SAN DIEGO STATE UNIVERSITY

Computational Graph and Neural Network

Instructor: Yang Xu

Recap

- Evaluation of ML models
  - Train/dev/test
  - Performance of classification models
    - TP, TN, FP, FN
    - Accuracy, Precision, Recall, F1 score
  - Overfitting
  - L2 regularization
    - Minimize(Cost + Complexity)

Outline

- Computation graph basic concepts
- Vectorization of logistic regression
- One hidden layer neural network

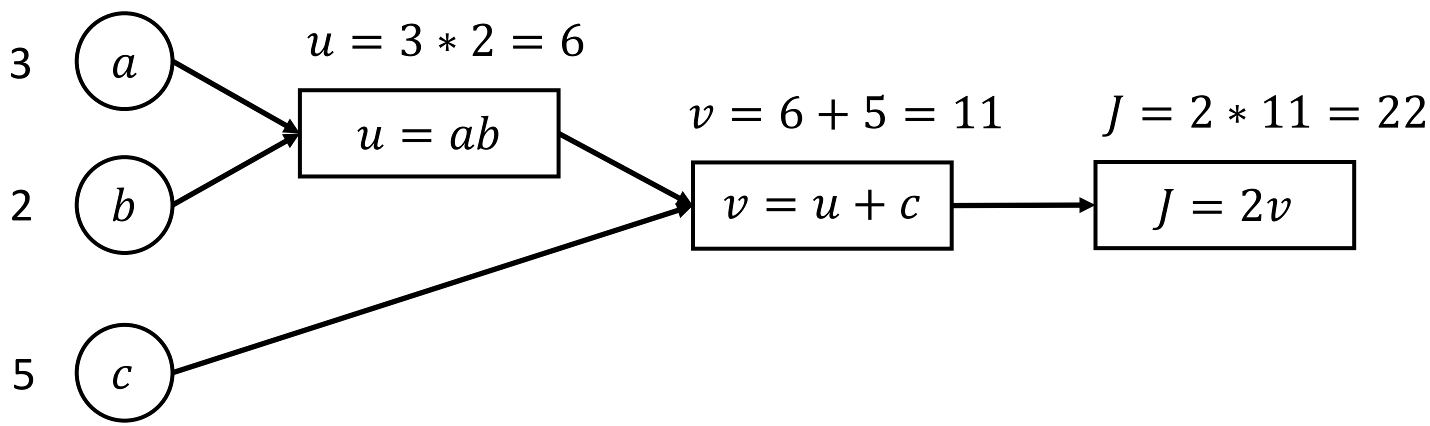
Computation Graph

- What is it?
