Derivatives

The derivative formula predicts that if you nudge a by a teny little bit then $y = f(a) = a^3$ should go up by $\frac{df(a)}{da} 3a^2$ and indeed it does turn-

---

Ex 2: if you bump up a by 0.001 you can expect $f(a)$ to go up by $\frac{1}{2}$ as much as a

So if $f(a) =$ log

- the increase in y is 3 times the increase in x
- Final output variable that I really care about

$\frac{dJ}{da} = da$ in code

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

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

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

- My job when I implement logistic regression, is to learn parameters w and b so that $\hat{y}$ becomes a good estimate of the chance of y being equal to one.

given x wants $\hat{y} = P(y=1 \mid x)$

If $y=1$, we wanna make $\hat{y}$ large

$\hat{y} \simeq 1$

If $y=0$, we wanna make $\hat{y}$ small

$\hat{y} \simeq 0$

Binary classification:

output either 0 or 1

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

$$db = dz$$

$$dw_2 = x_2 \cdot dz$$

$$z = w_1 x_1 + w_2 x_2 + b$$

$$a = \sigma(z) \quad | = \mathcal{L}(a, y)$$

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

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

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

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

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

$$\frac{\partial}{\partial w_1} J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})$$

Algorithm

$J=0$; $dw_1=0$; $dw_2=0$; $db=0$

for $i=1$ to m {

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

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

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

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

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

$dw_2 += x_2^{(i)} dz^{(i)}$

$db += dz^{(i)}$ } $n=2$

$J /= m$

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

---

compute w^T $\boxed{z = w^T x + b}$

$z = 0$

for i in range(n_x): {

$z += w[i] * x[i]$;

$z += b$

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

$a = np.array([1, 2, 3, 4])$

print(a)

import time

$a = np.random.rand(1000000)$

#million dimensional array

$b = np.random.rand(1000000)$

tic = time.time()

$c = np.dot(a, b)$

toc = time.Time()

$v = np.zeros((n, 1))$

$u[i] = math.exp(v[i])$

9 training inputs stacked together in different columns like this

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

$$w^T X + [b \; b \; \cdots \; b]$$

$1 \times m$

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

$$= [z^{(1)} \; z^{(2)} \; \cdots \; z^{(m)}]$$

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

$$A = [a^{(1)} \; a^{(2)} \; \cdots \; a^{(m)}]$$

$$dz^{(1)} = a^{(1)} - y^{(1)}$$

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

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

$$dZ = [dz^{(1)} \; dz^{(2)} \; \cdots \; dz^{(m)}]$$

$$dZ = A - Y$$

$Y = [y^{(1)} \cdots y^{(m)}]$

$dw = 0$

$dw \mathrel{+}= x^{(1)} dz^{(1)}$

$dw \mathrel{+}= x^{(1)} dz^{(2)} \cdots$

$dw \mathrel{+}= x^{(m)} dz^{(m)}$

$db = 0$

$db \mathrel{+}= dz^{(1)}$

$db \mathrel{+}= dz^{(2)} \cdots$

Vec

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

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

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

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

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

get rid of 2 for loop

$$Z = w^T X + b$$

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

$$A = \sigma(Z)$$

$$dZ = A - Y$$

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

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

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

Gradient descent update

$$w := w - \alpha\, dw$$

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

```
cal = A.sum(axis = 0)
print(cal)
percentage = 100 * A / cal.reshape(1, 4)
```

```
a = np.random.randn(5, 1)
assert(a.shape == (5,1))
```

$a = a.\text{reshape}((5, 1))$

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

II D = Identity for independently distributed

$P(\text{labels in training set}) = \prod_{i=1}^{m} P(y^{(i)} | x^{(i)})$

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

Quiz - 2

(10) Computation graph

$J = u + v - w$

$= a*b + a*c - (b+c)$

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

$= (a-1)(b+c)$

gradient = vector quantity
derivative = scalar quantity

gradient of a sigmoid func

sigmoid_derivative(x)

