Deep Learning Specialization

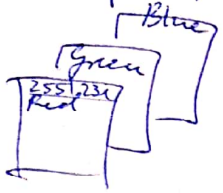
week 2 course I

Binary Classification

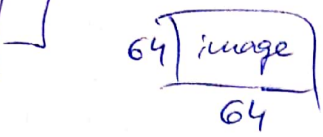
with
↳ Logistic Regression

cat image → 1 (cat) / 0 (non cat) } output label y

How is image represented?



Pixels 64×64 } unroll in feature vector x



$$x = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ \} R \\ \} G \\ \} B \end{bmatrix} \rightarrow \text{total dimension } 64 \times 64 \times 3$$

R, G, B

$L = 12288$

$n_x = 12288$

Notation

(x, y) $x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$

m training examples $\{(x^{(1)}, y^{(1)}), \dots (x^{(m)}, y^{(m)})\}$

m_{train}, m_{test}, ($m_{CV} \dots$)

matrix X = vectors $x^{(1)}, x^{(2)} \dots x^{(m)}$, n_x rows, m columns

$X \in \mathbb{R}^{n_x \times m}$ (rows, columns)

Python command for finding shape of matrix: $X.shape = (n_x, m)$

$Y = [y^{(1)}, y^{(2)} \dots y^{(m)}]$ (labeled)

$Y \in \mathbb{R}^{1 \times m}$ row vector

$Y.shape = (1, m)$

Logistic Regression

Given X, we want $\hat{y} = P(y=1|x)$ $0 \le \hat{y} \le 1$

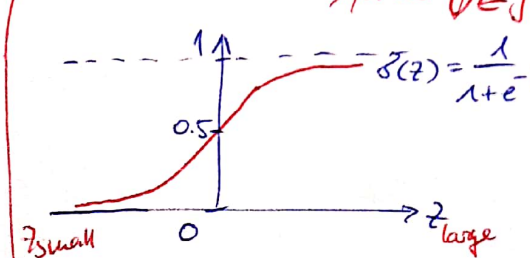
eg. picture $x \in \mathbb{R}^{n_x}$

Parameters: $w \in \mathbb{R}^{n_x}$ (weights), $b \in \mathbb{R}$ (bias, intercept term)

w is n_x-dimensional vector; b is real number

Output $\hat{y} = w^T x + b$ ~~$w^T x + b$~~ Linear Regression approach doesn't work because $\hat{y}$ could be any number but has to be $0 \le \hat{y} \le 1$!

$\hat{y} = g(w^T x + b)$ → forces $0 \le \hat{y} \le 1$!



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

If z is large $\sigma(z) \approx \frac{1}{1+0} = 1$

If z is low $\sigma(z) \approx \frac{1}{1+e^{-z}} \approx \frac{1}{1+big} \approx 0$

→ first solve linear term, then activation!

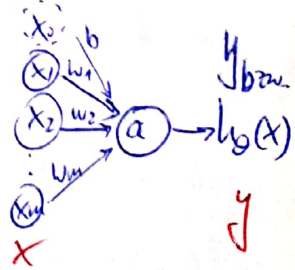
feature vector x and weight vector w must have same dimension

$x_0 = 1$, $x \in \mathbb{R}^{n_x+1}$ not using this notation in this course (theta)

$\hat{y} = \sigma(\theta^T x)$

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{n_x} \end{bmatrix}$ } b → keep separate! } w

parameter vector



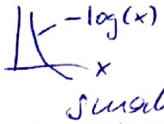
Logistic Regression cost function

$\hat{y} = \sigma(w^T x + b)$ with $\sigma(z) = \frac{1}{1+e^{-z}}$

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$ i-th example

Loss (error) function: ~~$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$~~ → don't do in logistic regression

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$$



want $\hat{y}$ large

If $y = 1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$ ← want as small as possible

If $y = 0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ ← want $\log(1-\hat{y})$ large, want $\hat{y}$ small as possible

Cost function: $J(w,b) = \frac{1}{m} \sum \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})\right]$

average over training examples

Loss function computes the error for a single training (m) example, the cost function is the average of the loss functions of the entire training set

