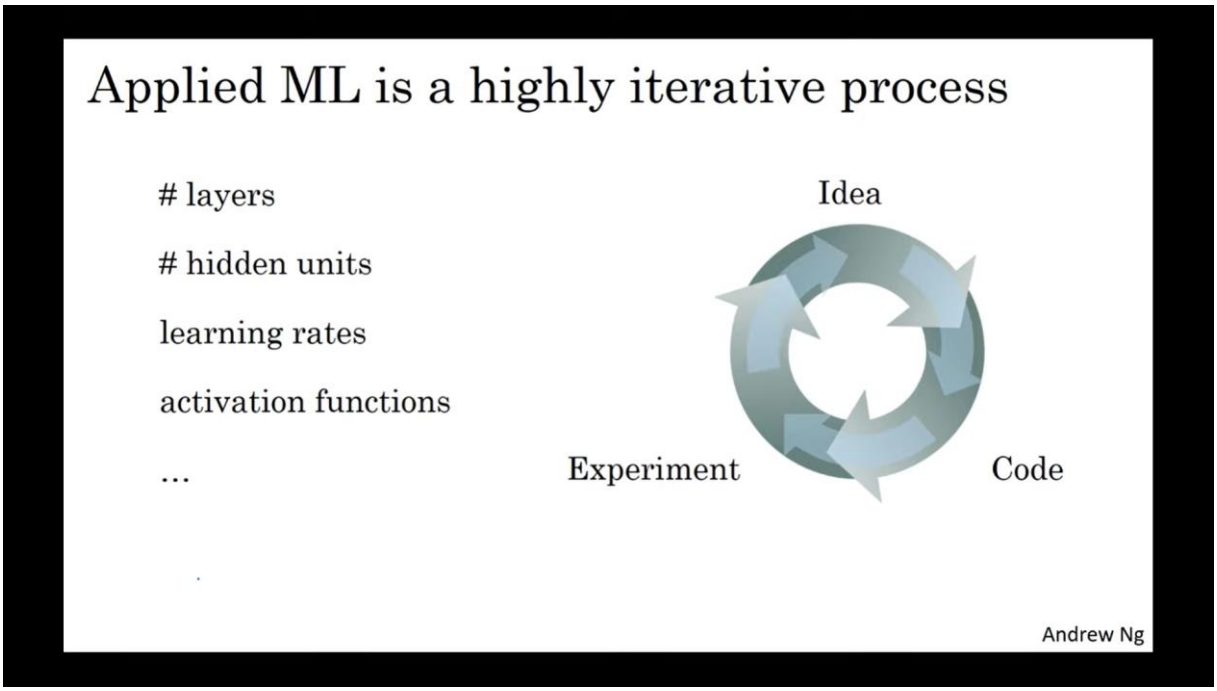
COURSE 2 W1

Vid 1

Train/Dev/Test Sets

2. course boyunca NN'imizin daha iyi çalışması için gerekli olan pratik bilgileri edineceğiz. Hyperparameter tuning, how to set of our data, how to make sure that our optimization algorithm runs quickly.

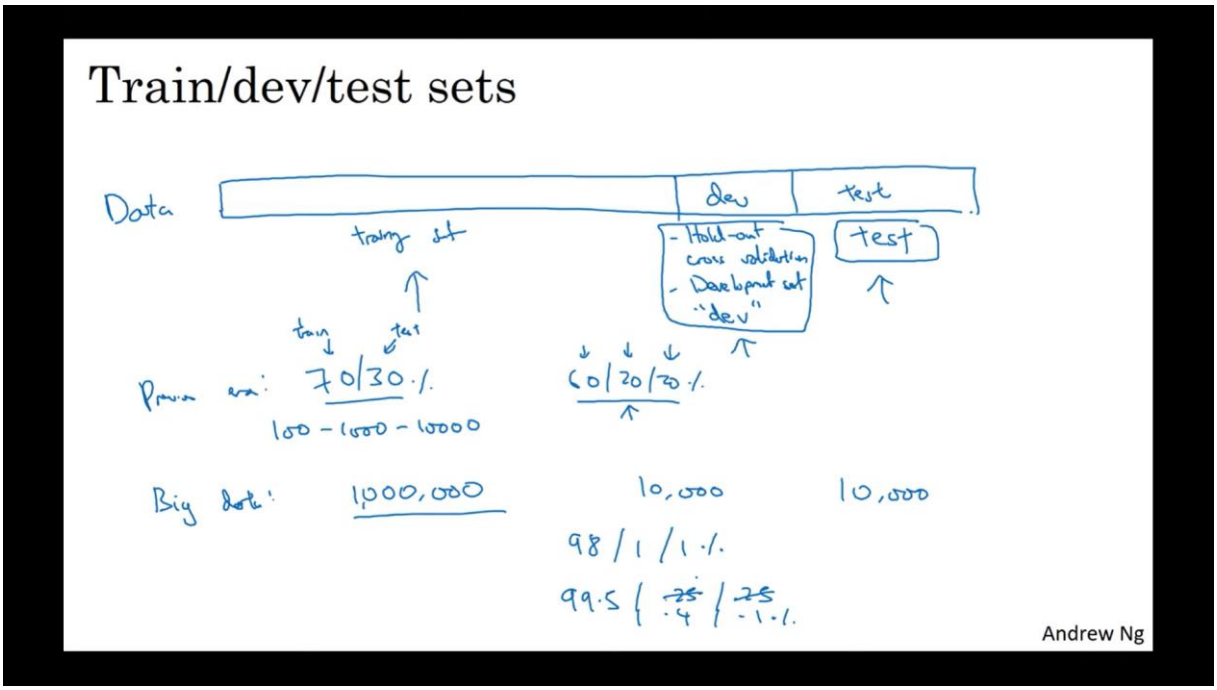
Bir model eğitirken layer sayısı, hidden unit sayısı, learning rate, activation function seçimi gibi bir sürü hyperparameter vardır, bunları direkt doğru seçmenin bir yolu yok, iteratif bir yaklaşım uygulamalıyız.



Sonuçta bizim amacımız yukarıdaki cycle'ı nasıl hızlı ve verimli bir şekilde ilerletebiliriz onu anlamak.

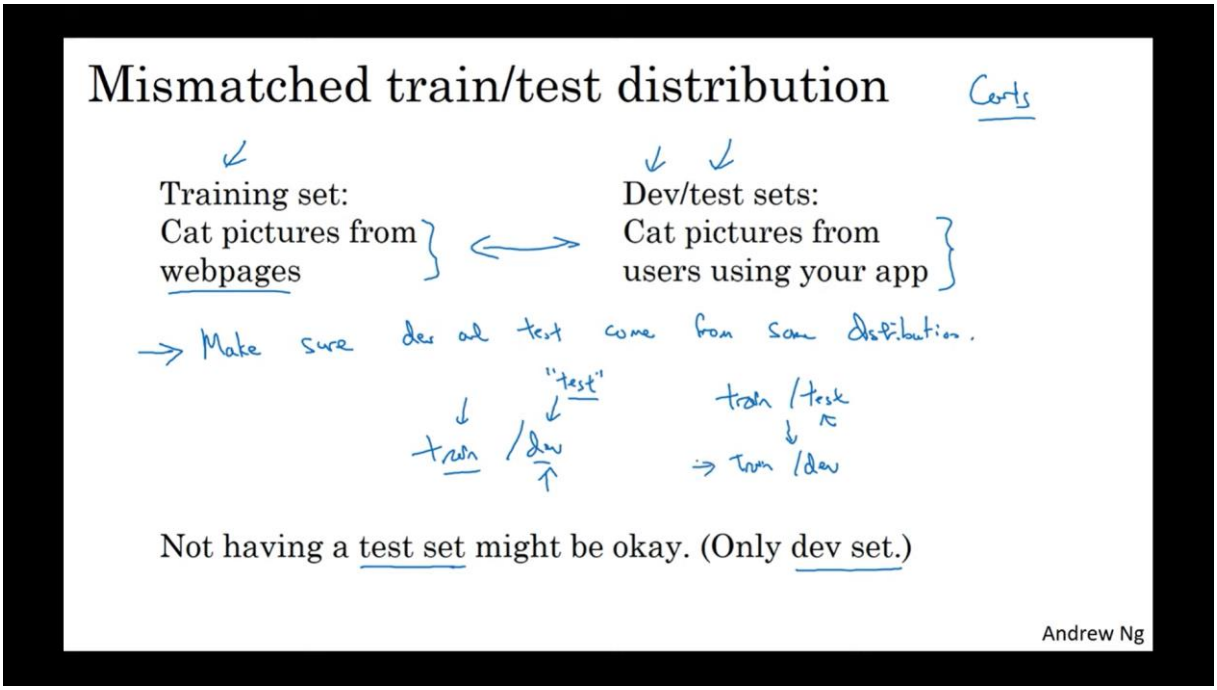
Datamızı iyi bir şekilde train development ve test olarak 3 sete bölmek iyi bir tercih, ve böylece bu iteratif süreci verimli bir şekilde tamamlayabiliriz.

Sonuçta datayı train, dev ve test olarak üçe ayırırız, train set ile belirlenen hyperparametreler ile modeli eğitiriz, bu ve diğer hyperparameter için bulunan modelleri dev set ile deneriz ve en iyi hyperparameter kombinasyonunu bulmuş oluruz, son olarak elde edilen final modeli test set üzerinde deneyerek score elde edebiliriz.



5-10 yıl önce common practice daha önce gördüğümüz gibi %70'e %30 train ve test olarak ayırmak veya dev var ise %60 %20 %20 olarak ayırmaktı. Bu sistem küçük veriler için hala mantıklı yani elimizde 10 000 civarı veya daha az veri var ise bu ayırımlar mantıklı olacaktır.

Ancak new era ile şunu gördük, eğer elimizdeki datanın m=1 000 000 gibi bir rakam veya daha fazla ise, böyle büyük bir datanın %20'sini dev'e %20'sini test'e ayırmak israf, çünkü asıl olay modelin eğitilmesi yani training set. Dev'in amacı basitçe hangi hyperparametreler ile eğitilen modelin performansının daha iyi olduğuna karar vermek o kadar, test de bir score oluşturacak. E o zaman ben bunlara 400 000 datayı feda etmeyeyim, 10 000 civarı data versem hangi performansın daha iyi olduğu veya score'un ne olduğu üç aşağı beş yukarı belli olur.



Modern era'da gördüğümüz bir başka practice de şu, bir çok insan mismatcehd train ve test data üzerinde eğitim gerçekleştiriyor.

Örneğin bir app build ediyoruz ve kullanıcılar bir çok kedi fotoğrafı yüklüyorlar ve bu fotoğrafları kedi veya değil olarak ayırmak istiyoruz. Bu durumda training set'i internetten veya başka bir kaynaktan elde edilen fotoğraflar ile oluşturabiliriz, kullanıcı fotoğraflarını da dev ve test set olarak kullanırız.

Fakat bu noktada, internetten elde edilen fotoğrafların profesyonal çekilmiş high quality fotoğraflar olması, bunun yanında app fotoğraflarının casual ortamlarda çekilmiş blurry fotoğraflar. Bu fotoğraflar farklı distribution'lardan geliyor demek oluyor.

Bu noktada rule of thumb olarak dev ve test set'in aynı distribution'dan gelmesine dikkat etmeli. Ancak deep learning modelleri data'ya aç oldukları için, test set'i başka yerlerden çekmek mümkün.

