Multi-Layer Networks

CSCI 447/547 MACHINE LEARNING

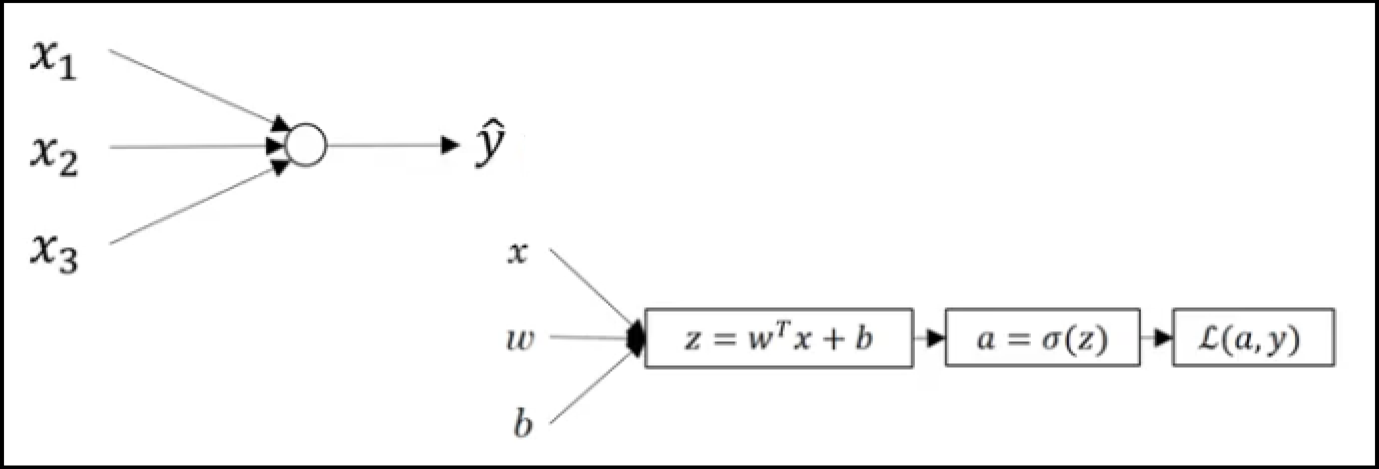
Outline

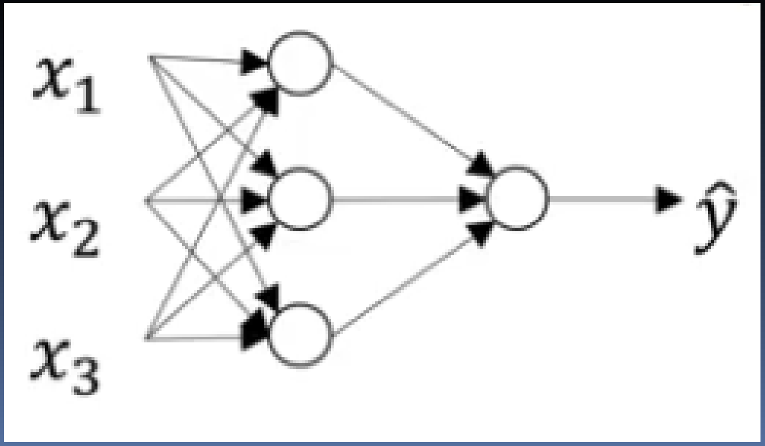
- Overview
- Representation
- Computing Output
- Vectorizing across Examples
- Activation Functions
- Gradient Descent
- Backpropagation

Overview

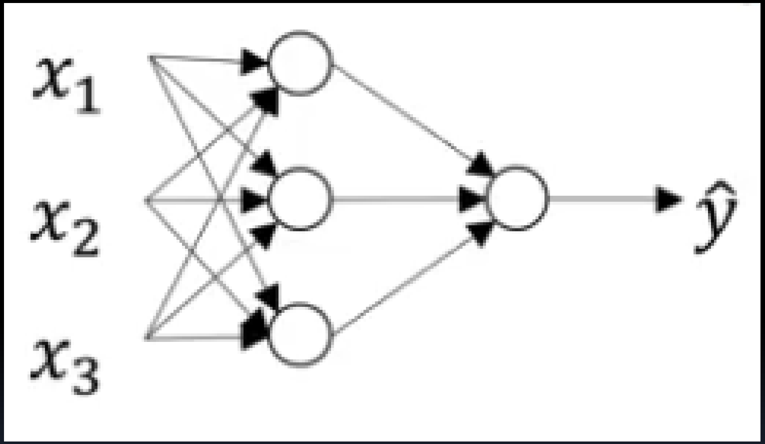
- Can think of a neural network as a function that approximates another function
    - The more neurons, the more complexity our function can have