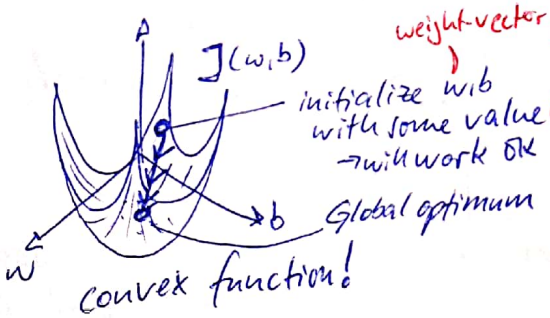
Gradient Descent (optimization)

Recap: $\hat{y} = \sigma(w^T x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}}$

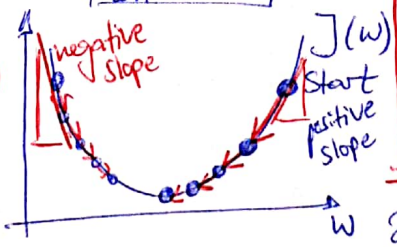
$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})\right]$$

cost function — average — loss function

→ Want to find w, b that minimize $J(w,b)$



weight-vector, Scalar



$\frac{\partial J(w)}{\partial w} = ?$

Repeat { learning rate

$w := w - \alpha \cdot \frac{\partial J(w)}{\partial w}$

dw Gname for derivative term

}

with w and b

$J(w,b)$ $w := w - \alpha \frac{\partial J(w,b)}{\partial w}$ → in code "dw"

$b := b - \alpha \frac{\partial J(w,b)}{\partial b}$ → in code "db"

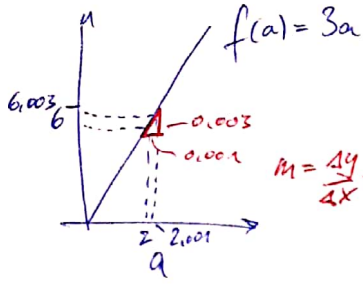
partial derivatives

Deep Learning Specialization

Week 2 | course I

derivative is slope of a function

Derivatives



for $a=2 \rightarrow f(a) = 3 \cdot 2 = 6$

$a = 2{,}001 \quad f(a) = 6{,}003$

slope (derivative) of $f(a)$ at $a=2$ is 3

$m = \frac{\Delta y}{\Delta x} \rightarrow \frac{0{,}003}{0{,}001} = 3$

$\frac{\partial f(a)}{\partial a} = 3 = \frac{\partial}{\partial a} f(a)$

formal definition:

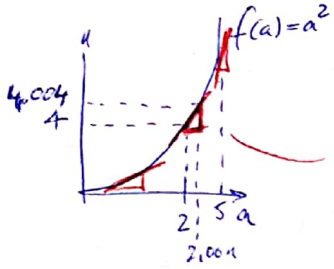
slope after infitesimal shift?

a is now 5, $f(a) = 15$

$a = 5{,}001 \quad f(a) = 15{,}003$

slope still = 3 at $a=5$

→ slope always same for linear function



$a = 2 \quad f(2) = 4$

$a = 2{,}001 \quad f(2{,}001) = 4{,}004$

slope at $a=2$ is 4

$a = 5 \quad f(5) = 25$

$a = 5{,}001 \quad f(5{,}001) = 25{,}010$

$\frac{\partial}{\partial a} f(a) = 10$ at $a=5$

$\frac{\partial}{\partial a} f(a) = \frac{\partial}{\partial a} a^2 = 2a$

→ slope varies

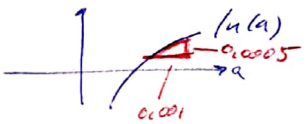
$f(a) = a^2 \quad \frac{\partial}{\partial a} f(a) = 2a \quad a=2 \rightarrow f(a)=4 \quad a=2{,}001 \rightarrow f(a) \approx 4{,}004$

$f(a) = a^3 \quad \frac{\partial}{\partial a} f(a) = 3a^2 \quad a=2 \rightarrow f(a)=8 \quad a=2{,}001 \rightarrow f(a)=8{,}012$

$f(a) = \log(a)$ ($\ln(a)$) $\quad \frac{\partial}{\partial a} f(a) = \frac{1}{a}$

