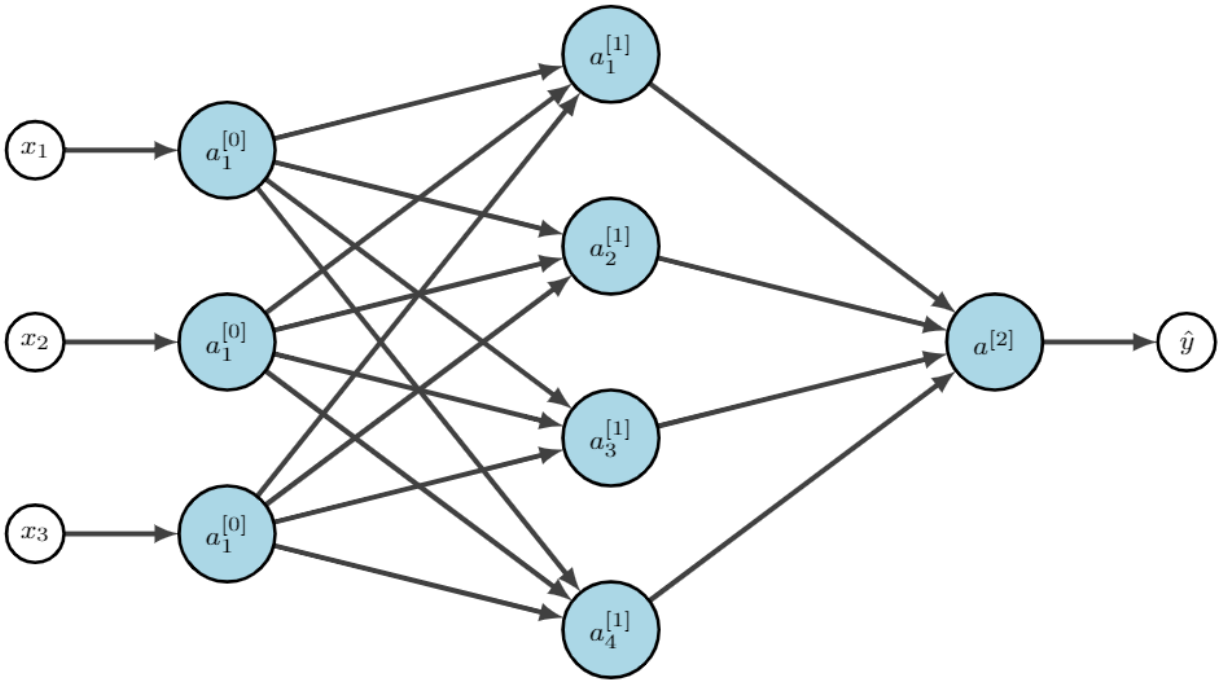
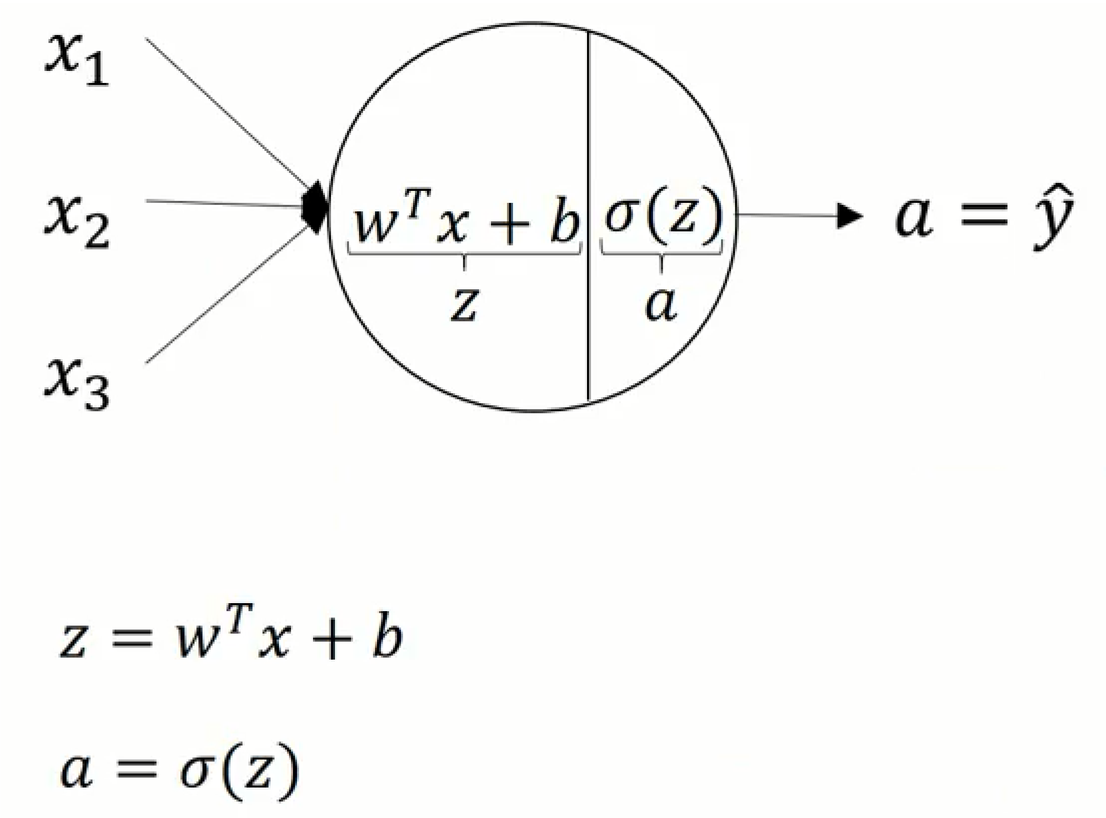
Shallow neural network overview