Comparison to Logistic Regression





- 2 layer neural network
  - Don’t count input layer



Input Layer

Hidden Layer

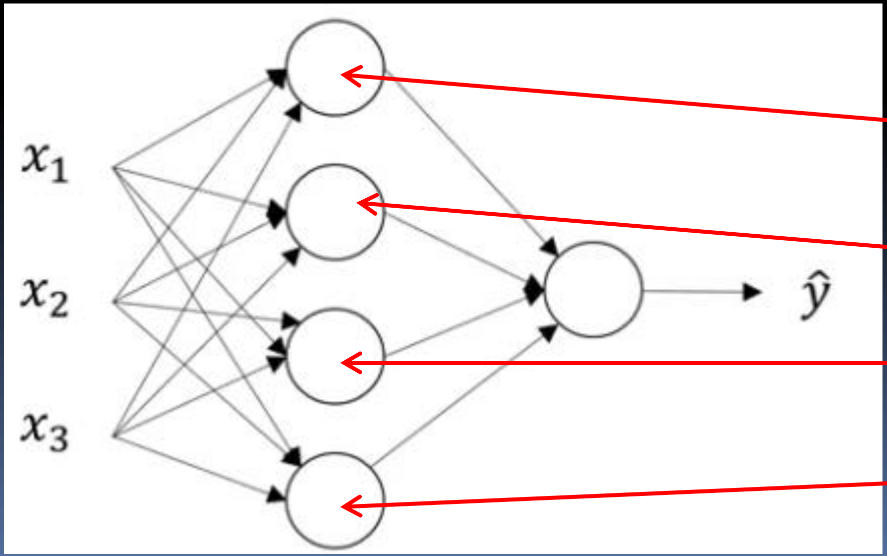
Output Layer

The Matrices / Vectors

- $a^{[0]}$ = x, the input vector
- $a^{[1]}$ = activation function of the nodes
  - $a_1^{[1]}$ for first node, $a_2^{[1]}$ for second, etc.
- $a^{[2]} = \hat{y}$
- Hidden and output layers have weight and bias matrices/vectors

Forward Propagation

- Each node calculates two things:
  - Summation of (weighted) input
  - Transfer function – output of that node



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \ a_4^{[1]} = \sigma(z_4^{[1]})$$

Vectorization

Given input x:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

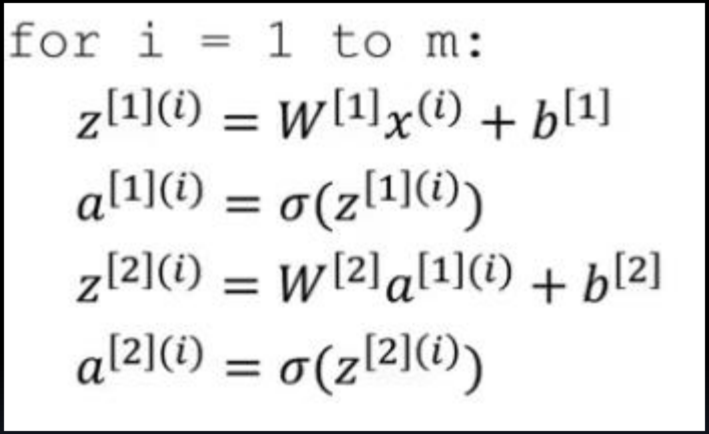
$$a^{[2]} = \sigma(z^{[2]})$$

$$\begin{bmatrix} \cdots & w_1^{[1]T} & \cdots \\ \cdots & w_2^{[1]T} & \cdots \\ \cdots & w_3^{[1]T} & \cdots \\ \cdots & w_4^{[1]T} & \cdots \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T}x + b_1^{[1]} \\ w_2^{[1]T}x + b_2^{[1]} \\ w_3^{[1]T}x + b_3^{[1]} \\ w_4^{[1]T}x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = Z^{[1]}$$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} == \sigma(Z^{[1]})$$

Vectorizing Input

- We could iteratively process each example:



```
for i = 1 to m:
    $z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$
    $a^{[1](i)} = \sigma(z^{[1](i)})$
    $z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$
    $a^{[2](i)} = \sigma(z^{[2](i)})$
```