- A useful tool to understand/visualize ML models

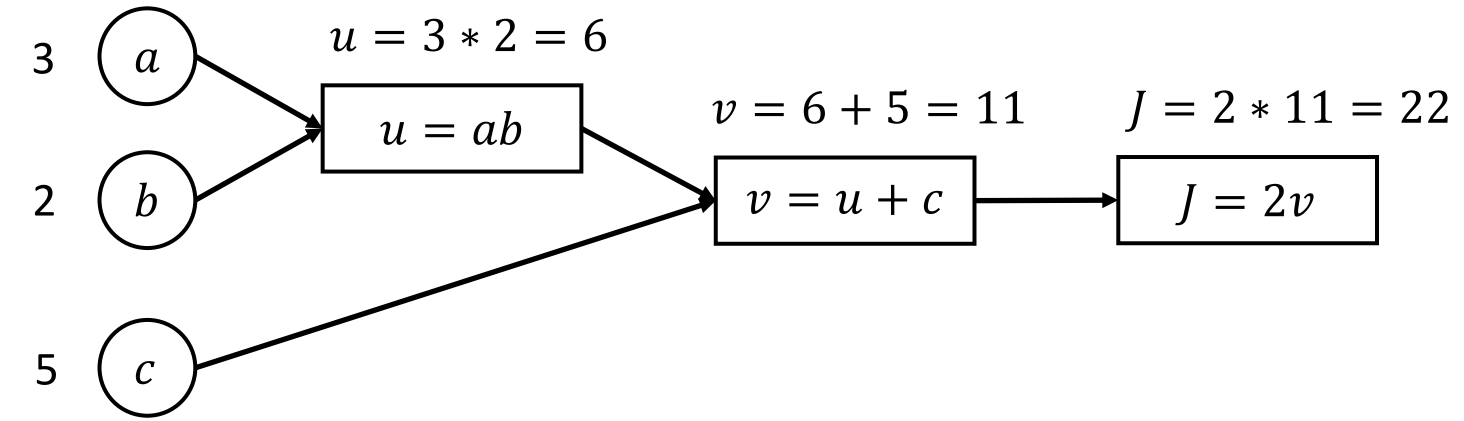
- Simple arithmetic example

$J(a, b, c) = 2(ab + c)$ Set intermediate variables $u = ab$

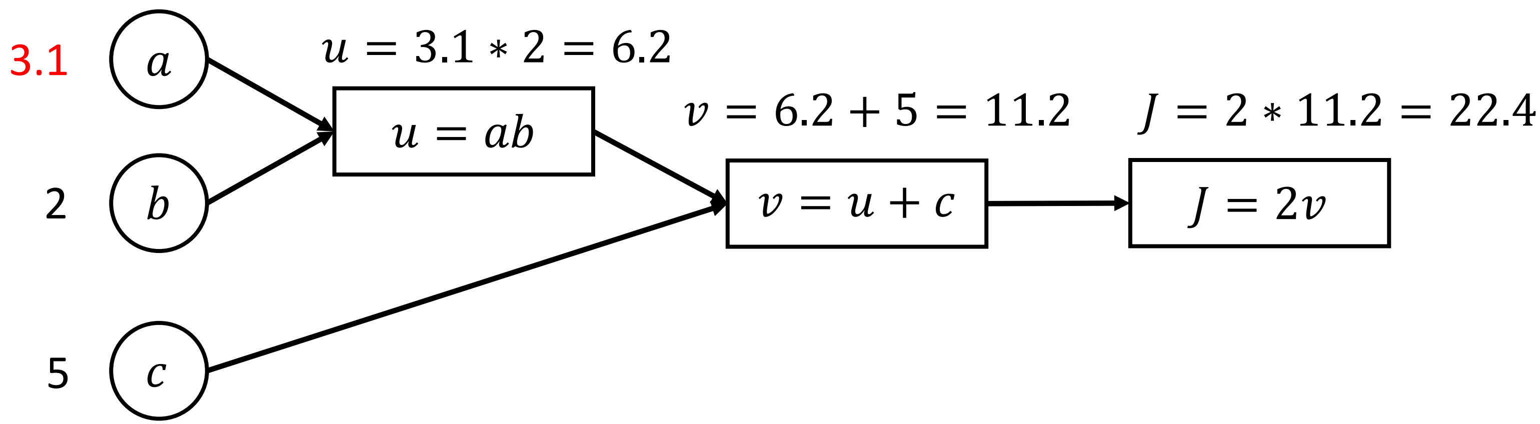
$v = u + c$



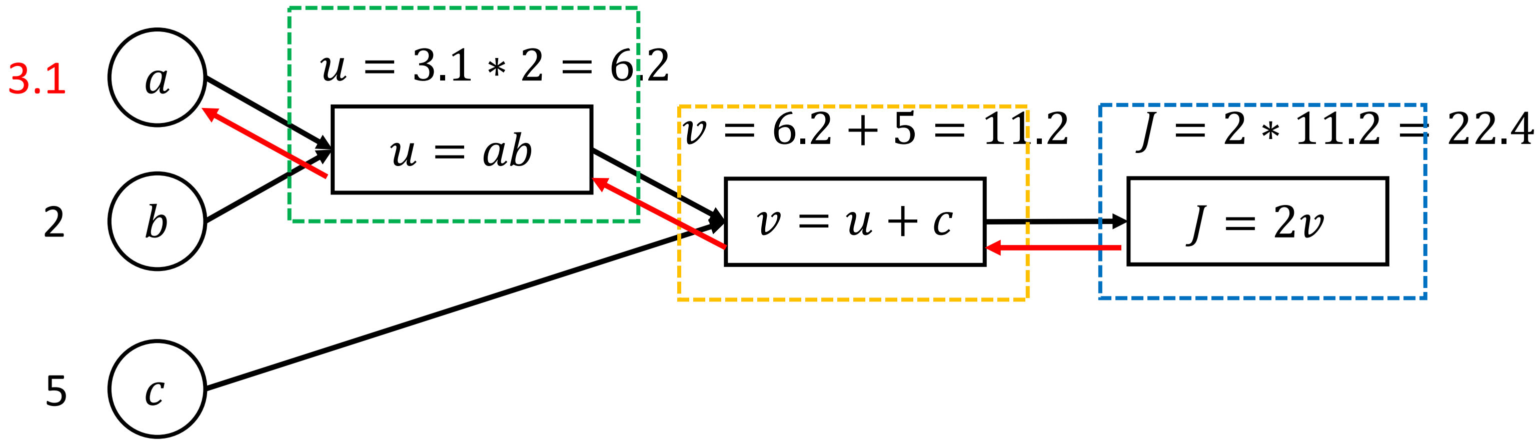
Input change leads to output change



How much will J change if a changes a little bit?



Derivative of J w.r.t a



$$\frac{\partial J}{\partial a} = \frac{22.4 - 22}{3.1 - 3} = 4$$

J will change by 4 units if a changes by 1

What about going Backward step by step:

$$\frac{\partial u}{\partial a} = \frac{6.2 - 6}{3.1 - 3} = 2$$

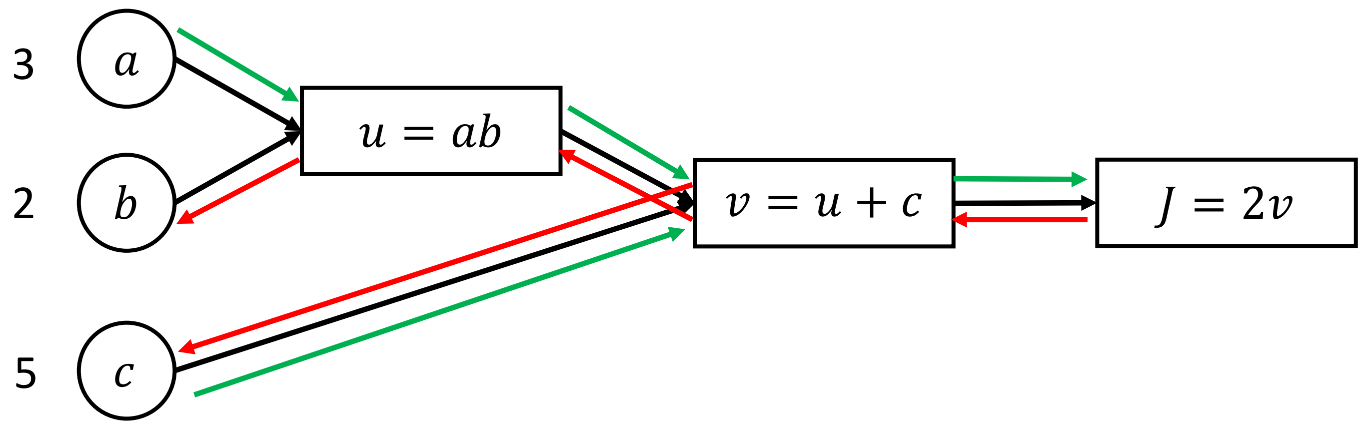
$$\frac{\partial v}{\partial u} = \frac{11.2 - 11}{6.2 - 6} = 1$$

$$\frac{\partial J}{\partial v} = \frac{22.4 - 22}{11.2 - 11} = 2$$

Apply chain rule:

$$\frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial a} = 2 * 1 * 2 = 4 = \frac{\partial J}{\partial a}$$

Derivative of J w.r.t all inputs



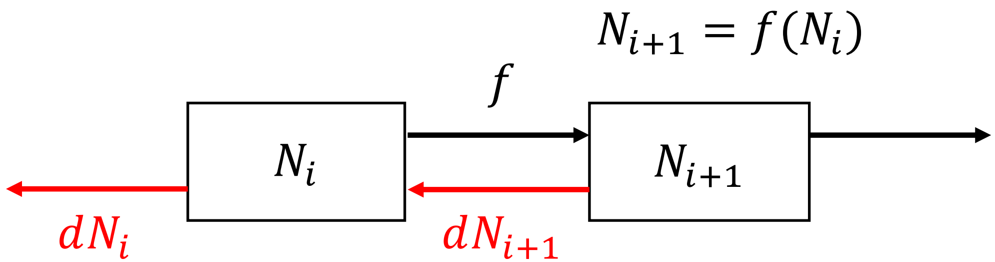
$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial b} = 2 * 1 * 3 = 6$$

$$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial c} = 2 * 1 = 2$$

Forward pass (green), compute the cost J

Backward pass (red), compute the derivative $\frac{\partial J}{\partial ?}$

The function of nodes



$$\frac{\partial L}{\partial N_i} = ? \qquad \boxed{\frac{\partial L}{\partial N_{i+1}}}$$

$$= \frac{\partial N_{i+1}}{\partial N_i} \cdot \frac{\partial L}{\partial N_{i+1}}$$

$$= \boxed{f' \frac{\partial L}{\partial N_{i+1}}}$$

Computation graph for logistic regression

San Diego State University

- Recap

$$z = w^T x + b$$

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

$$L(a, y) = -(y \log(a) + (1-y)\log(1-a))$$

a for "activation"

