

daiwk xx

55467ee on 20 Aug 2017

0 contributors

201 lines (128 sloc) 7.96 KB

contents

- [logistic regression as a neural network](#)
 - [binary classification](#)
 - [logistic regression](#)
 - [logistic regression cost function](#)
 - [gradient descent](#)
 - [derivatives](#)
 - [more derivative examples](#)
 - [computation graph](#)
 - [derivatives with a computation graph](#)
 - [logistic regression gradient descent](#)
 - [gradient descent on m examples](#)
- [python & vectorization](#)
 - [vectorization](#)
 - [more examples of vectorization](#)
 - [vectorizing logistic regression](#)
 - [vectorizing logistic regression's gradient output](#)
 - [broadcasting in python](#)
 - [a note on python/numpy vectors](#)
 - [quick tour of jupyter/ipython notebooks](#)
 - [explanation of logistic regression cost function](#)

logistic regression as a neural network

binary classification

维度为(64, 64, 3)的图片 ==> img vector: x =维度为(64*64*3=12288, 1)的列向量。($n_x = 12288$)

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

$m = m_{train}$ 个训练样本: $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$, m_{test} 个测试样本。

X 表示一个 $n_x * m$ 的训练样本矩阵,在python里就是 $X.shape=(n_x, m)$ Y 表示一个 $1 * m$ 的向量,在python里是 $Y.shape=(1, m)$

logistic regression

given x , want $\hat{y} = P(y = 1|x), x \in R^{n_x}$

params: $w \in R^{n_x}, b \in R$

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

logistic regression cost function

$(x^{(i)}, y^{(i)})$ 表示第*i*个样本。

Loss(error) function只针对一条训练样本:

- square error的loss function:

$$L(\hat{y}, y) = 1/2 * (\hat{y} - y)^2$$

- logistic regression的loss function:

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

if $y = 1$, $L(\hat{y}, y) = -\log \hat{y}$, want $\log \hat{y}$ large, want \hat{y} large

if $y = 0$, $L(\hat{y}, y) = -\log(1 - \hat{y})$, want \hat{y} small

Cost function针对全体训练样本:

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

gradient descent

lr的 $J(w, b)$ 是一个凸函数, 所以有全局最优。因为有全局最优, 所以lr的初始化一般是0, 不用随机。梯度下降: 不断重复 $w = w - \alpha \frac{dJ(w)}{dw}$ 直到收敛。后续, 用 dw 来指代 $\frac{dJ(w)}{dw}$ 。梯度下降的公式:

$$w = w - \alpha dw$$

$$b = b - \alpha db$$

derivatives

derivative = slope, 就是 $dy/dx = \Delta y/\Delta x$

more derivative examples

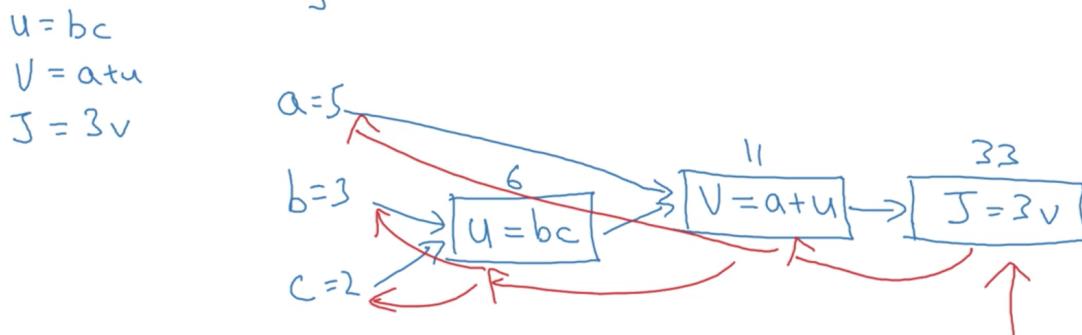
$$f(a) = \log_e(a) = \ln(a), df(a)/da = \frac{1}{a}$$