Son olarak, şuna değinelim, test set olmazsa olmaz bir şey değil. Test'in tek amacı son karar verilen modelin performasını belirlemek bu bize sadece bilgi verir, test performasına göre modeli değiştirmeyiz, bunun için dev seti kullanırız. Bu durumda sadece train ve dev set ile de en iyi modeli elde edebiliriz ancak performası konusunda unbiased bir veriden yoksun kalırız.

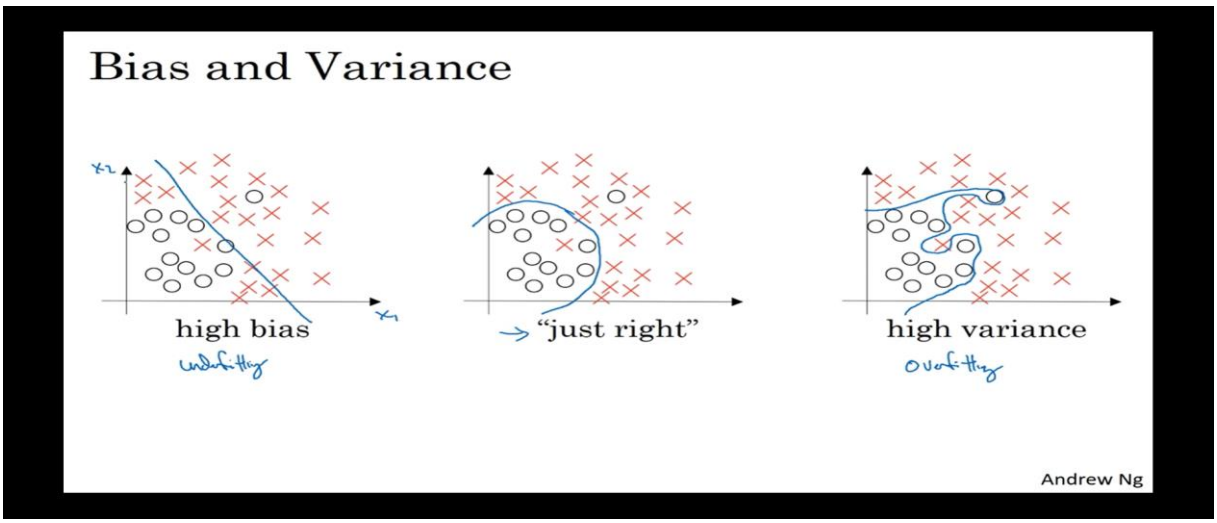
Böyle bir durumda bazı insanlar train ve dev set olarak adlandırma yapmak yerine train ve test set adlandırması yapıyorlar. Bu pek doğru bir terminology seçimi değil çünkü test set'e overfit etmiş olunuyor.

Vid 2

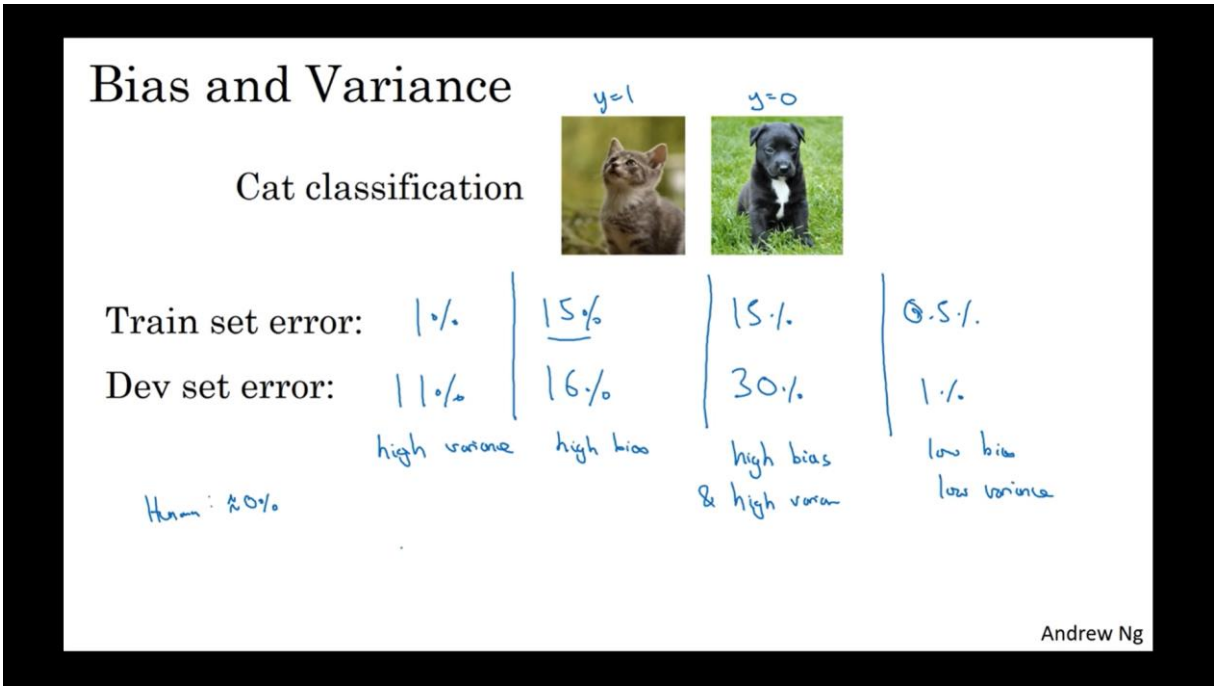
Bias and Variance

Bias ve Variance kavramlarını anlamak, iyi bir ML practioner olmak için çok önemli. They are easy to learn but difficult to master.

Deep learning ile bias variance tradeoff'undan pek bahsetmemeye başlandı.



2D örnekler için datayı ve model'i plot edip yukarıdaki gibi görseller elde edebiliriz ve böylece bias and variance'ı görselleştirebiliriz ancak daha yüksek dimension'lı örnekler için bu mümkün değil, başka metodlar kullanırız.



Burada training set error'a bakara bias hakkında karar verebiliriz, dev set error'un train'e göre durumuna bakarak da variance hakkında karar verebiliriz.

Tabii ki öncelikle elimizde beklenen bir optimal error değeri olmalı, bu durumda fotoğraftan kedi köpek ayırma işi için insan bunu hatasız yaptığı için beklenen error'a %0 diyebiliriz.

Eğer training error %1 fakat dev set %11 ise burada overfitting söz konusu, low bias high variance.

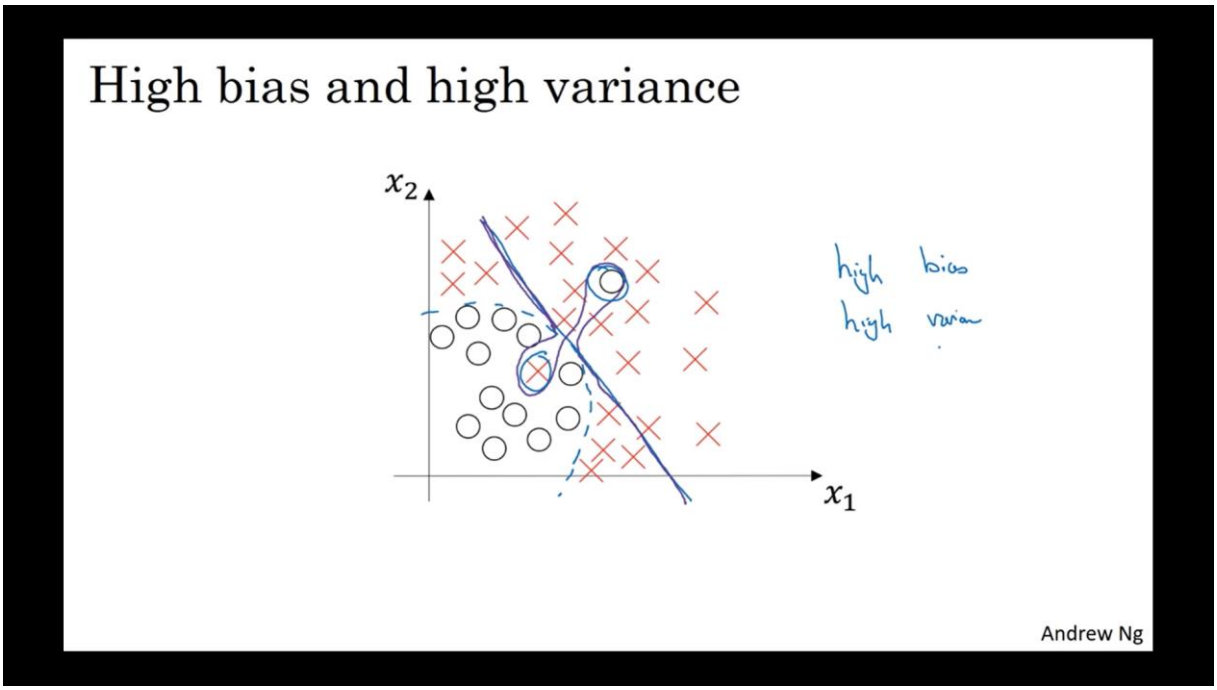
%15 - %16 durumu için training set için bile hata beklenenden çok fazla high bias low variance deriz, underfitting söz konusu.

%15-%30 durumu için yine bias kötü, high bias var, underfitting söz konusu ancak bunun yanında bir de high variance söz konusu, hem training set'e iyi oturmamış bir de training ile dev arasında da fark yaratıyor.

%0.5 - %1 durumu bu şartlar altında low bias ve low variance olarak düşünülebilir. Gayet iyi.

Dediğimiz gibi bu örnekte optimal veya bayes error %0 olarak belirlendi, eğer bir başka örnekte bu değer %15 olarak belirlenseydi o zaman %15-%16 sağlayan classifier'a high bias demeyecektik, gayet iyi diyecektik.

Son olarak 3. Classifier'ın yani hem high variance hem high bias'ın nasıl olduğunu anlamaya çalışalım. Bu durumda, hem underfitting hem overfitting söz konusu olabilecek en kötü durum heralde. Bazı bölgelerde high bias görürken bazılarında high variance görüyoruz.



Sonuç olarak train ve dev set hatalarına bakara, modelin bias ve variance'ı hakkında fikir edinebilirim, bu fikir ileriki adımlarda neler yapacağıma karar vermek için çok önemli.