$a = 2 \rightarrow f(a) = 0{,}69314$

$a = 2{,}001 \rightarrow f(a) = 0{,}69365 \quad \Delta = 0{,}0005$



$\frac{\partial}{\partial a} f(a) = \frac{1}{2}$

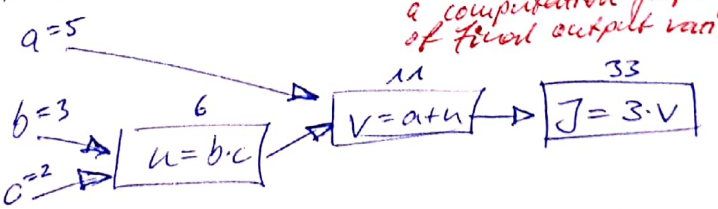
Computation Graph → organizes a computation

$J(a,b,c) = 3(a + \underbrace{bc}_{u})$

→ compute $u = bc$

→ compute $V = a + u$

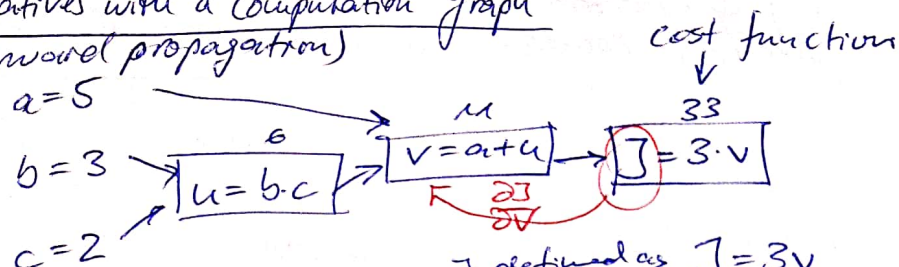
→ compute $J = 3 \cdot V$



One step of backward propagation on a computation graph yields derivative of final output variable.

Derivatives with a Computation Graph

(Forward propagation)



J defined as $J = 3v$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

$\frac{\partial J}{\partial b} = ? = 3$

$\frac{\partial J}{\partial a} = ? = 3$ → $a = 5 \rightarrow 5.001$ (changes)

$v = 11 \rightarrow 11.001$ (changes)

$J = 33 \rightarrow 33.003$

↳ $\frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial a} = 3$ ($\frac{\partial J}{\partial v} = 3$, $\frac{\partial v}{\partial a} = 1$)

Chain-rule!

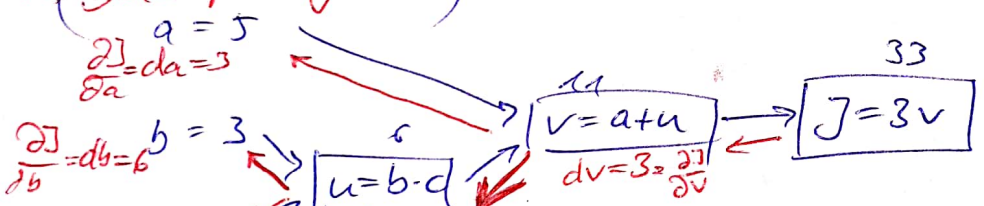
Final output variable, usually target for optimization in this case is J

$\frac{\partial \text{Final Output Var}}{\partial \text{var}}$ → "dvar"

↳ derivative of final output variable J, w.r.t. to various intermediate quantity

In this course

(Backpropagation)



$\frac{\partial J}{\partial u} \rightarrow u = 6 \rightarrow 6.001$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

$\frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial u} = 1 \cdot 3$

$b = 3 \rightarrow 3.001$

$u = 6 \rightarrow 6.002$

$v = 11 \rightarrow 11.002$

$J = 33.006$

$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial u} \cdot \frac{\partial u}{\partial b} = 6$ ($3 \cdot 2$)

$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial u} \cdot \frac{\partial u}{\partial c} = 9$

Deep Learning Specialization

week 2 course I

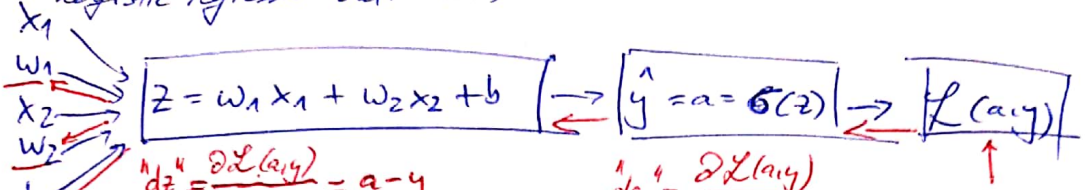
Logistic Regression Gradient Descent

$z = w^T x + b$

$\hat{y} = a = \sigma(z)$

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

logistic regression derivatives



"da" $= \frac{d\mathcal{L}(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$

