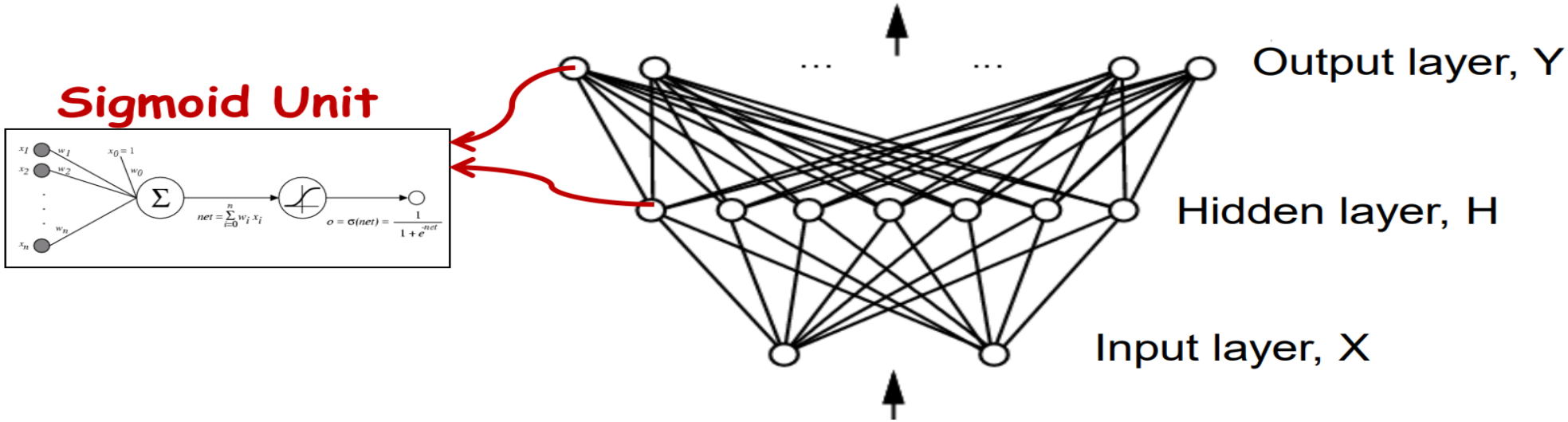
$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta \left. \frac{\partial l(\mathbf{w})}{\partial w_i} \right|_t$$

Source: Aarti Singh

Neural Networks to learn f: X → Y

- f can be a non-linear function
- X (vector of) continuous and/or discrete variables
- Y (vector of) continuous and/or discrete variables

- Neural networks - Represent f by *network* of logistic/sigmoid units, we will focus on feedforward networks:



Source: Aarti Singh

Forward Propagation for prediction

Sigmoid unit:

$$o(\mathbf{x}) = \sigma(w_0 + \sum_i w_i x_i)$$

1-Hidden layer,
1 output NN:

$$o(\mathbf{x}) = \sigma\left(w_0 + \sum_h w_h \underbrace{\sigma(w_0^h + \sum_i w_i^h x_i)}_{o_h}\right)$$

Prediction – Given neural network (hidden units and weights), use it to predict the label of a test point

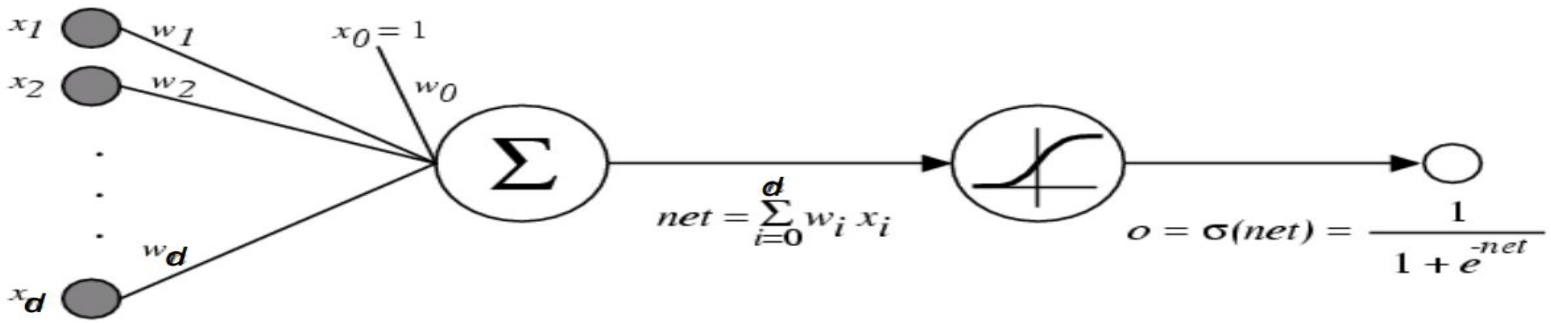
Forward Propagation –

Start from input layer

For each subsequent layer, compute output of sigmoid unit

Source: Aarti Singh

Sigmoid Unit



$\sigma(x)$ is the sigmoid function/activation function (also linear, threshold)