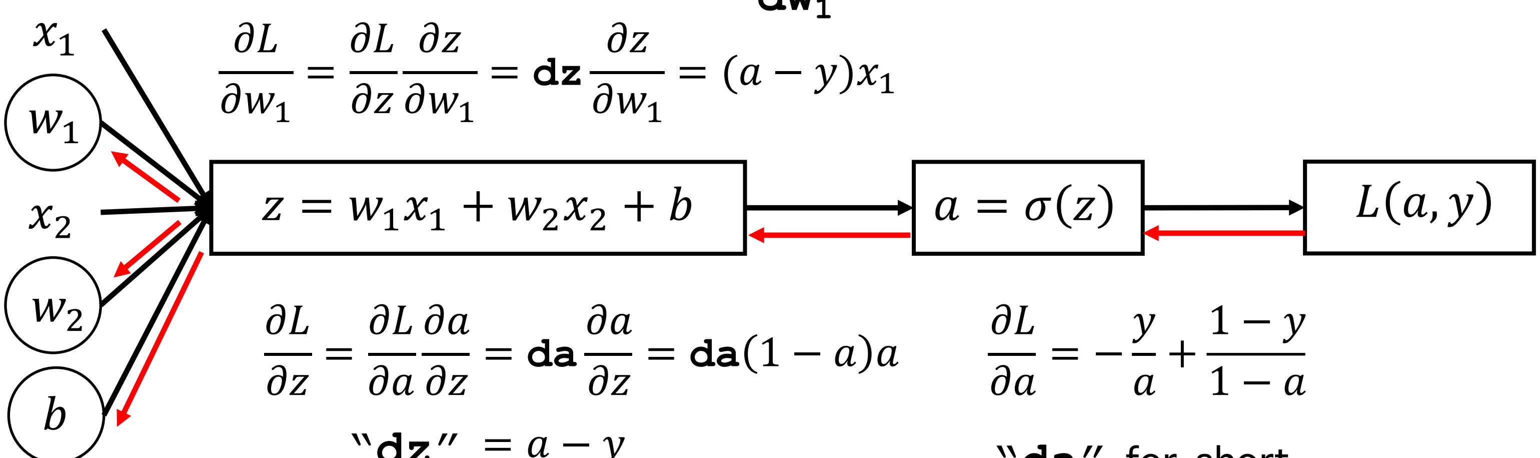
loss function $L(a, y)$

= The **cost** of a single training example

$$w_1 = w_1 - \alpha \mathbf{dw_1}$$
$$w_2 = w_2 - \alpha \mathbf{dw_2}$$
$$b = b - \alpha \mathbf{db}$$

"$\mathbf{dw_1}$"

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_1} = \mathbf{dz}\frac{\partial z}{\partial w_1} = (a-y)x_1$$



$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z} = \mathbf{da}\frac{\partial a}{\partial z} = \mathbf{da}(1-a)a$$

"$\mathbf{dz}$" $= a - y$

$$\frac{\partial L}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a}$$

"$\mathbf{da}$" for short

What about *m* examples?

Cost function: $J(w,b) = \frac{1}{m}\sum_i L(a^{(i)} - y^{(i)})$

$$\Rightarrow \frac{\partial J(w,b)}{\partial w_1} = \frac{1}{m}\sum_i \frac{\partial}{\partial w_1} L(a^{(i)} - y^{(i)}) = \frac{1}{m}\sum_i dw_1^{(i)}$$

Implementation with *for* loops

$dw_1 = 0; dw_2 = 0; \dots dw_n = 0, db = 0$

For i = 1 to m:

Forward $\begin{cases} z^{(i)} = w^T x^{(i)} + b \\ a^{(i)} = \sigma(z^{(i)}) \end{cases}$

Backward $\begin{cases} dz^{(i)} = a^{(i)} - y^{(i)} \\ \textbf{For } j = 1 \text{ to n:} \\ \quad dw_j \mathrel{+}= x_j^{(i)} dz^{(i)} \\ db \mathrel{+}= dz^{(i)} \end{cases}$

$dw_1 = \frac{dw_1}{m}; dw_2 = \frac{dw_2}{m}; \dots dw_n = \frac{dw_n}{m}, db = \frac{db}{m}$

Vectorization needed!

$w_1 = w_1 - \alpha \mathbf{dw_1}$

$w_2 = w_2 - \alpha \mathbf{dw_2}$

$b = b - \alpha \mathbf{db}$

Vectorization

$$z = w^T x + b \qquad w \in \mathbb{R}^n, x \in \mathbb{R}^n$$

Non-vectorized code

```
z = 0
for i in range(n):
    z += w[i]*x[i]
z += b
```

Vectorized code

```
z = np.dot(w, x) + b
```

This is way faster!

Because CPU/GPU has parallelization instructions, SIMD (single instruction multiple data).
Parallelism is enable in built-in function such as np.dot()

Demo: vectorization vs. non-vectorization

In [6]:
```python
import numpy as np
import time

a = np.random.rand(1000000)
b = np.random.rand(1000000)
```

In [8]:
```python
# Vectorized
start_time = time.time()
c = np.dot(a, b)
end_time = time.time()

print('Vectorized dot product: {} ms'.format(1000*(end_time - start_time)))
```

```
Vectorized dot product: 1.02996826171875 ms
```

In [9]:
```python
# Non-vectorized
start_time = time.time()
c = 0
for i in range(len(a)):
    c += a[i]*b[i]
end_time = time.time()

print('Non-vectorized dot product: {} ms'.format(1000*(end_time - start_time)))
```

```
Non-vectorized dot product: 342.3280715942383 ms
```

Vectorization is more than X300 faster

Whenever possible, avoid using explicit for loops

Vectorize logistic regression (forward)

SAN DIEGO STATE UNIVERSITY

$z^{(1)} = w^T x^{(1)} + b$ $\quad$ $z^{(2)} = w^T x^{(2)} + b$ $\quad$ ... $\quad$ $z^{(m)} = w^T x^{(m)} + b$

$a^{(1)} = \sigma(z^{(1)})$ $\quad$ $a^{(2)} = \sigma(z^{(2)})$ $\quad$ $\quad$ $a^{(m)} = \sigma(z^{(m)})$

$$X \in \mathbb{R}^{n \times m} = \begin{bmatrix} | & | & | & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & | & | \end{bmatrix}$$

$$Z = [z^{(1)}, z^{(2)}, \cdots, z^{(m)}] = w^T X + [b, b, \cdots, b]$$

One line numpy command: `Z = np.dot(w.T, X) + b` → Automatic "broadcast" form number to vector

$$A = \left[a^{(1)}, a^{(2)}, \cdots, a^{(m)}\right] = [\sigma(z^{(1)}), \sigma(z^{(2)}), \cdots, \sigma(z^{(m)})]$$

numpy command: `A = 1 / (1+np.exp(-Z))`

Vectorizing gradient computation (backward)