"dz" $= \frac{d\mathcal{L}(a,y)}{dz} = a - y$

$dz = \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$

$\downarrow$

$a(1-a)$

$\frac{\partial \mathcal{L}}{\partial w_1} =$ "dw_1" $= x_1 dz$

$\frac{\partial \mathcal{L}}{\partial w_2} =$ "dw_2" $= x_2 dz$

$\frac{\partial \mathcal{L}}{\partial b} =$ "db" $= dz$

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

Gradient Descent on m examples

→ Logistic regression on m examples

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

$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$

$dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$ } one training example $(x^{(i)}, y^{(i)})$

$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})}_{\text{"}dw_1^{(i)}\text{"} - (x^{(i)}, y^{(i)})}$

compute the derivatives & average

gives overall gradient to implement in gradient descent

for one iteration:

Initialize $J = 0$, $dw_1 = 0$, $dw_2 = 0$, $db = 0$ (or close to 0)

for $i = 1$ to m (Python for "in m:") ↳ training examples

1st for loop

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

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

$J \mathrel{+}= -\left[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})\right]$

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

2nd for loop

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$ } all features, here only 2

$dw_n \mathrel{+}= x_n^{(i)} dz^{(i)}$

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

$J /= m$

$dw_1 /= m$; $dw_2 /= m$; $db /= m$

$dw_1 = \frac{\partial J}{\partial w_1}$

no superscript because as accumulators to sum over entire training set

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$w_n := w_n - \alpha\, dw_n$

$b := b - \alpha\, db$

2 weaknesses:

- 2 for loops: one for m training examples, one for all features

In DL having explicit for loops lead to less efficiency

↳ Vectorization (5)

↳ get rid of for-loops

Vectorization → get rid of for-loops

Whenever possible avoid explicit for-loops

$z = w^T x + b$ $\quad w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$

non-vectorized normal

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

vectorized (Python)

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

computes $w^T x$

450 ms vs 1,5 ms → for 1 million-dimensional vectors w, b

GPU / CPU } SIMD = single instruction multiple data → enable Python Numpy for better parallelism

more examples

non-vectorized

$u = Av$

$u_i = \sum_j A_{ij} v_j$

```
u = np.zeros(n,1)
for i...
    for j...
        u[i] += A[i][j] * v[j]
```

vectorized

```
u = np.dot(A,v)
```

$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$ want to apply exponential operation on every element of a matrix/vector $u \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ e^{v_n} \end{bmatrix}$

→ non-vectorized

```
u = np.zeros((n,1))
for i in range(n):
    u[i] = math.exp(v[i])
```

vectorized

```
import numpy as np
u = np.exp(v)
```

np.log(v) → elementwise log
np.abs(v) → absolute values
np.maximum(v,0)
v**2
1/v

} Numpy built-in functions

Logistic regression derivatives
→ problem with 2 derivatives

```
dw = np.zeros((n_x, 1)) → initialization
```

operation

$dw += x^{(i)} dz^{(i)}$ → 2nd for loop

$dw /= m$ → averaging

for loop looping over training examples remain

Deep Learning Specialization
Course 1, week 2

Vectorizing Logistic Regression

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

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

matrix $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ n_x $\quad (n_x, m)$ $\quad \in \mathbb{R}^{n_x \times m}$

dimension of $X = \in \mathbb{R}^{n_x \times m}$

compute $z^{(1)} \ldots z^{(m)}$ ← m →

in 1 line of code:

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

$1 \times m$ matrix

row vector $w^T = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$

$= [w^T x^{(1)} + b,\ w^T x^{(2)} + b,\ \ldots\ w^T x^{(m)} + b]$

$1 \times m$

one line of code that calculates Z that contains all of lowercase z's

$Z = np.dot(w.T, x) + b$

vector

vectorized

$1 \times 1 \to \mathbb{R}$ real number

↳ Python automatically expands the real number out to a $1 \times m$ row vector

↳ "Broadcasting"

↳ next want to find a way to compute

$A = [a^{(1)} a^{(2)} \ldots a^{(m)}] = \sigma(Z)$

activations!