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

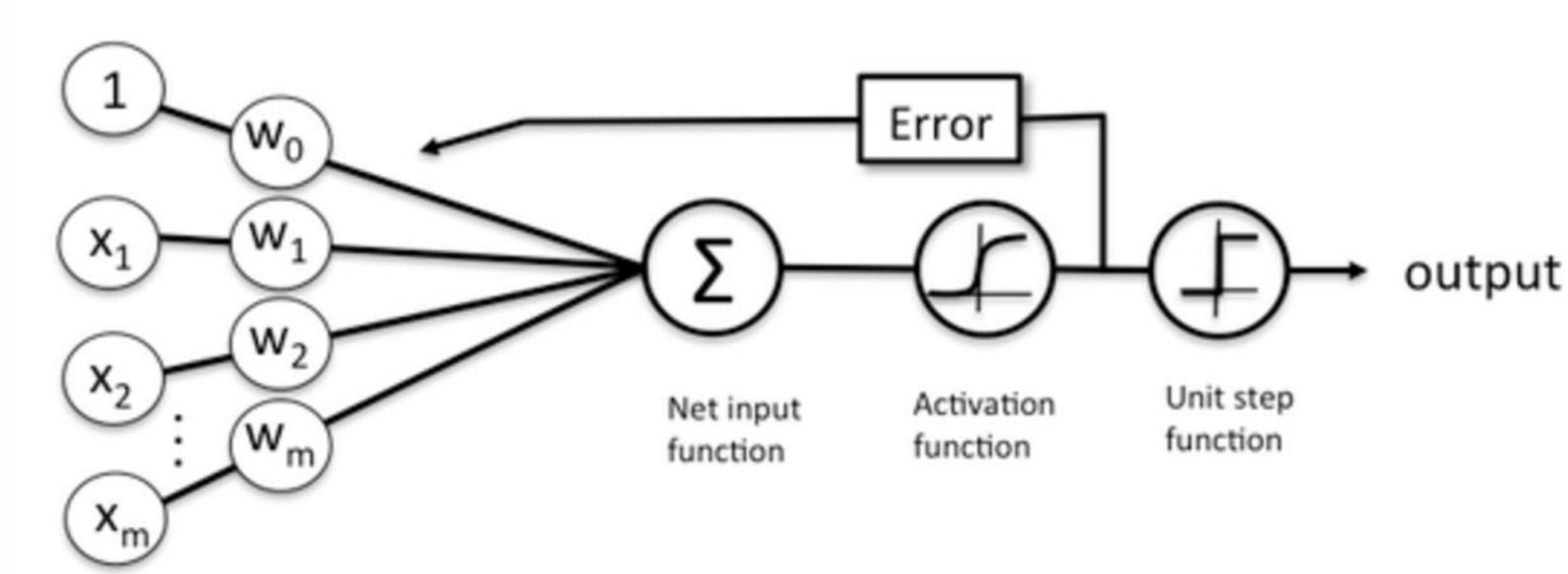
Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$ Differentiable

We can derive gradient decent rules to train

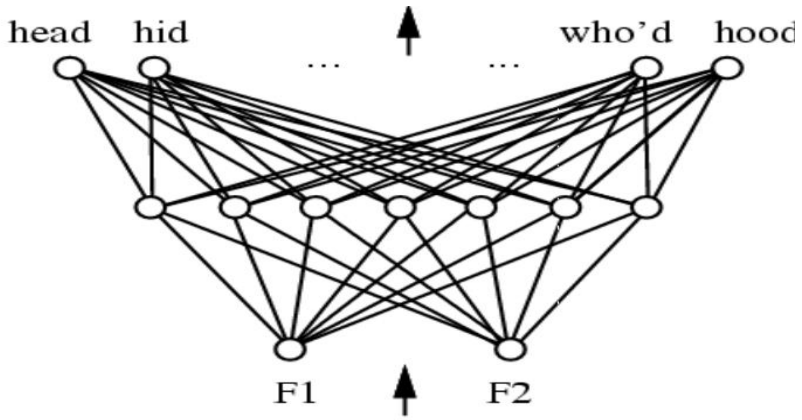
- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