computation graph

正向计算图算出输出, 算每个参数的梯度就反向算。

Computation Graph

$$\begin{aligned} J(a, b, c) &= 3(a + bc) &= 3(5 + 3 \times 2) &= 23 \\ &\underbrace{\quad}_{\text{v}} \quad \underbrace{\quad}_{\text{v}} \quad \underbrace{\quad}_{\text{J}} \\ u &= bc \\ v &= a + u \\ J &= 3v \end{aligned}$$



derivatives with a computation graph

$$J = 3v, v = a + u, u = bc$$

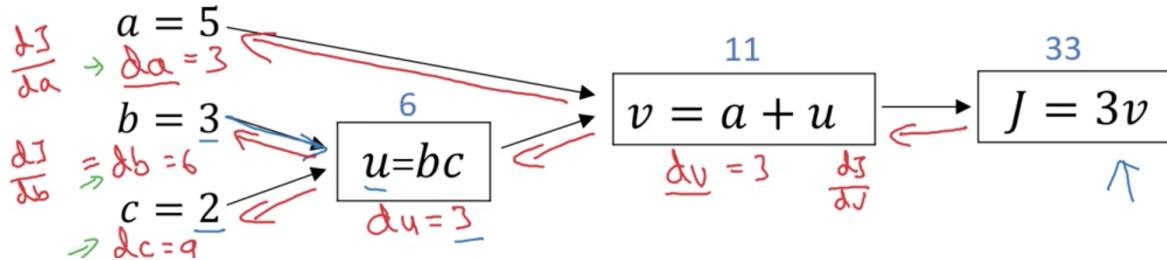
$$\frac{dJ}{dv} = 3, \frac{dv}{da} = 1$$

$$so, \frac{dJ}{da} = \frac{dJ}{dv} \frac{dv}{da} = 3 * 1 = 3$$

$$if b = 2, then \frac{dJ}{dc} = \frac{dJ}{dv} \frac{dv}{du} \frac{du}{dc} = 3 * 1 * b = 3 * 1 * 2 = 6$$

写代码时，将 $\frac{dFinalOutputVar}{dvar}$ 记为 $dvar$ (最后输出对这个变量的偏导)

Computing derivatives



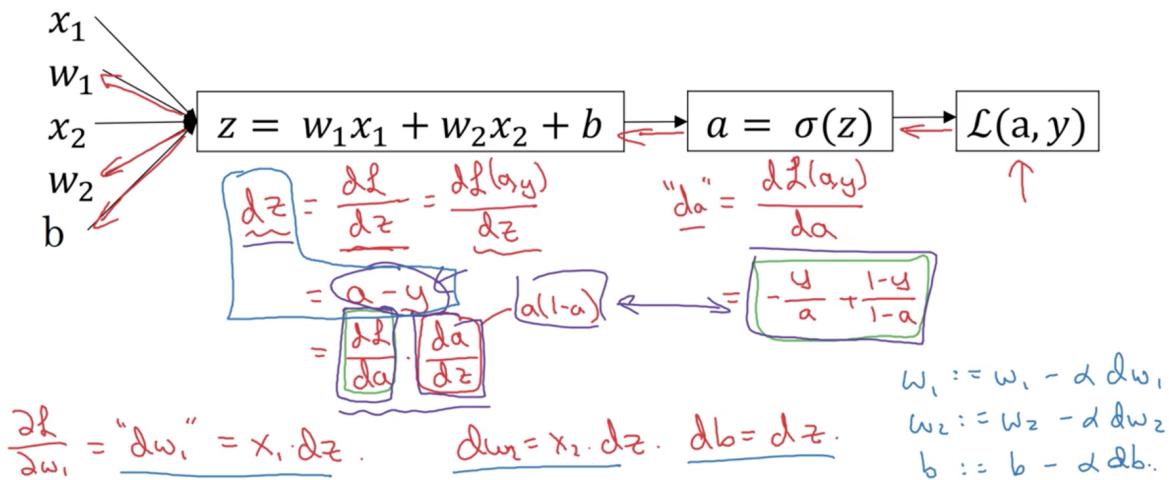
logistic regression gradient descent

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

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

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

Logistic regression derivatives



gradient descent on m examples

首先，根据J的公式，可以知道 dJ/dw_1 其实就是对每个样本的 dw_1 求和，然后/m。

Logistic regression on m examples

$$\underline{J(\omega, b)} = \frac{1}{m} \sum_{i=1}^m \underline{\ell(a^{(i)}, y^{(i)})}$$