$dz^{(1)} = a^{(1)} - y^{(1)}$ $\quad dz^{(2)} = a^{(2)} - y^{(2)}$ $\quad \cdots \quad$ $dz^{(m)} = a^{(m)} - y^{(m)}$

Vectorize: $A = \left[a^{(1)}, a^{(2)}, \cdots, a^{(m)}\right]$ $\quad Y = \left[y^{(1)}, y^{(2)}, \cdots, y^{(m)}\right]$

$$dZ = A - Y = [a^{(1)} - y^{(1)}, a^{(2)} - y^{(2)}, \cdots, a^{(m)} - y^{(m)}]$$

$dw_1 = 0; dw_2 = 0; \dots dw_n = 0, db = 0$

For i = 1 to m:

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

$\quad \cancel{a^{(i)} = \sigma(z^{(i)})}$

$\quad \cancel{dz^{(i)} = a^{(i)} - y^{(i)}}$

$\quad$ For j = 1 to n:

$\quad\quad dw_j \mathrel{+}= x_j^{(i)} dz^{(i)}$

$\quad \cancel{db \mathrel{+}= dz^{(i)}}$

$dw_1 = \frac{dw_1}{m}; dw_2 = \frac{dw_2}{m}; \dots dw_n = \frac{dw_n}{m}, db = \frac{db}{m}$

$$db = \frac{1}{m} \sum_i dz^{(i)}$$

numpy:

db = np.sum(dZ) / m

What about dw?

Vectorizing gradient computation (cont.)

$dw_1 = 0; dw_2 = 0; \ldots dw_n = 0, db = 0$

For i = 1 to m:

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

$\cancel{a^{(i)} = \sigma(z^{(i)})}$

$\cancel{dz^{(i)} = a^{(i)} - y^{(i)}}$

For j = 1 to n:

$\cancel{dw_j += x_j^{(i)} dz^{(i)}}$

$\cancel{db += dz^{(i)}}$

$dw_1 = \frac{dw_1}{m}; dw_2 = \frac{dw_2}{m}; \ldots dw_n = \frac{dw_n}{m}, db = \frac{db}{m}$

$$\boxed{dw = \frac{1}{m} X dz^T}$$

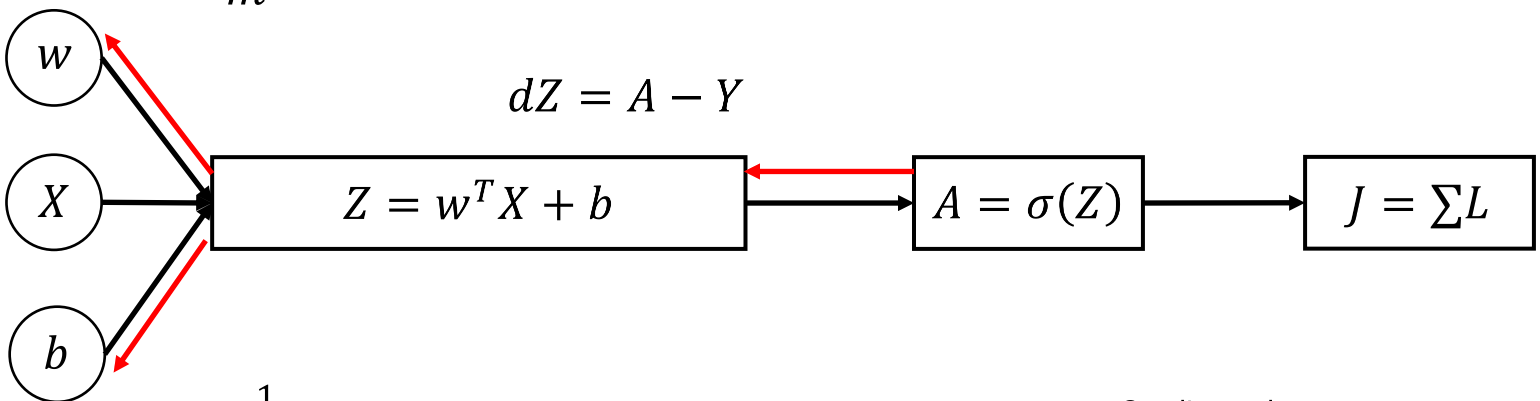
$$= \frac{1}{m} \begin{bmatrix} | & & | \\ x^{(1)} & \ldots & x^{(m)} \\ | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

$$= \frac{1}{m} [x^{(1)} dz^{(1)} + \cdots + x^{(m)} dz^{(m)}]$$

$$\frac{1}{m} \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \ldots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \ldots & x_2^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ x_n^{(1)} & x_n^{(2)} & \ldots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ dz^{(2)} \\ \vdots \\ dz^{(m)} \end{bmatrix} = \begin{bmatrix} \frac{1}{m}(x_1^{(1)} dz^{(1)} + \cdots + x_1^{(m)} dz^{(m)}) \\ \frac{1}{m}(x_2^{(1)} dz^{(1)} + \cdots + x_2^{(m)} dz^{(m)}) \\ \vdots \\ \frac{1}{m}(x_n^{(1)} dz^{(1)} + \cdots + x_n^{(m)} dz^{(m)}) \end{bmatrix} = \begin{bmatrix} dw_1 \\ dw_2 \\ \vdots \\ dw_n \end{bmatrix}$$