- hidden layer



Neural network representation

-



- $$\begin{cases} z_i^{[l]} = (w_i^{[l]})^T x + b_i^{[l]} \\ a_i^{[l]} = \sigma(z_i^{[l]}) \end{cases} \tag{1}$$
- $$a_{i \leftarrow node\ in\ layer}^{[l] \leftarrow layer} \tag{2}$$
- $$Z^{[1]} = \underbrace{\begin{bmatrix} \cdots & (w_1^{[1]})^T & \cdots \\ \cdots & (w_2^{[1]})^T & \cdots \\ \cdots & (w_3^{[1]})^T & \cdots \\ \cdots & (w_4^{[1]})^T & \cdots \end{bmatrix}}_{W^{[1]} \in \mathbb{R}^{4\times3}} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}_{b^{[1]} \in \mathbb{R}^{4\times1}} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \tag{3}$$
- $$a^{[1]} = \sigma(Z^{[1]}) \tag{4}$$

Computing a neural network output

- $Given\ input\ x:$
  - $a^{[0]} = x$
  - $z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$
  - $a^{[1]} = \sigma(z^{[1]})$
  - $z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$
  - $a^{[2]} = \sigma(z^{[2]})$

Vectorizing across multiple examples

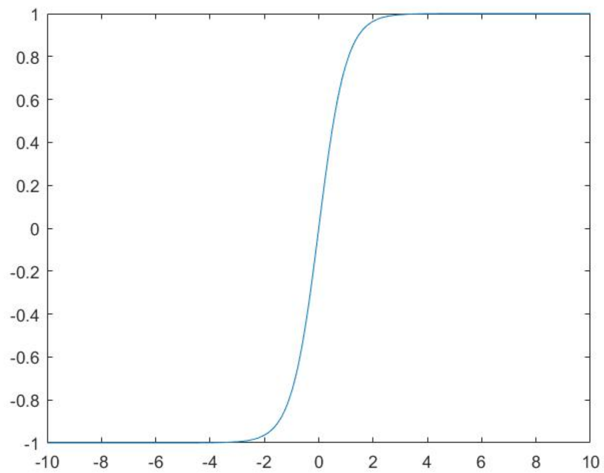
- $a^{[i](j)},\ [i]\ is\ i^{th}\ layer,\ (j)\ is\ j^{th}\ example$
- `for i = 1 to m:`
    - $a^{[0](i)} = x^{(i)}$
    - $z^{[1](i)} = W^{[1]}a^{[0](i)} + b^{[1]}$
    - $a^{[1](i)} = \sigma(z^{[1](i)})$
    - $z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$
    - $a^{[2](i)} = \sigma(z^{[2](i)})$
- Vectorizing
    - $Z^{[1]} = W^{[1]}X + b[1]$
    - $A^{[1]} = \sigma(Z^{[1]})$
    - $Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
    - $A^{[2]} = \sigma(Z^{[2]})$