$$\rightarrow \underline{a^{(i)}} = \underline{\hat{y}^{(i)}} = \sigma(z^{(i)}) = \sigma(\omega^\top x^{(i)} + b)$$

$$\underline{dw_1^{(i)}}, \underline{dw_2^{(i)}}, \underline{db^{(i)}}$$

$$(x^{(i)}, y^{(i)})$$

$$\underline{\frac{\partial}{\partial \omega_1} J(\omega, b)} = \frac{1}{m} \sum_{i=1}^m \underbrace{\frac{\partial}{\partial \omega_1} \ell(a^{(i)}, y^{(i)})}_{dw_1^{(i)}} - (x^{(i)}, y^{(i)})$$

每一次迭代，遍历m个样本，算出J/dw1/dw2/db，然后用这些梯度去更新一次w1/w2/b。

Logistic regression on m examples

$$J = 0; \underline{dw_1} = 0; \underline{dw_2} = 0; \underline{db} = 0$$

$$\rightarrow \text{For } i = 1 \text{ to } m$$

$$z^{(i)} = \omega^\top x^{(i)} + b$$

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

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

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

$$\begin{aligned} dw_1 &+= x_1^{(i)} dz^{(i)} \\ dw_2 &+= x_2^{(i)} dz^{(i)} \\ db &+= dz^{(i)} \end{aligned} \quad \left. \begin{aligned} n &= 2 \\ &\vdots \end{aligned} \right.$$

$$J /= m \leftarrow$$

$$dw_1 /= m; dw_2 /= m; db /= m. \leftarrow$$

$$dw_1 = \frac{\partial J}{\partial \omega_1}$$

$$\omega_1 := \omega_1 - \alpha \underline{dw_1}$$

$$\omega_2 := \omega_2 - \alpha \underline{dw_2}$$

$$b := b - \alpha \underline{db}$$

Vectorization

但这的for loop太多了。。所以我们需要vectorization!

python & vectorization

vectorization

What is vectorization?

$$z = \underbrace{\omega^T x + b}$$

$$\omega = \begin{bmatrix} : \\ : \\ : \end{bmatrix} \quad x = \begin{bmatrix} : \\ : \\ : \end{bmatrix} \quad \begin{array}{l} \omega \in \mathbb{R}^{n_x} \\ x \in \mathbb{R}^{n_x} \end{array}$$

Non-vectorized:

$$z = 0$$

for i in range(n_x):
 $z += \omega[i] * x[i]$

$$z += b$$

Vectorized

$$z = \underbrace{\text{np.dot}(\omega, x)}_{\omega^T x} + b$$

\Rightarrow GPU } SIMD - single instruction
 CPU } multiple data.

对于两个100w维的向量进行点乘，vectorization(1.5ms) 比for loop(470ms+)快

more examples of vectorization

Neural network programming guideline

Whenever possible, avoid explicit for-loops.

$$u = \underbrace{Av}_{\substack{\text{矩阵} \\ \text{向量}}}$$

$$u_i = \sum_j A_{ij} v_j$$

$$u = \underbrace{\text{np.zeros}(n, 1)}_{\substack{\uparrow \\ \uparrow}}$$

for i ... \leftarrow
 for j ... \leftarrow

$$u[i] += A[i, j] * v[j]$$

$$u = \text{np.dot}(A, v)$$

Vectors and matrix valued functions

Say you need to apply the exponential operation on every element of a matrix/vector.

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \rightarrow u = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

$\rightarrow u = np.zeros((n, 1))$

$\rightarrow \boxed{\text{for } i \text{ in range}(n):}$

$\rightarrow u[i] = \text{math.exp}(v[i])$

import numpy as np
 u = np.exp(v) ←
 ↗
 np.log(u)
 np.abs(u)
 np.maximum(v, 0)
 v ** 2 ↓
 1/v

Logistic regression derivatives

$$J = 0, \boxed{dw1 = 0, dw2 = 0}, db = 0 \quad dw = np.zeros((n_x, 1))$$

$\rightarrow \boxed{\text{for } i = 1 \text{ to } n:}$

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

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

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

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

\downarrow

$\boxed{\text{for } j=1 \dots n_x:}$

$\boxed{dw_j += x_j^{(i)} dz^{(i)}} \quad | \quad n_x=2$

$\boxed{dw_1 += x_1^{(i)} dz^{(i)}}, \quad dw_2 += x_2^{(i)} dz^{(i)}$

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

$J = J/m, \boxed{dw1 = dw1/m, dw2 = dw2/m}, db = db/m$

$dw /= m$.