Vectorizing logistic regression (recap)

$$dw = \frac{1}{m} X dZ^T$$



$$db = \frac{1}{m} \text{np.sum}(dZ)$$

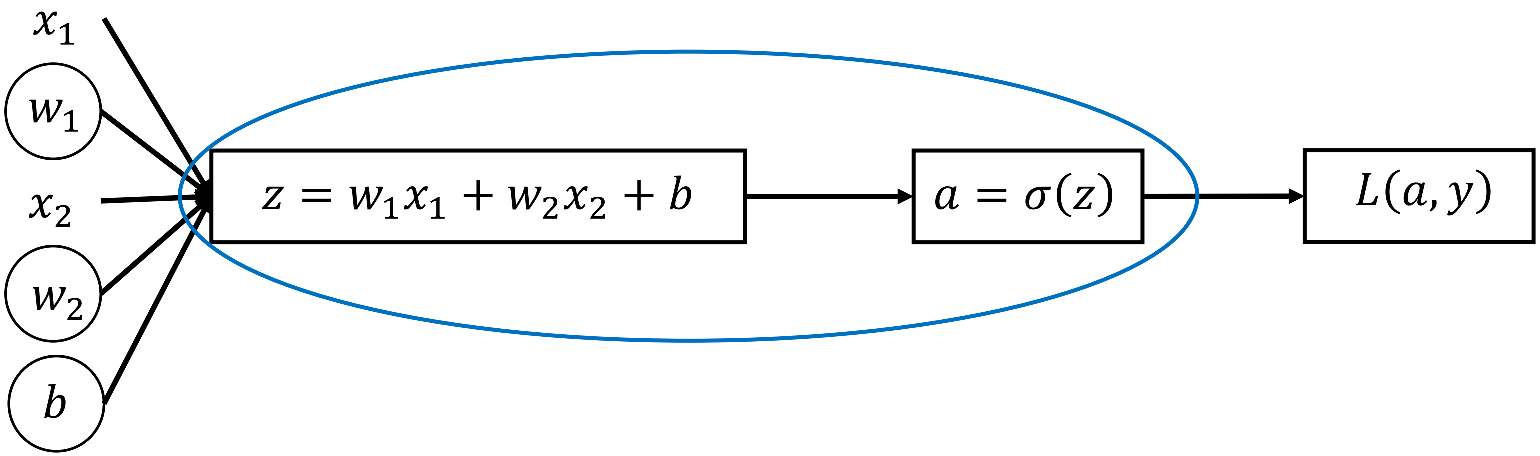
Gradient descent

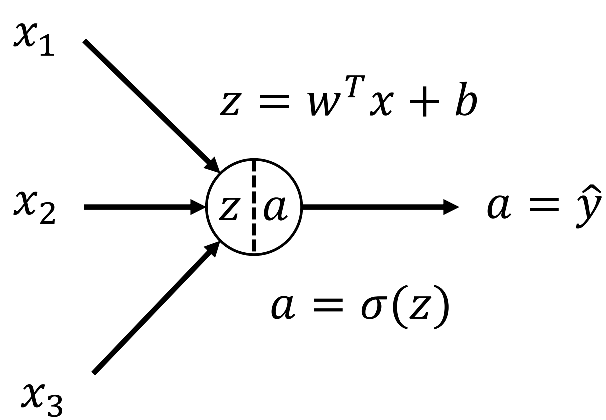
For loop → Repeat until s.t. {

$$Z = w^T X + b$$
$$A = \sigma(Z)$$
$$dZ = A - Y$$
$$dw = \frac{1}{m} X dZ^T$$
$$db = \frac{1}{m} \text{np.sum}(dZ)$$
$$w = w - \alpha dw$$
$$b = b - \alpha db$$