$= \sigma'(x) = \sigma(x)(1-\sigma(x))$

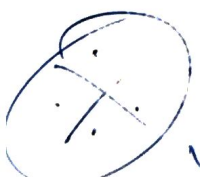
$\frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right)$

grad ≅ slope ≅ derivative

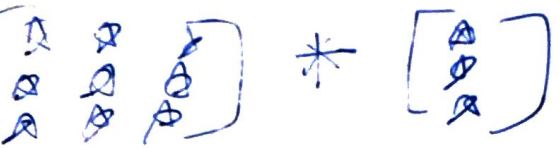
grad descent converges faster after normalization

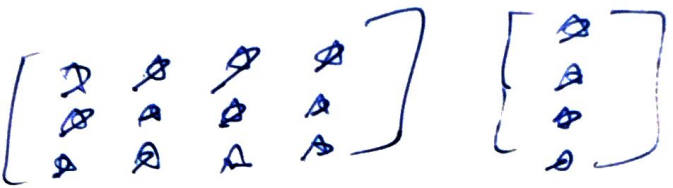
```
x_norm = np.linalg.norm
(x, ord=2, axis=1, keepdims
                =True)

x_sum = np.sum(x_exp,
        axis=1, keepdims=True)
```



dot product of a row vector & row vector is [1]

(g) 

(h) 

O/p of a neuron is

$a = g(Wx + b)$ where g is the activation function (sigmoid, tanh, ReLU)

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

known as logistic functia

- math lib when inputs of functions are real numbers
- But when inputs of funcs are matrices & vectors numpy lib more useful

- The loss is used to evaluate performance of model.

Bigger loss = more difference predictions ($\hat{y}$) from true values (y)

Gradient Descent = optimization algorithm = train model & minimize cost

if $x = [x_1, x_2, \ldots, x_n]$

then np.dot(x, x)

$$= \sum_{j=0}^{n} x_j^2$$

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

(a, b, c, d) → (b*c*d, a)

Standardize the dataset

- Subtract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array

Common steps for preprocessing a new dataset

(i) figure out the dimensions and shapes of the problem (m_train, m_test, num_px, ...)

(ii) reshape the datasets such that each example is now a vector of size (num_px * num_px * 3, 1)

(iii) "Standardize" the data

Learning rate α determines how rapidly we update the parameters

2 Overfitting: training accuracy much larger than Test accuracy

Week 3

1 Hidden layer not visible values in the training set

In a layer we get $w^{[1]}$ and $b^{[1]}$ as well

$w^{[1]}$ → (4, 3) mat

$b^{[1]}$ → (4, 1) vec

4 hidden units in the layer and 3 input layer features

So the hidden layer has associated with it these parameters

Just stack the nodes' z's vertically

To form the vector $z^{[1]}$

$$z^{[1]} = \begin{bmatrix} -w_1^{[1]T}- \\ -w_2^{[1]T}- \\ -w_3^{[1]T}- \\ -w_4^{[1]T}- \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$+ \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T}x + b_1^{[1]} \\ w_2^{[1]T}x + b_2^{[1]} \\ w_3^{[1]T}x + b_3^{[1]} \\ w_4^{[1]T}x + b_4^{[1]} \end{bmatrix}$$

$(4,1) \quad (4,3)(3,1) \quad (4,1)$

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

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

$x = a^{[0]}$ $\quad \hat{y} = a^{[2]}$

alias for vector of input features x

$W^{[2]}$ $b^{[2]}$

$(1,4)$ $(1,1)$

$z^{[2]}$ $(1,1) = (1,4)(4,1) + (1,1)$

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

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

Superscript roundbracket i refers to the i^{th} training example

Superscript squarebracket l refers to the l^{th} stack of nodes called a layer.

for $i = 1$ to m,

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

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

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