computes all lower case a's

new variable

Vectorizing Logistic Regression's Gradient Computation

$dz^{(m)} = a^{(m)} - y^{(m)}$

$dZ = [dz^{(1)} dz^{(2)} \ldots dz^{(m)}]$ row vector

$1 \times m$ matrix

$A = [a^{(1)} \ldots a^{(m)}]$ $\quad Y = [y^{(1)} \ldots y^{(m)}]$

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

got rid of 1st for loop but still had

$dw = 0$
$dw += x^{(1)} dz^{(1)}$
$dw += x^{(2)} dz^{(2)}$
$\vdots$
$dw /= m$

$db = 0$
$db += dz^{(1)}$
$db += dz^{(2)}$
$\vdots$
$db += dz^{(m)}$
$db /= m$

for multiple iterations still need for-loop

implemented single iteration of Gradient Descent for Log. Reg.

vectorized implementation

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

vectorized implementation of derivative calculation

Python ⇒ $\frac{1}{m} \cdot np.sum(dz)$

$dw = \frac{1}{m} \cdot X dz^T$ ⇒ compute with 1 line of code

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

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

$n \times 1$ vector

FP and BP without using a for loop

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

7

Repeat:

for iter in range(1000): ← Still need for-loop for multiple iterations

implementation of single iteration of Gradient Descent for logistic Regression

$$Z = w^T X + b$$
$$= np.dot(w.T, x) + b$$
$$A = \sigma(Z)$$
$$dz = A - Y$$
$$dw = \frac{1}{m} X dz^T$$
$$db = \frac{1}{m} np.sum(dz)$$
$$w := w - \alpha dw$$
$$b := b - \alpha db$$

} forward propagation and backpropagation without using a for-loop

Broadcasting in Python → technique to run code faster

Calories in 100g:

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9
	59cal			

Calculate % of calories from Carb, protein, fat

Apple: $\frac{56}{59} \approx 94{,}9\,\%$

→ Can this be done without for-loop?

→ $[\,\,] = A_{(3,4)}$

Calculation we want is to sum up each of the columns of the matrix to get (here) the total number of calories in 100g of apple, beef ... and then to divide throughout the matrix to get the % without explicit for loop

↳ 2 lines of code

```
A = np.array([[56.0, 0, 4.4, 68],
              [1.2, 104, 52, 8],
              [1.8, 135, 99, 0.9]])
A.shape = (3,4)
cal = A.sum(axis=0)   # want Python to sum vertically
```

→ cal = [59, 229, 155, 76] (apple, beef, eggs, potato)

```
percentage = A/cal.reshape(1,4) * 100
```

← Python broadcasting: 3×4 matrix divided by 1×4 matrix (cal is actually already in right shape here)

```
[[99.9   0    2.83  38.42]
 [ :     :    33..   :  ]
 [3.05   :    6%..  1.17]]
```

↳ $\frac{A}{cal} = \frac{3\times4 \text{ matrix}}{1\times4 \text{ matrix}}$

more examples:

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100 \xrightarrow{\text{Python broadcasting}} \begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix}$$

changes/expands number into vector of desired dimension

⇒ works for row and line vectors!!!

5) $\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}_{(m,n)} + \begin{bmatrix}100&200&300\end{bmatrix}_{(1,n)}$ → copy matrix to (m,n)

$$\rightarrow \begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + \begin{bmatrix}100&200&300\\100&200&300\end{bmatrix} = \begin{bmatrix}101&202&303\\104&205&306\end{bmatrix}$$

c) $\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + \begin{bmatrix}100\\200\end{bmatrix} \rightarrow \begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + \begin{bmatrix}100&100&100\\200&200&200\end{bmatrix} = \begin{bmatrix}101&102&103\\204&205&206\end{bmatrix}$

8

Deep Learning Specialization

Course 1 | week 2

Matlab/octave → bsxfun

Broadcasting in Python

General Principle / more form of broadcasting

$(m,n) + (1,n) \rightarrow (m,n)$

matrix $+$ / $(m,1) \rightarrow (m,n)$

$(m,1) + \mathbb{R}$

$\begin{bmatrix}1\\2\\3\end{bmatrix} + 100 = \begin{bmatrix}101\\102\\103\end{bmatrix}$