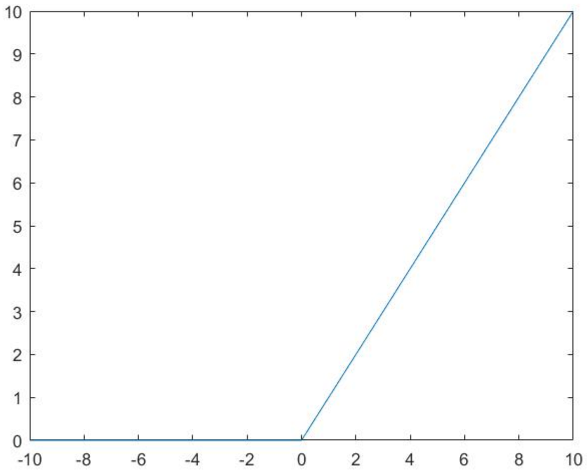
$$X = \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}, X \in \mathbb{R}^{\mathrm{n_x} \times \mathrm{m}} \tag{5}$$

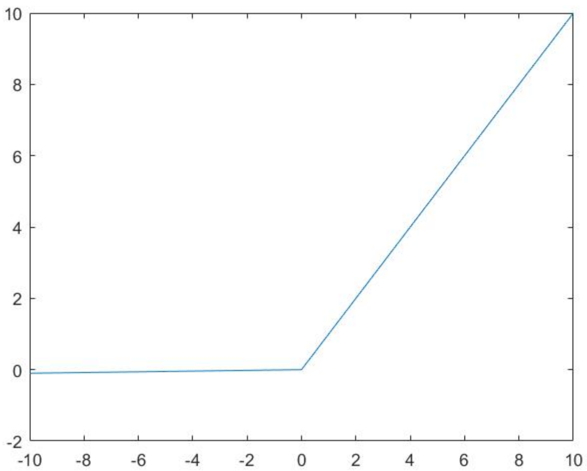
$$Z^{[i]} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ z^{[i](1)} & z^{[i](2)} & \dots & z^{[i](m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

$$A^{[i]} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ a^{[i](1)} & a^{[i](2)} & \dots & a^{[i](m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

Activation functions

- **tanh function** $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
    - $g^{[i]}(z^{[i]}) = tanh(z^{[i]})$
    - 

- when placed in **hidden layer**, *tanh* performs better than *sigmoid* functions, because it inclines to center on 0, which makes learning easier.
- In **output layer**, in the case of **binary classification**, sigmoid is still frequently used.
- Downside of *sigmoid* and *tanh*
  - when input z either very large or small, the slope tents to be trivial, which slows the learning process
- **ReLU function**
  - 

- General principle
  - hidden layer: *ReLU*(in practice preferred)/*tanh*/*leaky ReLU*
  - binary classification output: *sigmoid*
- **leaky ReLU**
  - $a = max(0.01z, z)$
  - 

Why need non-linear activation functions

- The two composite linear functions is still a linear function
  - $$a^{[2]} = w^{[2]}(w^{[1]}x + b^{[1]}) + b^{[2]} = (w^{[1]}w^{[2]})x + b^{[1]}b^{[2]} = w'x + b' \tag{6}$$
- General cases, using linear activation functions is not recommeded
- The only exception could be in the output layer

Derivatives of activation functions

- $a = g(z) = tanh(z)$
  - $$g'(z) = 1 - (tanh(z))^2 = 1 - a^2 \tag{7}$$
- $a = g(z) = ReLU(z) = max(0, z)$
  - $$g'(z) = \begin{cases} 0 & if\ z < 0 \\ 1 & if\ z \geq 0 \end{cases} (not\ mathematically\ correct) \tag{8}$$
- $a = g(z) = leaky\_ReLU(z) = max(0.01z, z)$
  - $$g'(z) \begin{cases} 0.01 & if\ z < 0 \\ 1 & if\ z \geq 0 \end{cases} \tag{9}$$