Vid 3

Basic Recipe for Machine Learning

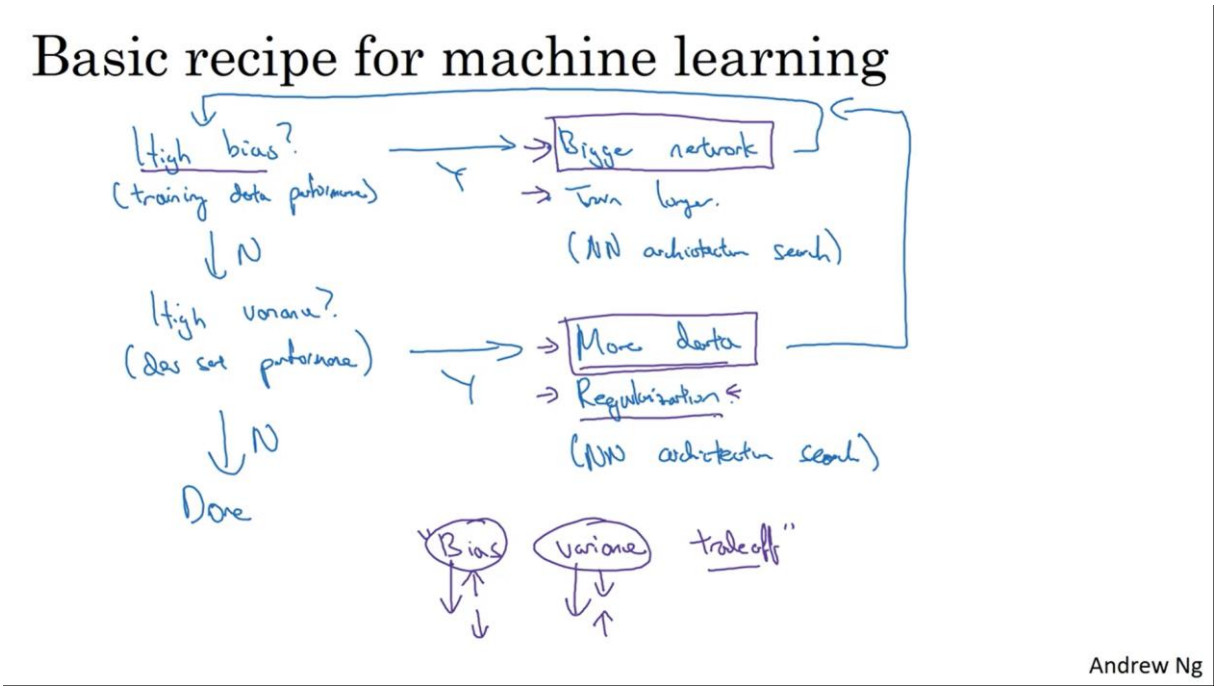
Modelimizin bias ve variance durumuna karar verdikten sonra geliştirme için temel yollar mevcut.

Diyelim ki model high bias'ten muzdarip, yani underfitting söz konusu bunu training set'e bakarak söyleriz. Bu durumda yapılabilecek şeyler: Bigger network eğitmek yani daha fazla layer ve/veya hidden unit. Bu method almost always işe yarar. Bunun yanında eğitim süresini uzatmak nadiren işe yarasa da don't hurt. Son olarak başka NN architecture denemek belki işe yarayabilir bu architecture'lardan daha sonra bahsedilecek.

Bu adımlar iteratif olarak high bias'ten kurutulana kadar deneyebilirim.

Bias'ı istenilen değerlere çektinden sonra, şunu sormalıyız modelin variance problemi var mı? Buna da dev set'e bakarak karar veririz. Eğer high variance söz konusu ise, data sayısını artırmak işe yarayabilir ama her zaman data bulamayız. Regularization işe yarar. Son olarak yine başka bir NN architecture işe yarayabilir.

Sonuçta iteratif olarak low bias ve low variance bir model elde edene kadar bu tavsiyeler uyarım.



Bir diğer nokta şu: daha önceden machine learning'de bias-variance tradeoff hakkında bir çok tartışma olurdu. Çünkü yapılanların bir çoğunda, bias artarsa variance düşüyor, variance artarsa bias düşüyordu yani arada bir tradeoff söz konusuydu.

Çünkü o zamanlarda sadece bias veya sadece variance'ı değiştirecek ve bu değişim sırasında diğerine dokunmayacak tool'larımız yoktu. Ancak deep learning için var.

Bigger network kullanımı doğru regularization kullanıldığında almost always sadece bias'ı düşürürken, More data kullanımı almost always sadece variance'ı düşürür.

Özetle artık bias ve variance'a özel olarak müdahale edebiliyoruz bu da deep learning'in önemli özelliklerinden birisi. Doğru regularization ile daha büyük bir network eğitmek almost never hurts!!! Sadece, computation yer.

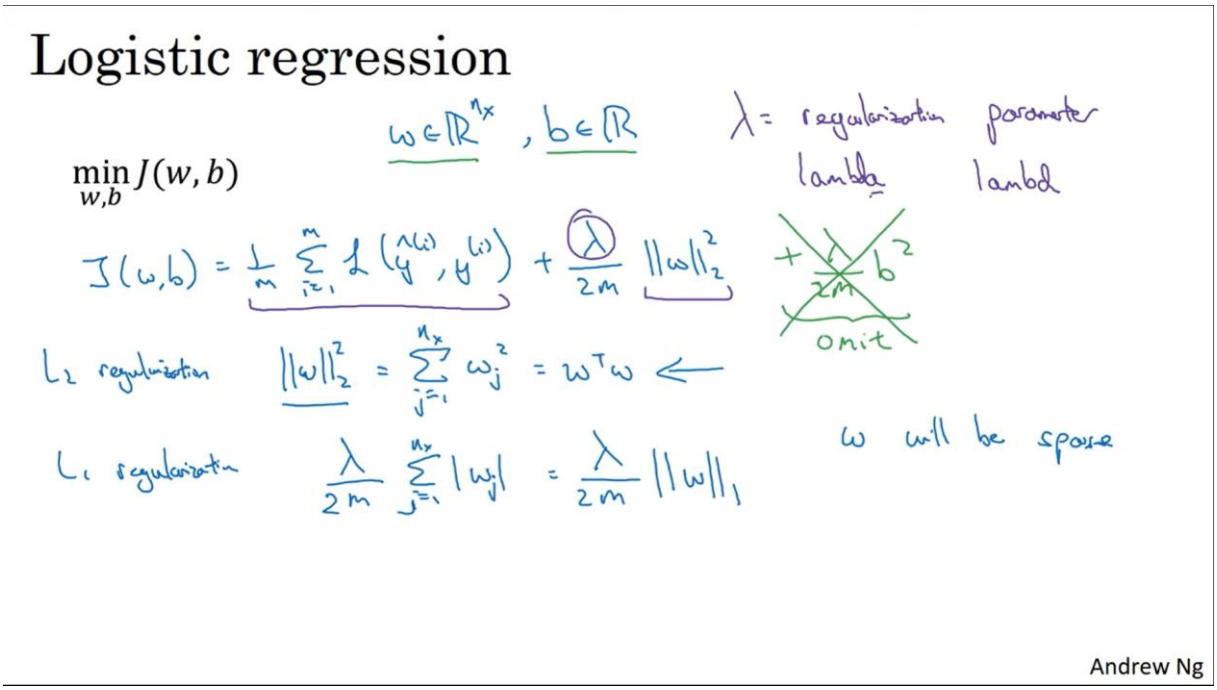
Regularization için az da olsa bias variance tradeoff söz konusudur ancak çok değil.

Vid 4

Regularization

Eğer modelimiz high variance'tan muzdarip ise yani overfitting söz konusu ise deneyeceğimiz ilk şey regularization olmalı. High variance'ı address etmenin bir diğer metodu ise daha fazla training data kullanımı.

Bu kısımda regularization'ın nasıl çalıştığını anlamaya çalışalım.

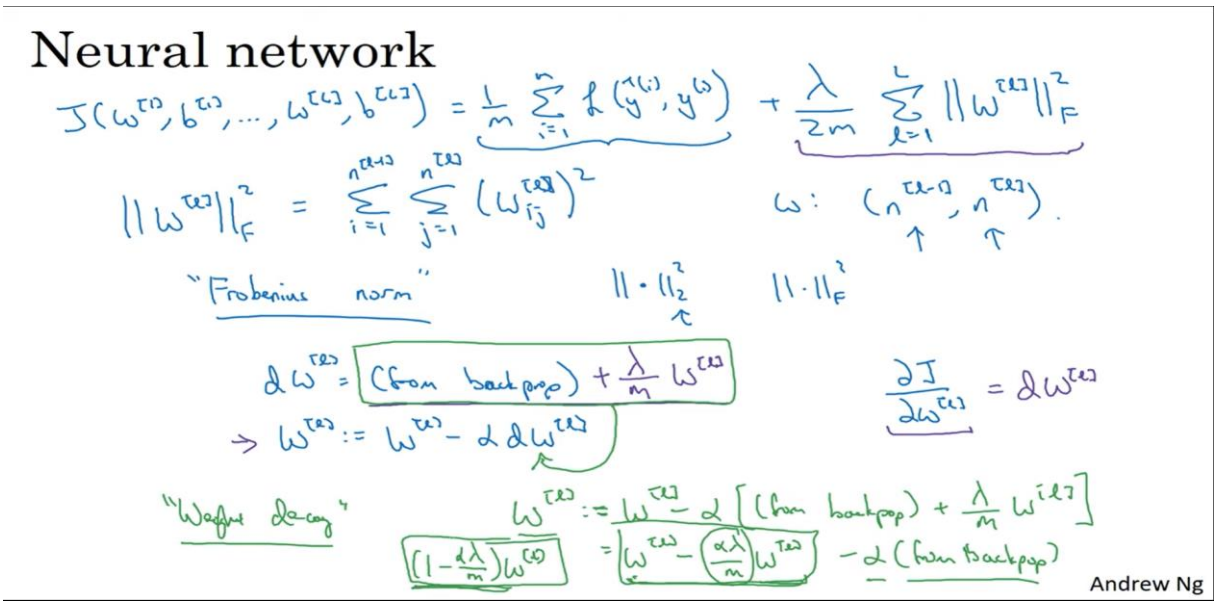


Diyelim ki bir logistic regression modelimiz var, J(w,b) daha önce bildiğimiz gibi hesaplanıyor. Eğer buna yukarıda görülen term'i eklersek k, buna L2 regularization denir o halde regularization yapmış oluyoruz. Burada b de bir parametre olduğu için bu da denkleme eklenebilir ancak, eklemesek de genelde sıkıntı olmaz çünkü genelde w içinde bir sürü parametre vardır, tek bir b parametresinin önemi kalmaz.

Burada lambda regularization parameter'dır ve overfitting ile underfitting dengesini ayarlar. Buna benzer L1 regularizationdan da bahsetmek mümkün ama L2 kullanımı daha yaygındır.

Son olarak practice hw'lerde göreceğimiz üzer python da lambda bir reserved keywordtür bu yüzden lambda yerine lambd ismi kullanılacak.

Peki bir NN için regularizaton nasıl olacak?:



NN için de durum çok benzer, J denklemine her w katsayısının karelerinin toplamını ekliyoruz, ve bunu da minimize ettiğimiz zaman elimizde regularized bir NN modeli kalıyor.

Bu matrix norm'una Frobenius norm dendiği için yanına F koyuyoruz ve öyle gösteriyoruz.

Peki bu formdaki cost function ile gradient descent nasıl oluyor?

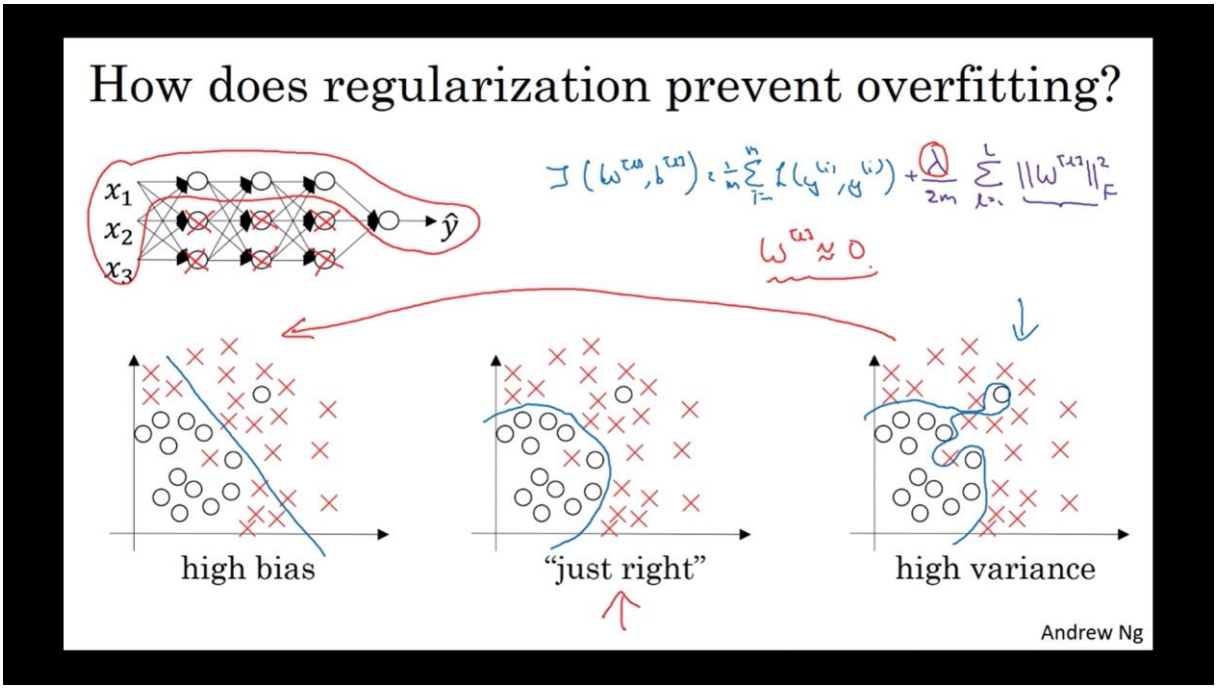
dW'in eski formuna morla gösterilen (lambda/m)*W(l) ekleniyor o kadar. Bundan sonra gradient step'i eskisi gibi atabiliriz, zaten olacağı da o sadece gradient formülü değişir.

L2 regularization'a Weight Decay de denir çünkü yeşille görülen kısımda gradient step formüşü açtığımız zaman aslında eskisine çok benzer. Tek fark önceden Wl'den bir step eksiltirken şimdi Wl-bişeybişey formundan bir step eksiltiyoruz yani sanki weight'i azaltmışız gibi bir etki yapıyor.

Özetle L2 regularization uyguladığımızda, gradient descent neredeyse aynı çalışıyor tek farklı W'den değilde 0.99 gibi lambda ile belirnen bir oranla çarpılmış W'den bir step atılıyor olması bu da overfitting'i önlemiş oluyor.

Vid 5

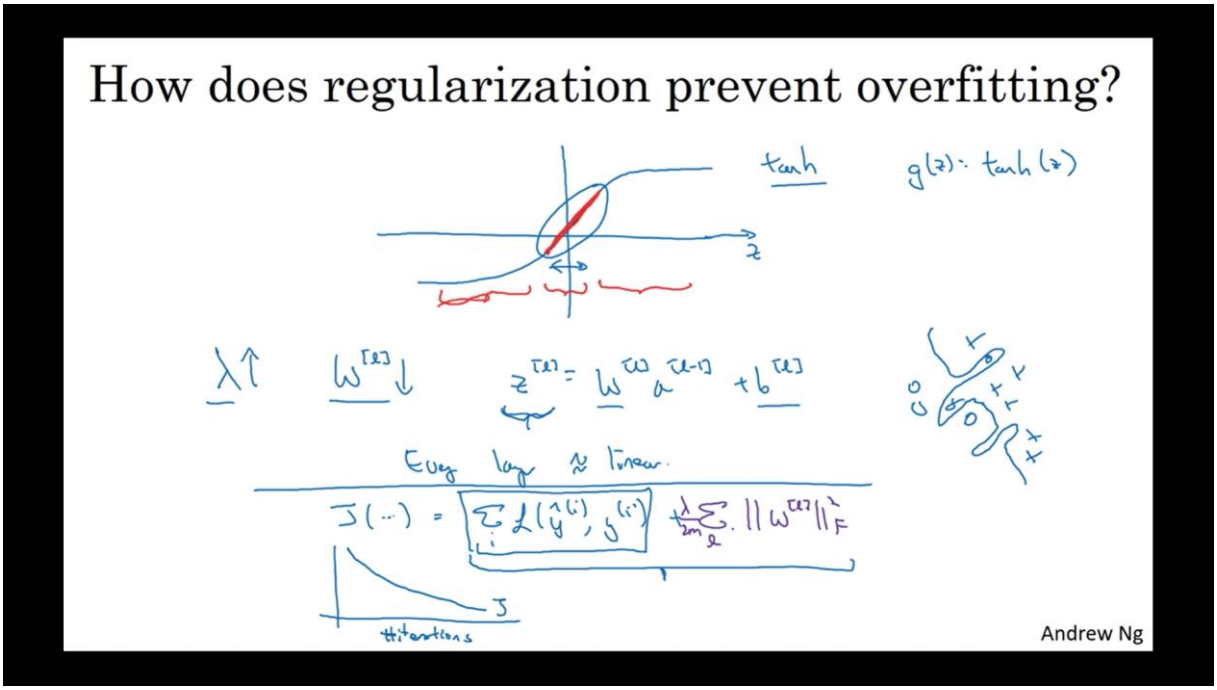
Why does regularization reduces overfitting?



Intuiton olarak şöyle düşünebiliriz, lambda çok büyürse regularized J'yi minimize ettiğimizde W parametreleri neredeyse 0 olacak bu yüzden de bizim complex NN yapımız bozulacak ve geriye daha sade bir yapı kalacak bu da bizi en sağdaki overfitting durumundan kurtaracaktır.

Gerçekte olan tam olarak bu değil, W ler 0'lanmıyor unitlerde tamamen elimine olmuyor ama effect'leri azalıyor ve bu da benzer bir etki yapıyor, yani simpler bir NN ile çalışmış oluyoruz.

Bir başka örnekle intuiton kazanmaya çalışalım:



Diyelim ki activation function'lar tanh, lambda arttı o zaman W düşecek demiştik, böyle olunca z de 0'a yakın olacaktır. Bu da şu demek elimde rougly linear activation function'lar olacak. Daha önce de dediğimiz gibi eğer tüm layer'lar linear ise sonuçta da linear bir ilişki elde ederiz yani complex feature'lar hesaplayamayız.

Bu da overfitting'den kurtulmak demektir.

Son olarak bir implementational detail şu, regularization ile J tanımı değişti. Bu yüzden her iterasyonda J'in düşmesini beklememiz doğru ancak, yanlışlıkla eski J tanımına bakarsak bu düşmek zorunda değil, buna dikkat et diyor.

Vid 6

Dropout Regularizaton

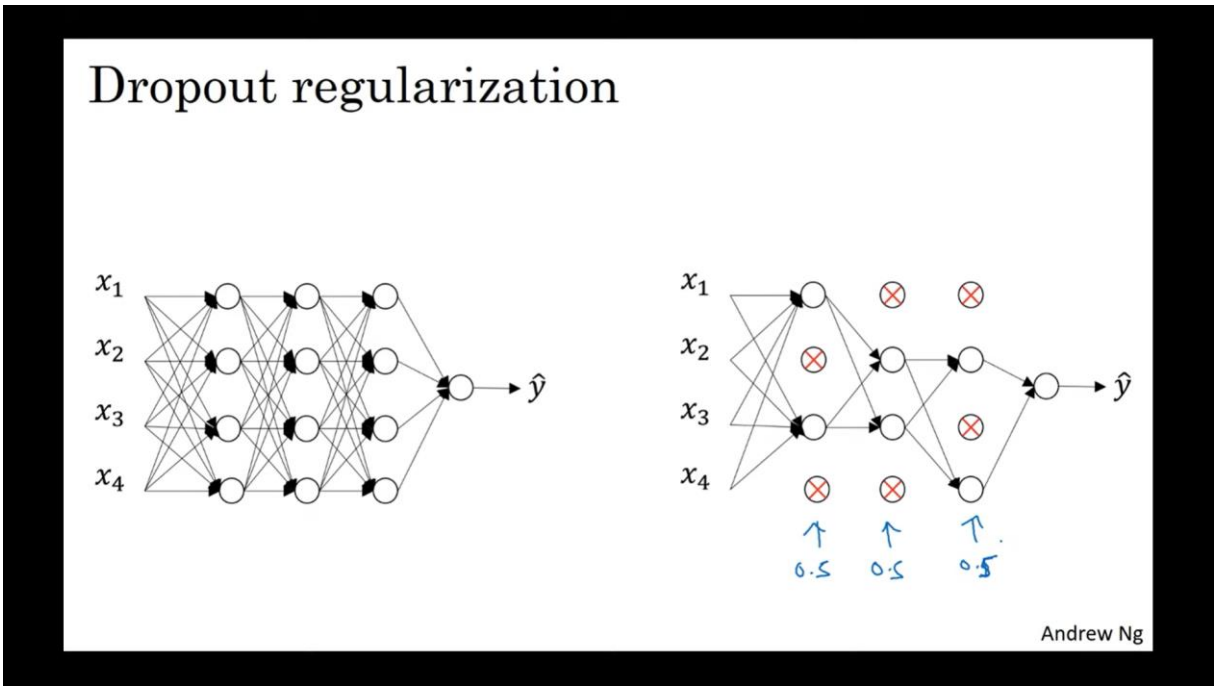
En sık L2 regularization kullanılmasına rağmen bazen dropout da kullanılır.

Diyelim ki aşağıda solda gördüğümüz gibi bir NN eğittik ve modelimiz overfitting'den muzdarip.

Dropout regularization ile yapacağımız şey şu: go through each of the layers of the network and set some probability of eliminating a node in NN. Yani bir olasılık belirliyoruz mesela %50 her layer'daki her node %50 olasılıkla kaldırılıyor. Yaklaşık %50 daha az node'dan oluşan simpler bir NN yapısı kalıyor.