如上图，将 n_x 维的dw变为一个np.array即可干掉内层的for loop。

vectorizing logistic regression

可见，整个求Z的过程可以变成一句话，而求A时，需要封装一个基于numpy的sigmoid函数。

Vectorizing Logistic Regression

$$\begin{aligned} \rightarrow z^{(1)} &= w^T x^{(1)} + b \\ \rightarrow a^{(1)} &= \sigma(z^{(1)}) \end{aligned}$$

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

$$\begin{aligned} z &= \begin{bmatrix} z^{(1)} & z^{(2)} & \dots & z^{(m)} \end{bmatrix} = w^T X + [b \ b \ \dots \ b]_{1 \times m} = \begin{bmatrix} w^T x^{(1)} + b \\ w^T x^{(2)} + b \\ \vdots \\ w^T x^{(m)} + b \end{bmatrix} \\ \rightarrow z &= np.\text{dot}(w.T, X) + b \quad \mathbb{R}^{(1, m)} \end{aligned}$$

$$A = [a^{(1)} \ a^{(2)} \ \dots \ a^{(m)}] = g(z)$$

"Broadcasting"

vectorizing logistic regression's gradient output

Vectorizing Logistic Regression

$$\begin{aligned} dz^{(1)} &= a^{(1)} - y^{(1)} & dz^{(2)} &= a^{(2)} - y^{(2)} & \dots \\ dz &= [dz^{(1)} \ dz^{(2)} \ \dots \ dz^{(m)}]_m \leftarrow \end{aligned}$$

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

$$\rightarrow dz = A - Y = [a^{(1)} - y^{(1)} \ a^{(2)} - y^{(2)} \ \dots]$$

$$\begin{cases} \frac{\partial J}{\partial w_1} \\ \frac{\partial J}{\partial w_2} \\ \vdots \\ \frac{\partial J}{\partial w_m} \end{cases} \rightarrow dw = 0$$

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

$$dw += \frac{x^{(2)}}{m} dz^{(2)}$$

$$\vdots$$

$$dw/m = m$$

$$\begin{cases} \frac{\partial J}{\partial b} \\ \frac{\partial J}{\partial b} \\ \vdots \\ \frac{\partial J}{\partial b} \end{cases} \rightarrow db = 0$$

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

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

$$\vdots$$

$$db/m = m$$

$$\begin{aligned} db &= \frac{1}{m} \sum_{i=1}^m dz^{(i)} \\ &= \frac{1}{m} np.\text{sum}(dz) \end{aligned}$$

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

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

$$= \frac{1}{m} \begin{bmatrix} x^{(1)} dz^{(1)} + \dots + x^{(m)} dz^{(m)} \end{bmatrix}_{n \times 1}$$

Implementing Logistic Regression

```

J = 0, dw1 = 0, dw2 = 0, db = 0
for i = 1 to m:
    z(i) = wTx(i) + b ←
    a(i) = σ(z(i)) ←
    J += -[y(i) log a(i) + (1 - y(i)) log(1 - a(i))]
    dz(i) = a(i) - y(i) ←
    [dw1 += x1(i)dz(i)]
    [dw2 += x2(i)dz(i)]
    db += dz(i)
J = J/m, dw1 = dw1/m, dw2 = dw2/m
db = db/m

```

```

for iter in range(1000): ←
    z = wTX + b
    = np.dot(w.T, X) + b
    A = σ(z)
    dZ = A - Y
    dw = 1/m * X * dZT
    db = 1/m * np.sum(dZ)
    w := w - αdw
    b := b - αdb