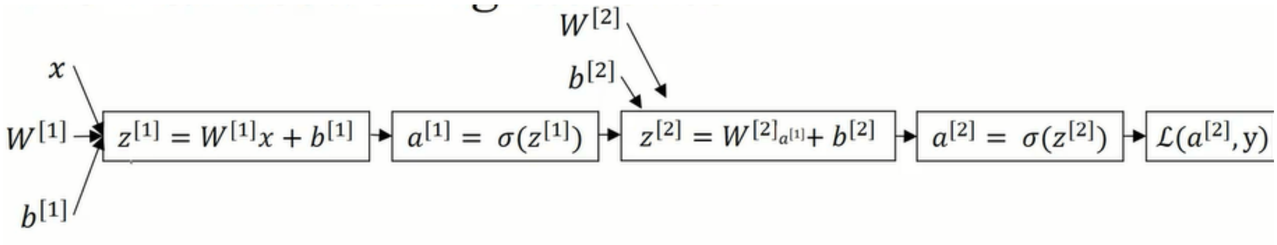
Gradient descent for neural network

- $Parameters: \underbrace{w^{[1]}}_{(n^{[1]}, n^{[0]})}, \underbrace{b^{[1]}}_{(n^{[1]}, 1)}, \underbrace{w^{[2]}}_{(n^{[2]}, n^{[0]})}, \underbrace{b^{[2]}}_{(n^{[2]}, 1)}$
- $Costfunction: J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{m} \mathscr{L}(\underbrace{\hat{y}}_{a^{[2]}}, y)$

Formulas for computing derivatives

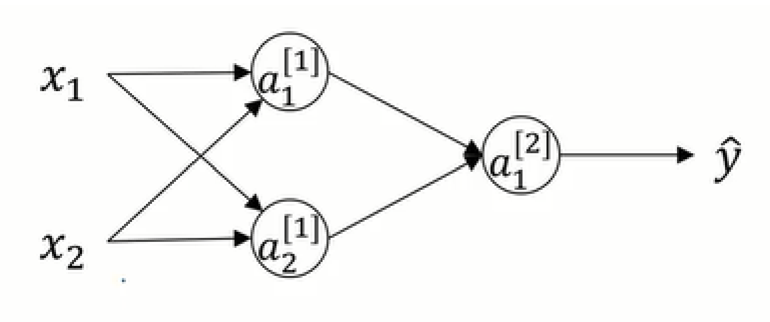
- $Forward\ propagation:$
  - $Z^{[1]} = w^{[1]}X + b^{[1]}$
  - $A^{[1]} = g^{[1]}(Z^{[1]})$
  - $Z^{[2]} = w^{[2]}A^{[1]} + b^{[2]}$
  - $A^{[2]} = g^{[2]}(Z^{[2]})$
- $Backward\ propagation$
  - $dZ^{[2]} = A^{[2]} - Y$
  - $dw^{[2]} = \frac{1}{m}dZ^{[2]}(A^{[1]})^T$
  - $db^{[2]} = \frac{1}{m}np.sum(dZ^{[2]},\ axis = 1,\ keepdims = True)$
  - $dZ^{[1]} = \underbrace{(w^{[2]})^T dZ^{[2]}}_{(n^{[2]}, m)} \underbrace{.*}_{element-wise\ product} \underbrace{g^{[1]'}(Z^{[1]})}_{(n^{[2]}, m)}$
  - $dw^{[1]} = \frac{1}{m}dZ^{[1]}X^T$
  - $db^{[1]} = \frac{1}{m}np.sum(dZ^{[1]},\ axis = 1,\ keepdims = True)$

Back propagation

- 

- $dz^{[2]} = a^{[2]} - y$
- $dW^{[2]} = dz^{[2]}(a^{[2]})^T \rightarrow dw = dz * x,\ W^{[2]}\ is\ a\ matrix\ stacked\ with\ many\ individual\ w$
- $db^{[2]} = dz^{[2]}$
- $dz^{[1]} = \underbrace{(w^{[2]})^T dz^{[2]}}_{(n^{[1]}, n^{[2]}) * (n^{[2]}, 1)} .* \underbrace{g^{[1]'}(z^{[1]})}_{(n^{[1]}, 1)}$
- $dW^{[1]} = dz^{[1]}(a^{[0]})^T$
- $db^{[1]} = dz^{[1]}$

Random initialization

- What if initializing the weights to zero
  - 

  - $then\ W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
  - $a^{[1]} = a^{[2]},\ the\ hidden\ layer\ would\ be\ symmetric$
- Random
  - $W^{[1]} = np.random.randn([2, 2]) * 0.01$
    - why choose scalar 0.01
    - if too large, the downside of sigmoid and tanh would be more severe, particularly in the classification case
  - $b^{[1]} = np.zeros([2, 1])$
    - b does't have any issue
  - $W^{[2]} = np.random.randn([1, 2]) * 0.01$
  - $b^{[1]} = 0$