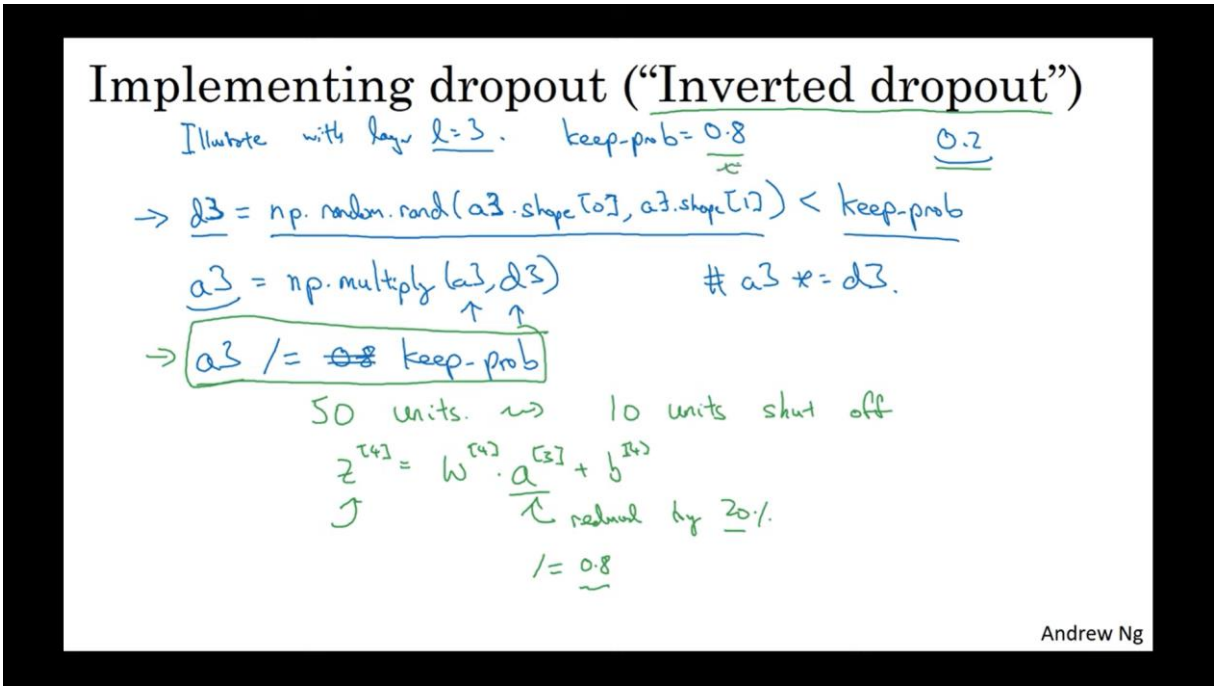
Her bir example için? veya her bir gradient step bu %50 coin toss işlemi tekrarlanır böylece her step için başka node'lar kullanılır.

Intuition olarak, simpler bir NN olduğu için regularization sağladığı düşünülebilir.



Dropout'ı ugulamanın birkaç farklı yolu var, en sık kullanılanı inverted dropout metodu, burada bundan bahsedeceğiz.

Örnepin bir NN'in 3. Layer'ı için dropout uygulayacağız diyelim.



Keep_prob adında bir hyperparameter belirlenir, buna 0.8 dersek bir node'un elimine edilme ihtimali %20 olmuş olur.

d3 vektörü, dropout vektör olarak adlandırılır, a3 ile aynı boyutta olacak, keep_prob olasılığıyla bağlantılı olarak bazı elemanları 0 bazıları ise 1 olacak. Bu bize dropout sağlayacak.

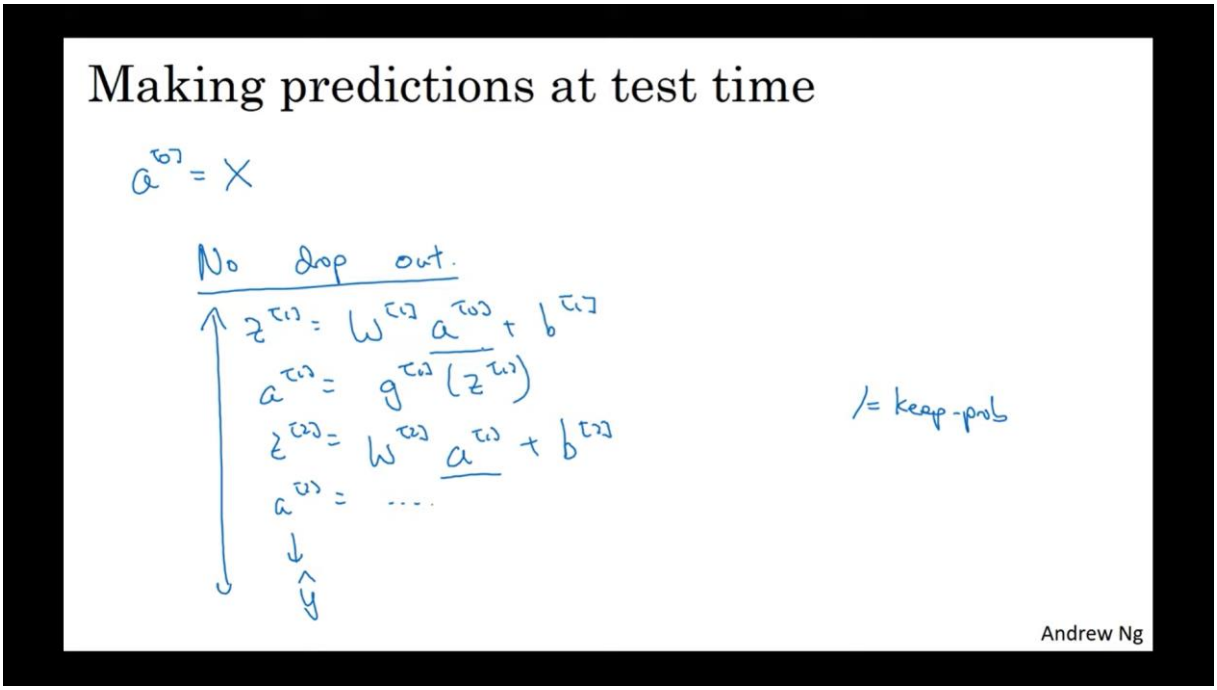
Bunun için öncelikle a3 ile d3'yi elementwise çarpıyoruz böylece bazı node'lar elimine edilecek. Ardından a3 vektörünü keep_prob parametresi ile scale ediyoruz yani a3'yi 0.8'e bölüyoruz.

Bunu neden yaptığımızı açıklayalım: diyelim ki 3. Hidden layerda 50 hidden unit var, yani a3 50x1 bir vektör veya 50xm. Eğer %20 ile elimine edersem, elimde ortalama 40 hidden unit kalacaktır. Bu yüzden z4'ın değeri ortalama %20 düşecektir çünkü z4=w4a3+b4 olarak hesaplanıyor.

Z4'un beklenen değerini %20 düşürmemek için a3'ü 0.8'e böleriz böylece sonuçta elde edilecek z4 aşağı yukarı dropout olmadan önce elde edilecek z4 ile aynı olacaktır.

Bu scaling işlemi nedeniyle bu metoda inverted dropout deniyor. Daha sonra göreceğiz ki bu sayede test yani NN evaluation daha kolay oluyor.

Şimdi diyekim ki modeli dropout ile train ettik ve şimdi test edeceğiz. Prediction yaparken, dropout kullanmayacağız. Çünkü prediction yaparken, output'umun random olmasını istemeyiz. Buna alternatif olarak yapılabilecek bir başka şey şu olur, prediction için dropout kullanırız ancak tek bir prediction için bir sürü kez random dropout yaparız ve sonuçların ortalamasını alırız. Ancak bu compuationaly inefficient ve aşağı yukarı aynı sonucu elde ederiz.



Dropout kullanmadan da sistemin çalışmasının olayı, dropout yaparken scale etmemiz, yani 0.8'e bölmüş olmamız.

because we have eliminated 20% of the neurons and thus 20% of the elements in the layer, so we want to compensate for that lost 20% by dividing by 0.8 to keep the same expected value of z4. Why do we want to do that? Remember that in the testing phase, we use all neurons without dropout, i.e. the full network, so we want to train the neurons to work in the full network, and give values relevant to the output in that network, rather than in the network with neurons turned off.

Vid 7

Understanding Dropout

Önceki kısımdan edindiğimiz ilk intuition şuydu, her iterasyonda farklı ama küçük bir NN ile çalıştığımız için overfitting'in önüne geçiyorduk.

İkinci intuiton'a bakalım: Tek bir unit'e odaklanırsak, her iterasyonda bazı unitlerin elimine edilmesi sayesinde aşağıda mor ile gösterilen unit, tek bir feature'a fazla bağlanamaz yani tek bir feature'a fazla ağırlık veremez bunun yerine ağırlığını tüm inputlara dağıtmak zorunda kalır. Bu da aynen L2 regularization'ın yaptğına benzer squared norms of weights'in minimize edilmesine neden olur.

Bir başka konuya gelirsek, diyelim ki sağdaki gibi bir NN olsun, burada her layer farklı keep_prob belirlenebilir, böylece çok neuron olan ve overfitting'e yol açabilecek layerlar için yüksek eliminasyon oranı set edebiliriz ve diğerleri için ise düşük oranlar set edebiliriz.

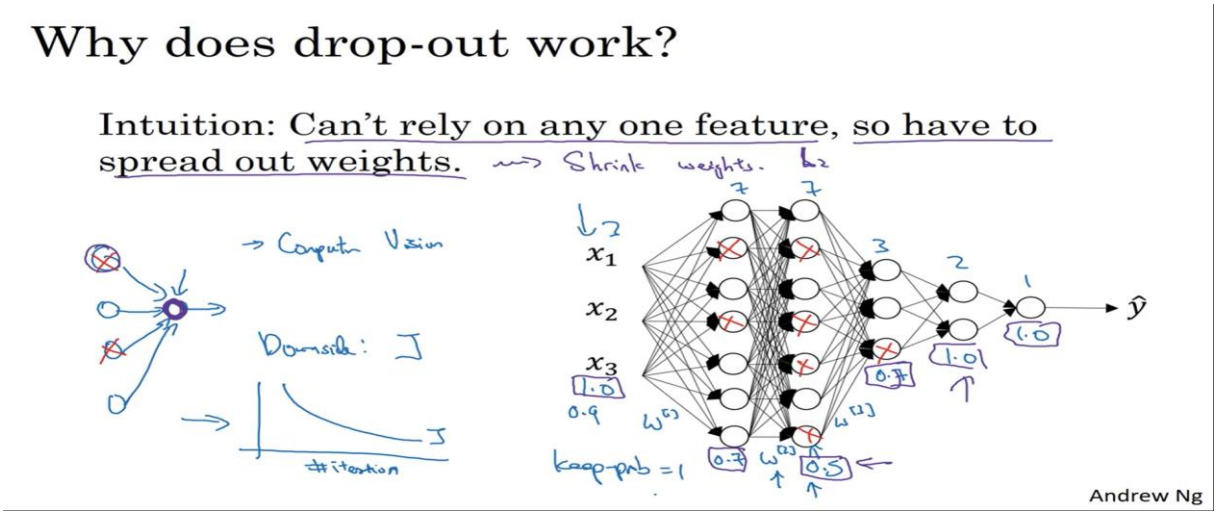
Fakat bu da daha fazla hyperparameter demek. Buna bir alternatif keep_prob'u aynı tutup bazı layer'lara dropout uygulamamaktır.

Teknik olarak dropout'u input layer'a da uygulayabiliriz ama çok önerilmez.

Son olarak bazı uygulama tyolarından bahsedelim. Dropout'ın ilk olarak başarıyla kullanıldığı yer computer vision problemleri, computer vision için input size is too big and we almost never have enough data bu yüzden dropout regularization computer vision problemlerinde sıklıkla kullanılır.

Ancak şunu unutma, eğer algoritma overfitting göstermiyorsa, dropout kullanmak anlamsız. Computer vision da neredeyse herzaman overfitting sözkonusu olduğu için gönül rahatlığıyla kullanılır ancak başka problemlerde bunun garantisi yok.