$[1,2,3] + 100 = [101, 102, 103]$

Note on Numpy vectors (bug avoidance)

expressibility → strength: flexibility

↳ weakness: bugs, subtle bugs

Don't use "rank-1 vectors/arrays"

example a):

```
a = np.random.randn(5)
```

rather do → `a = np.random.randn(5,1)` → column vector

→ or (1,5) would create a row vector

`print(a.shape)` → (5,1)

`print(a.T)` → ...

rather not use "rank 1 - array"! → a.shape → (5,1)

↳ behaves weird

`a = np.random.randn(5,1)` → column vector

`a = np.random.randn(1,5)` → row vector

If not sure of dimention of one of the vectors → `assert(a.shape == (5,1))`

↳ inexpensive to execute

If you have a "rank-1 array" do → `a = a.reshape((5,1))` (or (1,5))

(optional) Explanation of logistic regression cost function

$\hat{y} = \sigma(w^T x + b) \quad \sigma(z) = \frac{1}{1+e^{-z}}$

$\hat{y} = P(y=1|x)$

If $y=1$: $P(y|x) = \hat{y}$

If $y=0$: $P(y|x) = 1-\hat{y}$

} $P(y|x)$

summarize: $P(y|x) = \hat{y}^y (1-\hat{y})^{(1-y)}$

If $y=1 \rightarrow \hat{y} \cdot \underbrace{(1-\hat{y})^{(1-1)}}_{1} = \hat{y}$

If $y=0 \rightarrow \underbrace{1 \cdot (1-\hat{y})^{(1)}}_{1-\hat{y}} = 1-\hat{y}$

$\log p(y|x) = \log \hat{y}^y (1-\hat{y})^{(1-y)} = y \log \hat{y} + (1-y) \log(1-\hat{y})$

$= -\mathcal{L}(\hat{y}, y)$

Cost on m examples

$\log p = \log \prod_{i=1}^{m} P(y^{(i)}|x^{(i)})$

$\log p(\dots) = \sum_{i=1}^{m} \log(y^{(i)}|x^{(i)})$

$= -\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

↳ In statistics: principle of maximum likelyhood estimation

↳ choose parameters that maximize

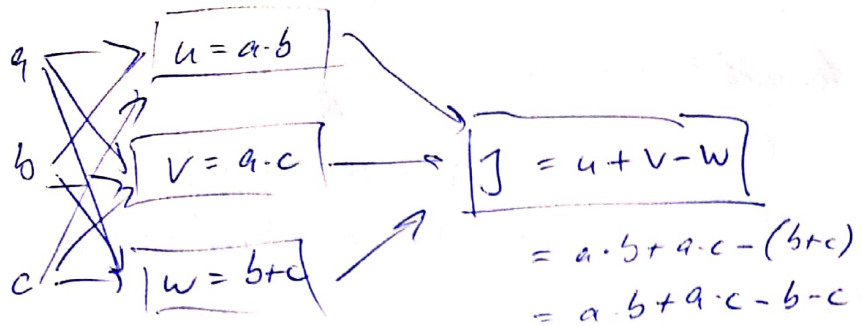
$-\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

Cost: minimize $J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

↑ average so results are better to scale

→ **Minimizing $\mathcal{L}$ corresponds to maximizing $\log p(y|x)$**

test question



$= a \cdot b + a \cdot c - (b+c)$

$= a \cdot b + a \cdot c - b - c$

$J = (c-1)\cdot(b+a) = b \cdot c + a \cdot c - b - a$

$J = (a-1)\cdot(b+c) = a \cdot b + \cdot a \cdot c - b - c$

$x = [x_1, x_2, \dots, x_n]$

$np.dot(x,x) = \sum_{j=0}^{m} x_j^2$

$L_2(\hat{y}, y) = \sum_{i=0}^{m} \underbrace{(y^{(i)} - \hat{y}^{(i)})^2}_{x_j}$

↳ np.sum(np.dot((y-yhat), (y-yhat)))

don't really needed as np dot sum all up already

$L_1(\hat{y}, y) = \sum_{i=0}^{m} \underbrace{|y^{(i)} - \hat{y}^{(i)}|}$

↳ np.sum(abs(y-yhat))

image → 3D array of shape (length, height, depth = 3)

↳ convert to vector of shape (length · height · 3, 1)

if want to: array v of shape (a, b, c) in to shape (a · b, c)

↳ v = v.reshape((v.shape[0] * v.shape[1], v.shape[2]))

If we have "image" and want to return "v" as vector of shape ((l·h·d, 1)

