

# COURSE 2 W3

## Vid 1

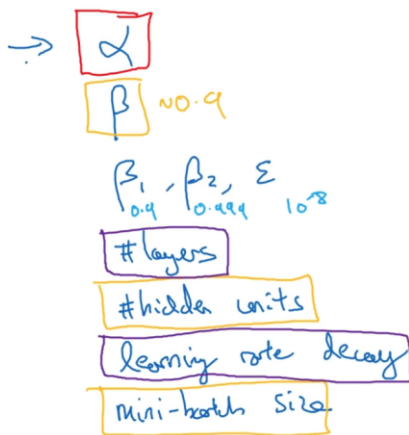
### Tuning Process

Bir NN eğitirken hyperparameter tuning önemli. Bu kısımda, sistematik olarak hyperparameter tuning'ı nasıl yapabileceğimize bakacağız.

Learning rate, Beta(if we use momentum), Beta1 Beta2 Epsilon (if we use adam) , number of layers, number of hidden units, learning rate decay, mini batch size gibi birsürü hyperparameters'dan söz edilebilir.

Bunlardan bazıları diğerlerinden daha önemlidir.

## Hyperparameters

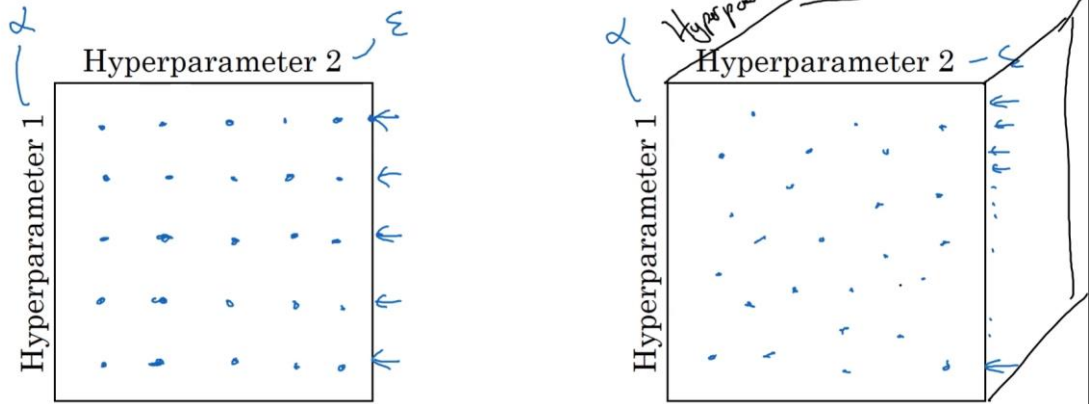


Andrew Ng

Genelde en önemlisi learning rate'tir. Daha sonra turuncu olanlar gelir, momentum için kullanılan Beta parametresi genelde 0.9 seçilir. Daha sonra # of layers ve learning rate gelir son olarak adam optimization parametreleri gelir bunları neredeyse hep 0.9, 0.999 ve  $10^{-8}$  olarak seçebiliriz.

Peki bazı hyperparametrelerim var diyelim bunları nasıl set edip test etmeliyim?

## Try random values: Don't use a grid



Andrew Ng

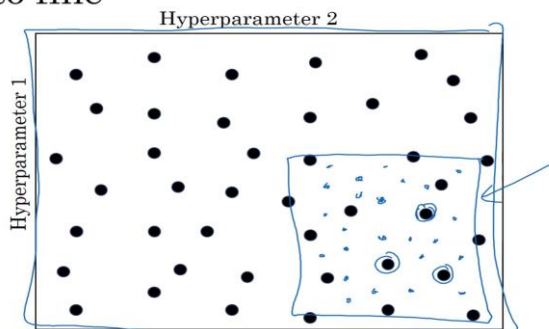
Diyelim ki 2 hyperparametrem var tune edeceğim, yukarıdaki gibi bir grid yapısı oluşturup her noktayı test edebilirim böylece 25 farklı nokta için crossvalidation set üzerinde performans ölçülür ve en iyisi bulunur.

Hyperparametre sayısı az ise bu yaklaşım uygulanabilir ancak, deep learning'de grid yöntemi yerine 25 tane random point denemek daha mantıklı. Buna **Random Sampling** denir.

Neden böyle? Diyelim ki hp1 learning rate olsun, hp2 ise epsilon yani hp1 çok önemli hp2 ise pek bir etki yaratmıyor, bu durumda soldaki yaklaşımda 25 denemenin 20 tanesi çöpe gitmiş oluyor çünkü hp2'yi değiştirmemizin hiçbir önemi yok. Ancak sağdaki yaklaşımda ise 25 farklı alpha denemiş oluyoruz burada doğru parametreyi bulma şansımız daha fazla.

2'den fazla hparametre olursa da aynı mantık işler.

### Coarse to fine



Andrew Ng

Ayrıca **course to fine yaklaşım** da sık kullanılır diyelim ki ilk uzayda 25 random noktada cross validation error bulduk sağ alttaki 3 nokta bariz iyi sonuçlar verdi, o zaman o çevrede daha hassas 25 nokta sample'ı daha yaratıyoruz ve onu test ediyoruz ki daha hassas bir hyperparametre setine ulaşalım.

# Vid 2

## Using an Appropriate Scale

Anladık ki hparameter tuning için parametrelerden random dağılmış sample alacağız ve bunu test edeceğiz, ancak bu random dağılımın nasıl bir dağılım olacağı da önemli.

Her hyperparameter için uniformly dağılmış bir random sample kullanmak doğru değil. Bu kısımda bundan bahsedeceğiz.

Diyelim ki bir hidden layer l için hidden unit sayısını yani nl'i set etmeye çalışıyorum. Aşağı yukarı min ve max range'i 50 ve 100 olarak belirlemişiz. Bu durumda 50 ile 100 arasında uniform dağılan bir random sample denemek mantıklı.

Benzer şekilde bir NN için # of layers set etmek istiyorum ve 2 ile 4 arasında hangisi daha iyi denemek istiyorum, bu durumda da 2 3 ve 4'ü denemek mantıklı.

## Picking hyperparameters at random

$$\rightarrow n^{T2} = 50, \dots, 100$$



$$\rightarrow \#layers \quad L: \quad 2 - 4$$

$$2, 3, 4$$

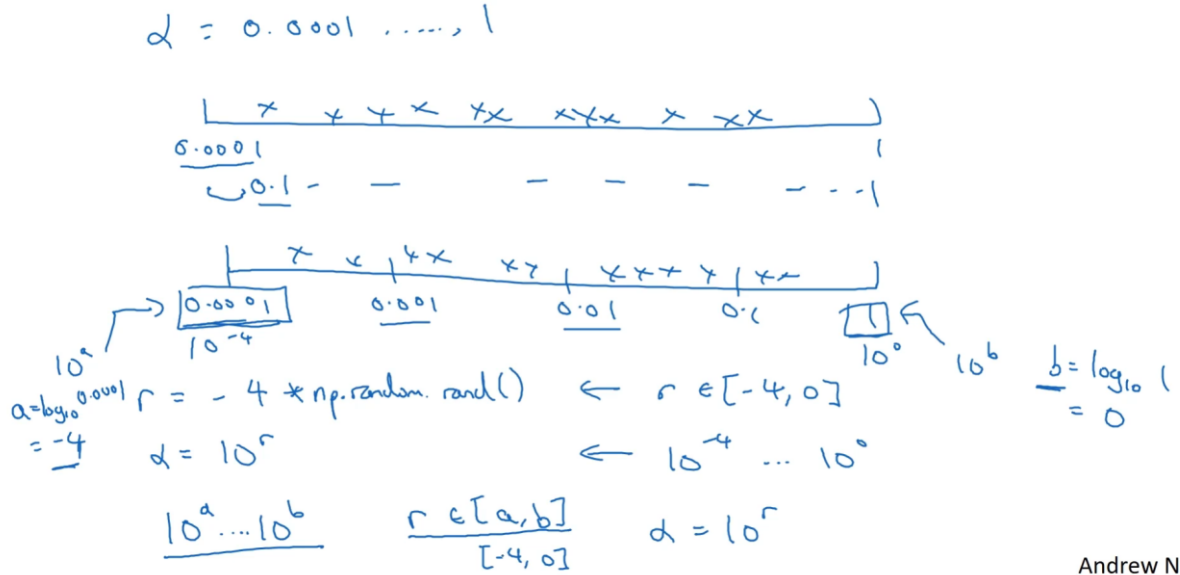
Andrew Ng

Ama bu durum tüm hyperparametreler için geçerli değil. Diyelim ki learning rate'i tune etmek istiyorum, minimum değeri 0.0001 ve max değeri 1 olarak belirledik. Bu iki değer arasında uniformly distributed bir random variable ile sample üretirsek bu sample'ın %90'ı 0.1'den büyük olacaktır. Yani totalde 100 değer deniyorsak bunun 90 tanesi 0.1 ile 1 arasındaki değerleri denerken, sadece 10 tanesi 0.0001 ile 0.1 arasına odaklanır burada doğru olmayan bir şeyler var.

Böyle yapmak yerine learning rate için 0.0001 ile 1 arasında log scale kullanarak sample alırız. Yani 100 sample varsa 4'te biri 0.0001 ile 0.001 arasında 4'te biri 0.001 ile 0.01 arasında 4'te biri 0.01 ile 0.1 arasında ve 4'te biri ise 0.1 ile 1 arasında olur.

Bunu python'da implement etmek için, -4 ile 0 arasında bir random number  $r$  generate ettiririz ve  $\alpha = 10^r$  olarak tanımlarız. Daha genel bir form için,  $r$ 'yi  $a$  ve  $b$  arasında randomly generate ederiz,  $a = \log_{10}(\min)$  ve  $b = \log_{10}(\max)$  olarak bulunabilir bu örnekte  $\min$  dediğimiz 0.0001 ve  $\max$  ise 1.

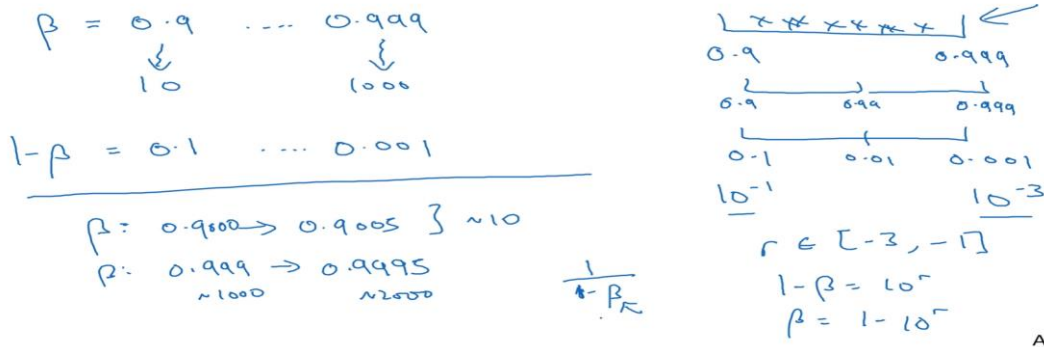
## Appropriate scale for hyperparameters



Son olarak bir diğer tricky case ise exponentially weighted averages hesaplamak için kullanılan Beta'yı tune etmek. Diyelim ki Beta 0.9 ile 0.999 arasında test edilsin istiyoruz.

Beta 0.9 demek son 10 data için average alıyor demek, 0.999 demek son 1000 data için average alıyor demek. Burada da learning rate'e benzer bir durum var, uniform distribution ile random sample almak doğru değil, çünkü 0.9 ile 0.99 arasına 90 örnek düşüyorsa 0.99 ile 0.999 arasına sadece 10 örnek düşecek, bunu da logaritmik olarak set etmeliyiz, peki nasıl yaparız?

## Hyperparameters for exponentially weighted averages



Beta'yı değil de  $1 - \text{Beta}$ 'yı set ettiğimizi düşünelim, bu durumda  $1 - \text{Beta}$ 'yı 0.1 ile 0.001 arasına set edeceğiz, learning rate'e benzer şekilde 0.1'in  $10^{-1}$  e ve 0.001'in  $10^{-3}$ 'e denk geldiğini bulurum ve daha sonra  $r$  için -3 ile -1 arasında random alırım ve  $1 - \text{Beta}$   $10^r$  olarak tanımlanır dolayısıyla  $\text{Beta} = 1 - 10^r$  olur.

Peki uniform olarak almak neden bu kadar yanlış? Yani 0.9 ile 0.99 arasında 0.99 ile 0.999'a göre daha fazla sample olması neden yanlış? Çünkü Beta büyüdükçe hassaslaşıyor yani şöyle beta 0.9 veya 0.9005 olmuş çok farketmez ortalama 10 örnek için average alır ancak Beta'nın 0.999 değil de 0.9995 olması önemli burada 1000 yerine 2000 örnek için average almasından bahsediyoruz, bu yüzden uniform test edersek, sallıyorum 10 örnek için sample alacak sonra 100, 1000, 10000 gibi artacak ben bunun daha hassas artmasını istiyorum 30 50 300 500 de güme gitmesin gibi düşün.

Ama bu dünyanın sonu da değil, sonuçta course to fine methodunu uygularsam diyelim ki 100 ile 10000 iyi çıktı burada scale'i yoğunlaştırıp uniform bile dağıtsam öyle ya da böyle düzgün bir sonuca ulaşırım.

## Vid 3

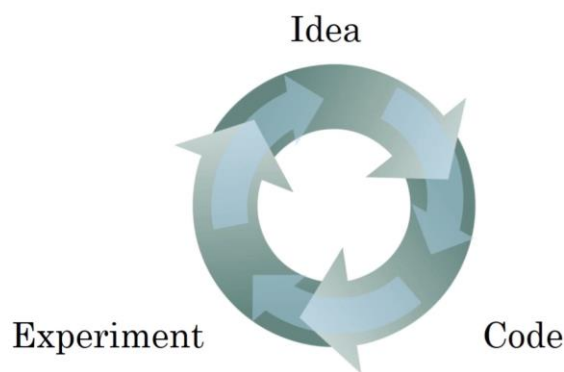
### Hyperparameter Tuning in Practice

Anlatılanları toparlamadan önce son birkaç tip ve tricksten bahsedelim.

Bildiğin gibi deep learning'in NLP, CV, Speech Recognition, Logistics gibi bir sürü farklı uygulama alanı var bu alanların birinde bulunan fikirler diğerine de uygulanabiliyor mesela Computer Vision için bulunan resnets fikri speec recognition'a da uygulanıyor. Bu yüzden bir alanda çalışanlar diğer alanlarla ilgili yenilikleri takip edip uyarlama yapmaya çalışırlar.

Ancak hyperparameter tuning için farklı alanlarda edinilen intuition'lar genelde diğerinde işlevsiz kalır.

## Re-test hyperparameters occasionally



- NLP, Vision, Speech,  
Ads, logistics, ....
- Intuitions do get stale.  
Re-evaluate occasionally.

Andrew Ng

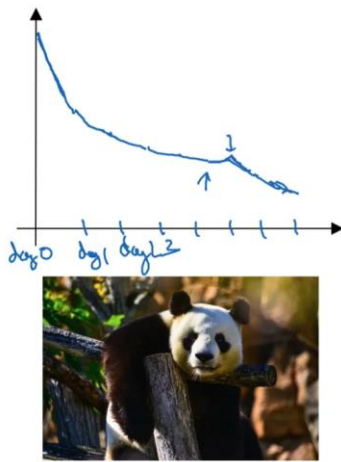
Aynı alanda bile data'da algoritmada veya başka şeylerde zamanla değişimler olabilir bu sebeple belirli aralıklarla tekrar hyperparameter tune etmekte fayda var.

Son olarak hyperparameter tune etmek için 2 farklı yaklaşımdan söz etmek mümkün birincisi Babysitting one model, bu yöntem genelde büyük bir datasetimiz olduğunda ancak buna karşın çok computation power'ımız olmadığında kullanılır, yani bir ve birkaç model dışında daha fazla model eğitmek çok fazla zaman alacaksa bu methoda başvurulur.

Yapılan şey elimizdeki modele training süresince bakıcılık yapmak ve hyperparametrelerini tune etmek ve değişimi gözlemleyerek bir sonraki adımda ne yapacağımıza karar vermektir. Aşağıdaki gibi diyelim ki ilk gün parametreleri random seçtik, 2. Gün dedik ki biraz learning rate'i artırayım başka bir şeyi düşüreyim vesaire, böyle böyle hergün sonuca göre el ile değişiklik yapıp modeli eğitebiliriz.

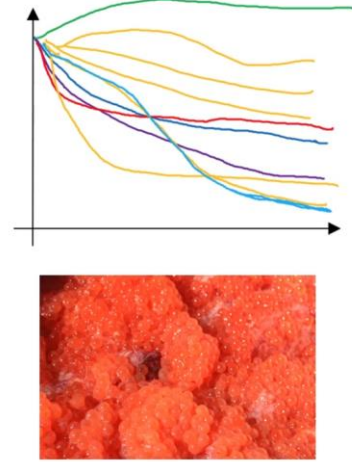
Ancak bu modelde structure'ı değiştirmek mümkün olmamalı çünkü o zaman her şeyi baştan hesaplamak zorunda kalacak, yani bu yöntemle sadece learning rate, beta vesaire gibi değiştiğinde eğitimin sıfırlanmasını gerektirmeyen parametreler tune edilebilir.

## Babysitting one model



Panda

## Training many models in parallel



Caviar

Andrew Ng

İkinci yöntem ise, training many models in parallel, yani temelde her bir farklı hyperparameter seti için başka bir modeli eğitirim ve performansını test ederim ona göre en iyisini seçerim.

Eğer computation power'ım varsa elbetteki caviar approach'ı seçmek daha mantıklı ama bazı uygulamalarda bu olmaz.

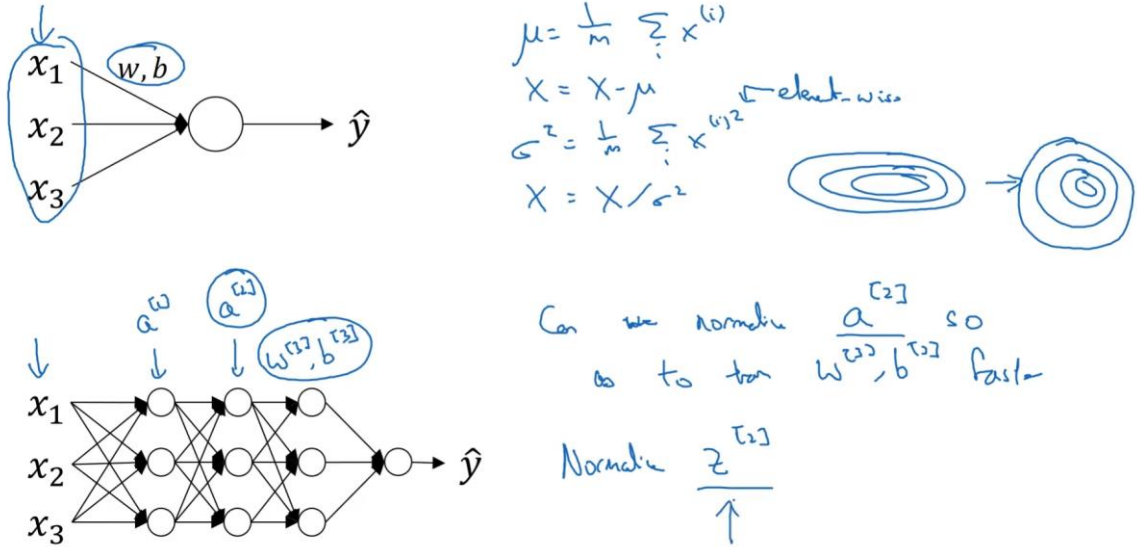
# Vid 4

## Normalizing Activations in a Network

Deep learning'in yükselişinde önemli fikirlerden biri **Batch Normalization** oldu. **Batch normalization makes our hyperparameter search easier, it makes our NN much more robust to the choice of hyperparameters yani daha geniş bir rangedeki hyperparametreler iyi çalışır. Ayrıca daha kolay bir şekilde oldukça derin networkler eğitilebilir.**

Biliyoruz ki bir model train ederken, input features'ı normalize etmek learning process'i hızlandırıyor. Bunu aşağıdaki şekilde yapıyorduk, yani dataset'in mean'ini buluyoruz, sonra her datadan bu mean'i çıkarıyoruz, benzer şekilde variance'ı buluyoruz ve her datayı buna bölüyoruz. Böylece cost function'ı elongated formlardan alıp daha round ve optimization'ın daha iyi olduğu bir forma getirebiliriz.

## Normalizing inputs to speed up learning



Andrew Ng

Peki ya elimizde yukarıdaki gibi daha derin bir network varsa? Elimizde sadece input features değil bunun yanında activations  $a_1, a_2$  vesaire de var.

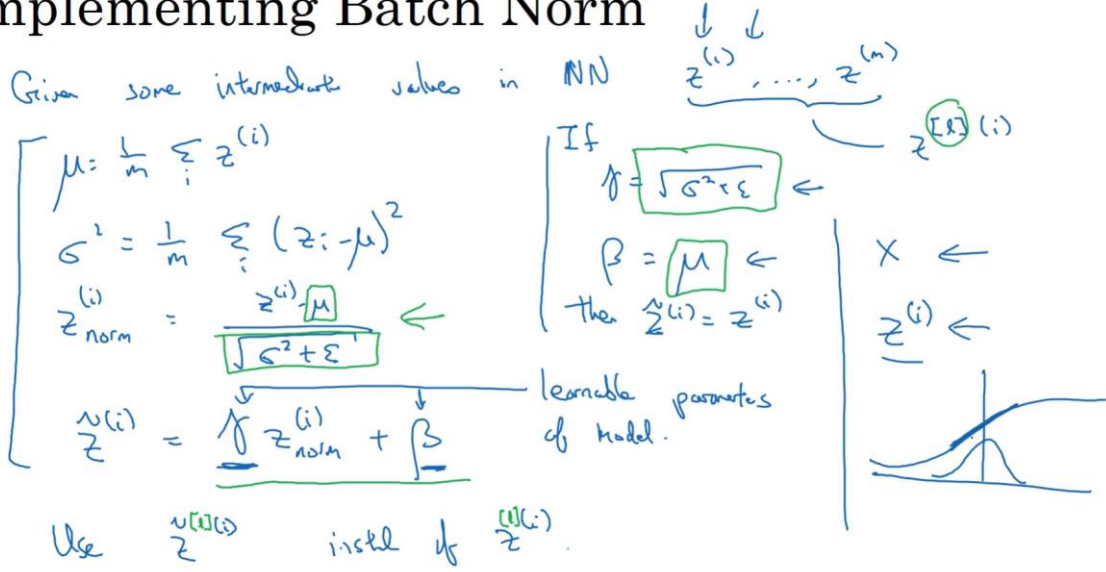
Fikir şu, eğer input feature'ları normalize ederek, bir logistic regression unit için  $w, b$  parametrelerini öğrenmeyi kolaylaştırabiliyorsak, acaba deep NN için mesela  $a_2$ 'yi normalize etsek,  $w_3$  ve  $b_3$  parametrelerini öğrenmeyi kolaylaştırır mıyız?

Burada teknik olarak  $a_2$  yerine  $z_2$ 'yi normalize edeceğiz, ama bu konuda tartışmalar var, kimisi  $a_2$ 'yi normalize etmemiz gerektiğini söyler.

Uygulanışına bakalım:



# Implementing Batch Norm



Andrew Ng

Diyelim ki elimizde herhangi bir layer için  $z_1, z_2, \dots, z_m$  vektörleri mevcut, her örnek için bir  $z$  vektörü. Burada computation'a devam etmeden önce bu  $z$ 'leri normalize edeceğiz:

Aynı input feature normalize eder gibi  $z_{\text{norm}}(i)$  hesaplanır daha sonra eskiden farklı olarak  $z_{\text{tilda}}(i)$ 'ye geçilir çünkü  $z_{\text{norm}}$ 'un dağılımı için  $\text{mean}=0$  ve  $\text{variance}=1$  olur biz bunu input için istesek de  $z$  için istemeyebiliriz çünkü linearity'den yararlanamayız. Sağ alttaki şekile bakarsan eğer  $z$ 'nin dağılımı böyle olursa sürekli 0 etrafında gelecek e o zaman benim activation'ı nonlinear seçmemin bir anlamı kalmaz, sürekli linear alanda çalışırım.

Bu yüzden bu formdan kurtulmak için  $z_{\text{tilda}}(i)$  kullanılıyor, bu formdaki 2 parametre öğrenilebilir parametreler, eğer yukarıda ortadaki gibi seçilirse zaten  $z_{\text{tilda}}$  ile  $z$  birbirine eşit oluyor.

Sonuçta ileriki hesaplamalar için  $z$  yerine  $z_{\text{tilda}}$  kullanılır.  $z_{\text{tilda}}$  içindeki iki dağılım parametresini learning algorithm öğrenir.

İleriki kısımda önce bu uygulamayı genelleştirip tüm layerlar için uygulanışına bakacağız daha sonra da batch norm'un neden işe yaradığını intuitive biçimde anlamaya çalışacağız.

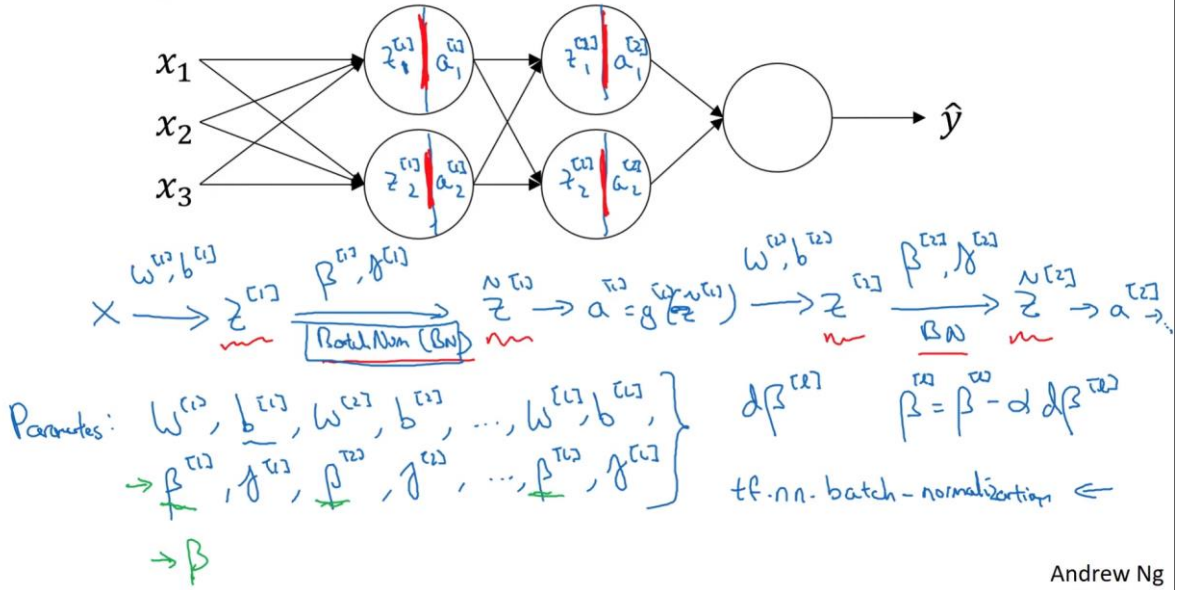


# Vid 5

## Fitting Batch Norm Into a NN

Önceki kısımda batch norm'un tek bir layer için implemantation'ınını gördük, şimdi tüm NN için bunu nasıl genelleştireceğimizi göreacağız.

## Adding Batch Norm to a network



Andrew Ng

Daha önce olduğu gibi inputlar ile  $w_1$  ve  $b_1$  kullanılarak  $z_1$  hesaplanır daha sonra eskiden bunu doğrudan  $a_1$  için kullanıyorduk ancak burada önce batch norm uygulanıyor ve  $z_1$  elde ediliyor ve  $a_1$  bu  $z_1$  ile elde ediliyor, daha sonra aynı işlem networkun sonuna doğru devam ediyor.

Yani batch norm'un uygulandığı nokta mimaride görünen kırmızı çizginin olduğu yer,  $z_1$  ile  $a_1$  arasındaki geçişte uygulanıyor.

Sonuçta yeni networkde eski parametrelerin yanına her layer için Beta ve Gamma parametreleri ekleniyor buradaki Beta'nın momentum betası ile hiçbir alakası yok.

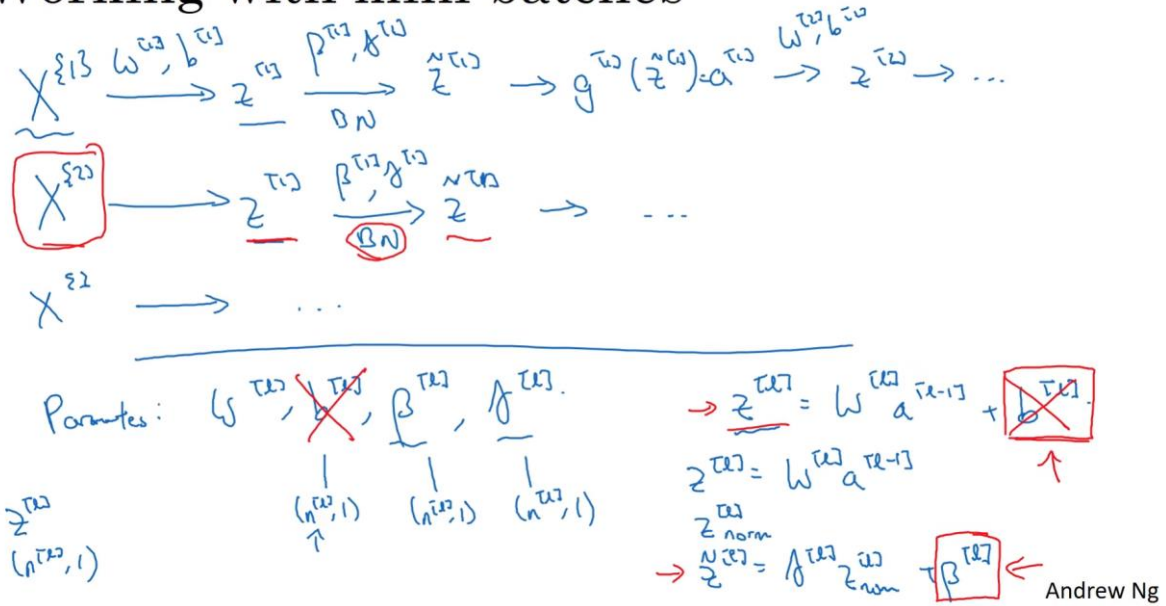
Sonuçta artık modelimin parametreleri bunlar ve learning için istediğim optimizasyonu kullanabilirim, mesela gradient descent kullanırım ve aynı diğer parametreler gibi Beta ve Gamma'yı da update ederek modelimi iyileştiririm.

Batch norm'un nasıl hesaplandığını mean ve variance bularak geçen derste açıklamıştık ama genelde pratikte bunu kendimiz uygulamayız, tensorflow ile `tf.nn.batch_normalization` ile bunu uygulayabiliriz.

Şimdiye kadar batch norm uygulamasında bir gradient step için tüm dataset'in kullanıldığını durumu ele aldık, yani batch gradient descent kullandığımız varsaydık, pratikte batch norm genelde minibatch gradient descent ile kullanılır.

Mini batch gd için batch normalization uygulamasına bakalım:

## Working with mini-batches



Önce ilk batch data alınır, buradan  $z_1$  hesaplanır, sadece bu minibatch için mean ve variance hesaplanır sonuçta bu batch için  $z$ tilde hesaplanır sonra aktivasyon function uygulanır böyle çıkışa kadar ilerler. Sonuçta bir minibatch gd step tamamlanır.

Daha sonra ikinci minibatch için aynı işlemler tekrar edilir. İkinci minibatch için  $z$ tilde hesaplanırken ilk minibatch datası ile hiçbir işlemiz olmaz, herşeyi sıfırdan hesaplarız. Bu şekilde diğer minibatchler için de aynı adımlar tekrarlanır.

Daha önce demiştik ki her layer için  $w$   $b$  beta ve gama parametrelerinden bahsedilebilir, ancak aslında batch norm kullanıldığında  $b$  parametresi kullanılmaz. Çünkü batch norm sırasında yapılan işlemlerden sonra bu parametrenin bir önemi kalmaz, mean subtraction step ile bu eklenen constant sıfırlanacak.

Bu yüzden  $z$ 'i  $b$ 'siz hesaplarız ve gama ve beta parametreleri ile scale edilir, sonuçta beta parametresi  $b$ 'nin yerini tutmuş oluyor. Vektör boyutlarına bakarsak, her hidden unit için bir beta ve gama parametresi olduğunu görürüz, yani her hidden unitin mean ve variance'ı scale edilir.

## Implementing gradient descent

for  $t = 1 \dots \text{num MiniBatches}$   
 Compute forward pass on  $X^{\{t\}}$ .  
 In each hidden layer, use BN to replace  $z^{\{t\}}$  with  $\tilde{z}^{\{t\}}$ .  
 Use backprop to compute  $\frac{dL}{dW^{\{t\}}}, \frac{dL}{d\beta^{\{t\}}}, \frac{dL}{d\gamma^{\{t\}}}$ .  
 Update parameters  $\left. \begin{aligned} W^{\{t+1\}} &:= W^{\{t\}} - \alpha \frac{dL}{dW^{\{t\}}} \\ \beta^{\{t+1\}} &:= \beta^{\{t\}} - \alpha \frac{dL}{d\beta^{\{t\}}} \\ \gamma^{\{t+1\}} &:= \gamma^{\{t\}} - \alpha \frac{dL}{d\gamma^{\{t\}}} \end{aligned} \right\} \leftarrow$   
 Works w/ momentum, RMSprop, Adam.

Son olarak yukarıda tüm adımlar yer alıyor, gradient descent'in nasıl uygulanacağı da yazıyor, diğer optimizasyon algoritmalarında kullanılması mümkün.

## Vid 6

### Why does batch normalization works?

Bu kısımda batch norm'un neden işe yaradığını anlamaya çalışacağız.

İlk neden şu: Input features'ın mean=0 ve variance=1 olarak normalize edilmesinin learning'i nasıl hızlandırdığını zaten biliyoruz. Yani bazı feature'ların 0-1 arası bazılarının ise 0-1000 arası bir scale'de olması yerine normalizasyon ile hepsinin benzer scale'lerde olması learning'i hızlandırabilir. Benzer şekilde batch norm da aynı şeyi uyguluyor, bunu sadece inputs için değil, hidden units için de uyguluyor.

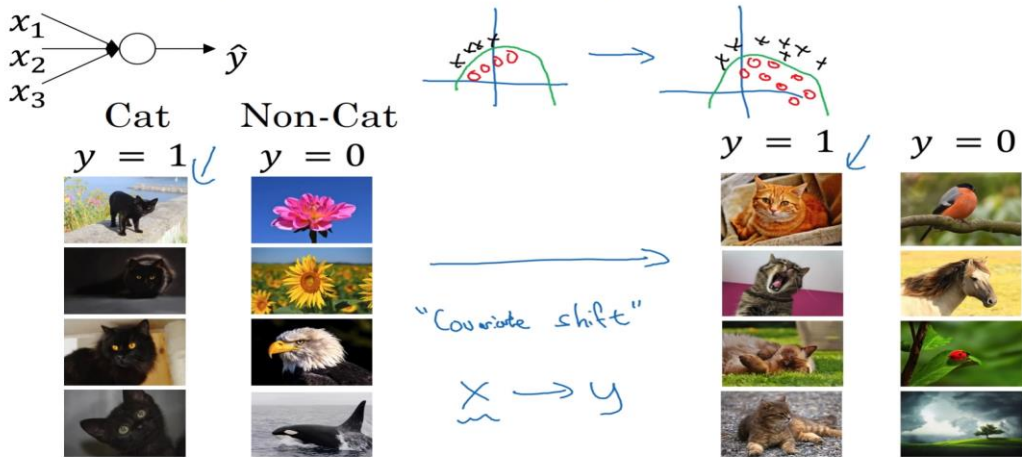
Bu batch norm'un tek bir boyutu, batch norm'u daha iyi anlayabilmek için birkaç farklı intuitiondan da sözedilebilir.

İkinci neden ise şu: batch norm derin katmanların weight'lerini sık katmanların weight değişimlerine karşı daha robust bir hale getirir. Bunu daha iyi anlamak için bir örneğe bakalım:

Mesela elimizde shallow bir network olsun (log reg. veya kısa bir NN) sonuçta amacı cat classification olsun. Modeli siyah kedilerle eğittik diyelim, ancak testi renkli kedilerle yapıyoruz, sonuçlar pek de iç açıcı olmaz. Bunun sebebi aşağıda çizilen data dağılımına bakarak anlaşılabilir, soldaki kedilerle oluşturulan dağılım soldaki grafikteki gibi olur aslında sağdaki ile çelişmiyor sadece, ikisi bir bütünün parçaları ancak sadece birini kullanarak bir eğitim yaparsak diğerine uymayabilir.

Bu data distribution'ın değişme olayına covariant shift denir. Ana fikir şu, bir  $x \rightarrow y$  map'i öğrendiysek ve  $x$ 'in dağılımı değişirse learning algorithm'i tekrar train etmemiz gerekir.

### Learning on shifting input distribution



Andrew Ng

Bu covariant shift problemi NN için nasıl oluyor?

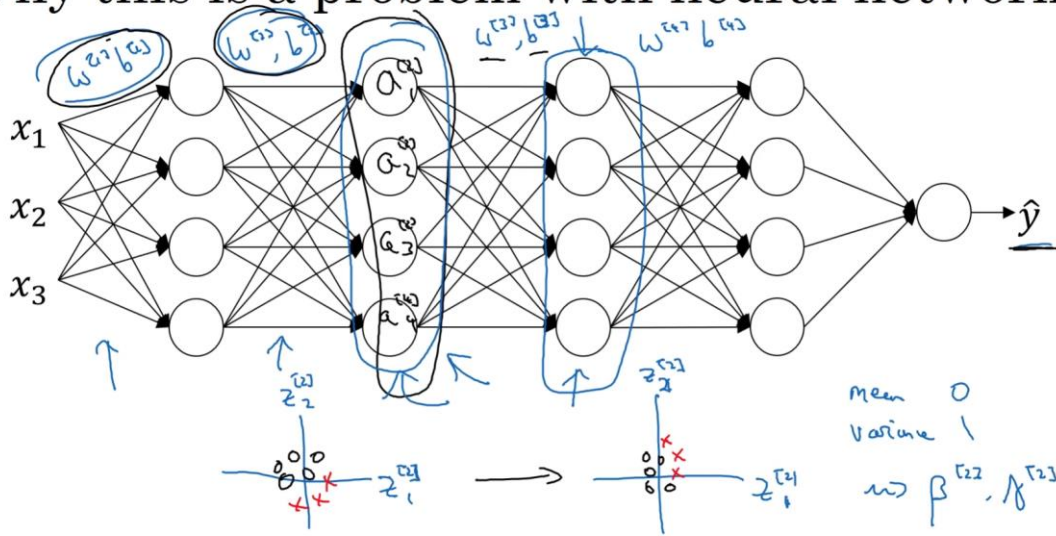
Diyelim ki elimizde aşağıdaki gibi bir deep NN olsun ve diyelim ki 3. Layer'ın learning process'ine odaklanıyoruz. Bu layer'ın öğrenmesi gereken parametreler  $w_3$  ve  $b_3$ .

Şimdilik sadece  $w_3$  ve  $b_3$ 'ün hesaplanmasına odaklandığımız için, önceki kısmı düşünmeyelim ve buraya girdi olarak  $a_{21}$   $a_{22}$   $a_{23}$   $a_{24}$  girdiğini biliyoruz. Sonuçta 3. Hidden layer'ın amacı bu değerleri alıp, bir şekilde  $y_{\text{hat}}$ 'ile map edecek çıktıyı üretmek. Yani 2. Layerdan sonrası aslında  $a_{21}$   $a_{22}$   $a_{23}$  ve  $a_{24}$ 'ü input olarak alıyor ve  $y_{\text{hat}}$ 'e en yakın sonucu üretecek parametreleri öğrenmeye çalışıyor.

Ancak 2. Layerdan önceki parametreler yani  $w_1$   $b_1$  ve  $w_2$   $b_2$  değiştikçe 2. Layer'ın çıkışları  $a_{21}$   $a_{22}$   $a_{23}$  ve  $a_{24}$  de değişir bu yüzden bu a değerleri sürekli değişir ve az önce bahsedilen covariance shift probleminden muzdariptirler, yani 2. Layerdan önceki parametreler her değiştiğinde  $a_2$  o kadar değişir ki, 3. Ve 4. Layer parametrelerinin geçmişte öğrendikleri önemini yitirmeye başlar, baştan öğrenmesi gerekir.

İşte batch norm tam olarak bu etkiyi azaltır, önceki parametreler değişse de  $a_2$  nin mean ve variance'ı yani dağılım parametreleri aşağı yukarı aynı kalır. Sonuçta böylece bir layer ondan önceki layer parametrelerinin değişimlerinden daha az etkilenir ve kendi içinde öğrenmeye odaklanır bu da overall learning'i hızlandırıyor.

## Why this is a problem with neural networks?



Andrew Ng

What it does is: it limits the amount to which updating the parameters in the earlier layers can affect the distribution of values that the 3rd layer now sees therefore that has to learn on.

Yani batch norm ile  $a$ 'nın dağılımı daha stable oluyor ve, önceki parametrelerin değişiminden daha az etkileniyorlar.

Bunların yanında batch norm'un az da olsa bir regularization etkisi de var. Ancak batch norm'un kullanım amacı regularization değildir.

# Batch Norm as regularization

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values  $z^{[l]}$  within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

mini-batch : 64

512

Andrew Ng

Batch norm ile her mini batch için ayrı bir mean ve variance hesaplanıyor, bu yüzden bu mean ve variance değerleri ztilde'ya bir noise ekler, bu noise da aslında dropout'a benzer bir etki yapar ve az da olsa bir regularization sağlar.

Minibatch size'in büyük olması bu etkiyi düşürür.

Son olarak batch norm konusunu kapatmadan önce, prediction yaparken batch norm'un nasıl uygulandığını göreceğiz. Çünkü training sırasında batch norm her minibatch için ayrı bir ztilde hesaplıyor ve bir sonraki katmana öyle geçiyor, ancak prediction için bir minibatch söz konusu değil.

## Vid 7

### Batch Norm at Test Time

Training sırasında her minibatch için batch normalization uyguladığımızı biliyoruz. Ancak test sürecinde yani prediction için example'lar tek tek hesaplanıyor bu durumda network'ü nasıl adapt edeceğiz?

Aşağıda solda training süresince nasıl batch norm uyguladığımızı hatırlıyoruz, burada m bir minibatch size olarak kullanılmış. Minibatch için mean ve variance bulduktan sonra, normalizasyon ile ztilde elde ediliyor.

Peki ya prediction için bu normalizasyon nasıl yapılacak? Elimizde bir Mean ve Variance değeri olması gerekiyor, çünkü tek bir example için bunu hesaplayamayız.

Yapılan şey şu: training sırasında her minibatch için ayrı ayrı bulunan mean ve variance'ların exponentially weighted average'ı alınır ve eğitim sonunda elimizde tek bir mean ve variance değeri oluşur, bu değeri prediction sürecinde batch norm uygulamak için kullanırız.



## Batch Norm at test time

Handwritten notes and formulas for Batch Normalization at test time:

Left side (Formulas):

- $\mu = \frac{1}{m} \sum_i z^{(i)}$
- $\sigma^2 = \frac{1}{m} \sum_i (z^{(i)} - \mu)^2$
- $z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$
- $\tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta$

Right side (Diagram and Notes):

$\mu, \sigma^2$ : estimate using exponentially weighted average (across mini-batches).

$X^{(1)}, X^{(2)}, X^{(3)}, \dots$

Arrows point from  $X^{(1)}, X^{(2)}, X^{(3)}$  to  $\mu^{(1)}, \mu^{(2)}, \mu^{(3)}$  and  $\sigma^{(1)}, \sigma^{(2)}, \sigma^{(3)}$ .

Below these,  $\theta_1, \theta_2, \theta_3$  are shown, with arrows pointing to  $\mu$  and  $\sigma^2$ .

Bottom right:  $\tilde{z} = \gamma z_{\text{norm}} + \beta$

Andrew Ng

Yukarıda görüldüğü gibi her batch için ayrı bir mean ve variance bulunur, bunların ewa ile ortalaması tutularak en sonunda tek bir mean ve variance average değeri elde edilir, prediction sürecinde bu verilen kullanılır, gama ve beta ise zaten model tarafından öğrenilmiş olacaktır.

Tabi bu mean ve variance, tüm training set kullanılarak da hesaplanabilir, ancak bunun yerine pratikte eğitim süresince ewa ile bu değere yakın bir average tutulur.

## Vid 8

### Softmax Regression

Şimdiye kadar bahsedilen classification problemleri hep binary classification yapıyordu. Logistic regression'ın generalized versiyonu olan softmax regression'ı 2 den fazla class'lı classification problemler için kullanabiliyoruz.

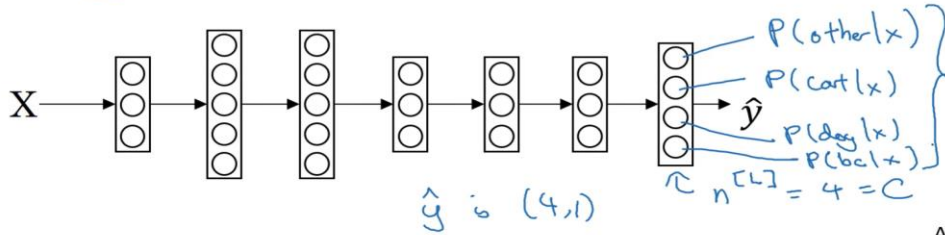
Örneğin aşağıdaki gibi bir classification örneğimiz olsun, 4 class'ımız var kedi köpek civciv ve other. Bunun için output layerda 4 neuron var, ve her biri bir olasılığa karşılık geliyor, ayrıca bu olasılıkların toplamı da 1 olmalı. Bunu sağlamak için output layer için softmax layer kullanılır.

# Recognizing cats, dogs, and baby chicks , other



3      1      2      0      3      2      0      1

$C = \#classes = 4$  (0, ..., 3)



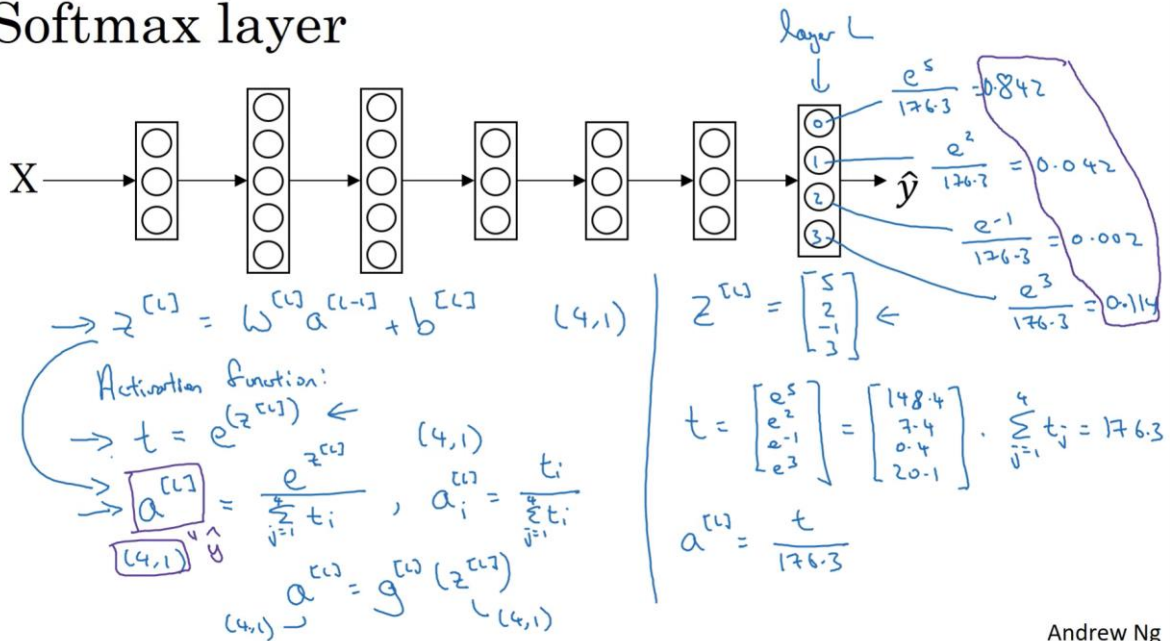
Andrew Ng

Peki bu layer'ın activation'ı nedir biraz buna bakalım? Temelde bu layera kadar her şey bildiğimiz gibi, bu softmax layer için  $z$  alınır ve  $g(z)$  hesaplanır bu hesaplamamızın diğer aktivasyon fonksiyonlarından farkı, tüm layer boyunca yapılması, yani layerdaki bir unit'in sonucu hesaplanırken diğer unitleri de hesaba katar. Ancak mesela sigmoid için input bir real number olur, buna karşılık bir başka real number output verilir.

İşleme bakarsak, diyelim ki sağ alttaki gibi bir  $z$  vektörümüz elde edilmiş, bu durumda  $g(z)$ 'ye geçmek için önce  $e^z$  yani her eleman için exponential alınır,  $t$  elde edilir. Daha sonra elde edilen  $t$  vektörünün eleman sum'ı bulunur.

Son olarak her eleman bu sum'a bölünür, böylece olasılık değerleri elde edilmiş olur.

## Softmax layer



Andrew Ng



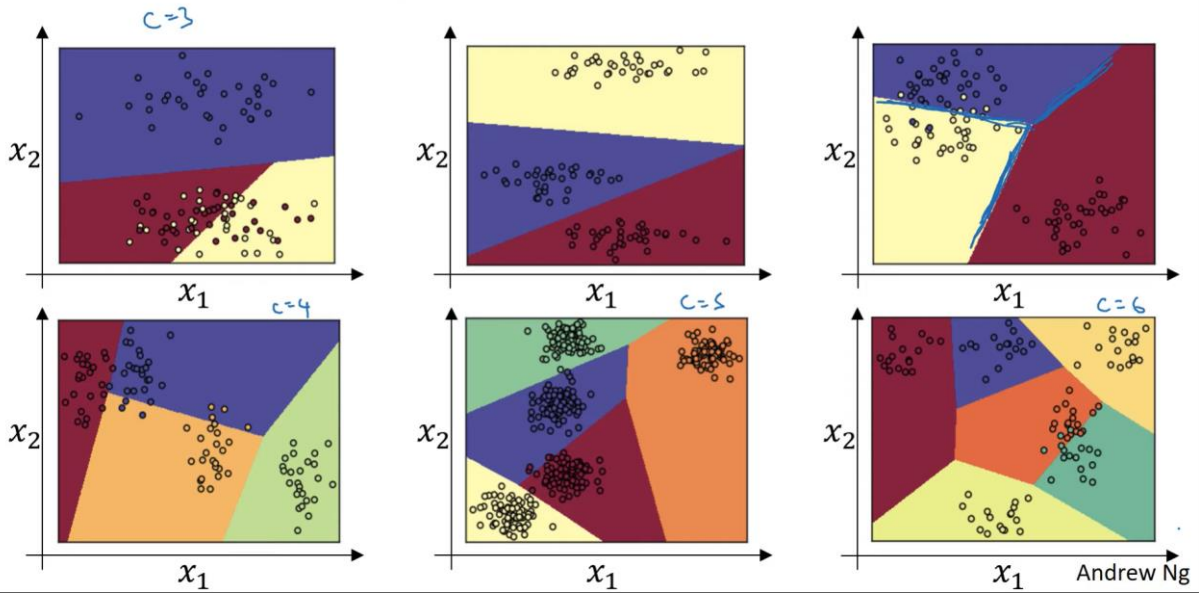
Şimdi sadece tek bir softmax layer ile nasıl classificationlar yapılabileceğine bakalım, aşağıda görülen örnekler için 2 tane input feature'ımız var ve bir softmax layer  $c=3$  veya daha fazla class için ayırım yapıyor.

Temelde mantığı logistic regression'a benziyor, logistic regression ile  $g(z)$  için eğer  $z=0$  yani  $wx+b=0$  bir doğru denklemini temsil ediyor, bu doğrunun bir tarafı için  $g(z)>0.5$  verirken diğer tarafı için  $<0.5$  veriyordu, bu doğruyu oturtmak demek, modeli eğitmek demektir.

Burada da aslında  $c$  farklı log. Reg. varmış gibi düşün, ilk örnekler için 3 farklı unit var yani 3 farklı seti ayırmaya çalışıyor.

## Softmax examples

$$\begin{matrix} x_1 \\ x_2 \end{matrix} \rightarrow \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \rightarrow \hat{y}$$
$$z^{(i)} = W^{(i)}x + b^{(i)}$$
$$a^{(i)} = \hat{y} = g(z^{(i)})$$



Elbette eğer, sadece softmax kullanmak yerine daha öncesinde nonlinear activationlı layerlar kullanılırsa daha kompleks decision boundaries de ortaya çıkar.

## Vid 9

### Training a Softmax Classifier

Aşağıda softmax'ın hesaplaması tekrar gösterilmiş diyelim ki  $c=4$  yani 4 class'ı ayırmaya çalışıyoruz ve zı aşağıdaki gibi bulunmuş bu durumda softmax yine aşağıdaki gibi uygulanır ve her class için olasılıklar bulunur.

Burada önemli bir nokta şu: softmax dediğimiz logistic regression'ın genelleştirilmiş halidir, yani eğer 2 class varsa softmax kullanmak ile logistic regression kullanmak aynı kapıya çıkar.

Softmax ismi de hardmax isminden türemiştir, hardmax denilen durumda,  $z$  vektöründe en büyük olan yere 1 yazılır diğerleri 0 alınır ve böylece activation elde edilmiş olur, softmax bunun softu.

# Understanding softmax

$(4,1)$

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$$

$C=4$   $g^{(u)}(\cdot)$

"soft max"

$$a^{[u]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5 + e^2 + e^{-1} + e^3) \\ e^2/(e^5 + e^2 + e^{-1} + e^3) \\ e^{-1}/(e^5 + e^2 + e^{-1} + e^3) \\ e^3/(e^5 + e^2 + e^{-1} + e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

"hard max"

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Softmax regression generalizes logistic regression to  $C$  classes.

If  $C=2$ , softmax reduces to logistic regression.  $a^{[u]} = \begin{bmatrix} 0.842 \\ -0.158 \end{bmatrix}$

Andrew Ng

Şimdi softmax layer'ı olan bir NN için loss function'ın nasıl tanımlandığına bakalım:

## Loss function

$(4,1)$

$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$  - cat  $y_2=1$

$(4,1)$

$\hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$   $C=4$

$y_1 = y_3 = y_4 = 0$

$\mathcal{L}(\hat{y}, y) = - \sum_{j=1}^4 y_j \log \hat{y}_j$

$-y_2 \log \hat{y}_2 = -\log \hat{y}_2$  Make  $\hat{y}_2$  big.

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

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

$(4, m)$   $(4, m)$

$= \begin{bmatrix} 0 & 0 & 1 & 0 & \dots \\ 1 & 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \end{bmatrix}$   $= \begin{bmatrix} 0.3 & \dots \\ 0.2 & \dots \\ 0.1 & \dots \\ 0.4 & \dots \end{bmatrix}$

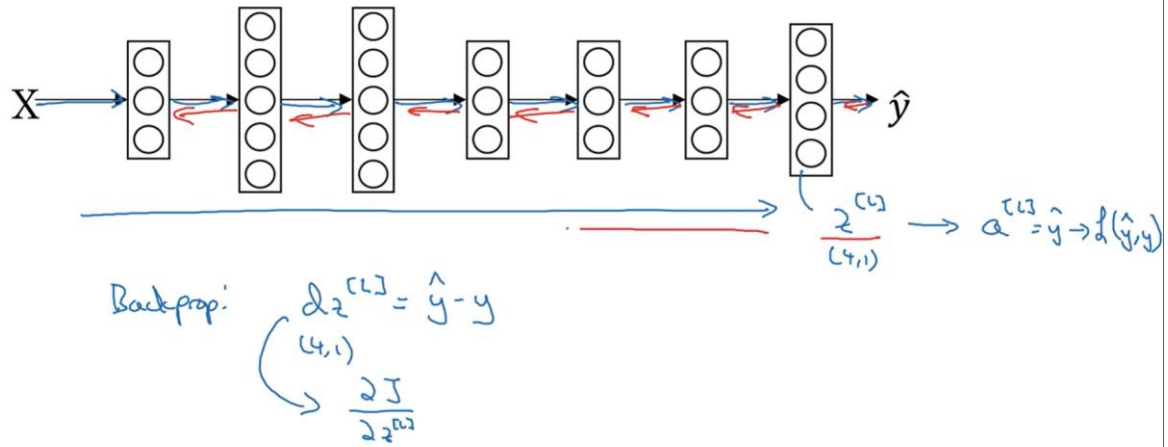
Andrew Ng

Tek bir example için loss yukarıda görüldüğü gibi tanımlı aslında bu şuna denk geliyor: Diyelim ki 2. Unit 1 yani example kedi, fakat prediction kedi olasılığına 0.2 vermiş, o halde  $-\log(0.2)$  cost geliyor,  $-\log$ 'un içi 1'e yaklaştıkça loss 0'a yaklaşır, içi 0'a yaklaştıkça loss büyür.

Cost ise tahmin edildiği gibi ortalama loss olarak hesaplanıyor, vektörizasyon yapılırsa  $Y$  ve  $\hat{Y}$  yukarıdaki gibi olacaktır.

Son olarak backprop denkleminde bahsedelim:

## Gradient descent with softmax



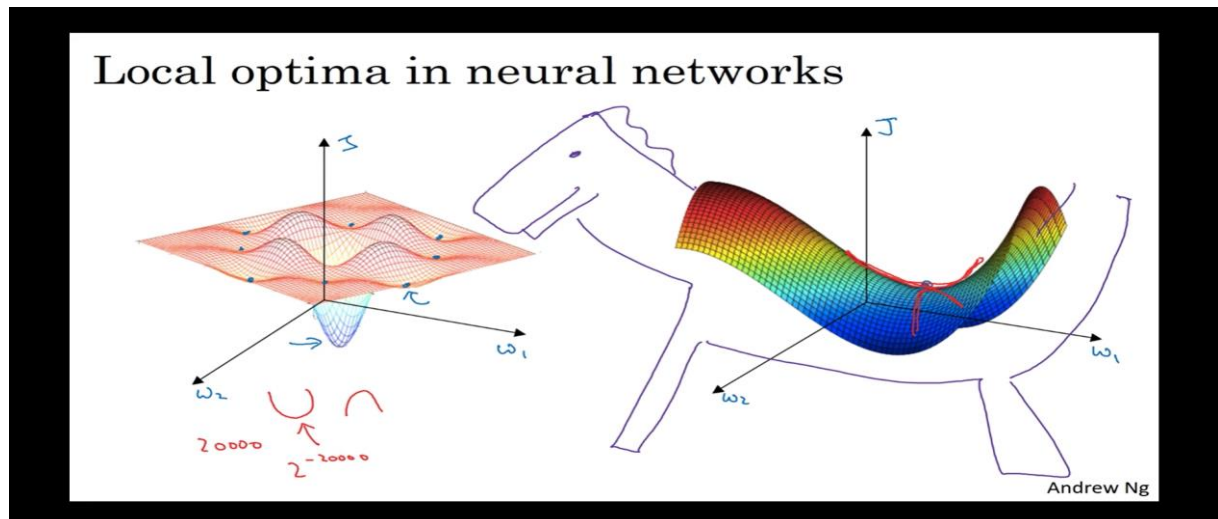
Andrew Ng

Ancak tensorflow gibi bir framework kullanılacağı için, sadece forward prop'u implemant etmek yeterli oluyor, framework backpropagation kısmını kendi hallediyor.

## Vid 10

### Problem of Local Optima

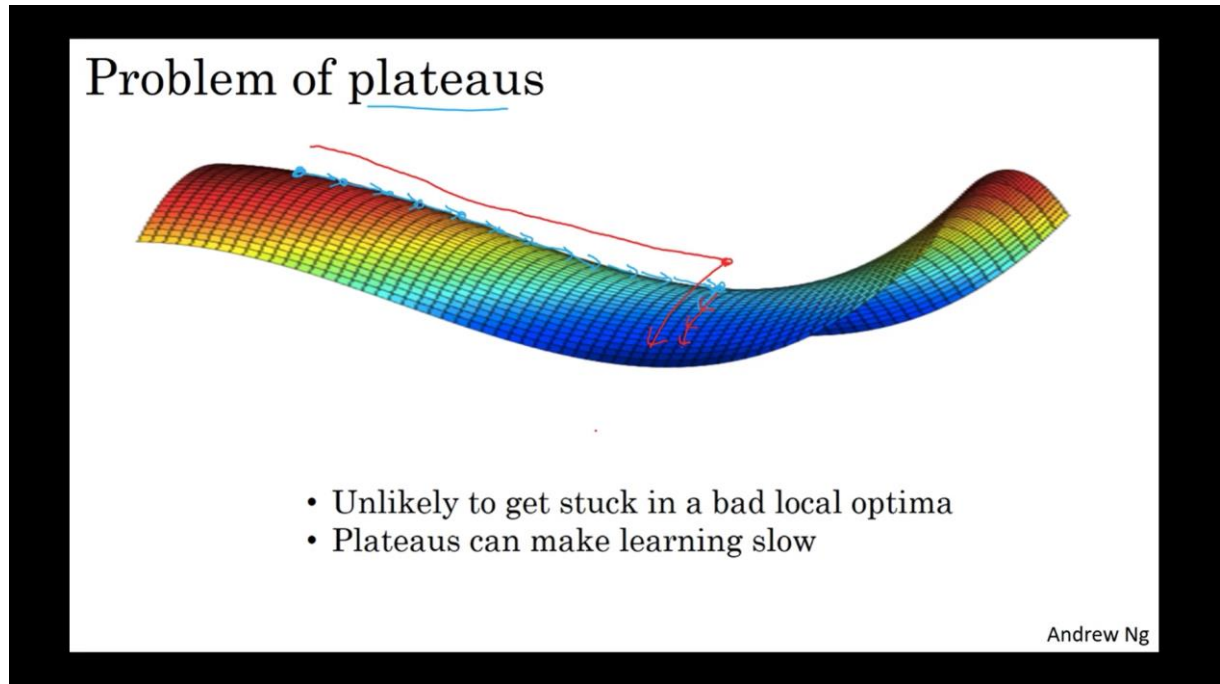
Eskiden insanlar deep NN için optimizasyon yaparken, kötü bir local optima'da takılmaktan çok korkarlardı. Ancak zamanla local optima anlayışımız da değişiyor.



Eskiden local optimada takılmak yüksek olasılık gibi düşünülüyordu çünkü, cost function dediğimizde aklımıza sol üstteki gibi bir grafik geliyordu.

Ancak 2 boyutta elde edilen bu intuition aslında pek de doğru değildir. Yüksek boyutlar düşünüldüğünde, gradient'in 0 olduğu bu noktaların birçoğu aslında global optima değil de saddle pointtir.

Saddle eyer demek, yani burada şunu açıklamaya çalışıyor, eğer ki çok dimensionlı bir cost function söz konusu ise ve gradients=0 çıkıyorsa bu büyük ihtimalle global optimadır, çünkü N dimension'ın hepsi düşünüldüğünde convex veya concave bir fonksiyon oluşması düşük bir olasılık, global optima dışındakiler genelde sağdaki gibi saddle point olur yani bir ekseninde concave ise diğerinde convex olur hepsi aynı olmaz, zaten bu noktalarda da tıkanmayız.



Local optima'lar problem olmasa da plato'lar problem çıkarır, plato dediğimiz yerler gradients'in 0'a yakın olduğu yerlerdir bu yerlerde learning algorithm çok küçük adımlar atabildiği için eğitim süresi uzar.

Bu nedenle Momentum, RMSProp veya Adam gibi optimizasyon algoritmaları önem arzeder.

# Vid 11

## Tensorflow

Birsürü DL programming frameworks var, bunlardan biri TensorFlow. Bu kısımda basic structure'ı öğreneceğiz ve bu haftanın excercise'ı ile pekiştireceğiz.

Motivating example olarak minimize etmemiz gereken bir cost function  $J$ 'imiz olduğunu düşünelim. Bu cost function aşağıdaki gibi tanımlı olsun,  $w=5$  değeri için minimize oluyor.

### Motivating problem

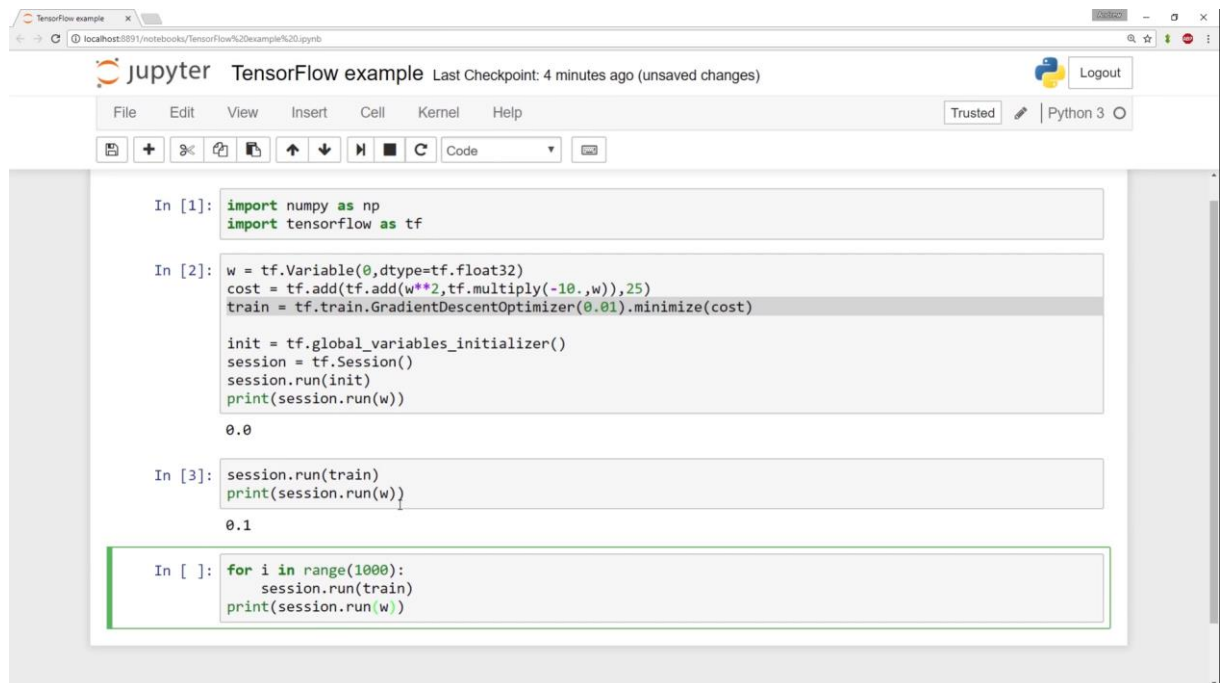
$$J(w) = w^2 - 10w + 25$$

$(w-5)^2$   
 $w=5$

$$J(w, b)$$

Andrew Ng

Böyle bir cost function'ı tensorflow ile nasıl minimize edeceğimize bakacağız, bunu yapabilirsek, gerçek bir NN için elde edilen çok daha karmaşık bir  $J(W,b)$ 'yi de nasıl minimize edeceğimizi anlamış oluruz.



```
In [1]: import numpy as np
import tensorflow as tf

In [2]: w = tf.Variable(0, dtype=tf.float32)
cost = tf.add(tf.add(w**2, tf.multiply(-10., w)), 25)
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

0.0

In [3]: session.run(train)
print(session.run(w))

0.1

In [ ]: for i in range(1000):
    session.run(train)
    print(session.run(w))
```

- Öncelikle  $J(w)$ 'yi minimize etmek için gerekli variable'ı yani optimizasyon parametresini tanımlıyoruz.

- Daha sonra cost function'ı tanımlıyoruz. Bu fonksiyonu aynı np kütüphanesini kullanarak gibi yukarıdaki şekilde de tanımlayabiliriz, veya direkt  $cost = w^2 - 10w + 25$  şeklinde de tanımlamak mümkün.
- Train içinde hangi algoritma ile hangi fonksiyonu minimize edeceğimizi belirtiyoruz.
- Bundan sonraki 3 line are quite idiomatic. Burada bir tensorflow session başlatıyoruz, ancak henüz bir optimizasyon yapılmadı, bu yüzden session.run(w) dediğimizde w ilk halinde duruyor yani 0.
- session.run(train) dediğimizde gradient descent bir adım için çalışıyor, bu yüzden tekrar session.run(w) ile w'yi kontrol edersek cost'u minimize edecek olan değere yani 5'e bir adım yaklaştığını görüyoruz.
- Şimdi 1000 gradient descent'le 1000 adım atalım, bunun için bir for loop içinde aynı kodu kullanıyoruz. 1000 gradient step sonucunda  $w = 4.99999$  olarak bulunacaktır.

Tensorflow'un güzelliği optimize edeceğimiz cost function'ı tanımlamamız gradient descent'in çalışması için yetiyor, gradients'i tensorflow kendi hesaplıyor, normalde derivative'ini de bizim tanımlamamız gerekiyordu. Örneğin matlabda advanced learning algorithm kullanıyorduk orada daha hem J'yi hem de dJ'yi tanımlıyorduk ki optimizasyon algoritması çalışsın.

Yani forwardpropagation işlemini biz tanımlarsak backpropagation işlemini (ki bu işlem dW, db gibi optimizasyon parametrelerinin türevlerini bulmaktan başka bir işe yaramıyordu) tensorflow kendi hallediyor ki bu bulunmaz nimet.

Yukarıdaki örnek için J(w)'nin sonucu sadece w'ye bağlı, bunu etkileyen başka hiçbir faktör yok ve J'yi sadece w'yi kullanarak minimize ediyoruz. NN problemlerinde ise, yine J'yi w variable'ı ile minimize etsek de aslında J sadece w'nin değil bunun yanında x'in yani dataset'in de bir fonksiyonu. Elbette biz dataset'i değiştiremeyeceğimiz için minimizasyon işlemini sadece w'yi kullanarak yaparız, yine de şunu unutma aslında hiçbir problem için aynı cost function kullanılmıyor biz aynı cost function kullanılıyor dediğimizde x'i parametrik aldığımız için öyle diyoruz, numerik sonuçlara baktığımızda aslında x her problem için farklı bir dataset.

Şimdi, J'nin sadece w'ye değil x'e de bağlı olduğu bir durum için nasıl optimizasyon yaparız ona bakalım:

```
In [1]: import numpy as np
import tensorflow as tf

In [8]: coefficients = np.array([[1.], [-10.], [25.]])

w = tf.Variable(0, dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
#cost = tf.add(tf.add(w**2, tf.multiply(-10.,w)),25)
#cost = w**2 - 10*w + 25
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

0.0

In [6]: session.run(train, feed_dict={x:coefficients})
print(session.run(w))

0.1

In [7]: for i in range(1000):
```

- Coefficients içinde yukarıdaki gibi tanımlı dataset'imiz olduğunu varsayalım elbetteki bu değişken.
- Bunun için öncelikle placeholder olarak bir x tanımlıyoruz, bu şu demek [3,1] boyutunda bir x vektörü daha sonra sağlanacak. Böyle yaptığımız zaman aynı formata başka coefficients da besleyebileceğiz.
- Elbette cost function da artık içinde x içeriyor, normalde y de içerir bu yüzden y'yi de placeholder olarak besleriz.
- Idiomatic kısım aynı kalıyor, gradient step atan kodun içine x'i dde yukarıdaki gibi besleriz daha sonra eskisi gibi loop içinde gradient step attirabiliriz. Şuan x'in parametreleri bir önceki problemle aynı seçildiği için sonucun yine 5 çıkmasını bekliyoruz, eğer x değişirse sonuçta değişecektir.
- Eğer minibatch kullanacaksak, session.run(train,dict=...) kısmında her seferinde başka coefficients (training setden minibatch'ler) beslenir.

Dediğimiz gibi tensorflow'un gücü sadece cost function'ı tanımlayarak optimizasyon işlemi yapabilmemizde yatıyor, optimizasyon için ihtiyaç duyduğu derivatives'i kendi hesaplıyor.

Son olarak aşağıda farklı kodun nispeten temiz halini görüyoruz. Aşağıda idiomatic olan 3 line koda alternatif bir koda tanımlanmış, bazen sağdaki kullanılır, temelde aynı şey, sağdaki error handling konusunda daha başarılı.

Tensorflow'un yaptığı şeye gelirsek aslında cost'u tanımlarken, bir computation graph kullanıyor, ve forward propagation hesaplarken backward'ı da computation graph ile hesaplıyor, böylece derivatives'i elde etmiş oluyor ve daha sonra optimizasyonu yapmak zaten iş değil.

Son olarak eğer tensorflow documentation'a bakarsan, sağ alttaki gibi değil de onun altındaki gibi bir computation graph görmek mümkün sonuçta mantık aynı derivatives'i kendi hesaplar.



# Code example

```
import numpy as np
import tensorflow as tf
```

```
coefficients = np.array([[1], [-20], [25]])
```

```
w = tf.Variable([0], dtype=tf.float32)
```

```
x = tf.placeholder(tf.float32, [3,1])
```

```
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0] # (w-5)**2
```

```
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
```

```
init = tf.global_variables_initializer()
```

```
session = tf.Session()
```

```
session.run(init)
```

```
print(session.run(w))
```

```
with tf.Session() as session:
```

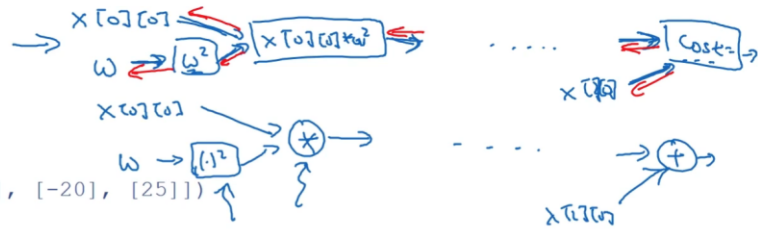
```
    session.run(init)
```

```
    print(session.run(w))
```

```
for i in range(1000):
```

```
    session.run(train, feed_dict={x:coefficients})
```

```
print(session.run(w))
```



Andrew Ng