Source: Aarti Singh

Training Steps



Backpropagation Algorithm (MLE)



Initialize all weights to small random numbers.
Until satisfied, Do

- For each training example, Do
  1. Input the training example to the network and compute the network outputs → Using Forward propagation
  2. For each output unit k

$$\delta_k^l \leftarrow o_k^l(1 - o_k^l)(y_k^l - o_k^l)$$

  3. For each hidden unit h

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

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

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

where

$$\Delta w_{i,j}^l = \eta \delta_j^l o_i^l$$

y_k = target output (label) of output unit k

$o_{k(h)}$ = unit output (obtained by forward propagation)

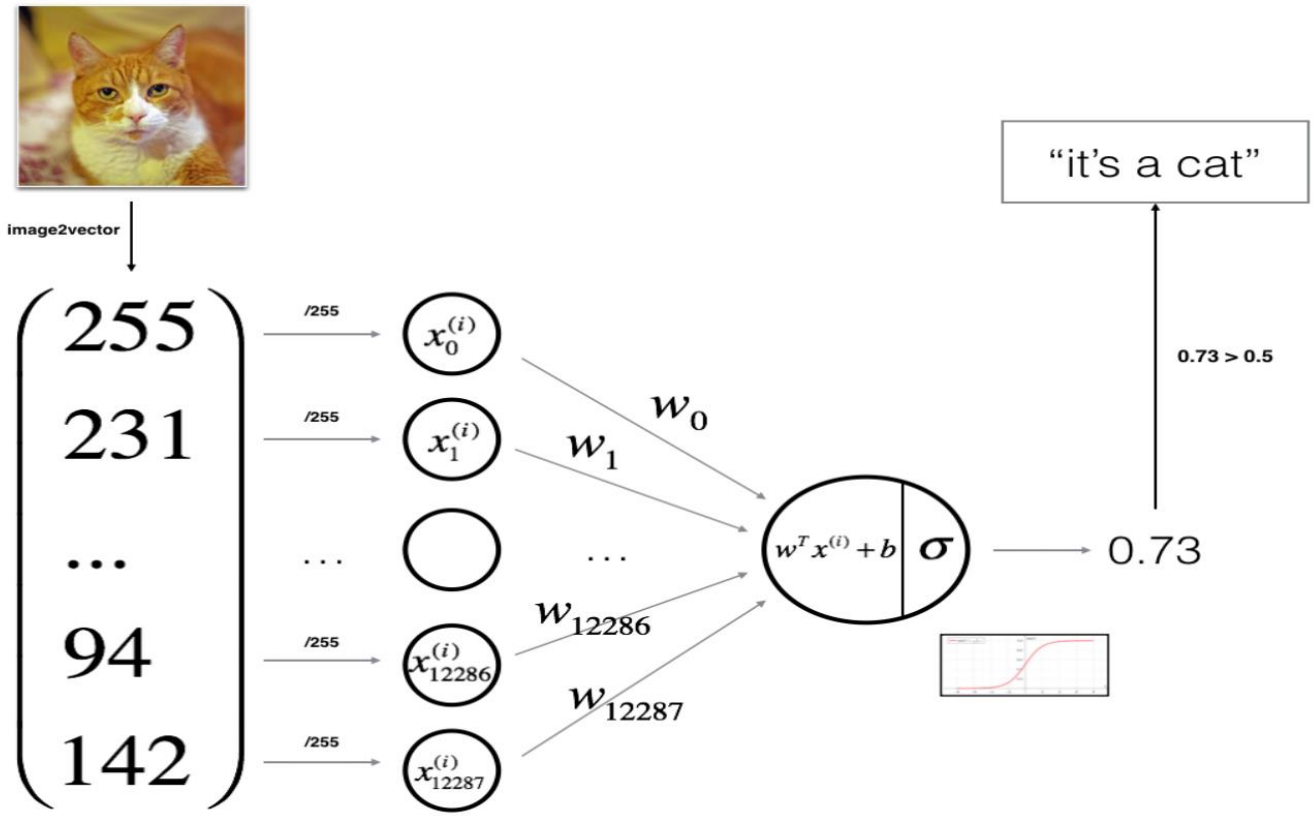
w_{ij} = wt from i to j

Note: if i is input variable, $o_i = x_i$

Source: Aarti Singh

NN training Steps

- 1. Define the neural network structure (# of input units, # of hidden units, etc).
  2. Initialize the model's parameters
  3. Loop:
  - Implement forward propagation
  - Compute loss
  - Implement backward propagation to get the gradients
  - Update parameters (gradient descent)



