

COURSE 1 W1&2

Vid4

Sigmoid vs Relu

Sigmoid yerine Relu kullanmak computationally cheaper. Sigmoid'in dezavantajlarından biri şu ki z 0'a veya 1'e yakinken gradient neredeyse sıfır oluyor bu sebeple learning işlemi çok yavaşlıyor. Sonuç olarak Sigmoid yerine Relu kullanmak gradient descent algoritmasını büyük oranda hızlandırıyor.

Vid7

Notation

X ve Y data matrislerinin transpose'unu kullanmak daha verimli diyor yani eskiden X 'in her satırı bir training ex. İken şimdi her sütunu bir training ex. Benzer şekilde Y bir sütun vektörüken şimdi satır vektörü.

Vid8

Log. Reg. Notation

Önceden logistic regression için z 'yi tanımlarken $\theta_0 * x_0 + \theta_1 * x_1$ diye giden bir notation'ımız vardı, $x_0 = 1$ olarak tanımlıyorduk. Implementation için bu notation'ı salmak daha yararlı oluyormuş. Artık $x_0=1$ tanımını unut, onun yerine $z = b + x_1 * w_1 + x_2 * w_2$ diye gidiyor, b ve w parametrelerini ayrı tanımlıyoruz sonuçta matematiksel olarak değişen bir şey yok. $\theta_0 * x_0 = \theta_0$ yerine b koymuş oluyoruz, w 'yi de ayrı kullanıyoruz.

Logistic Regression (C1W2L02)

Given x , want $\hat{y} = P(y=1|x)$
 $x \in \mathbb{R}^{n_x}$
 $0 \leq \hat{y} \leq 1$

Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$.

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

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

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

If z large negative number $\sigma(z) = \frac{1}{1+e^{-z}} \approx \frac{1}{1+\text{Big num}} \approx 0$

$x_0 = 1, x \in \mathbb{R}^{n_x+1}$
 $\hat{y} = \sigma(\theta^T x)$

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix}$ $b \leftarrow \theta_0$
 $w \leftarrow \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix}$

Vid9

Log. Reg. Cost Function

Logistic Regression cost function

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

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

$x^{(i)}$
 $y^{(i)}$
 $z^{(i)}$ [-th example.

Loss (error) function: $\mathcal{L}(\hat{y}, y) = \frac{1}{2} (\hat{y} - y)^2$

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

If $y=1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y} \leftarrow$ want $\log \hat{y}$ large, want \hat{y} large.

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

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

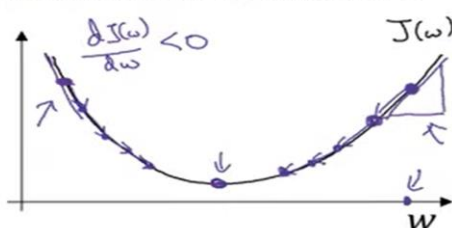
Andrew Ng

Bildiğimiz gibi squared error loss kullanmıyoruz çünkü cost function convex olmuyor, bir çok global minimum oluyor, optimization problemi karmaşılaşıyor. Loss function dediğimiz tek bir example için tanımlı fonksiyon, cost function dediğimiz tüm dataset boyunca tanımlanan hata fonksiyonu.

Vid10

Gradient Descent

Gradient Descent



Repeat {
 $w := w - \alpha \frac{\partial J(w)}{\partial w}$
 }
 $w := w - \alpha dw$
 $\frac{\partial J(w)}{\partial w} = ?$

$J(w, b)$

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

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

Partial derivatives: $\frac{\partial J(w, b)}{\partial w}$ and $\frac{\partial J(w, b)}{\partial b}$ are shown in boxes. Arrows indicate the direction of change for dw and db .

Andrew Ng

Önceden bildiğimiz şey, w ve b parametrelerini her iterasyonda partial derivative olarak itere ediyoruz en nihayetinde bize minimum cost'u veren w ve b parametrelerini bulmaya çalışıyoruz. Ayrıca kod ile implementation için yukarıdaki partial derivative gösterimi yerine dw veya db gösterimini kullanacağız.

Vid13

Computation Graphs

The computations of a neural network are organized in terms of a forward propagation step in which we compute the output of the neural network followed by a backpropagation step which we used to compute gradients.

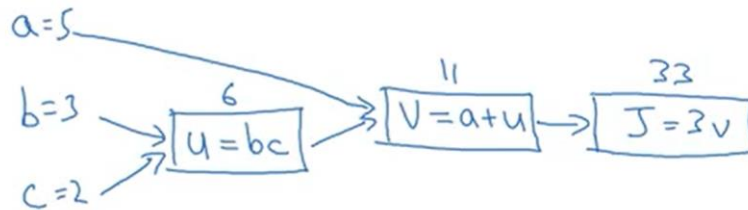
Bunun neden böyle olduğunu daha iyi anlamak için computation graphs'e bakacağız.

Diyelim ki bir $J(a,b,c)$ fonksiyonunu hesaplamaya çalışıyoruz. Bu hesaplamayı computation graph şeklinde gösterebiliriz:

Computation Graph

$$J(a,b,c) = 3(a + \underbrace{bc}_u) = 3(5 + \underbrace{3 \cdot 2}_v) = 33$$

$$\begin{aligned} u &= bc \\ v &= a + u \\ J &= 3v \end{aligned}$$



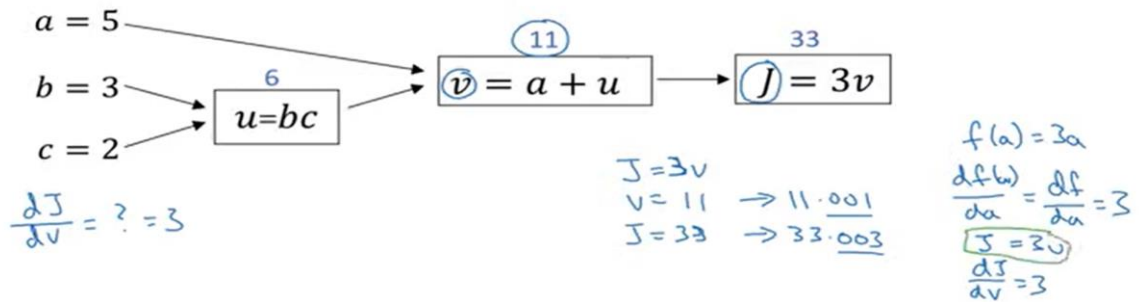
Andrew Ng

Soldan sağa bir path ile J fonksiyonunu hesaplayabiliyoruz, diğer kısımda göreceğiz ki derivative'leri hesaplayabilmek için right to left path izlemeliyiz.

Vid14

Derivatives with Computation Graphs

Computing derivatives

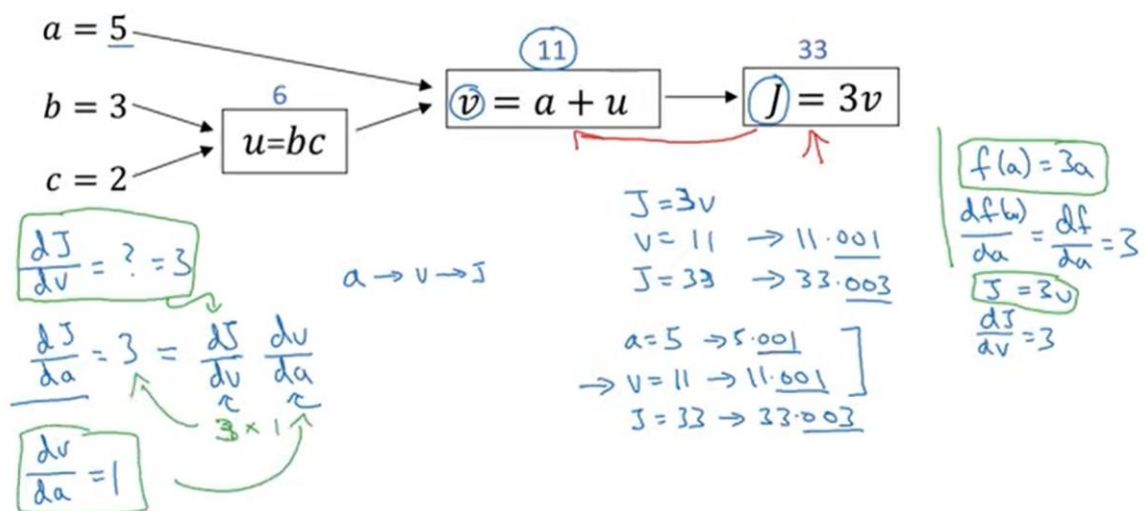


Andrew Ng

Yukarıda görüldüğü gibi dJ/dv dersek, sonucu 3 olarak bulabiliriz. Bu hesaplama için sadece son bloğu kullandığımıza dikkat et.

Ayrıca kod yazarken dJ/dv 'yi ifade edecek variable name'e direkt dv diyeceğiz çünkü NN için tüm gradient'ler J 'nin gradient'i oluyor.

Computing derivatives

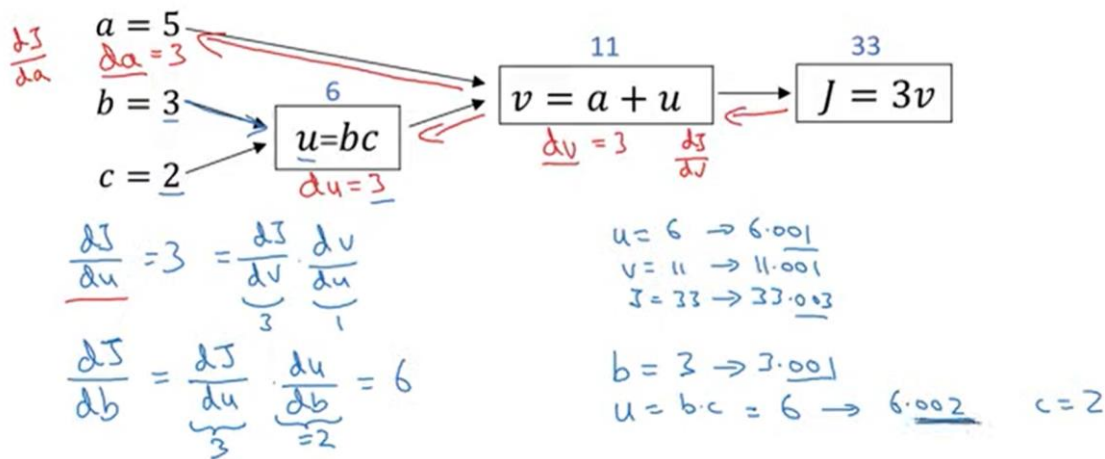


Andrew Ng

Benzer şekilde dJ/da hesaplamaya çalışırsak bunun için önce J 'nin v 'den nasıl etkilendiğini buluruz sonra v 'nin a dan ne kadar etkilendiğini buluruz ve iki sonucu çarpabiliriz. Bu hesaplama için sondan geriye doğru gittiğimize dikkat et.

Benzer şekilde diğer gradientler de bulunabilir:

Computing derivatives



Andrew Ng

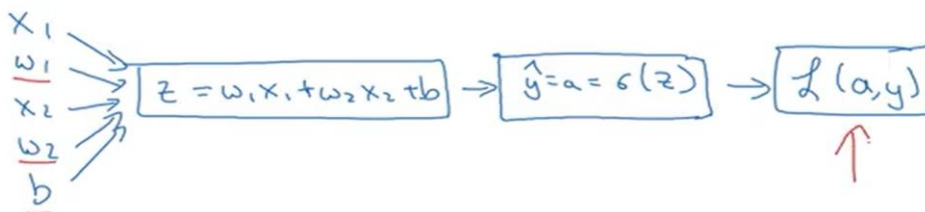
Vid15

Log. Reg. Gradient Descent – Single training example

Bu noktada computation graph kullanarak logistic regression için gradient hesaplamalarının nasıl yapıldığını anlamaya çalışacağız.

Logistic regression recap

$$\begin{aligned} \rightarrow z &= w^T x + b \\ \rightarrow \hat{y} &= a = \sigma(z) \\ \rightarrow \mathcal{L}(a, y) &= -(y \log(a) + (1 - y) \log(1 - a)) \end{aligned}$$



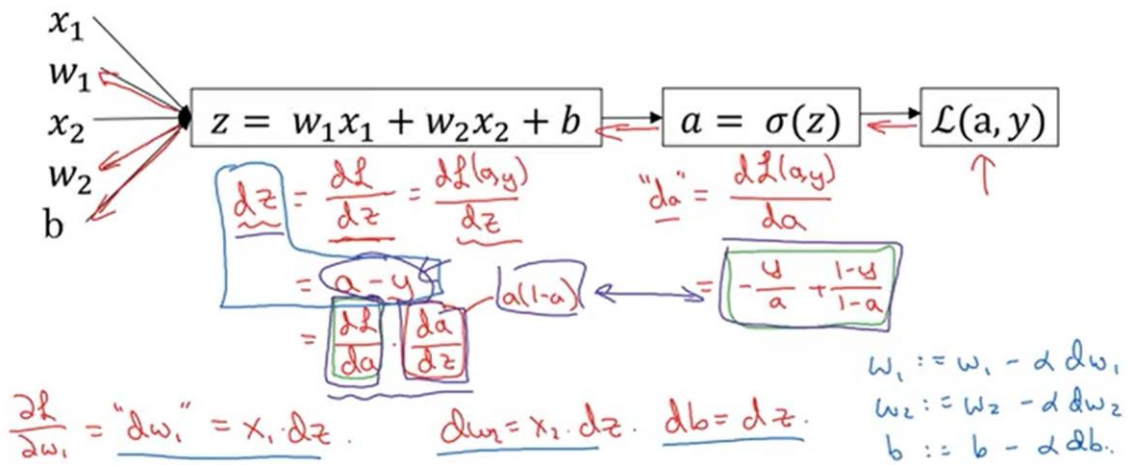
Andrew Ng

Burada a dediğimiz logistic regression'ın çıkışı temsil ediyor theta sigmoid function. Tek bir example için loss function da yukarıdaki gibi verilmiş.

Computation graphi yukarıdaki gibi çizebiliriz. Diyelim ki 2 inputumuz olsun. Z'yi hesaplamak için inputlar parametreler gerekli daha sonra bu z'den sigmoid ile çıkışı hesaplarız son olarak da loss hesaplanır.

Şimdi de ters yönde ilerleyerek gradients nasıl hesaplanır onu görelim:

Logistic regression derivatives



Andrew Ng

Sonuçta amacımız $dL/dw_1, dL/dw_2$ ve dL/db 'yi bulmak ve böylece parametreleri daha düşük bir loss'a doğru itere edebilirim.

Bunlara kısaca dw_1, dw_2 ve db diyordum, bunları elde etmek için sağdan sola doğru adım adım gidiyoruz, önce de yukarıdaki gibi elde edilebilir bunu loss function'ın türevini alarak bulabiliriz. Daha sonra dz 'yi $dL/da * da/dz$ şeklinde chain rule ile yazabilirim ve buradan da/dz sigmoidin türevi oluyor sonuçta çarpım ile $a-y$ değerini elde edebilirim. Son adımda ise dw_1, dw_2 ve db gradientleri gösterildiği gibi elde edilebiliyor.

Burada single training example varsaydık!!!

Bir sonraki başlıkta m tane training example için işlerin nasıl ilerlediğine bakacağız.

Vid16

Log. Reg. Gradient Descent – m training example

Bir önceki kısımda tek bir training example için loss function nasıl hesaplanıyor, bunun gradientleri nasıl hesaplanıyor bunları görmüştük. Bu kısımda m training example için gradient descent nasıl işliyor onu göreceğiz.

İlk olarak şunu hatırlayalım, cost function dediğimiz şey, aşağıda gösterildiği gibi her example için loss function'ın toplanması ve m'e bölünmesi ile elde ediliyor, bir başka deyişle ortalama loss.

Logistic regression on m examples

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

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

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

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

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

Andrew Ng

Bizim bir gradient step atabilmemiz için bize J 'nin gradientleri lazım. Bunları elde etmek için yukarıdaki eşitlikten yararlanabiliriz, J 'nin gradient'i aslında her example için L 'nin gradient'inin hesaplanması ve m'e bölünmesiyle elde edilebilir. Bu zaten türevin özelliği, sum'ın dışından içine geçiyor.

Şimdi gradient descent ile bir step atabilmek için nasıl bir algoritma gerektiğini yazmaya çalışalım.

Önce $J=0$, $dw_1=0$, $dw_2=0$ ve $db=0$ olarak atanır. Daha sonra 1'den m'e kadar her example için loss hesaplanır, ve her example için gradientler hesaplanır ve ilk değerlere toplanır sonuçta loop bitiminde bu değerler m ile bölününce, gradientler elde edilmiş olur. Bu gradientler ile bir step atılabilir, bir başka step için herşeyi en baştan yapmamız gerekir.

Logistic regression on m examples

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

For $i=1$ to m

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

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

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

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

$$dw_1 += x_1^{(i)} dz^{(i)} \quad \uparrow n=2$$

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

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

$$J /= m \leftarrow$$

$$\underline{dw_1} /= m; \quad \underline{dw_2} /= m; \quad \underline{db} /= m. \leftarrow$$

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

$$w_1 := w_1 - \alpha \underline{dw_1}$$

$$w_2 := w_2 - \alpha \underline{dw_2}$$

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

Andrew Ng

Yukarıda görülen tüm adımlar tek bir gradient step için gereken adımlar, bir başka adım için bu adımlar tekrarlanmalı.

Ayrıca, dw_1 , dw_2 ve db hesaplamasını biz şuan el ile yazdık, şuan için $n=2$ tane feature'umuz var ancak bizim daha fazla feature'ımız olabilir bu yüzden bu hesaplamaları kodlarken içeriye bir for loop daha girer, 2 for loop demek n^2 bigO demektir çok verimsiz!

Bu durumu çözmek için ve algoritmayı daha büyük datasetlere problemsiz scale edebilmek için vectorization kavramından yararlanmalıyız.

Önümüzdeki birkaç kısımda vectorization'dan bahsedeceğiz. Böylece yukarıdaki algoritmayı hiç for loop kullanmadan vectorler ile yapacağız.

Vid17

Vectorization

Vectorization'ın olayı, explicit for loops'dan kurtulmak.

What is vectorization?

$$z = \omega^T x + b$$

Non-vectorized:

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

$$\omega = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \quad x = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

$$\omega \in \mathbb{R}^{n_x}$$
$$x \in \mathbb{R}^{n_x}$$

Vectorized

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

Andrew Ng

Normalde z'yi hesaplarken vectorization kullanmazsak soldaki gibi n'e veya x'e kadar dönen bir for loop kullanmamız gerekiyor, her parametre için çarpımı bulup sonuca ekliyoruz. Vectorization uygularsak doğrudan $\omega^T x$ 'i bulabiliriz bunun için `np.dot(w,x)` kullanılıyor, daha sonra b'yi ekleyebiliriz.

```
In [1]: import numpy as np
a = np.array([1,2,3,4])
print(a)
[1 2 3 4]

In [14]: import time
a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized version: " + str(1000*(toc-tic)) + "ms")

c = 0
tic = time.time()
for i in range(1000000):
    c += a[i]*b[i]
toc = time.time()

print(c)
print("For loop: " + str(1000*(toc-tic)) + "ms")
```

File "c:\python-input-14-8728d229d5f0", line 18
print("For loop: " + str(1000*(toc-tic)) + "ms")

Yukarıda deneme amaçlı $n = 1\,000\,000$ için bu iki işlem karşılaştırılıyor, **vectorized için yaklaşık 1.5 ms alan işlem non vectorized için yaklaşık 500 ms alıyor. İnanılmaz bir fark söz konusu.**

Explicit for loop yerine, built-in python fonksiyonlarını kullanırsak (`np.dot()`) gibi, bu enables parallelism bu yüzden computations için büyük bir hız kazanırız. Bu parallisation hem GPU hem de CPU üzerinde yapılabilir, GPU daha başarılıdır ama CPU da fena sayılmaz.

WHENEVER POSSIBLE AVOID EXPLICIT FOR LOOPS!!!

Vid18

More Vectorization Examples

Neural network programming guideline

Whenever possible, avoid explicit for-loops.

$$u = Av$$

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

$$u = \text{np.zeros}(n, 1)$$

$$\text{for } i \dots \leq$$

$$\text{for } j \dots \leq$$

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

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

Andrew Ng

Soldaki bir A matrisi ise v vektörünün çarpımını looping ile yaparsak ne olacağını gösteriyor iç içe 2 loop, bigO'su çok kötü. Ancak numpy kullanarak yaparsak sağdaki gibi hem daha rahatça hem de daha verimli bir şekilde yapmış oluruz.

Vectors and matrix valued functions

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

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

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

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

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

$$\text{import numpy as np}$$

$$u = \text{np.exp}(v)$$

$$\text{np.log}(v)$$

$$\text{np.abs}(v)$$

$$\text{np.maximum}(v, 0)$$

$$v ** 2$$

$$1/v$$

Andrew Ng

Bir başka örnek yukarıdaki gibi elementwise exponential veya diğer elementwise işlemler olabilir, numpy kullanarak bu elementwise işlemleride sağda görüldüğü gibi rahatça halledebiliriz.

Logistic regression derivatives

```

J = 0, dw1 = 0, dw2 = 0, db = 0      dw = np.zeros((n_x, 1))
→ for i = 1 to 'm':
    z(i) = wTx(i) + b
    a(i) = σ(z(i))
    J += -[y(i) log ŷ(i) + (1 - y(i)) log(1 - ŷ(i))]
    dz(i) = a(i)(1 - a(i))
    dw1 += x1(i) dz(i)      n_x = 2      dw += x(i) dz(i)
    dw2 += x2(i) dz(i)
    db += dz(i)
J = J/m, dw1 = dw1/m, dw2 = dw2/m, db = db/m
dw /= m.

```

Andrew Ng

Logistic regression örneğine geri dönersek, hatırlarsan burada iç içe iki loop söz konusuydu, vectorization ile içerideki dw hesaplayan loptan kurtulabiliriz, yapılması gerekenler yukarıda yeşil ile işaretlenmiş. (Burada dzi sanırım yanlış ai -yi olacak)

Vid19

Vectorizing Logistic Regression

Vectorizing Logistic Regression

$z^{(1)} = w^T x^{(1)} + b$
 $a^{(1)} = \sigma(z^{(1)})$

$z^{(2)} = w^T x^{(2)} + b$
 $a^{(2)} = \sigma(z^{(2)})$

$z^{(3)} = w^T x^{(3)} + b$
 $a^{(3)} = \sigma(z^{(3)})$

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

$Z = \begin{bmatrix} z^{(1)} & z^{(2)} & \dots & z^{(m)} \end{bmatrix} = w^T X + \begin{bmatrix} b & b & \dots & b \end{bmatrix}$
 $\begin{matrix} (1, m) \\ \mathbb{R}^{1 \times m} \end{matrix}$
 $\begin{matrix} (1, n_x) \\ \mathbb{R}^{1 \times n_x} \end{matrix}$
 $\begin{matrix} (n_x, m) \\ \mathbb{R}^{n_x \times m} \end{matrix}$

$\Rightarrow Z = \text{np.dot}(w.T, X) + b$
 $\begin{matrix} (1, 1) \\ \mathbb{R} \end{matrix}$
"Broadcasting"

$A = \begin{bmatrix} a^{(1)} & a^{(2)} & \dots & a^{(m)} \end{bmatrix} = \sigma(Z)$

Andrew Ng

Şimdi, sanıyorum, log.reg.'in üstteki lopundan kurtulmaya yoğunlaştık ancak bu sadece forward propagation yani z ve a hesaplanıyor. Daha önce yaptığımız şey en üstte görüldüğü gibi z1 ve a1'i hesaplamak daha sonra z2 ve a2'yi ... Yani her training example için tek tek hesap yapmak. Bunu vectorize edersek, basitçe aşağıda kırmızı içine alınmış $z = \text{np.dot}(w.T, x) + b$ olarak buluruz. Burada b'yi numpy otomatik olarak 1xm vektör olarak alır yani vektörün her elamanına b ekler. Son olarak a'ya geçmek için input olarak vektör alan ve vektör üzerinde işlem yapan bir sigmoid fonksiyonu yazarız ve onu kullanırız. Böylece loop kullanmadan z'yi ve a'yı hesaplayabiliriz.

Vid20

Vectorizing Logistic Regression – Gradient Computation

Bir önceki kısımda, vectorization ile prection'ı nasıl hızlandırabileceğimizi yani z'yi ve a'yı nasıl vektörize edebileceğimizi gördük.

Şimdi ise m examples için gradient computation yapacakken vectorization'ı nasıl kullanacağımızı göreceğiz.

Son olarak da hepsini birleştireceğiz ve nasıl verimli bir logistic regression modeli elde edebiliriz ona bakacağız.

Vectorizing Logistic Regression

$d_z^{(i)} = a^{(i)} - y^{(i)}$ $d_z^{(2)} = a^{(2)} - y^{(2)}$...

$\underline{dz} = [\underline{dz}^{(1)} \quad \underline{dz}^{(2)} \quad \dots \quad \underline{dz}^{(m)}]$ \leftarrow
 $1 \times m$

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

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

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

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

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

$dw = \frac{1}{m} X dz^T$
 $= \frac{1}{m} \begin{bmatrix} x^{(1)} & \dots & x^{(m)} \\ 1 & & 1 \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$
 $= \frac{1}{m} [x^{(1)} dz^{(1)} + \dots + x^{(m)} dz^{(m)}]$
 $n \times 1$

Andrew Ng

dz'leri yukarıdaki gibi her example için hesaplayabildiğimizi biliyoruz ancak, bir önceki örnekte forward propagation'ı vektörize ettik ve A ile Y'yi elde etmiştik o halde dz basitçe A-Y olarak elde edilebilir.

Bir diğ er adım dw'yi elde etmek, burada i erdeki loop'u yani her ex. i in ayrı ayrı dw1, dw2, dw3 .. hesaplamasını vekt rize etmiřtik ancak hala her training example i in dw hesaplanıyor bunu da vekt rize edebiliriz.

Dw ve db vekt rizasyonu i in kırmızı circled formları kullanabiliriz b ylece artık iki loptan da kurtulmuř olduk.

Sonuç olarak toparlarsak, eski ve yeni formu yan yana g sterelim:

Implementing Logistic Regression

```
J = 0, dw1 = 0, dw2 = 0, db = 0
for i = 1 to m:
    z(i) = w^T x(i) + b
    a(i) = sigma(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):

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

Andrew Ng

Forward propagation ile Z ve A hesaplanıyor, backward propagation ile dZ ve dW ve db hesaplanıyor, sonu ta bu hesaplamanın sonunda tek bir gradient step atılıyor, convergence i in bu forward ve backward propagation'ı yine bir loop i ine alıyoruz maalesef bu son loptan kurtulamıyoruz.

Vid21

Broadcasting

Broadcasting python kodumuzu daha hızlı  alıřtırabilmek i in kullanabileceğimiz bir bařka tekniktir.

Bir  rnekle, broadcasting'i anlamaya  alıřalım.

Diyelim ki ařağıdaki gibi bir matrisimiz olsun, bu A matrisi i inde farklı besinlerin i indeki carb, protein ve fat kalorileri verilmiř. Benim istediğim her besin i in bu deėerleri y zdelik deėere  evirmek, yani elma i in 56 cal carb yerine %94.9 carb yazacak.

Bunu loop kullanmadan yapacaėız.

Broadcasting example

Calories from Carbs, Proteins, Fats in 100g of different foods:

	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

$= A$
(3,4)

59 cal
 $\frac{56}{59} \approx 94.9\%$

Calculate % of calories from Carb, Protein, Fat. Can you do this without explicit for-loop?

```
Broadcasting example
Last Checkpoint: 22 minutes ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help
Trusted Python 3

In [6]: import numpy as np

A = np.array([[56.0, 0.0, 4.4, 68.0],
              [1.2, 104.0, 52.0, 8.0],
              [1.8, 135.0, 99.0, 0.9]])

print(A)

[[ 56.   0.   4.4  68. ]
 [  1.2 104.  52.   8. ]
 [  1.8 135.  99.   0.9]]

In [7]: cal = A.sum(axis=0)
print(cal)

[ 59.  239. 155.4  76.9]

In [ ]: percentage = 100*A/cal.reshape(1,4)
print percentage
```

Yapılan şey önce, axis=0 yani vertical yönde matris elemanlarını toplayarak her besin için total kalori değerini elde etmek daha sonra, A matrisini bu kalori vektörüne bölüyoruz, bu bölme sıradan bir bölme değil, her sütunu bu vektör ile bölüyor yani elementwise işlem yapıyor, ama bilinen işlemlerin hiçbirine karşılık düşmüyor işte buna broadcasting diyoruz.

Burada reshape commandine gerek yok aslında redundant, ancak emin olmak için kullanıyoruz. Reshape command O(1) bir command yani don't be shy to call it. Peynir ekmek gibi çağır.

Kısaca birkaç örnekle broadcasting'in nasıl çalıştığına bakalım:

Broadcasting example

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{(m,n)}^{(2,3)} + \begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix}_{(1,n)}^{(2,3)} \rightarrow (m,n) = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{(m,n)} + \begin{bmatrix} 100 & 100 & 100 \\ 200 & 200 & 200 \end{bmatrix}_{(m,1)}^{(2,3)} \rightarrow (m,n) = \begin{bmatrix} 101 & 102 & 103 \\ 204 & 205 & 206 \end{bmatrix}$$

İlk örnekte 100'ü alıyor ilk vektörle toplanabilecek bir 100 vektörüne çeviriyor.

İkincide keza, vektörü matrisle toplanabilecek bir matrise dönüştürüyor.

Sonuncuda da yine aynı işlem gerçekleşiyor.

General Principle

$$\begin{array}{c} (m,n) \\ \text{matrix} \end{array} \begin{array}{c} + \\ - \\ * \\ / \end{array} \begin{array}{c} (1,n) \\ (m,1) \end{array} \rightarrow (m,n)$$

$$\begin{array}{c} (m,1) \\ \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \\ (1,2,3) \end{array} + \begin{array}{c} \mathbb{R} \\ 100 \\ 100 \end{array} = \begin{array}{c} \begin{bmatrix} 101 \\ 102 \\ 103 \end{bmatrix} \\ [101 \ 102 \ 103] \end{array}$$

Matlab/Octave: bsxfun

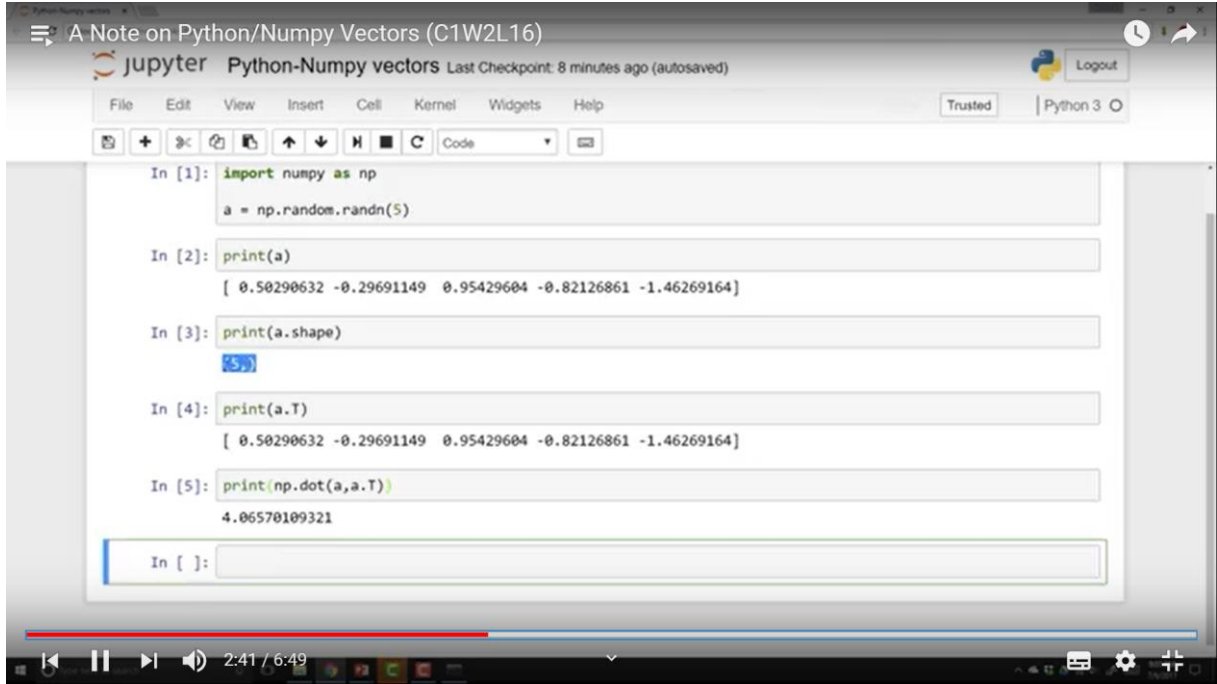
Genel olarak, bir matrise bir vektörü toplamaya çalıştığımızda vektörü alıyor, matrisle toplanabilecek şekilde genişletiyor bu sırada vektörü kopyalamaktan fazlasını yapmıyor, daha sonra bildiğimiz matris işlemleri, yani eleman elemana toplama, çıkarma, çarpma, bölme yapıyor.

Vid22

Python/Numpy Vectors

Bir önceki kısımda gördüğümüz broadcasting gibi numpy flexibilityleri işlevsel olduğu kadar kodu hataya açık hale getirir ve hataları bulması zorlaşabilir mesela bir row vector ile column vectorun toplamının hata vermesini beklerken, vermez.

Bu kısımda bazı tips and tricks'ten bahsedeceğiz:



```
A Note on Python/Numpy Vectors (C1W2L16)
jupyter Python-Numpy vectors Last Checkpoint: 8 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help
Trusted Python 3

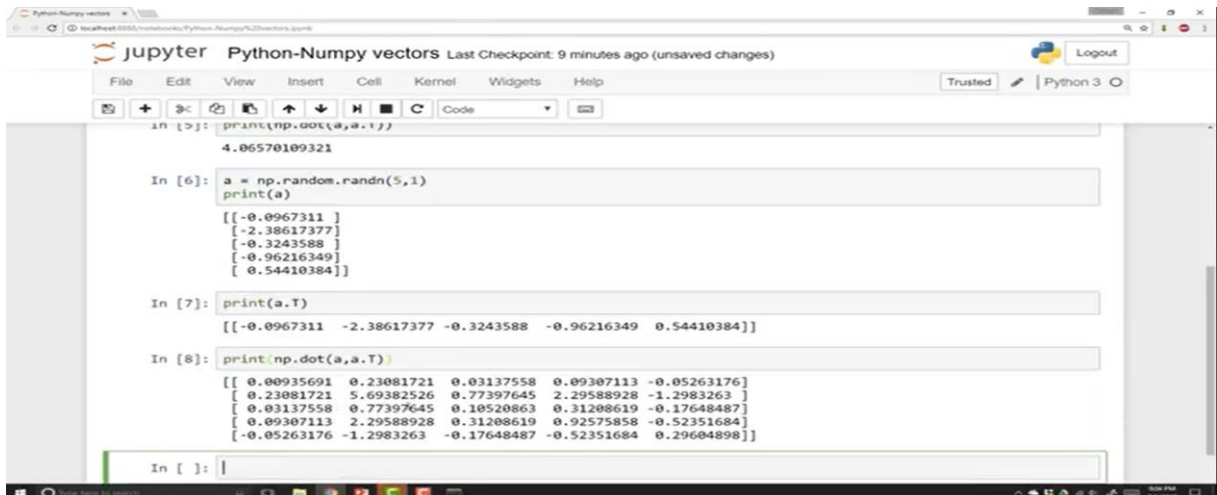
In [1]: import numpy as np
        a = np.random.randn(5)

In [2]: print(a)
[ 0.50290632 -0.29691149  0.95429604 -0.82126861 -1.46269164]

In [3]: print(a.shape)
(5,)
```

Yukarıdaki gibi bir a vektörü tanımladığımızda bunun shape'i (5,) olarak döner. Bu aslında tam olarak bir vektör gibi davranmaz buna RANK 1 ARRAY denir. a ile a.T çarpımının sonucu 5*5 bir matris çıkmasını beklerken tek bir sayı çıkıyor, tuhaf.

Bu formu kullanmaktan kaçın, shape dediğinde iki boyutu da görülmeli rank 2 array kullan.



```
Python-Numpy vectors
jupyter Python-Numpy vectors Last Checkpoint: 9 minutes ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help
Trusted Python 3

In [5]: print(np.dot(a,a.T))
4.06570109321

In [6]: a = np.random.randn(5,1)
        print(a)
[[-0.0967311 ]
 [ 2.38617377]
 [-0.3243588 ]
 [-0.96216349]
 [ 0.54410384]]

In [7]: print(a.T)
[[-0.0967311 -2.38617377 -0.3243588 -0.96216349  0.54410384]]

In [8]: print(np.dot(a,a.T))
[[ 0.00935691  0.23081721  0.03137558  0.09307113 -0.05263176]
 [ 0.23081721  5.69382526  0.77397645  2.29588928 -1.2983263 ]
 [ 0.03137558  0.77397645  0.10520863  0.31208619 -0.17648487]
 [ 0.09307113  2.29588928  0.31208619  0.92575858 -0.52351684]
 [-0.05263176 -1.2983263 -0.17648487 -0.52351684  0.29604898]]
```

Yukarıdaki gibi bir kullanım beklenen sonuçları karşılıyor.

Özetle:

Python/numpy vectors

```
a = np.random.randn(5)
a.shape = (5,)
"rank 1 array" } Don't use
```



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



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



```
assert(a.shape == (5,1)) ←
a = a.reshape((5,1))
```

Andrew Ng

Rank 1 array kullanma.

Eğer shape'den emin değilsek bir assertion uygula bunu kullanmak cheaptir ve bir bilgilendirme yerine geçer, eğer condition true ise hiçbir şey olmaz, false ise error throw eder.

Son olarak reshape komutu ile, arrayi istediğimiz formata getirebiliriz.

Vid23

iPython Jupyter Notebooks

Ara sıra çok açık bırakırsak veya çok uzun işler yaptırırsak kernel ölebilir.

Yukarıdan kernel'e tıkla, restart kernel de.

Vid24

Optional Cost Function

Logarithmic cost function ile ilgili bilgiler veriyor, atladım. Gerekirse bakılabilir.