Dropout'ın bir başka downside'ı ise şu artık cost function J is not well defined. Her iterasyonda farklı node'lar öldüğü için gradient descent'in performansını double check etmek zorlaşıyor. Bu sebeple önce dropoutsuz veya keep_prob=1 olarak algoritmayı çalıştırıp gradient descent'in doğru çalıştığından emin oluruz sonra dropout'ı uygularız.



Vid 8

Other regularization Methods

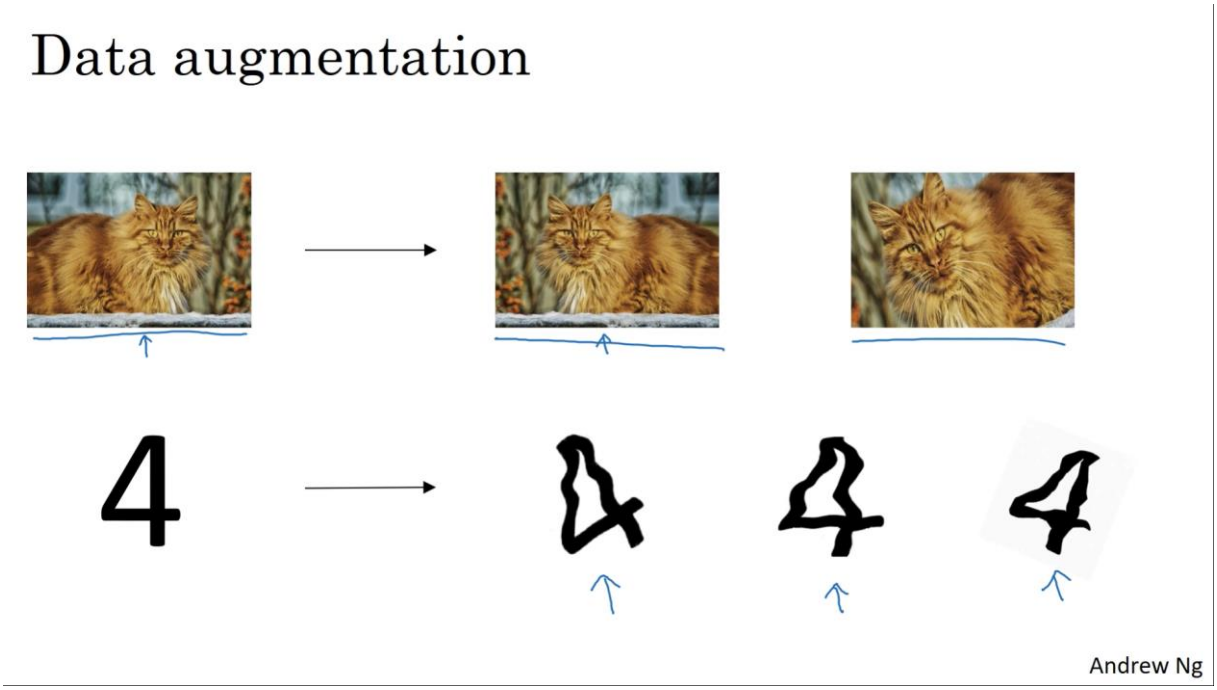
L2 ve dropout regularization'ın yanında overfitting'i reduce etmek için birkaç farklı teknikten de söz etmek mümkündür.

Eğer overfitting söz konusu ise getting more training data can help ancak bu her zaman mümkün değil. Ancak data augmentation ile training set'i artırabiliriz.

Mesela image input ile cat classifier yapıyoruz diyelim, training set'i alıp aynalarsak bu fotoğrafları yeni örneklermiş gibi kullanabiliriz. Benzer şekilde rotation ile random crop uygulayıp yeni örnekler elde etmek de mümkün.

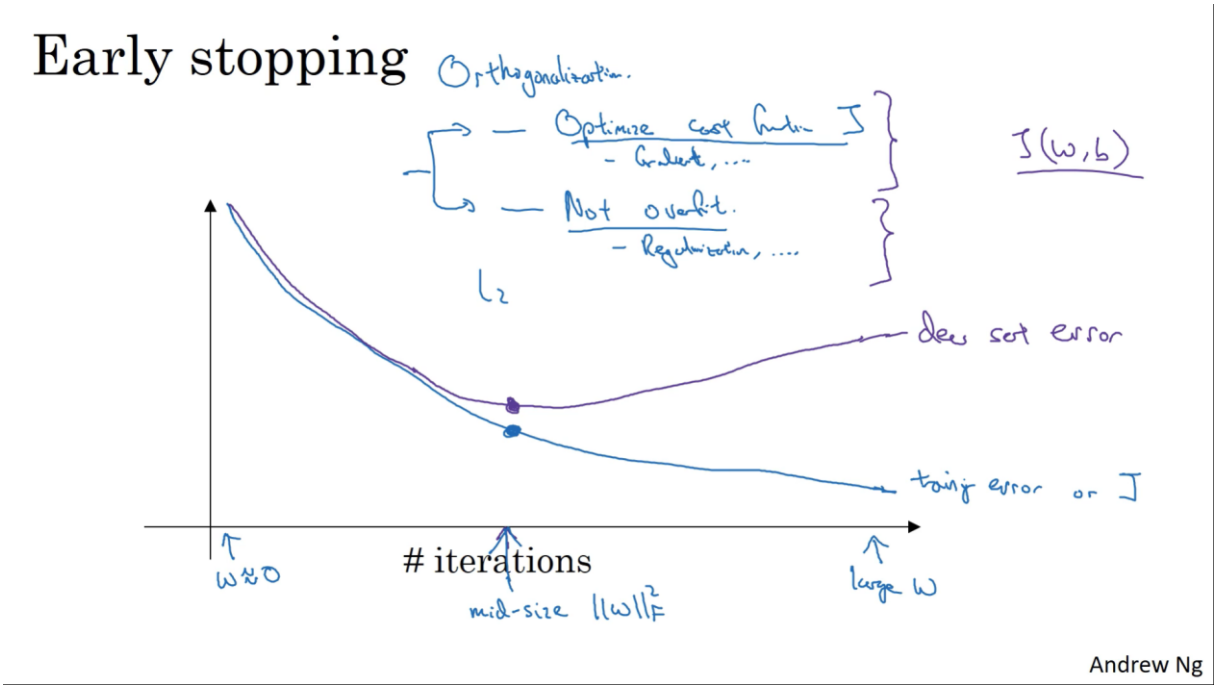
Elbette bu yöntemle elde edilen örnekler, independent örnekler kadar katkı sağlamaz ancak yine de iş yapar, ve rahat bir şekilde elde edilebilirler.

Benzer şekilde algoritmamız rakamları ayırmaya çalışıyorsa, rakamı rotate ettirebilirim ve/veya bazı distortionlar ekleyebilirim.



Sonuç olarak data augmentation ile data set'i artırabiliriz ve böylece overfitting'i azaltabiliriz.

Bir diğer overfitting azaltma yöntemi ise early stopping dir.



Early stopping'in olayı basit, optimizasyonu erken durduruyoruz böylelikle dev set error ile training error farkı açılmadan yani ortalarda bir yerde durdurmayı ve overfittingden muzdarip olmayan bir model elde etmeyi planlıyoruz.

Bu neden çalışıyor? Şöyle düşünebiliriz, ilk başta random initialization ile w değerleri 0'a çok yakın, her iterasyon ile w giderek büyüyor. Early stopping ile w değerleri çok büyümeden eğitimi durduruyoruz ve böylece L2 regularization'a benzer şekilde daha küçük w'lerden oluşan bir NN oluşturmuş oluyoruz.

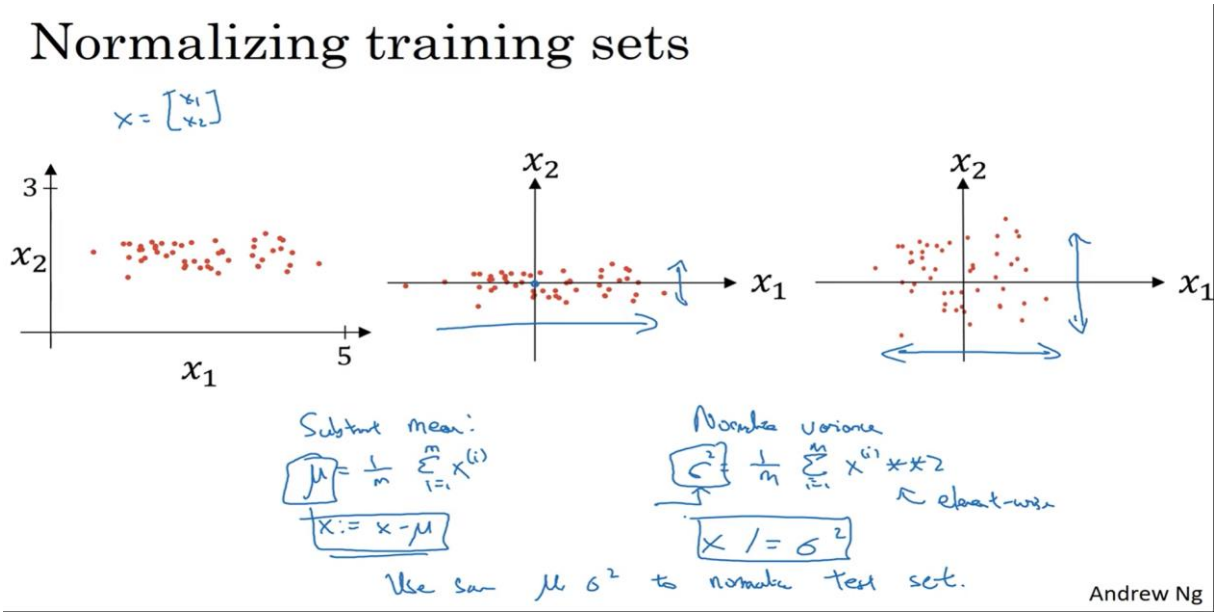
Early stopping'in bir downside'ı var o da şu: Biz normalde, cost function optimization ve overfitting çözümünü 2 farklı problem olarak ele alırız, bunları farklı toollar ile birbirinden bağımsız olarak çözeriz buna ortogonalization denir, daha sonra göreceğiz. Early stopping ile bu iki kavram birbirine girer, yani overfitting'i engelleyeceğim diye aslında optimizasyonu yapamamış oluyorum.

Andrew L2 kullanmanın onun için daha rahat olduğunu söylüyor ancak bu durumda lambda için bir çok değer denemesi gerekiyor, early stopping'in artısı da da gradient descent'i bir kez çalıştırarak küçük w, orta w, ve büyük w etkilerini görmüş oluyoruz, bir de lambda set etmekle uğraşmıyoruz.

Vid 9

Normalizing Inputs

Bir NN eğitirken, training'i hızlandıracak tekniklerden bir tanesi inputları normalize etmektir.