↳ v = image.reshape((image.shape[0] * image.shape[1] * image.shape[2]))

From optimal Numpy Lab — Important

- np.exp(x) works for any np.array and applies e^x to each element
- $\sigma(z) = \frac{1}{1+e^{-z}}$ $\sigma'(z) = \sigma(z)\cdot(1-\sigma(z))$
- image2vector is commonly used in deep learning
- np.shape is widely used, you'll see the keeping vector/matrix dimensions straight will go toward eliminating a lot of bugs
- numpy has efficient built-in functions
- broad casting is extremely useful

- Vectorization is very important in DL. It provides computational efficiency and clarity
- $\mathcal{L}_1(\hat{y}, y) = \sum_{i=0}^{m} |y^{(i)} - \hat{y}^{(i)}|$ → np.sum(abs(y-yhat)), $\mathcal{L}_2(\hat{y}, y) = \sum_{i=0}^{m} (y^{(i)} - \hat{y}^{(i)})^2$
- np.sum, np.dot, np.multiply, np.maximum ...

↳ np.sum(np.dot((y-yhat), (y-yhat)))

DL Spec Week 2 course 1

Coding Assignment – Overview

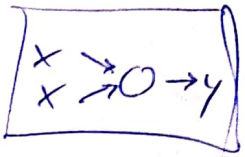
given:

training set → m_train images → cat (y=1)
→ non-cat (y=0)

test set → m_test images – " –

each image is of shape (num_px, num_px, 3)
3 RGB

each image is square num_px × num_px → (64, 64, 3)
↳ 12'288



train_set_x_orig
test_set_x_orig
train_set_y
test_set_y
= load_dataset()
→ will be pre-processed → train_set_x
→ test_set_x

1) Find values for m_train, m_test, num_px
↳ figure out dimensions

m_train = train_set_x_orig.shape[0] → 209
m_test = test_set_x_orig.shape[0] → 50
num_px = train_set_x_orig.shape[1] → 64

2) Reshape training and test data sets in to flattened arrays

with X_flatten = X.reshape(X.shape[0], -1).T

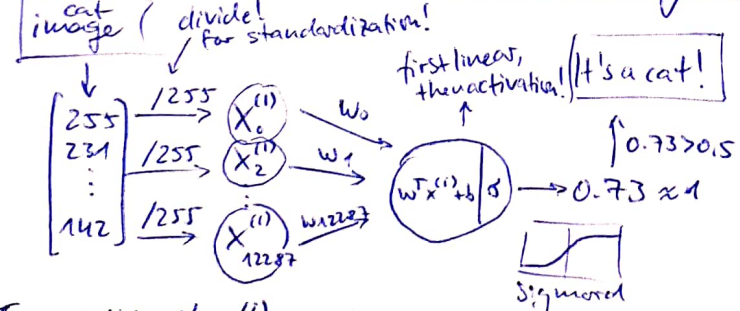
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
transpose!

" same for test_set_x_orig ...

3) Center and standardize dataset → $\frac{(X-\mu)}{\sigma}$ (mean, std. deviation)

↳ for pictures: divide every row by 255 (max. pixel value)

General Architecture of learning algorithm



Key steps in this exercise

→ Initialize all params
→ Learn them by minim. cost
→ Use learned params for predic.
→ Analyze results and conclude

For one example $x^{(i)}$:

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

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

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

$g^{(i)}$

The cost is computed by summing over all training examples

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

Main steps for building a Neural Network

1. Define model structure (e.g. # of input features)
2. Initialize the model's params
3. Loop: Calculate current loss (FP)
   " current gradient (BP)
   Update parameters (Gradient Descent)

↳ build 1 → 3 separately and integrate them into one function `model()`

ex: create sigmoid as helper function

Initialize params

$w = np.zeros((dim, 1))$

for a picture $(num\_px * num\_px * 3, 1)$
$= 64 \times 64 \times 3 = 12288$

Forward propagation

- get X
- compute $A = \sigma(w^T X + b) = (a^{(1)} \dots a^{(m)})$ (logistic regression)
- " cost $J = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$

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

$A = sigmoid(w.T, X) + b$
(np.dot)

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

$$X = \begin{bmatrix} 1 & 2 & -1 \\ 3 & 4 & -3.2 \end{bmatrix} \quad w = [1, 2] \quad b = 2$$

$$Y = [1, 0, 1]$$

Optimization: