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 yakınken gradient neredeyse sıfır oluyor bu sebeple learning işlemi çok yavaşlıyor. Sonuç olarak Sigmoid yerine Relu kullanmak gradient descent algoritmasını büyük oranda hızlandırıyor.

Vid7

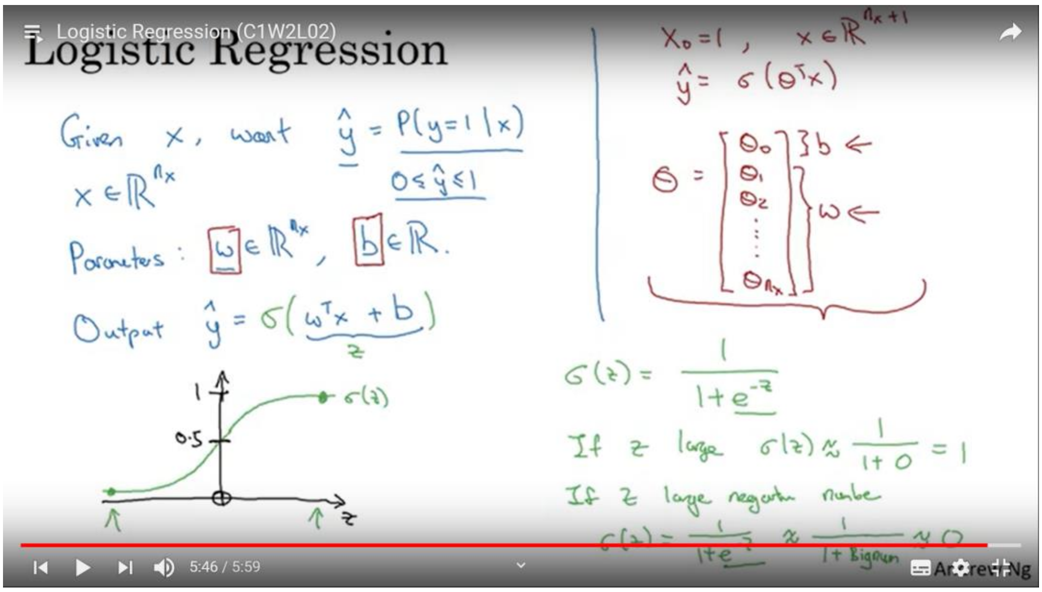
Notation

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 vektörüyken şimdi satır vektörü.

Vid8

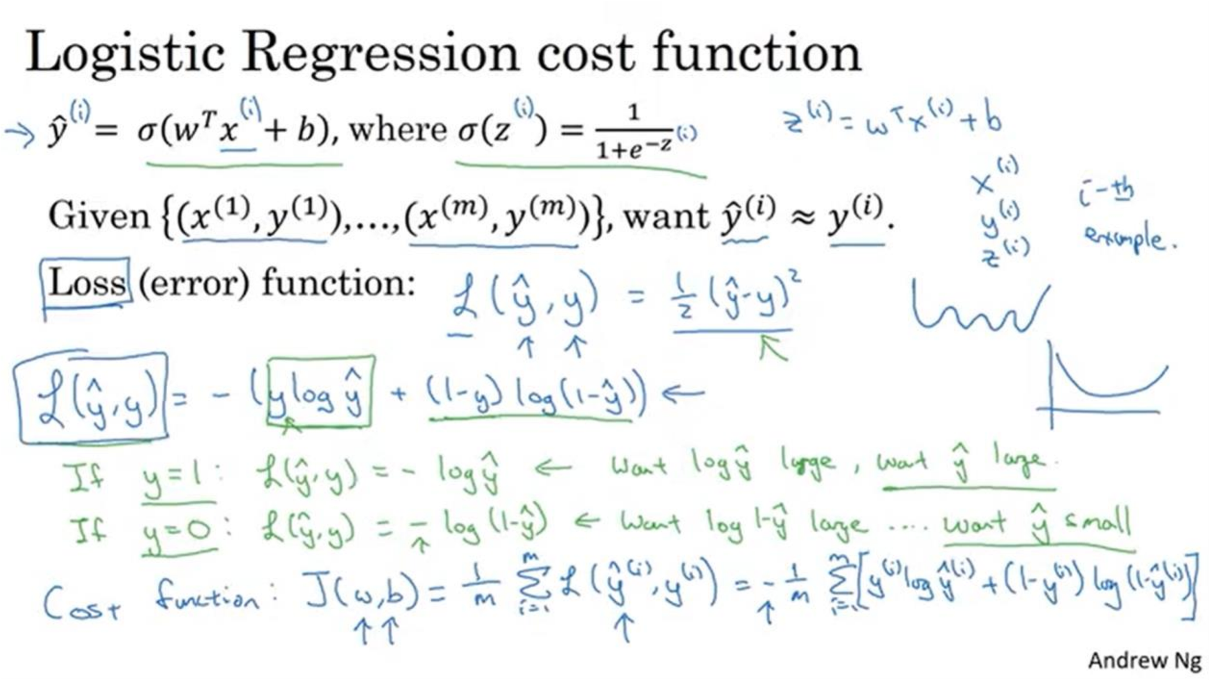
Log. Reg. Notation

Önceden logistic regression için z’yi tanımlarken theta0*x0 + theta1*x1 diye giden bir notation’ımız vardı, x0 = 1 olarak tanımlıyorduk. Implementation için bu notation’ı salmak daha yararlı oluyormuş. Artık x0=1 tanımını unut, onun yerine z = b + x1*w1 + x2*w2 diye gidiyor, b ve w parametrelerini ayrı tanımlıyoruzç sonuçta matematiksel olarak değişen bir şey yok. Theta0*x0 = Theta0 yerine b koymuş oluyoruz, w’yi de ayrı kullanıyoruz.



Vid9

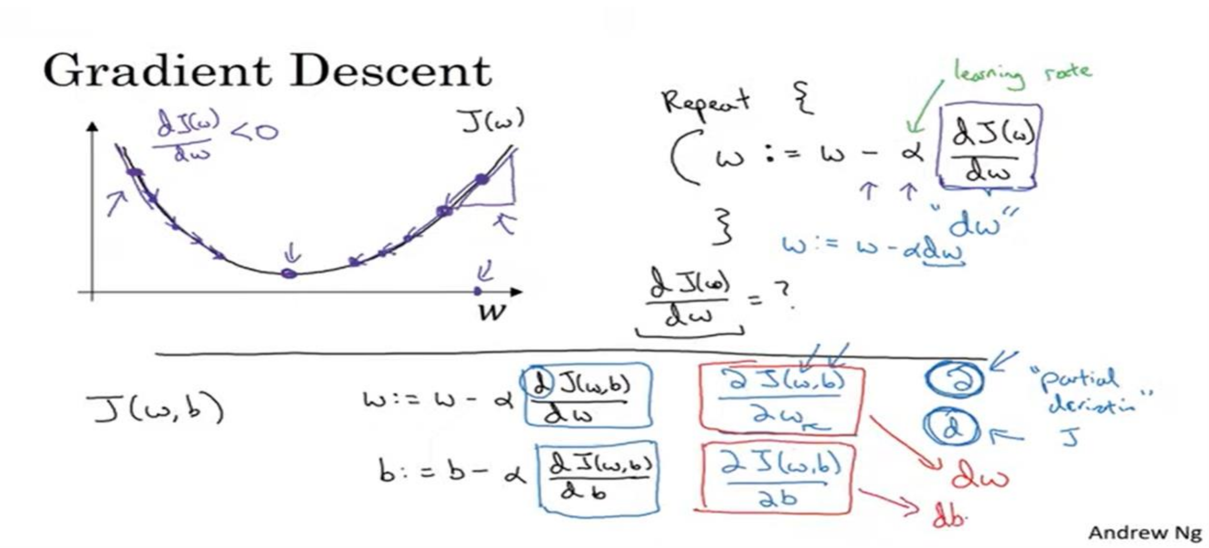
Log. Reg. Cost Function



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

Vid10

Gradient Descent



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

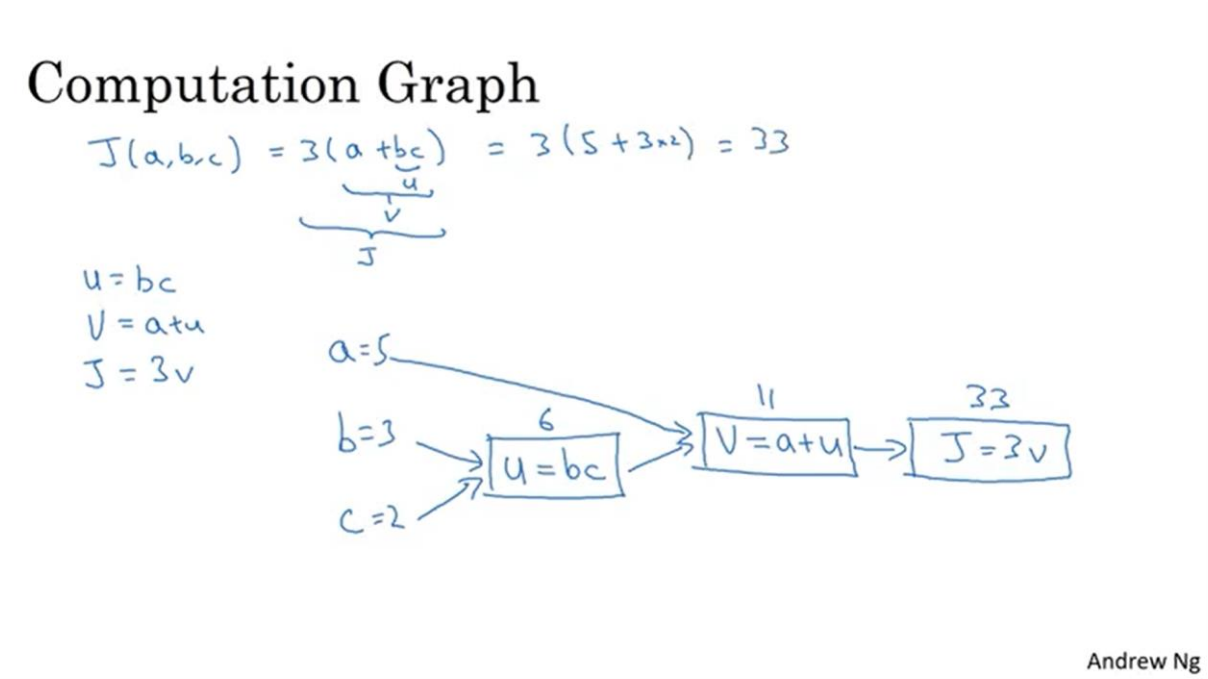
Vid13

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.

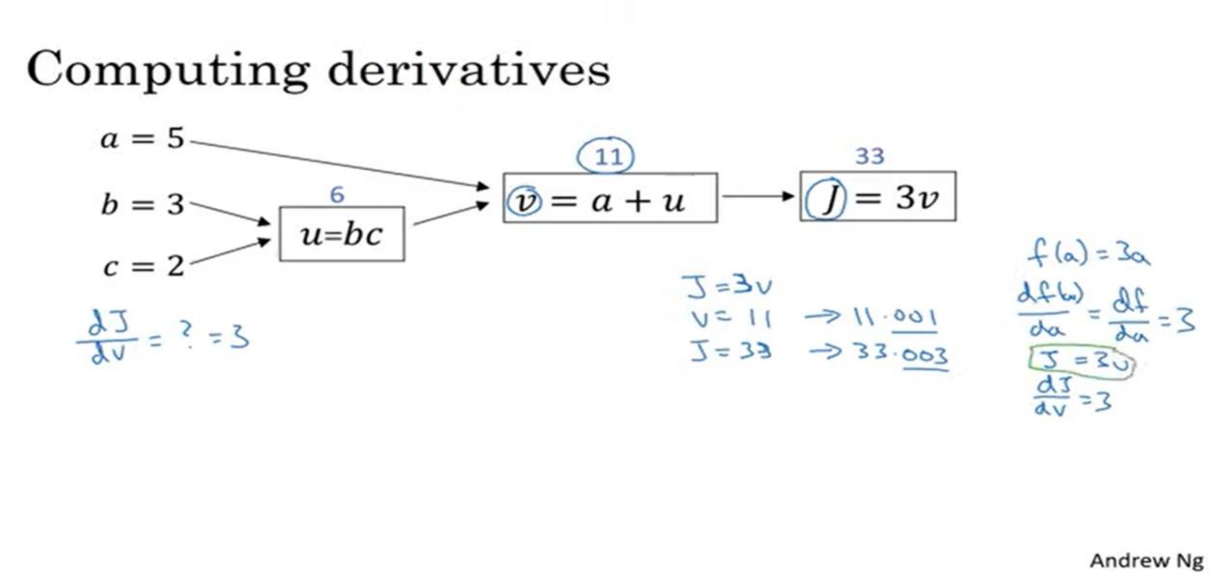
Diyelim ki bir J(a,b,c) fonksiyonunu hesaplamaya çalışıyoruz. Bu hesaplamayı computation graph şeklinde gösterebiliriz:



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

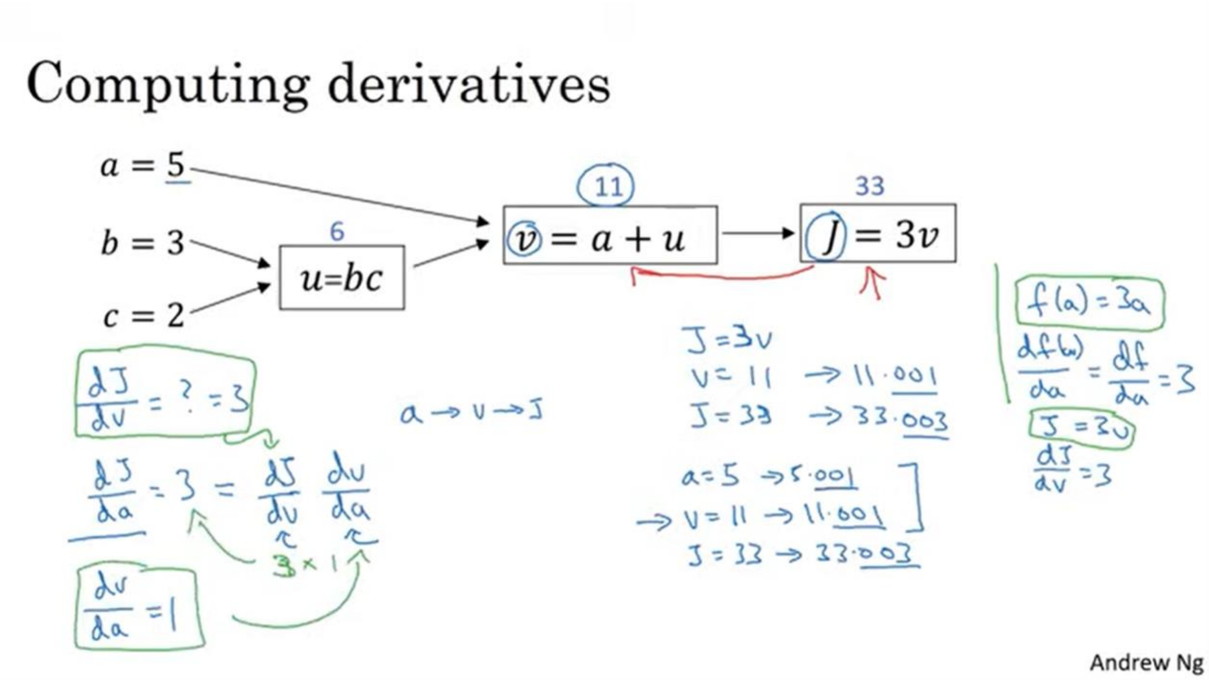
Vid14

Derivatives with Computation Graphs



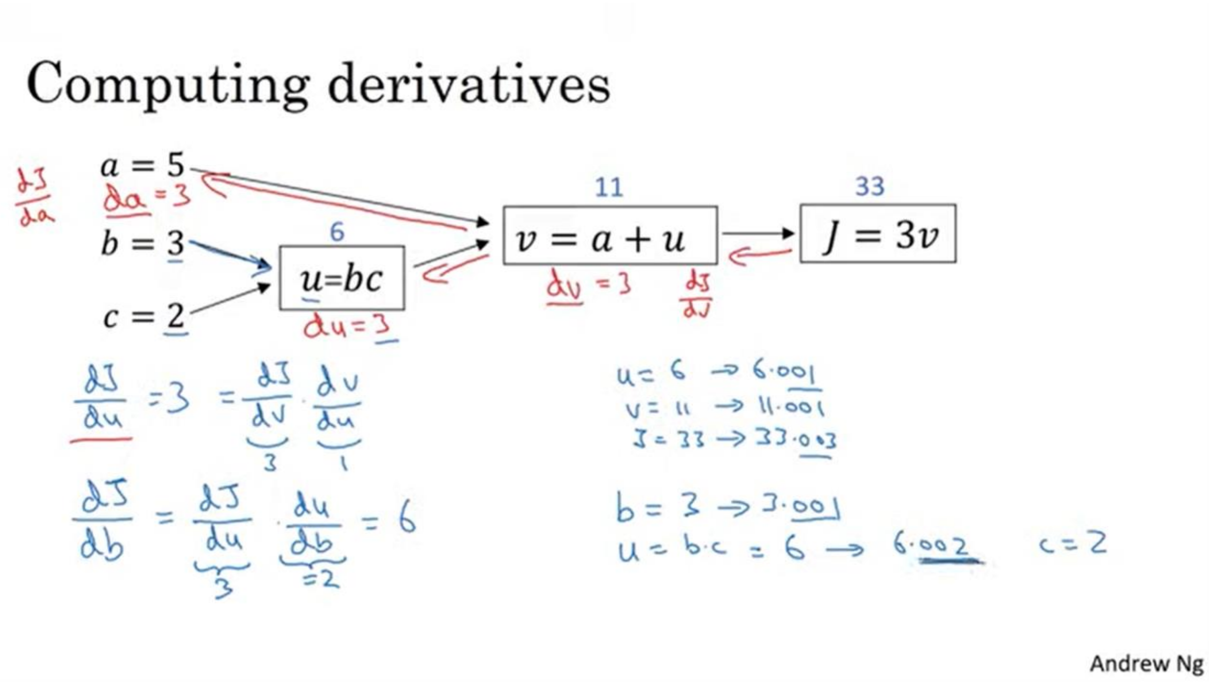
Yukarıda görüldüğü gibi dJ/dv dersek, sonucu 3 olarak bulabiliriz. Bu hesaplama için sadece son bloğu kullandığımıza dikkat et.

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.



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 çarparız. Bu hesaplama için sondan geriye doğru gittiğimize dikkat et.

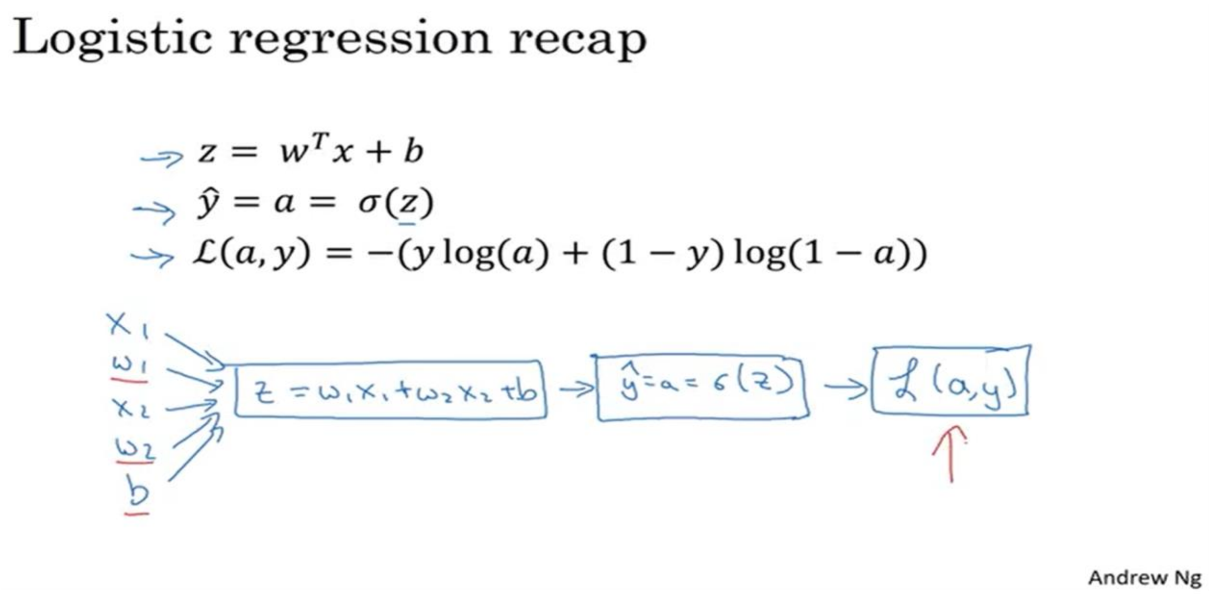
Benzer şekilde diğer gradientler de bulunabilir:



Vid15

Log. Reg. Gradient Descent – Single training example

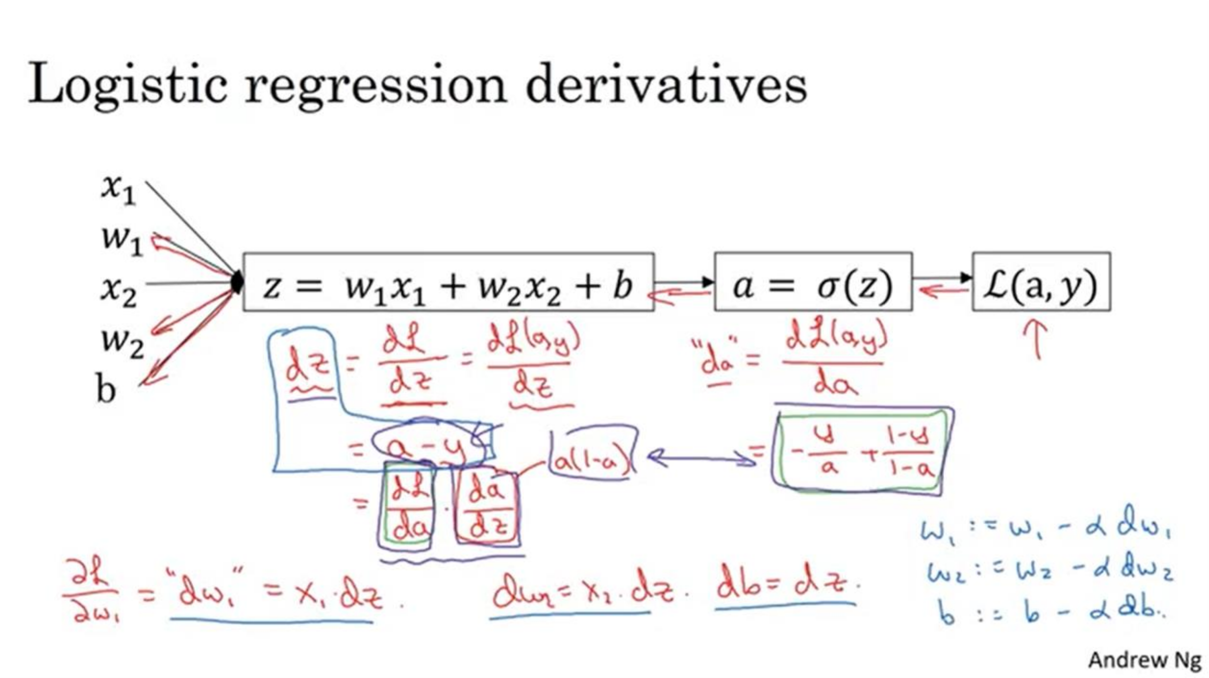
Bu noktada computation graph kullanarak logistic regression için gradient hesaplamalarının nasıl yapıldığını anlamaya çalışacağız.



Burada a dediğimiz logistic regression'ın çıkışını 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:



Sonuçta amacımız dL/dw1, dL/dw2 ve dL/db'yi bulmak ve böylece parametreleri daha düşük bir loss'a doğru itere edebilirim.

Bunlara kısaca dw1 dw2 ve db diyordum, bunları elde etmek için sağdan sola doğru adım adım gidiyoruz, önce da 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 dw1,dw2 ve db gradientleri gösterildiği gibi elde edilebiliyor.

Burada single training example varsaydık!!!

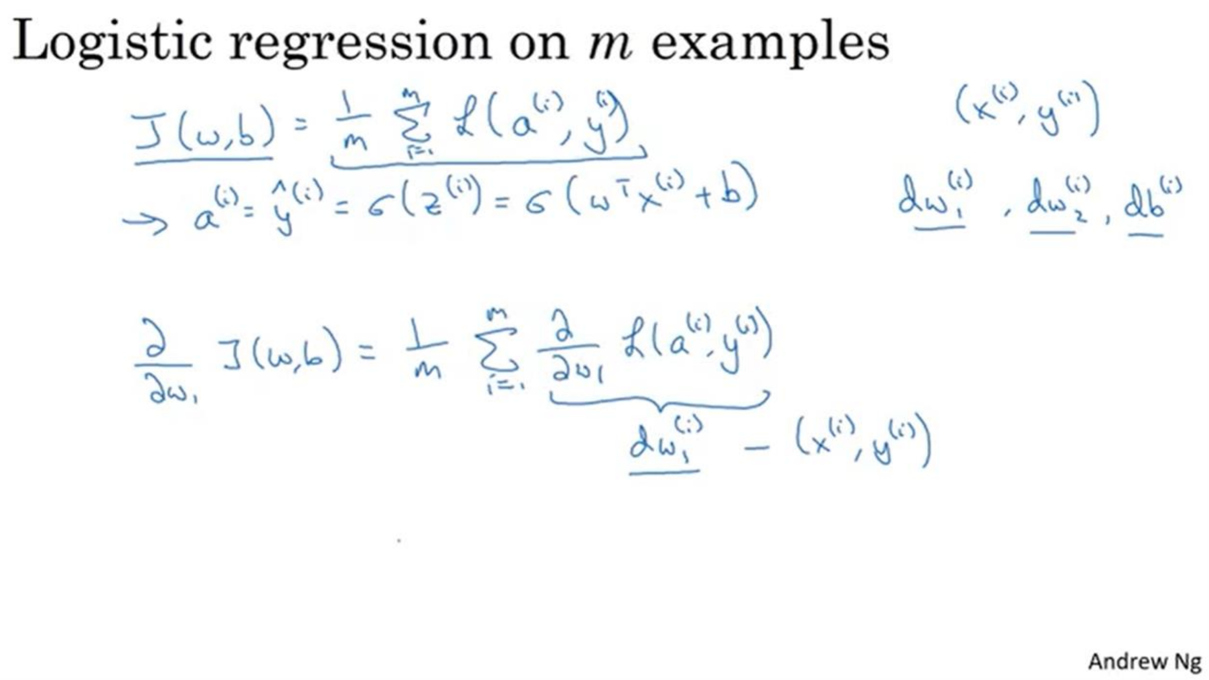
Bir sonraki başlıkta m tane training example için işlerin nasıl ilerlediğine bakacağız.

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.

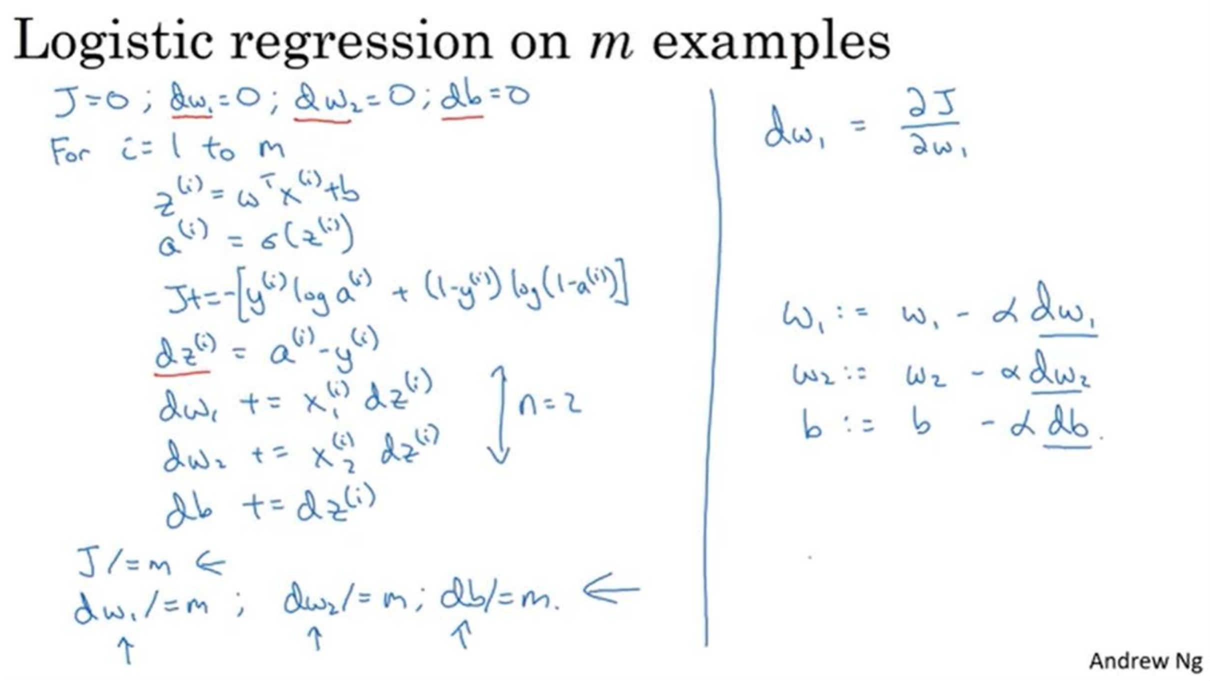
İlk olarak şunu hatırlayalım, cost function dediğimiz şey, aşağıda gösterildiği gibi her example için loss function'ın toplanması ve m'e bölünmesi ile elde ediliyor, bir başka deyişle ortalama loss.



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, dw1=0, dw2=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.



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

Ayrıca, dw1, dw2 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!

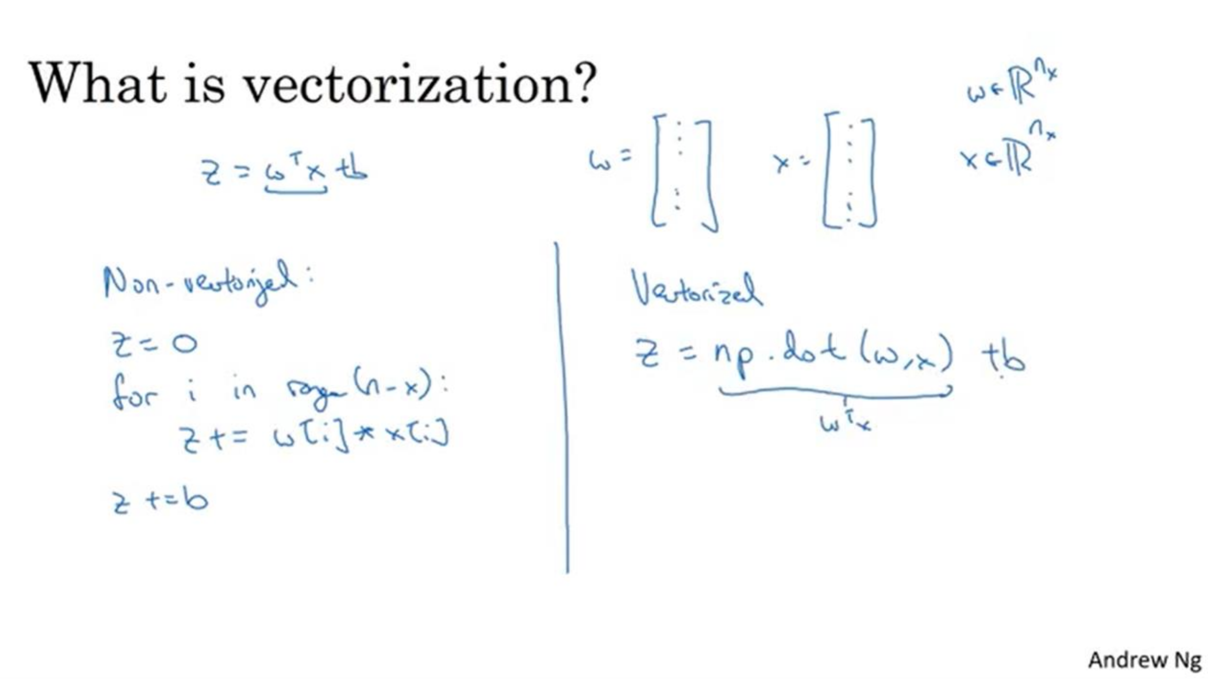
Bu durumu çözmek için ve algoritmayı daha büyük datasetlere problemsiz scale edebilmek için vectorization kavramından yararlanmalıyız.

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

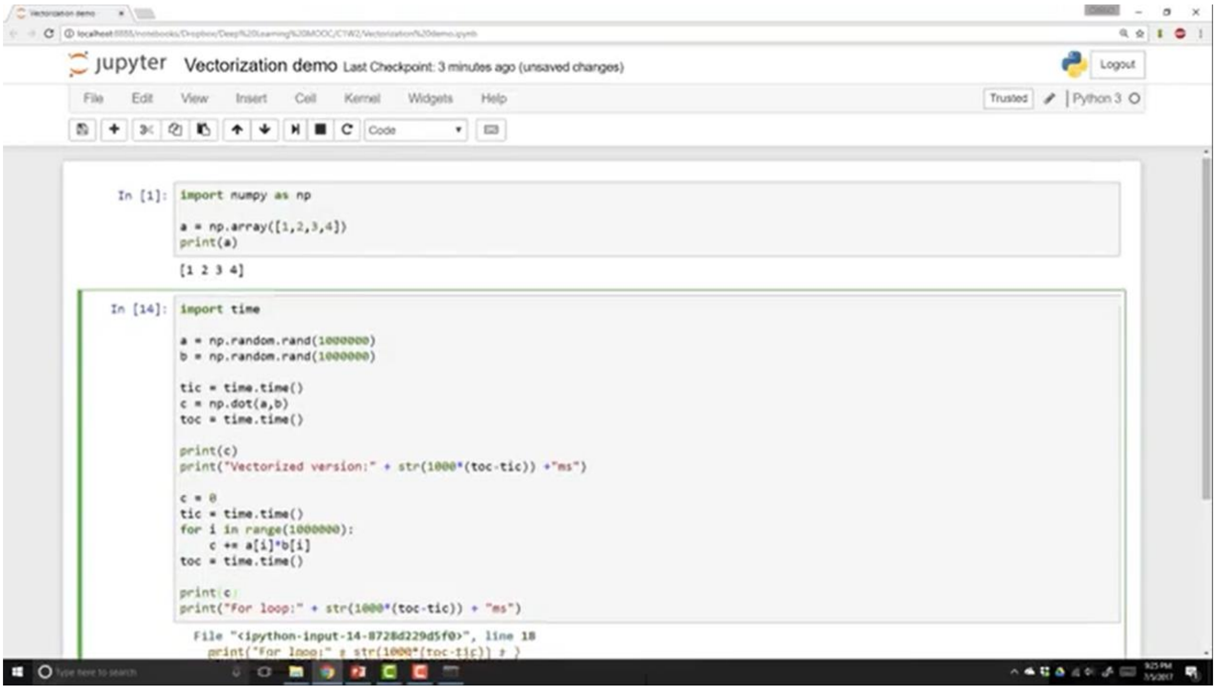
Vid17

Vectorization

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



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 wT*x'i bulabiliriz bunun için np.dot(w,x) kullanılıyor, daha sonra b'yi ekleyebiliriz.



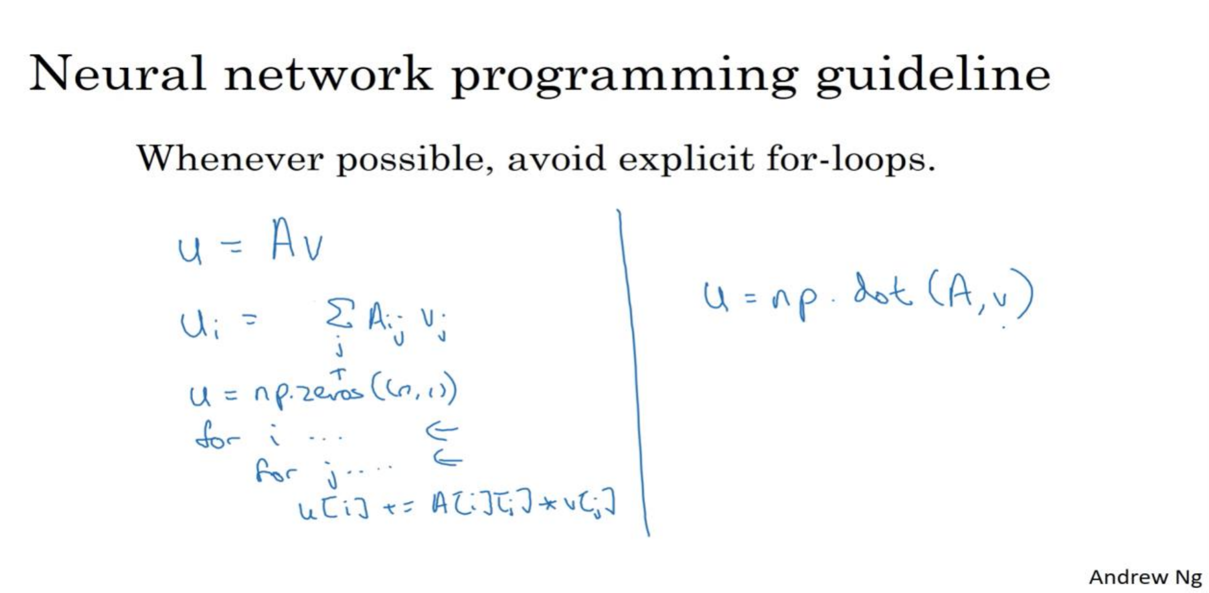
Yukarıda deneme amaçlı n = 1 000 000 için bu iki işlem karşılaştırılıyor, vectorized için yaklaşık 1.5 ms alan işlem non vectorized için yaklaşık 500 ms alıyor. İnanılmaz bir fark söz konusu.

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.

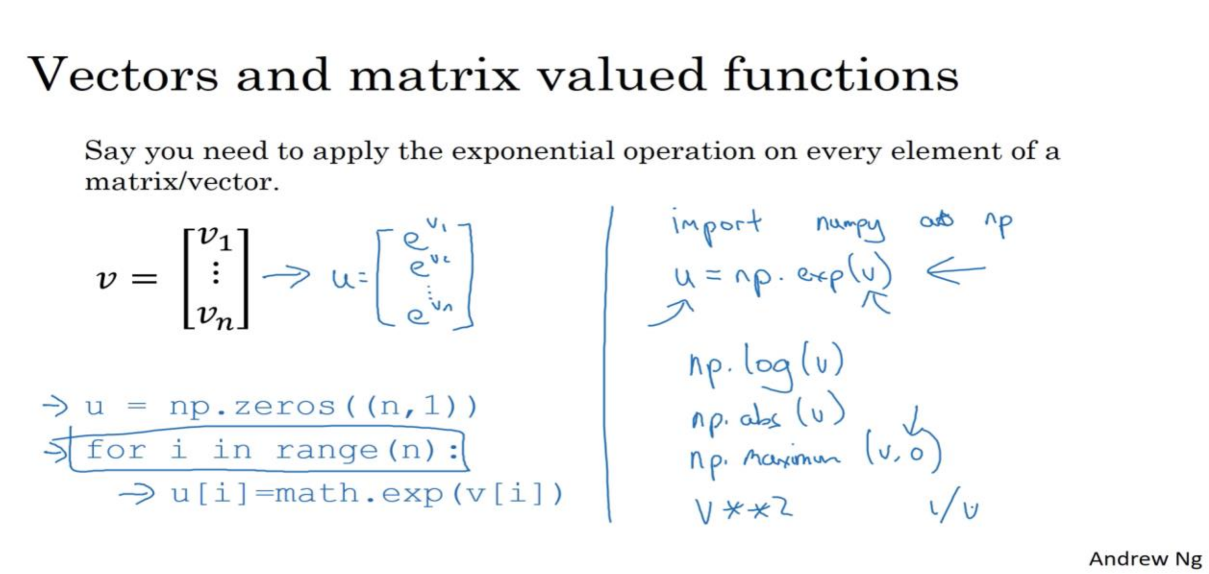
WHENEVER POSSIBLE AVOID EXPLICIT FOR LOOPS!!!

Vid18

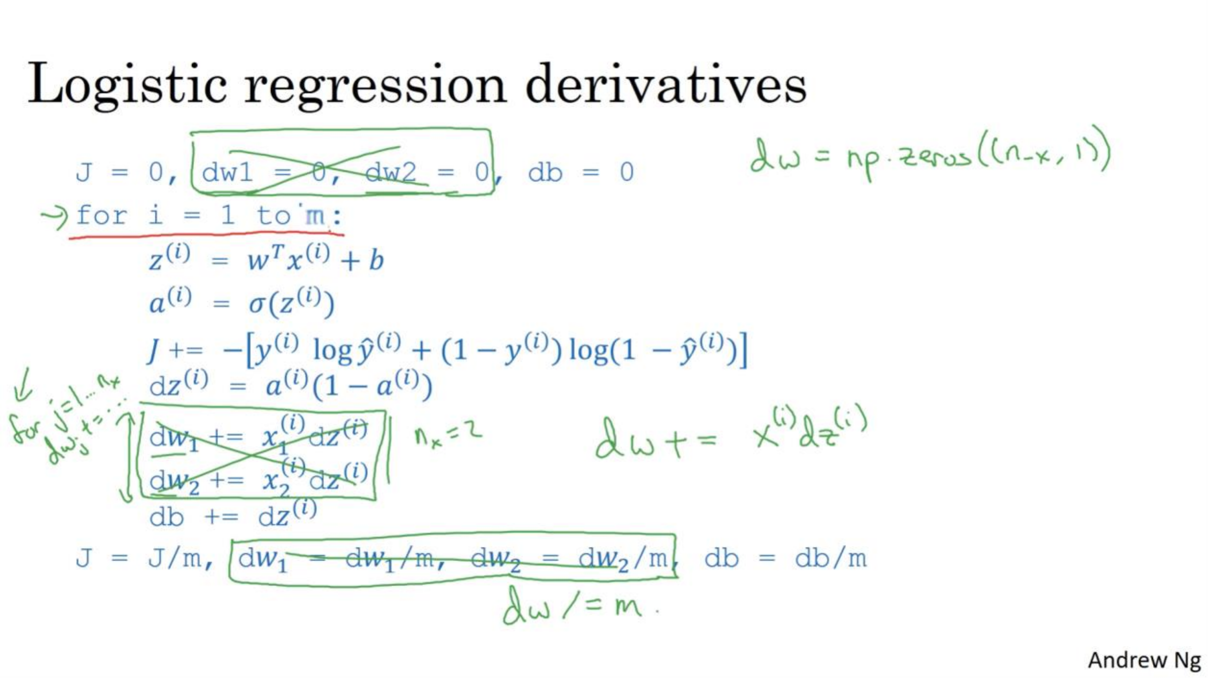
More Vectorization Examples



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.



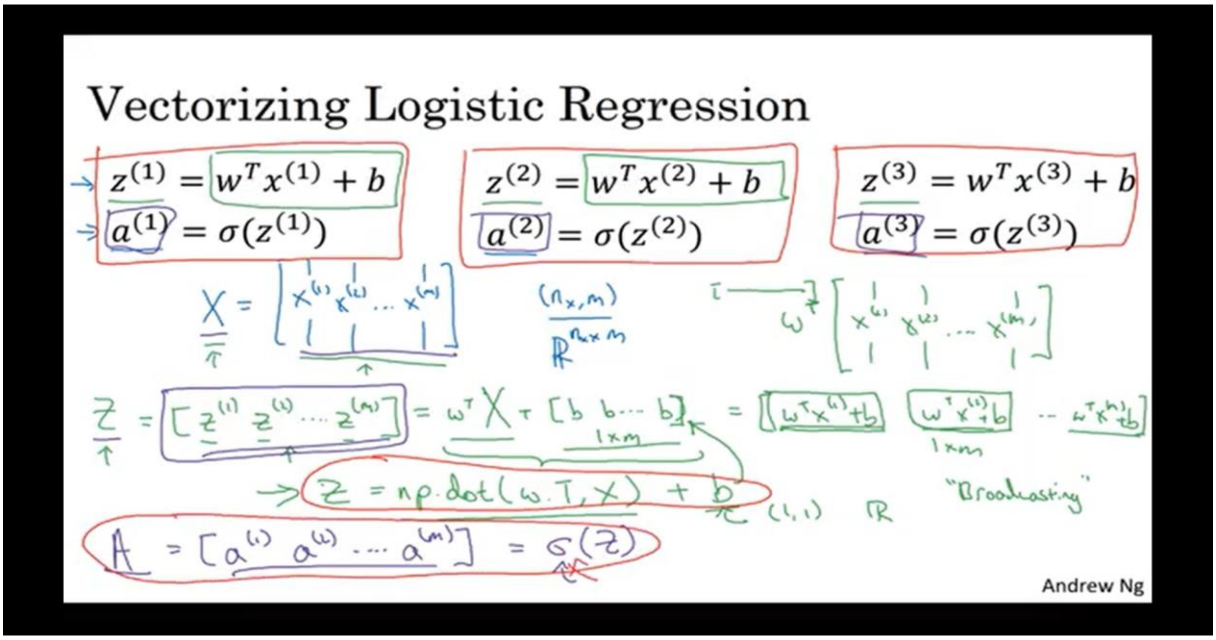
Bir başka örnek yukarıdaki gibi elementwise exponential veya diğer elementwise işlemler olabilir, numpy kullanarak bu elementwise işlemleride sağda görüldüğü gibi rahatça halledebiliriz.



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



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=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.

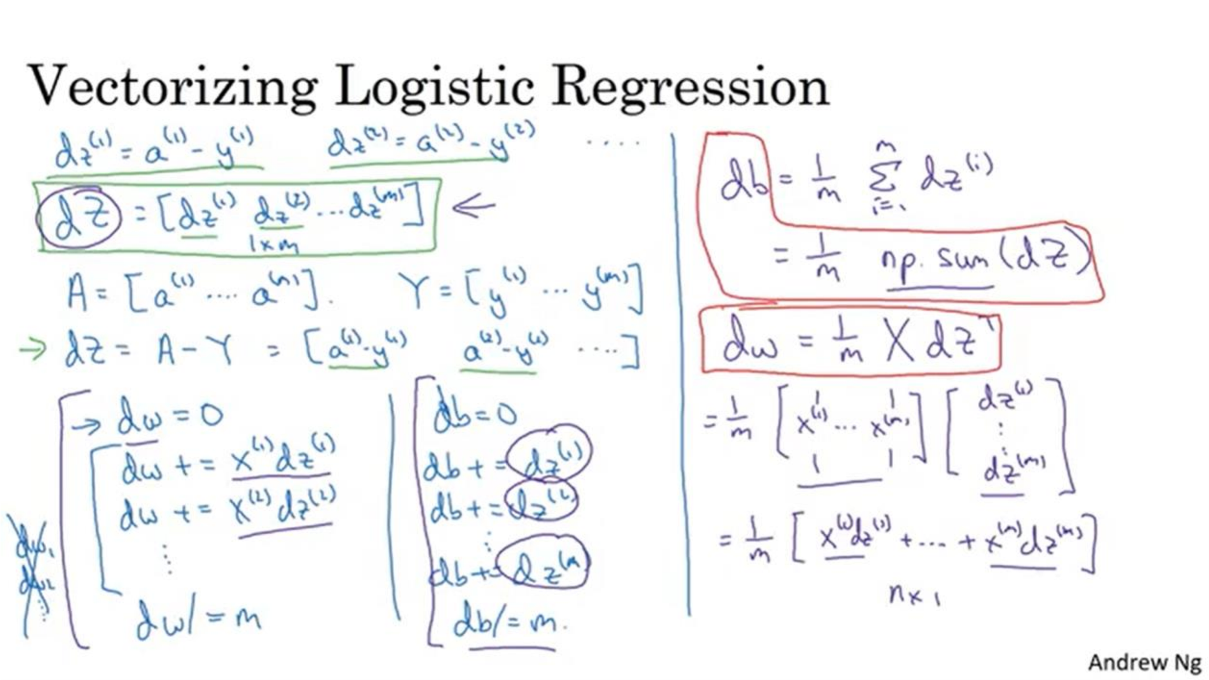
Vid20

Vectorizing Logistic Regression – Gradient Compuation

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.

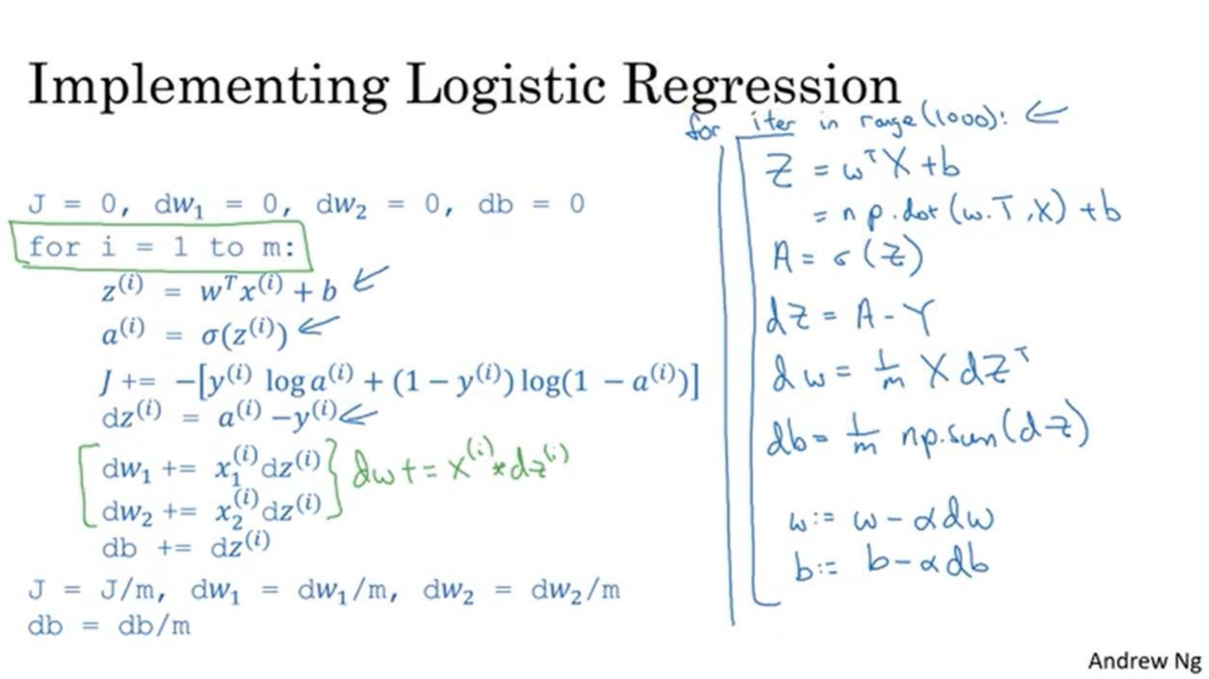


dz’leri yukarıdaki gibi her example için hesaplayabildiğimizi biliyoruz ancak, bir önceki örnekte forward propagation’ı vektörize ettik ve A ile Y’yi elde etmiştik o halde dz basitçe A-Y olarak elde edilebilir.

Bir diğer adım dw’yi elde etmek, burada içerdeki loop’u yani her ex. İçin ayrı ayrı dw1, dw2, dw3 .. hesaplamasını vektörize etmiştik ancak hala her training example için dw hesaplanıyor bunu da vektörize edebiliriz.

Dw ve db vektörizasyonu için kırmızı circled formları kullanabiliriz böylece artık iki loptan da kurtulmuş olduk.

Sonuç olarak toparlarsak, eski ve yeni formu yan yana gösterelim:



Forward propagation ile Z ve A hesaplanıyor, backward propagation ile dZ ve dW ve db hesaplanıyor, sonuçta bu hesaplamanın sonunda tek bir gradient step atılıyor, convergence için bu forward ve backward propagation’ı yine bir loop içine alıyoruz maalesef bu son loptan kurtulamıyoruz.

Vid21

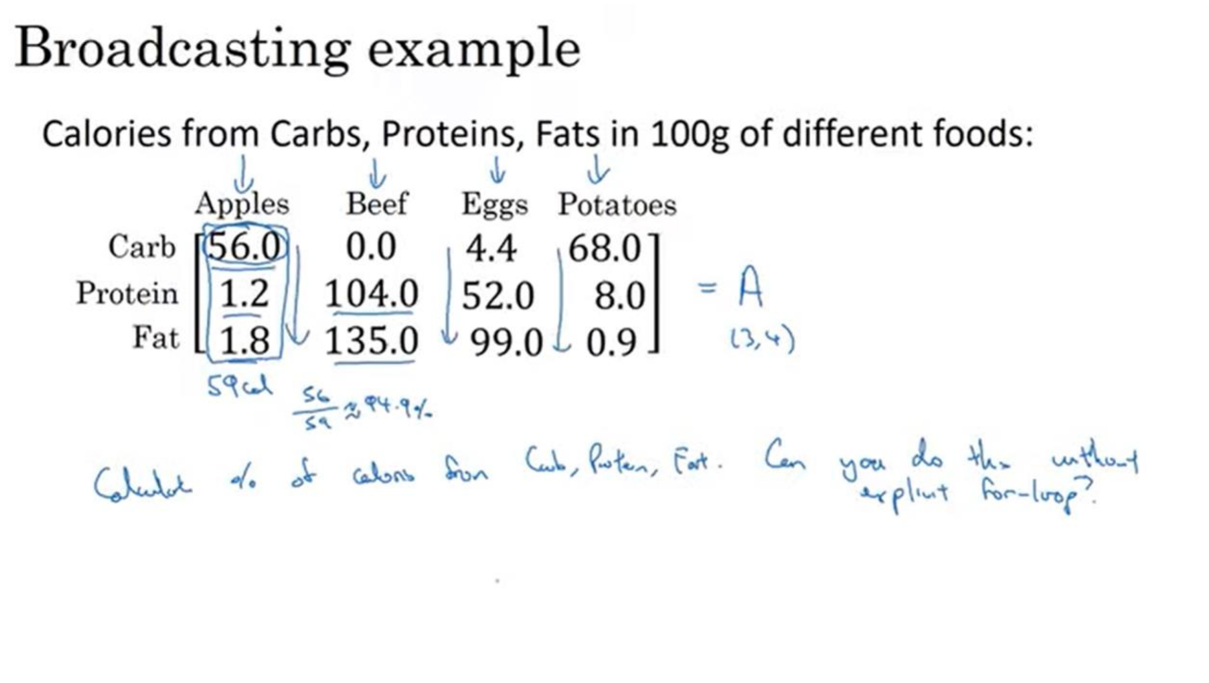
Broadcasting

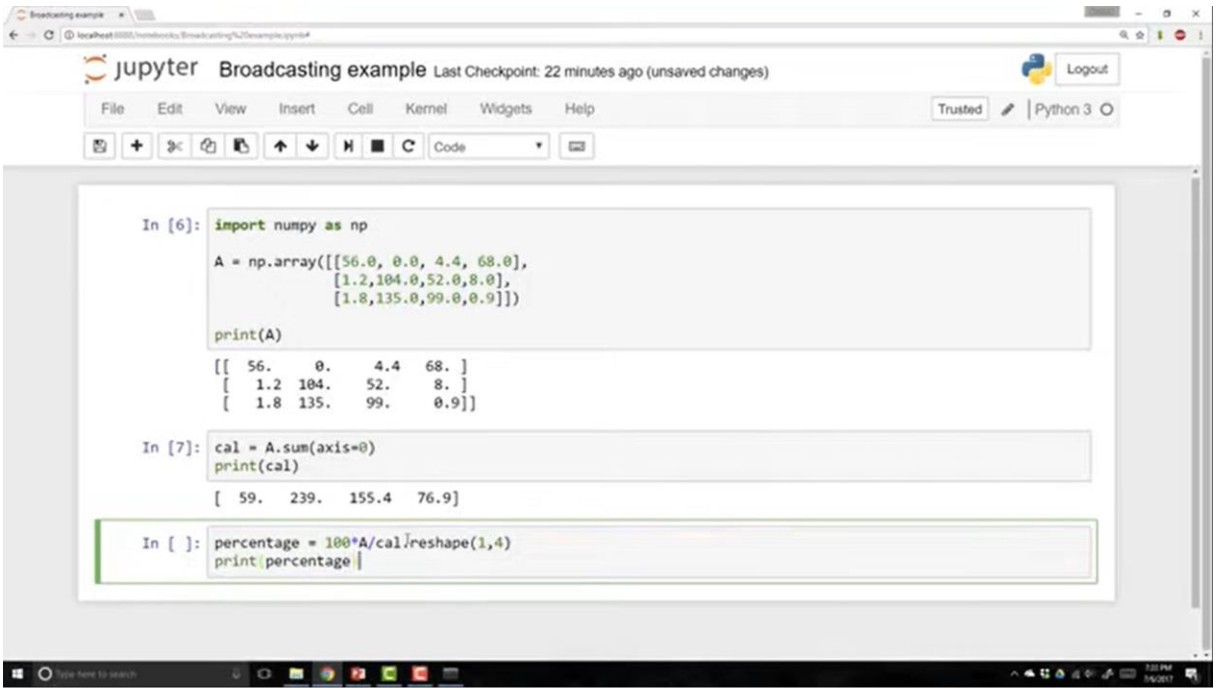
Broadcasting python kodumuzu daha hızlı çalıştırabilmek için kullanabileceğimiz bir başka tekniktir.

Bir örnekle, broadcasting’i anlamaya çalışalım.

Diyelim ki aşağıdaki gibi bir matrisimiz olsun, bu A matrisi içinde farklı besinlerin içindeki carb, protein ve fat kalorileri verilmiş. Benim istediğim her besin için bu değerleri yüzdelik değere çevirmek, yani elma için 56 cal carb yerine %94.9 carb yazacak.

Bunu loop kullanmadan yapacağız.

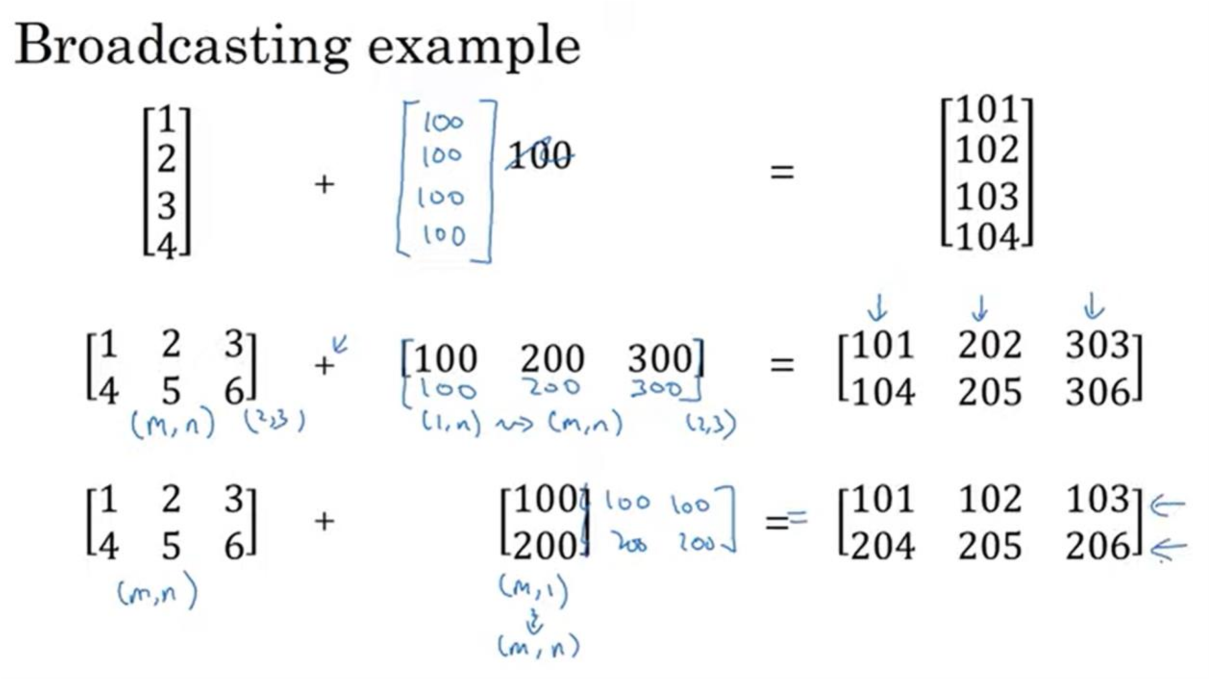




Yapılan şey önce, axis=0 yani vertical yönde matris elemanlarını toplayarak her besin için total kalori değerini elde etmek daha sonra, A matrisini bu kalori vekörüne bölüyoruz, bu bölme sıradan bir bölme değil, her sütunu bu vektör ile bölüyor yani elementwise işlem yapıyor, ama bilinen işlemlerin hiçbirine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

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

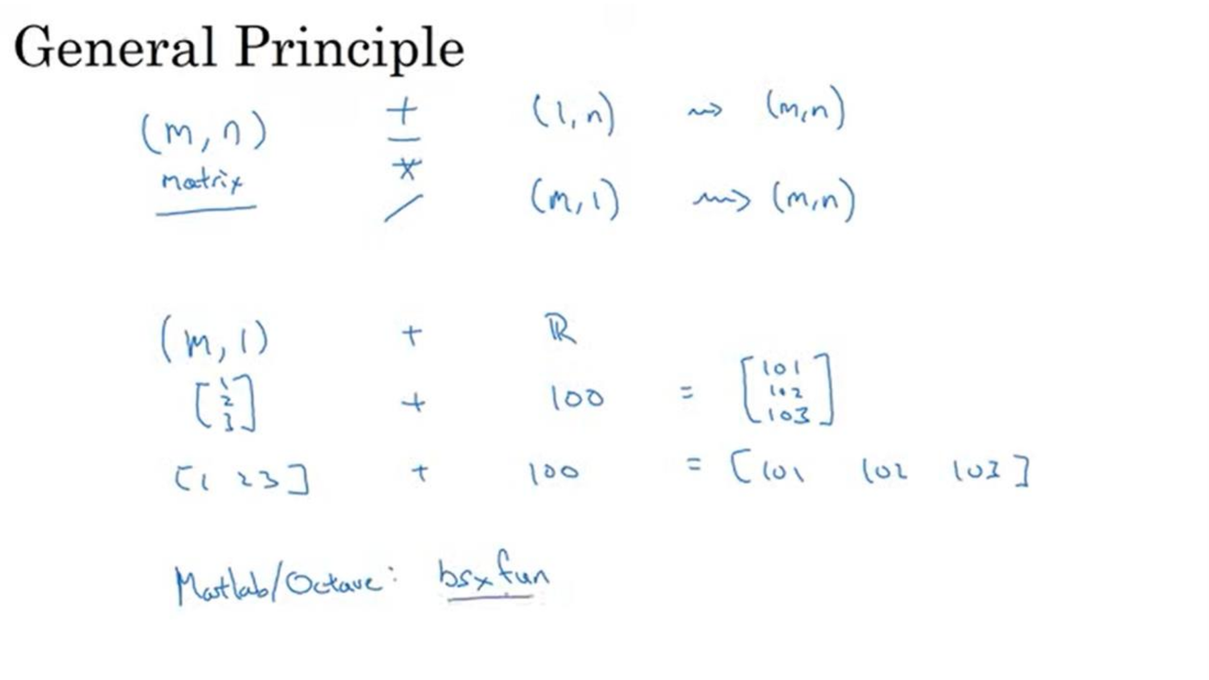
Kısaca birkaç örnekle broadcasting’in nasıl çalıştığına bakalım:



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.



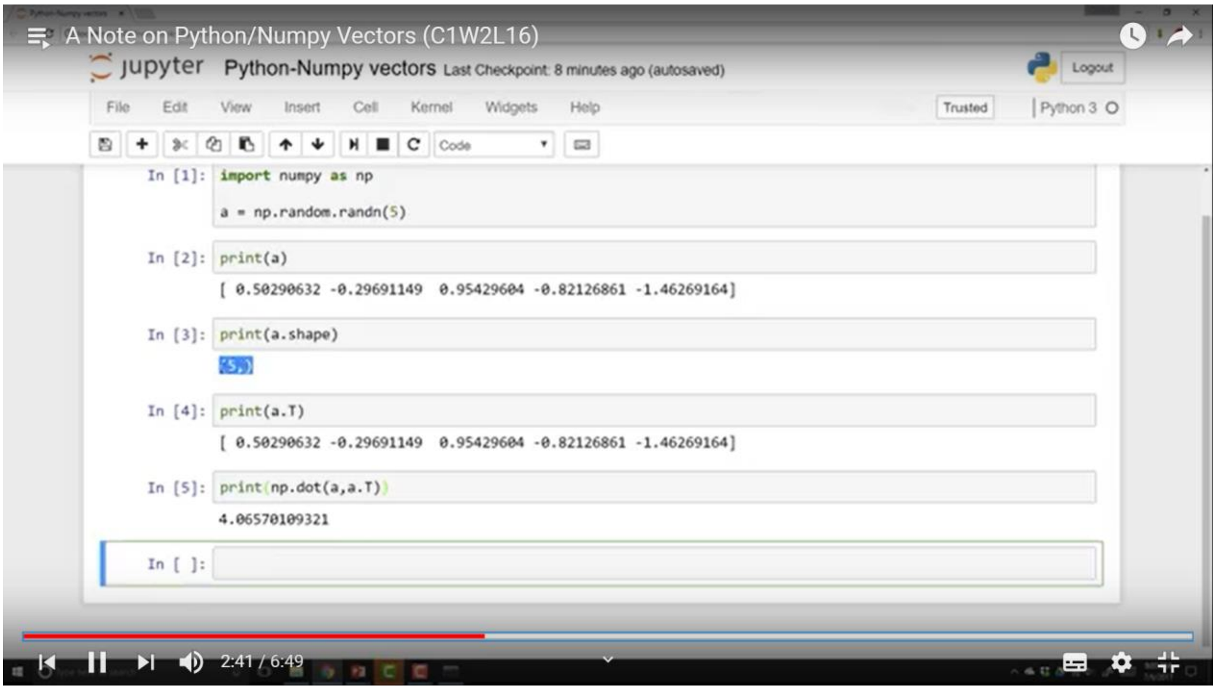
Genel olarak, bir matrise bir vektörü toplamaya çalıştığımızda vektörü alıyor, matrisle toplanabilecek şekilde genişletiyor bu sırada vektörü kopyalamaktan fazlasını yapmıyor, daha sonra bildiğimiz matris işemleri, yani eleman elemana toplama, çıkarma,çarpma,bölme yapıyor.

Vid22

Python/Numpy Vectors

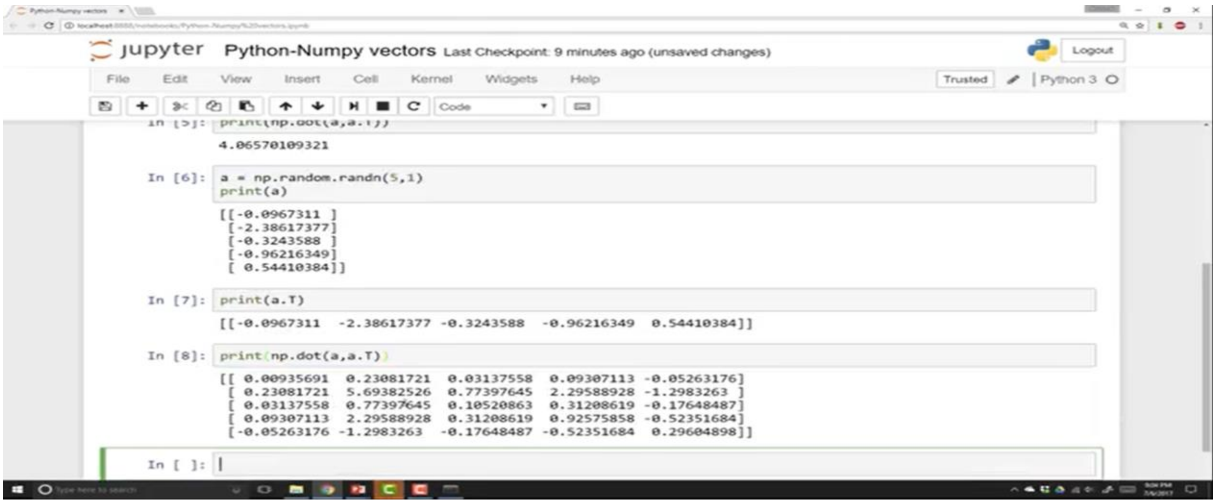
Bir önceki kısımda gördüğümüz broadcasting gibi numpy flexibilityleri işlevsel olduğu kadar kodu hataya açık hale getirir ve hataları bulması zorlaşabilir mesela bir row vector ile column vectorun toplamının hata vermesini beklerken, vermez.

Bu kısımda bazı tips and tricks'ten bahsedeceğiz:



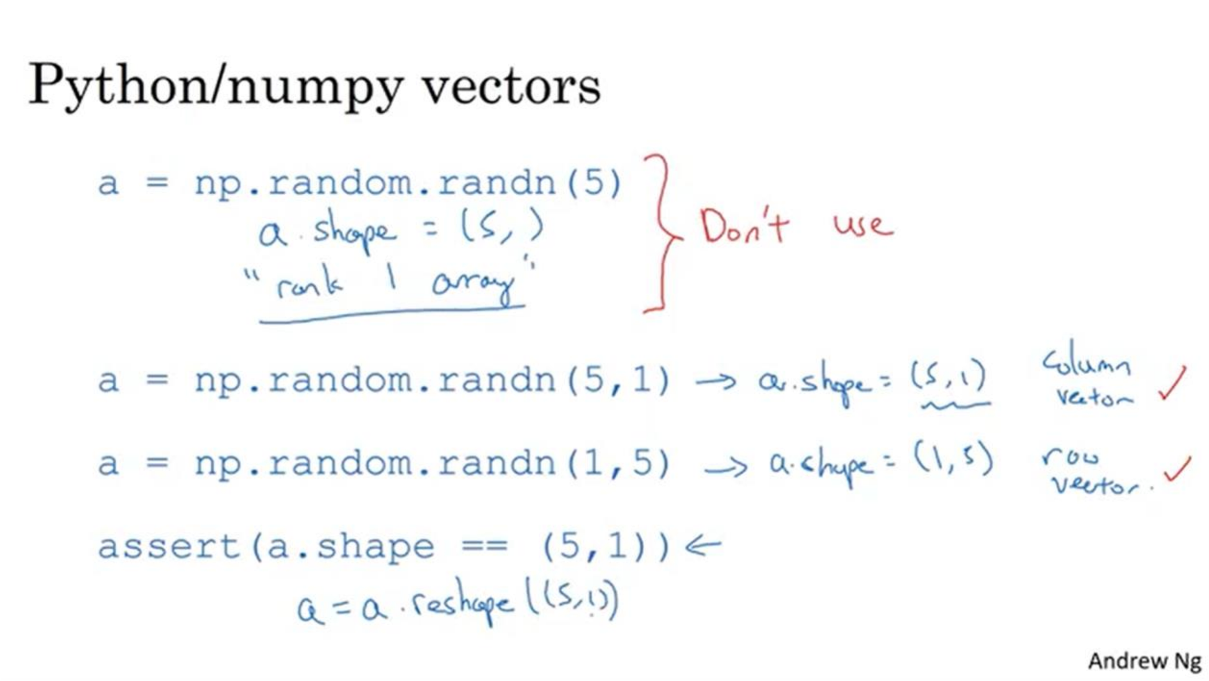
Yukarıdaki gibi bir a vektörü tanımladığımızda bunun shape'i (5,) olarak döner. Bu aslında tam olarak bir vektör gibi davranmaz buna RANK 1 ARRAY denir. a ile a.T çarpımının sonucu 5*5 bir matris çıkmasını beklerken tek bir sayı çıkıyor, tuhaf.

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



Yukarıdaki gibi bir kullanım beklenilen sonuçları karşılıyor.

Özetle:



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.