$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$

$$z^{(i)} = w^T x^{(i)} + b$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)})$$

cross-entropy loss

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)})$$

Source: Andrew NG

Cross-Entropy Error Measure

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)})$$

- Minus log-likelihood for the data y
- Alternative to sum-squared error for binary outputs; diverges when the network gets an output completely wrong.
- Can produce faster learning for some types of problems.
- Can learn some problems where sum-squared error gets stuck in a local minimum, because it heavily penalizes “very wrong” outputs.
- This formulation is often used for a newtwork with oneoutput predicting two classes.

Forward propagate and cost function

Forward Propagation:

- You get X
- You compute $A = \sigma(w^T X + b) = (a^{(0)}, a^{(1)}, \ldots, a^{(m-1)}, a^{(m)})$
- You calculate the cost function: $J = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$

cross-entropy loss

Here are the two formulas you will be using:

$$\frac{\partial J}{\partial w} = \frac{1}{m} X (A - Y)^T$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)})$$

Source: Andrew NG

m – number of example

w -- weights b -- bias, a scalar

X -- data of size

Y -- true "label" vector (containing 0 if non-cat, 1 if cat)

- A = sigmoid(np.dot(w.T, X) + b)
- cost = -1./m* np.sum(Y*np.log(A) + (1-Y)*np.log(1-A))
- dw = 1./m*np.dot(X, (A-Y).T)
- db = 1./m*np.sum(A-Y)

- Finally update each parameter
- W1 = W1-dW1*learning_rate,
  b1 = b1-db1*learning_rate
  W2 = W2-dW2*learning_rate
  b2 = b2-db2*learning_rate

Optimization

- update the parameters using gradient descent

  learn w and b by minimizing the cost function J.
  For a parameter θ, the update rule is $\theta = \theta - \alpha\, d\theta$,
  where α is the learning rate.