Vectorizing Input

- Or use a matrix of input examples, each one a column

$$X = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

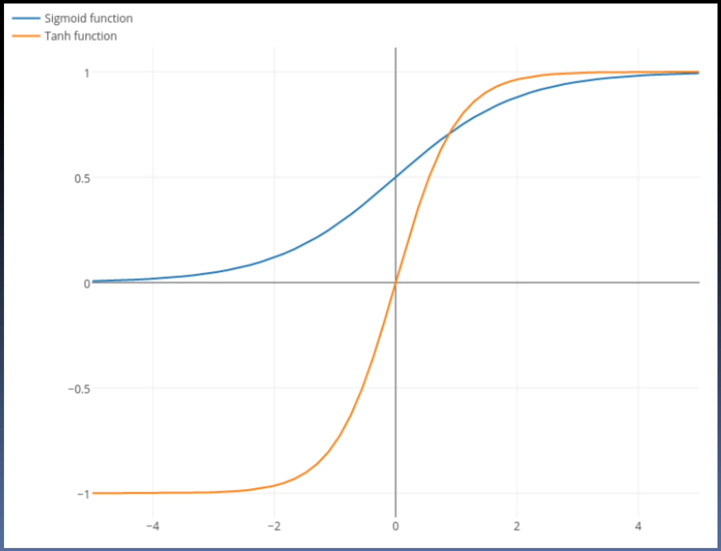
$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]}$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]}$$

- Horizontally across matrices, corresponds to different training examples
- Vertically, corresponds to different nodes in network layer

Activation Functions

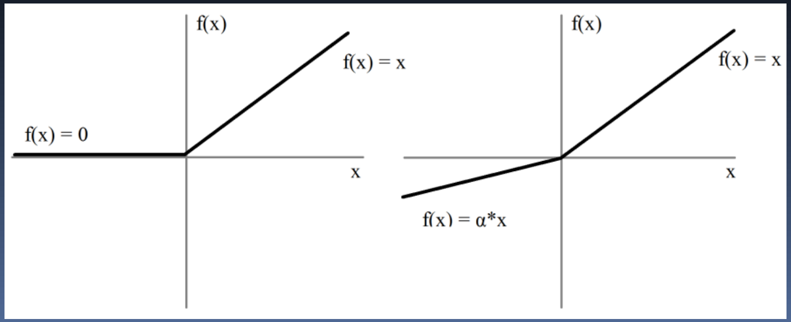
- Have been using sigmoid function
- Can use some other function
  - One that is non-linear
  - e.g. hyperbolic tangent function

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



Activation Function

- Drawback with both, if z is large or small, slope of curve is small, so gradient descent is slow
- Alternative is ReLU function
  - Rectified Linear Unit
  - a = max(0, z)
  - derivative is either 1 or 0 (can pretend it's 0 at discontinuity)
- Leaky ReLU:
  - a = max(z, 0.01z)



Why Use Activation Functions

- g(z)=z is the “identity activation function”
- If we do this, we are computing the output as a linear function of the input
- May want to do linear function at the output layer if your output warrants it, but not in the hidden layers

Derivatives of Activation Functions

- Sigmoid $g(z) = \frac{1}{1+e^{-z}}$ $\frac{d}{dz}g(z) = g(z)(1-g(z))$ $a = g(z), \text{ so } g'(z) = a(1-a)$
- Tanh

$$\frac{d}{dz}g(z) = 1 - (tanh(z))^2$$

$$g'(z) = 1 - a^2$$

- ReLU and Leaky ReLU

$$g(z) = max(0, z)$$

$$g'(z) = \begin{Bmatrix} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{Bmatrix}$$

$$g(z) = max(0.01z, z)$$

$$g'(z) = \begin{Bmatrix} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{Bmatrix}$$

Gradient Descent

- Parameters $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$
- Cost function
- Gradient Descent
  - Initialize parameters
  - Compute prediction
  - Compute derivatives
  - Update weights and bias

Backpropagation

- Similar to gradient descent in logistic regression

$$dZ^{[2]} = A^{[2]} - Y \text{ where } Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} \text{np.sum}(dZ^{[2]}, \text{axis} = 1, \text{keepdims} = \text{True})$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]}) \text{ (* is an element-wise multiplication)}$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$

$$db^{[1]} = \frac{1}{m} \text{np.sum}(dZ^{[1]}, \text{axis} = 1, \text{keepdims} = \text{True})$$

Random Initialization

- Used 0 for initialization in regression
- Should use random numbers for neural networks
  - If you initialize weights to 0 in network, end up with symmetric weight vectors across nodes, so they compute the same function
  - OK to initialize bias vector values to 0
  - Can use np.random.randn((2,2)) * 0.01
    - Normal distribution of small numbers
    - Use np.zero((2,1)) for bias numbers
  - Better to use small random numbers or you will end up on flat ends of sigmoid or tanh activation function

Summary

- Overview
- Representation
- Computing Output
- Vectorizing across Examples
- Activation Functions
- Gradient Descent
- Backpropagation