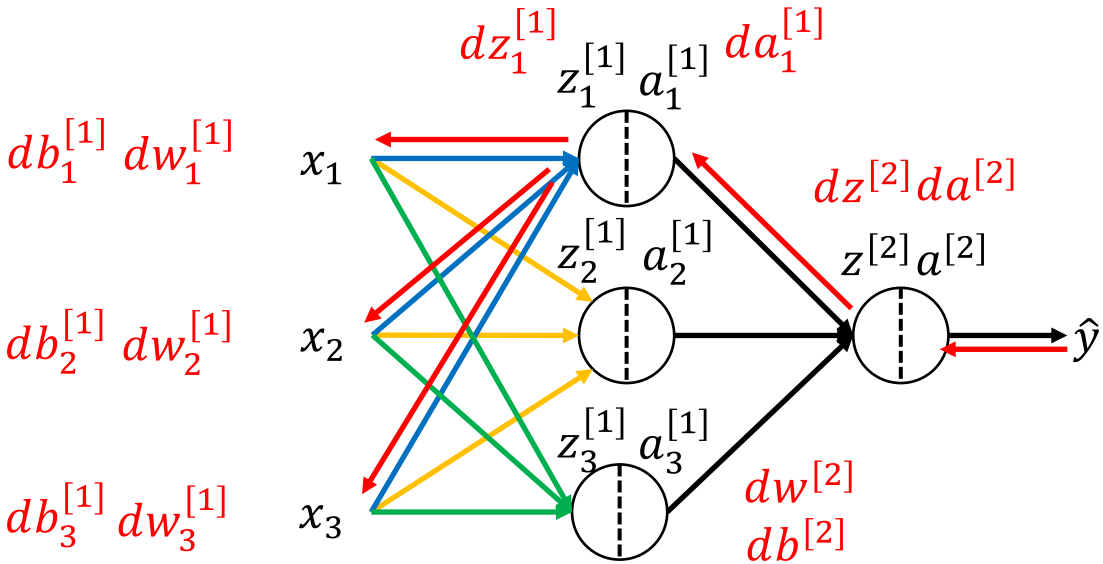
}

Wrapped into a single unit





A neural network



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

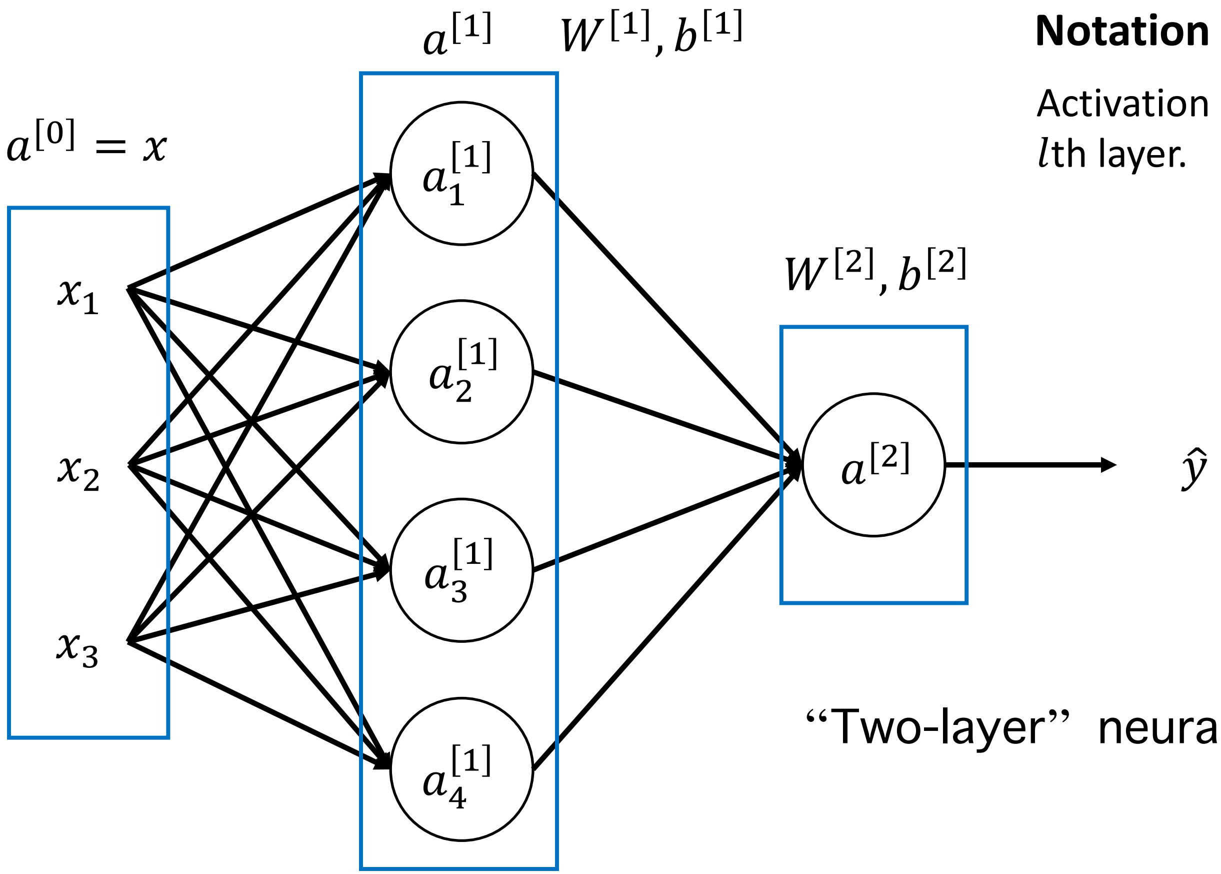
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]} \qquad a_2^{[1]} = \sigma(z_2^{[1]})$$