```

broadcasting in python

```

A = ndarray([[1,2,3,4],[2,3,4,5],[3,4,5,6]]) # 3*4
calc = A.sum(axis=0) # A的每列求和, 得到1*4
calc2 = A.sum(axis=1) # A的每行求和, 得到3*1
A/calc.reshape(1,4) # 得到一个3*4的矩阵, 就是broadcasting。其实等价于A/calc, 但为了保险, 可以调用reshape(1,4)来确保无误

```

小结:

General Principle

(m, n)	\pm	$(1, n)$	$\rightarrow (m, n)$
<u>matrix</u>	\times	$(m, 1)$	$\rightarrow (m, n)$

$(m, 1)$	$+$	R	$= \begin{bmatrix} 101 \\ 102 \\ 103 \end{bmatrix}$
$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$	$+$	100	$= \begin{bmatrix} 101 & 102 & 103 \end{bmatrix}$
$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$	$+$	100	

a note on python/numpy vectors

```

a=np.random.randn(5) # a.shape=(5,)是一个vector(rank 1 array), 不是一个矩阵, 所以a.T还是(5,), np.dot(a,a.T)=np.dot(a.T,a)
b=np.random.randn(5,1) # a.shape=(5,1), a.T.shape=(1,5), np.dot(a,a.T)是一个5*5的, np.dot(a.T,a)是一个1*1的矩阵 (类似a
## 可以加一句:
assert(a.shape == (5, 1))
## 如果不小心搞了个rank 1 array, 也可以手动a.reshape((5,1))=a.reshape(5,1)

```

quick tour of jupyter/ipython notebooks

explanation of logistic regression cost function

单个样本的loss function, log越大, loss越小:

Logistic regression cost function

$$\begin{aligned}
 &\rightarrow \boxed{\text{If } y = 1: p(y|x) = \hat{y}} \\
 &\rightarrow \boxed{\text{If } y = 0: p(y|x) = 1 - \hat{y}}
 \end{aligned}
 \quad \left. \right\} p(y|x)$$

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

$$\text{If } y=1: p(y|x) = \hat{y} \underbrace{(1-\hat{y})}_{=1}$$

$$\text{If } y=0: p(y|x) = \hat{y}^0 (1-\hat{y})^{(1-y)} = 1 \times (1-\hat{y}) = 1-\hat{y}$$

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

$$= \boxed{-\frac{1}{m} \sum_{i=1}^m \ell(\hat{y}^{(i)}, y^{(i)})}$$

如果是iid (独立同分布), 那么, m个样本的cost function, 其实就叫对数似然。对他求极大似然估计, 其实就是对m个样本求每个cost function的min:

Cost on m examples

$$\begin{aligned}
 \log p(\text{labels in training set}) &= \log \prod_{i=1}^m p(y^{(i)}|x^{(i)}) \leftarrow \\
 \log p(\dots) &= \sum_{i=1}^m \underbrace{\log p(y^{(i)}|x^{(i)})}_{-\ell(\hat{y}^{(i)}, y^{(i)})} \quad \text{Maximum likelihood estimator} \\
 &= -\sum_{i=1}^m \ell(\hat{y}^{(i)}, y^{(i)}) \\
 \text{Cost: } \underbrace{J(w, b)}_{(\text{minimize})} &= \frac{1}{m} \sum_{i=1}^m \ell(\hat{y}^{(i)}, y^{(i)})
 \end{aligned}$$

极大似然估计:
 $\max \log p$
即 $\min L(\hat{y}, y)$

programming assignments

squeeze

```

np.squeeze(a, axis=None)
## 删掉维数是一的部分, axis可以是Int/int数组, 表示只去掉指定下标的部分, 如果该部分维数不是1, 会报错
x = np.array([[[0], [1], [2]]])
x.shape=(1,3,1)
np.squeeze(x)=array([0,1,2]) # shape=(3,)
np.squeeze(x, axis=(2,))=array([[0, 1, 2]]) # shape=(1,3)

```

把一个shape是(a,b,c,d)的array转成一个type是(bcd,a)的array:

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

图片的预处理:

- Figure out the dimensions and shapes of the problem (m_{train} , m_{test} , num_{px} , ...)
- Reshape the datasets such that each example is now a vector of size ($num_{px} * num_{px} * 3, 1$)
- "Standardize" the data: 对图片而言, 所有元素除以255就可以了