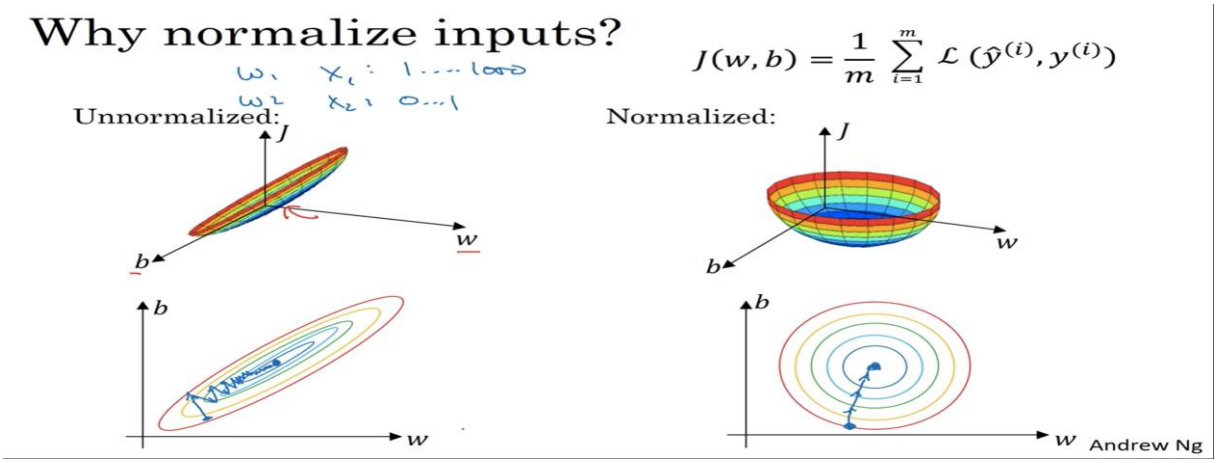


Göründüğü gibi normalizasyonda 2 stepten bahsedebiliriz, ilki dataset'in mean'ini 0'a çekmek, ikincisi ise her feature için variance'ı 1 yapmak. Böylece data set ilk halden sırasıyla 2. Ve 3. Hale geçer.

Bu noktada önemli bir not şu, test set'inormalize etmek için aynı Mü ve Sigma^2 değerlerini kullanmalıyız.

İyi ama neden normalizasyon kullanıyoruz? Ne gerek var? Eğer unnormalized input features kullanırsak cost function sol altta gördüğümüz istemediğimiz shape'lerde ortaya çıkabilir, ince ve uzun bir bowl gibi. Ancak inputları normalize edersek cost function sağda görüldüğü gibi olacaktır, daha simetrik bir yapı.

Soldaki gibi bir cost function için gradient descent'in minimuma ulaşması uzun ve zor olacaktır



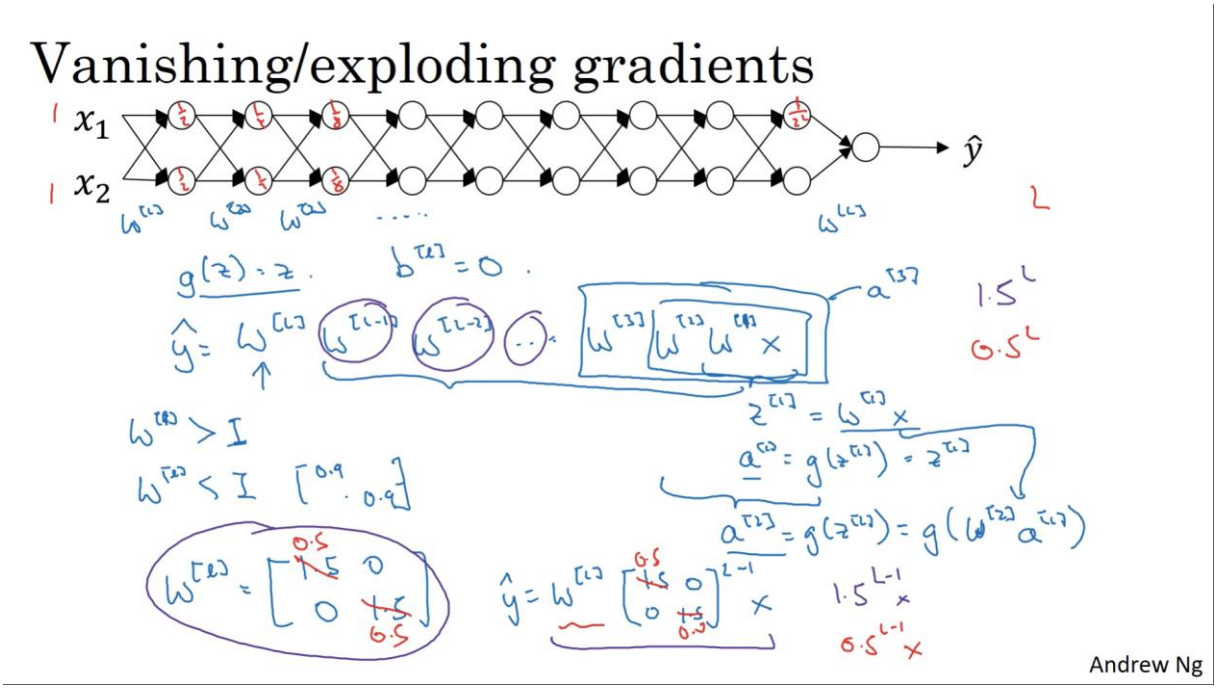
Normalizasyon almost never hurts, yani kullan gitsin. Eğer input scale'leri eşitse gerek yok ama yine de kullanılabilir.

Vid 10

Vanishing/Exploding Gradients

NN train ederken karşılaşılan problemlerden bir tanesi, özellikle deep NN eğitirken karşılılır: Vanishing and Exploding Gradients.

Bu şu demek, çok derin bir network eğitirken gradient değerleri bazen çok büyük veya çok küçük olabilir. Bu kısımda bu problemi anlayacağız ve, random weight initializiation'ı dikkatli yaparak bu problemi nasıl önleyebileceğimizi göreceğiz.



Burada problem açıklanmış yukarıdaki gibi basit ama derin bir NN olduğunu varsayarsak, ve activation'ın linear b'nin ise 0 olduğunu varsayarsak y_hat yukarıdaki gibi tanımlanabilir. Bu durumda wl değerleri Identity matrix'ten biraz büyük olursa, çıkış çok büyüyecek benzer şekikde, identityden biraz küçük olması durumunda çıkış 0'a çok yaklaşacaktır. Bu durum gradients için de geçerlidir. Bu durumda learning process zorlaşacaktır, özellikle gradients'in 0'a yaklaşması durumunda learning çok yavaşlayacaktır.

Bu problem uzun süreler boyunca derin ağlar eğtmenin önünde bir engel olarak kaldı. Bir sonraki başlıkta göreceğimiz üzer weight initialization'ını dikkatli yaparsak bu problem kısmen çözebiliyoruz.

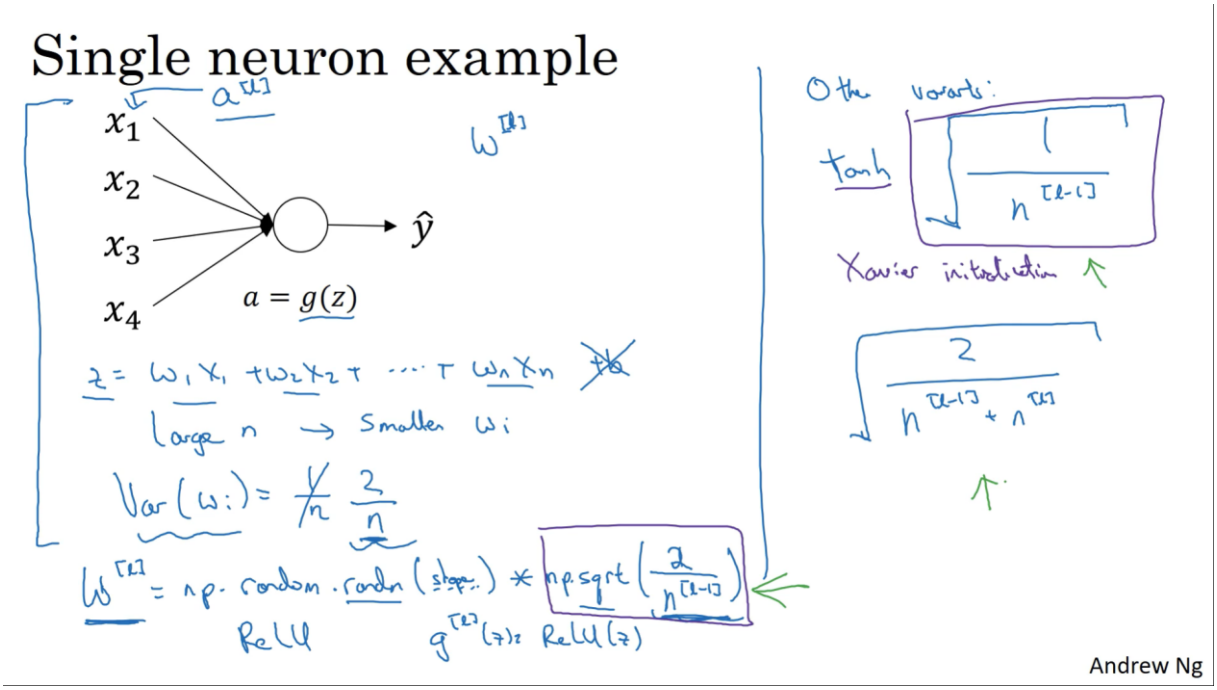
Vid 11

Weight Initialization in a Deep Network

Derin ağlarda karşımıza çıkan vanishing/exploding gradients problemine bir çözüm olarak careful weight initialization karşımıza çıkar, bunu daha iyi anlayalım.

Yukarıdaki örnekte problem neydi, her layer için neuron çıktılarının giderek küçülmesi veya büyümesi. Bu problemi nasıl çözeriz?

Öncelikle tek bir neuron için initialization'ı odaklanacağız daha sonra bu anlayışı genişleteceğiz.



Yukarıdaki gibi bir neuron'umuz olduğunu varsayalım. Z belirtildiği şekilde tanımlanır, b=0 kabul edelim for the sake of simplicity. N yani neuron'a giriş sayısı arttıkça o neuron'a ait w değerlerim küçülsün isterim ki sonuçta orantısız bir çıkış elde etmeyeyim.

Benzer şekilde yapılabilecek şeylerden biri wi'nin variance'ını 1/n olarak ayarlamak. Bunu yukarıda gösterilen şekilde gerçekleyebiliriz. Eğer Relu activation kullanılıyorsa 1/n yerine 2/n variance'ı kullanmak daha verimli oluyor.

Sonuçta amaçladığımız şey, weight matrixlerinin yani w'lerin 1'den çok büyük veya küçük olmamasını sağlamak böylece vanishing veya exploding probleminin ortaya çıkmasını geciktirmiş oluyoruz yani daha çok layer'da ortaya çıkmış olacak. Problemi tamamen çözmüyor.

Relu yerine tanh activation kullanılıyorsa 2/n yerine 1/n kullanılır. Ayrıca bazen son gösterilen formül de kullanılabilir.