- w = w - learning_rate * dw
- b = b - learning_rate * db

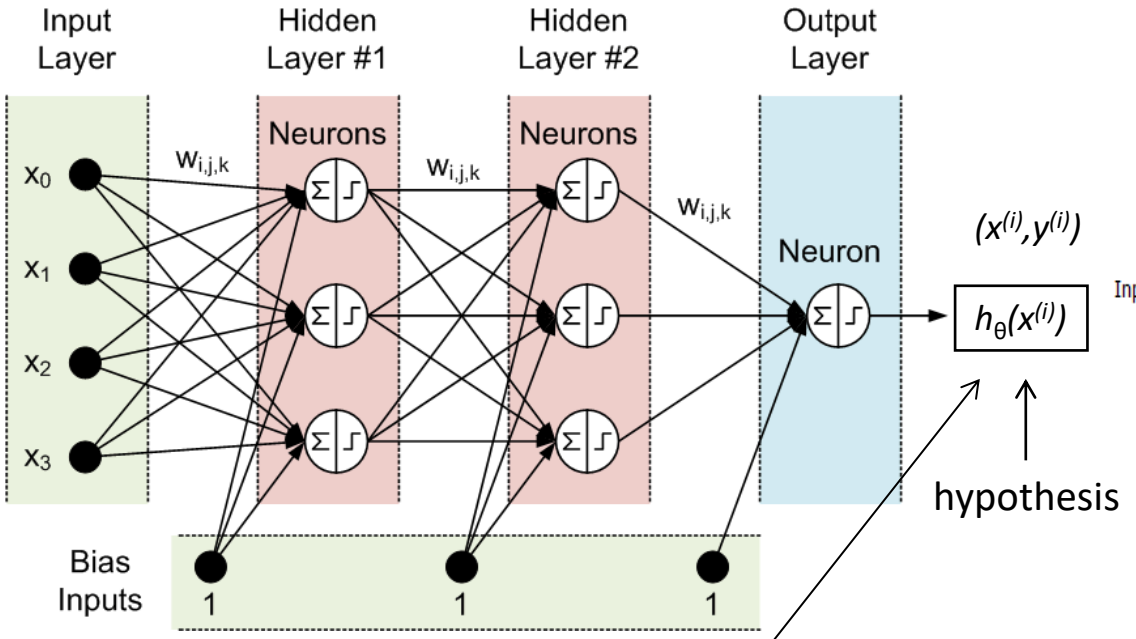
Source: Andrew NG

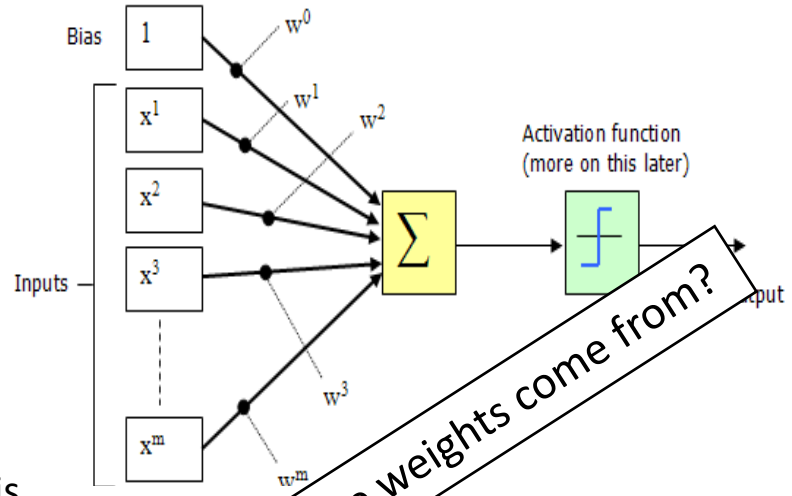
Learning with backpropagation

Solution of the complicated learning:

- calculate first the changes for the synaptic weights of the output neuron;
- calculate the changes backward starting from layer p-1, and propagate backward the local error terms.

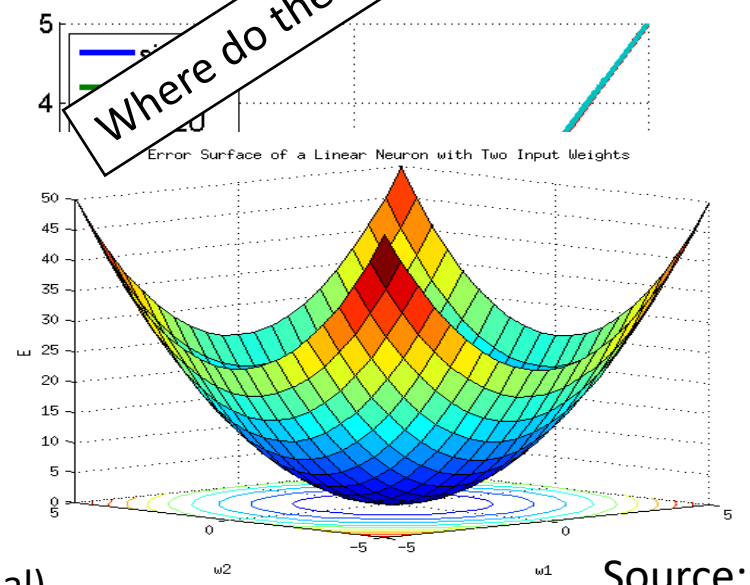
Deep Feed Forward Neural Nets





Forward Propagation

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$



Learning is the adjusting of the weights $w_{i,j}$ such that the cost function $J(\boldsymbol{\theta})$ is minimized (a form of Hebbian learning).

Simple learning procedure: *Back Propagation* (of the error signal)

Source: IETF

Issues in Training Neural Networks

- Initial values of parameters
  - Back-propagation finds local minimum
- Overfitting
  - Neural networks have too many parameters
  - Early stop and regularization
- Scaling of the inputs
  - Inputs are typically scaled to have zero mean and unit standard deviation
- Number of hidden units and layers
  - Better to have too many than too few
  - With ‘traditional’ back-propagation a long NN gets stuck in local minima and does not learn well

- http://scs.ryerson.ca/~aharley/neural-networks/
- http://cs231n.github.io/neural-networks-case-study/
- http://cs.stanford.edu/people/karpathy/cs231nfiles/minimal_net.html
- http://playground.tensorflow.org/
- https://github.com/janishar/mit-deep-learning-book-pdf

Any Questions?