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

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

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

Formal definition



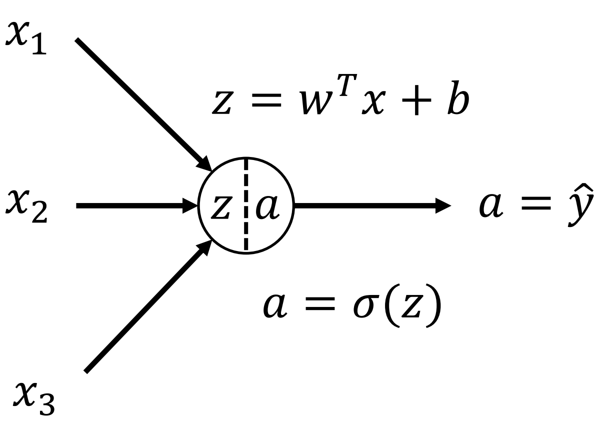
Notation $a_i^{[l]}$

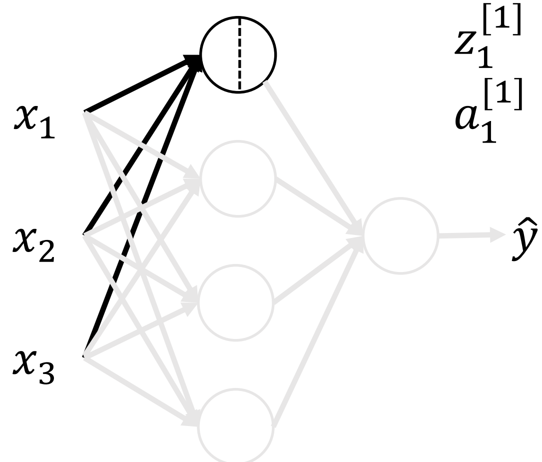
Activation of ith unit of lth layer.

“Two-layer” neural network

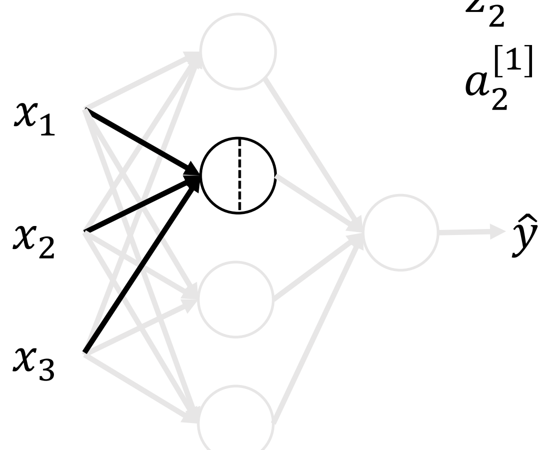
Input layer Hidden layer Output layer

Computing output (one training example)



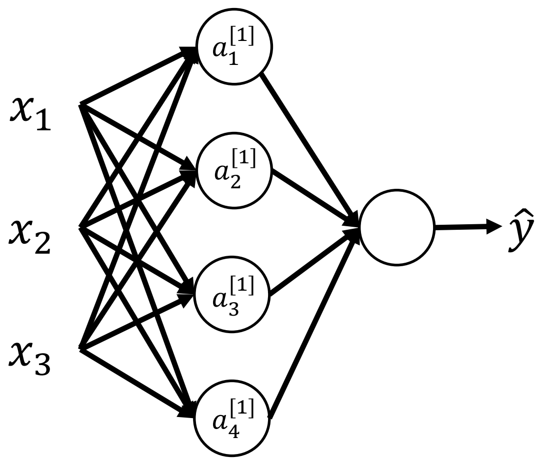


$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$$
$$a_1^{[1]} = \sigma(z_1^{[1]})$$



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

Vector representation



$$
\begin{aligned}
z_1^{[1]} &= \boxed{w_1^{[1]T} x} + \boxed{b_1^{[1]}}, \quad & a_1^{[1]} &= \boxed{\sigma(z_1^{[1]})} \\
z_2^{[1]} &= w_2^{[1]T} x + b_2^{[1]}, \quad & a_2^{[1]} &= \sigma(z_2^{[1]}) \\
z_3^{[1]} &= w_3^{[1]T} x + b_3^{[1]}, \quad & a_3^{[1]} &= \sigma(z_3^{[1]}) \\
z_4^{[1]} &= w_4^{[1]T} x + b_4^{[1]}, \quad & a_4^{[1]} &= \sigma(z_4^{[1]})
\end{aligned}
$$

$$
\underbrace{\begin{bmatrix} — & w_1^{[1]} & — \\ — & w_2^{[1]} & — \\ — & w_3^{[1]} & — \\ — & w_4^{[1]} & — \end{bmatrix}}_{4\times 3}
\underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{3\times 1}
+
\underbrace{\begin{bmatrix} b_1^{[1]} \\ b_1^{[2]} \\ b_1^{[3]} \\ b_1^{[4]} \end{bmatrix}}_{4\times 1}
=
\begin{bmatrix} w_1^{[1]T} x & +b_1^{[1]} \\ w_2^{[1]T} x & +b_1^{[2]} \\ w_3^{[1]T} x & +b_1^{[3]} \\ w_4^{[1]T} x & +b_1^{[4]} \end{bmatrix}
=
\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}
= \underset{4\times 1}{z^{[1]}}
$$

$\boxed{W^{[1]}}$ $\boxed{b^{[1]}}$

$$
\underset{4\times 1}{a^{[1]}} = \boxed{\begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}} = \sigma(z^{[1]})
$$