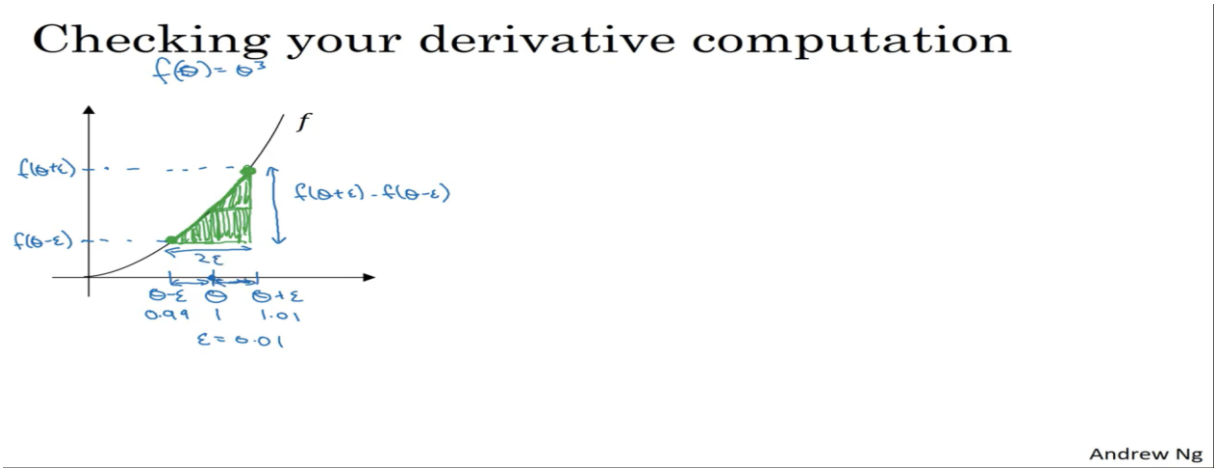
Vid 12

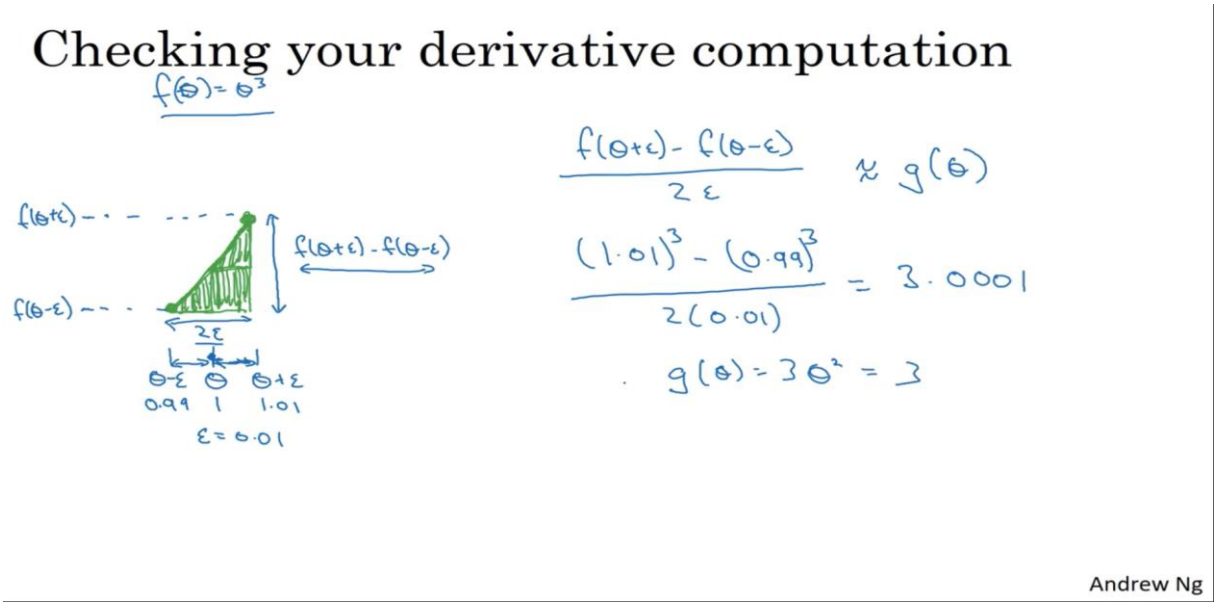
Numerical Approximations of Gradients

Backpropagation'ı implement ederken gradient descent checking ile backprop implementation'ın doğru olup olmadığını kontrol edebiliriz.

Öncelikle gradients'i numerik olarak nasıl hesaplayacağımızı göreceğiz sonra diğer kısımda ise gradient checking'i nasıl implement edeceğimizden bahsedeceğiz.

Aşağıdaki gibi bir f fonksiyonu olsun, theta noktasında gradient hesaplamak için +E, -E kullanarak oluşan üçgenin eğimini buluyorum. E burada 0.01 olarak seçildi. Daha önce sadece +E ile çizilen üçgen için de hesap yapmıştık.





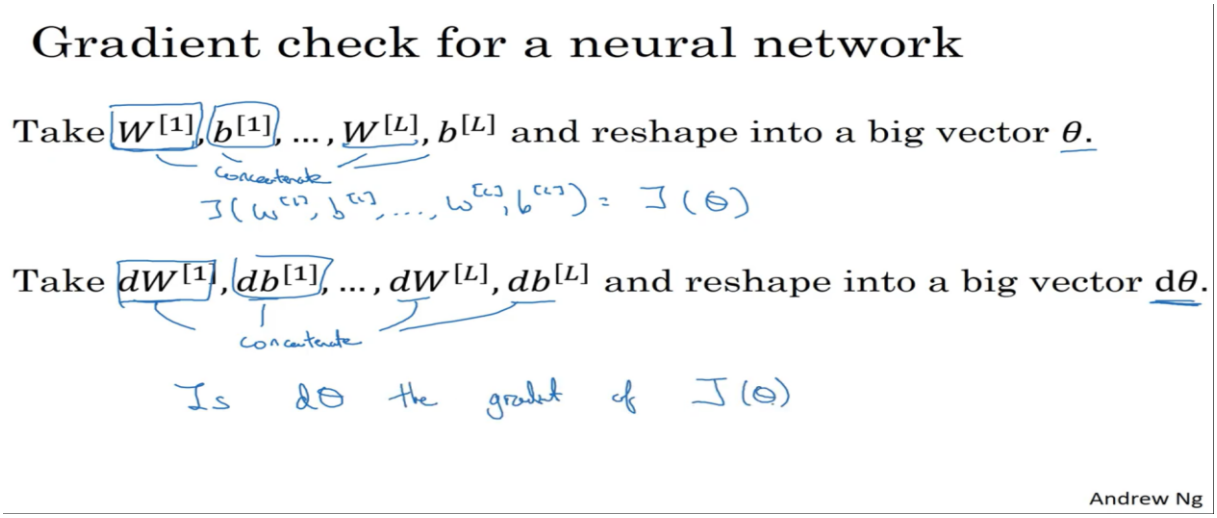
Sadece +E kullanırsak sonuç yine 3'e yakın olacak ama hata daha fazla olacak, 2E kullanmak daha kesin sonuç veriyor, computation biraz yavaşlasa da böylesi daha mantıklı.

Vid 13

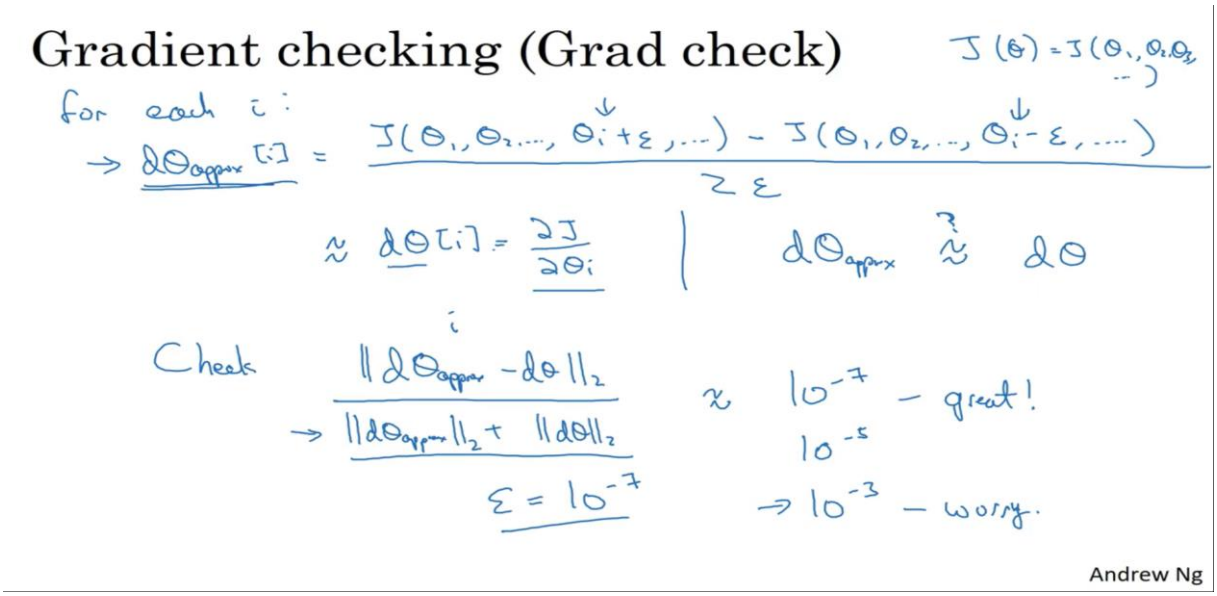
Gradient Checking

Gradient checking ile backpropagation implementationımızda olası hataları bulabilir ve kayda değer şekilde zaman kazanabiliriz.

NN için nasıl gradient checking yapacağımıza bakalım:



Sonuçta elimizde Theta ve dTheta vektörleri olacak, şimdi cevap aradığımız soru şu: dTheta JTheta'nın gradient'i mi?



Bunu anlamak için yaptığımız şey basit, her ayrı parametre için yukarıdaki gibi dThetaApprox hesaplarız ve sonucunda bu yeni dThetaApprox vektörünün, backprop ile bulunan dTheta'ya aşağı yukarı eşit olmasını bekleriz.

Bu eşitliği check etmek için ise en alttaki formülü kullanılırız, euclidian distance ile check ederiz ve eğer sonun 10^-7'den küçükse sıkıntı yok, 10^-5 civarında ise belki problem olabilir, 10^-3 civarında ise sıkıntı var demektir.

Vid 14

Gradient Checking Implementation Notes

dThetaApprox'u hesaplamaz oldukça yavaş bir işlem bu yüzden bu hesaplamayı training için kullanmamalıyız.

Yani dTheta'yı backprop ile hesapla, sonra debug için dThetaApprox'u hesapla check et eğer sıkıntı yoksa dThetaApprox hesaplamasını durdur.

İkinci tip şu: Eğer check sonucu bir farklılık çıkarsa, hangi elemanlarda hata çıktığına bak diyor belki tüm hatalar db veya dw hesaplamasından kaynaklanıyor, direkt gidip ona odaklanabiliriz.

Üçüncü tip: Eğer regularization term var ise, JTheta'nın regularization term ile kullanıldığına dikkat et!

Gradient check doesn't work with dropout. Çünkü J'yi hesaplamak zorlaşıyor. Bu yüzden önce grad check'i dropout yokken denerim, daha sonra dropout'u açarız ve umarız ki dropout implemantation doğru olsun.

Son olarak: Nadiren de olsa, backprop w ve b değerleri 0'a yakınken doğru çalışıp, gradient descent ilerledikçe ve bu değerler büyükdükçe hata çıkarabilir bu sebeple, pek uygulanmasa da hem başta hem daha sonra gradient checking yapılabilir.