

COURSE4 – W2

Vid 1

Why Look at Case Studies

Bu hafta CNN için bazı case studies'den bahsedilecek. Böylece amacımız, bir intuition kazanmak. Ayrıca, genelde bir uygulamada işe yarayan computer vision architecture, başka bir uygulamada da işe yarar, bu yüzden başkalarının çalışmalarından yararlanmak fayda sağlayacaktır.

Bu kısımda öncelikle aşağıda belirtilen 3 klasik network yapısını inceleyeceğiz. Bunlar görece eski fakat modern yapıların temelini oluşturan paperlar. Buradan öğrendiğimiz fikirleri kendi uygulamalarımızda da kullanabiliriz.

Outline

Classic networks:

- LeNet-5 ←
- AlexNet ←
- VGG ←

ResNet (152)

Inception

Andrew Ng

Daha sonra residual networks'ten bahsedilecek, biliyoruz ki neural networks giderek daha da derinleşiyor, tabi derinleştikçe eğitim de zorlaşıyor. ResNet trained a very very deep 152 layer neural network, has some very interesting tricks and ideas how to do that effectively.

Son olarak Inception NN case study'ı inceleyeceğiz.

Computer Vision problemleri ile uğraşmıyor olsak dahi, buradaki fikirler kullanışlıdır ve diğer alanlarda da kullanılabilirler, bu yüzden iyi anlamaya çalış.

Vid 2

Classic Network

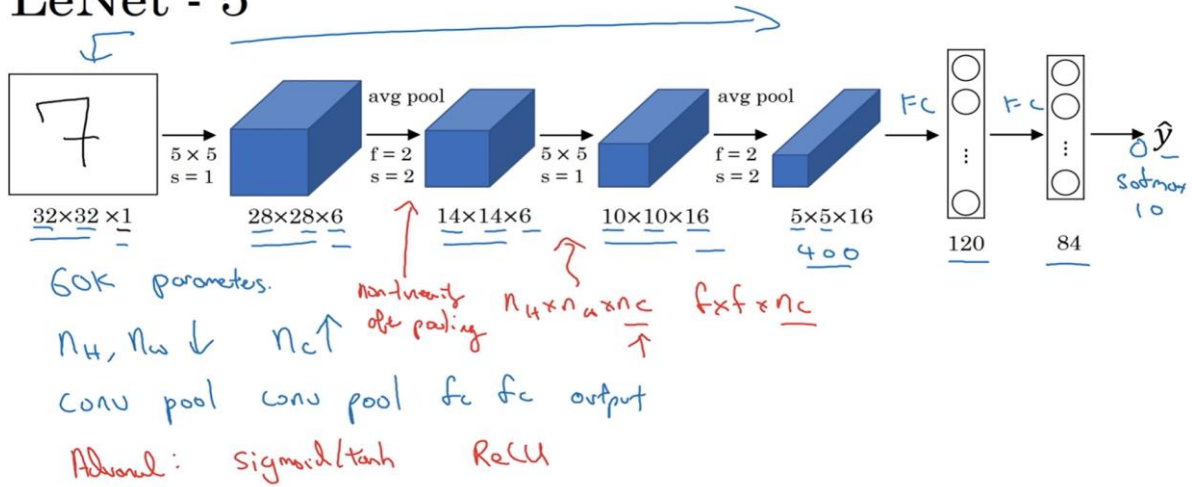
Bu kısımda 3 klasik NN architecture'ı öğreneceğiz:

***LeNet-5 *** AlexNet ***VGG-16

LeNet-5 ile başlayalım. LeNet-5'in amacı handwritten digits recognition idi.

- 32x32x1'lik grayscale input image ile başlıyor.
- Ardından 6 tane 5x5 filtre kullanılıyor, stride=1, no padding. Sonuçta 32-5+1=28 yani her filter için map 28x28 oluyor, 6 filtre olduğu için output 28x28x6.
- Ardından, average pooling uygulanıyor f=2, s=2. O zamanlarda max pooling ortada olmadığı için avg pooling kullanmış, max pooling daha iyi çalışıyor.
- Pooling ile height and width yarıya düşüyor ardından 16 tane 5x5 filtre kullanarak conv uyguluyoruz, sonra yine bir pooling uyguluyoruz.
- 5x5x16'lık output'u flatten edip, 400 inputu fully connected layer'a besliyoruz 120 – 84 – 1'lik bir FCNet'ten bahsediyoruz.
- Yani tek bir output unit var, 10 farklı değer alabiliyor. Bunun modern versiyonunda output olarak 10 unitli bir softmax layer kullanılırdı.

LeNet - 5



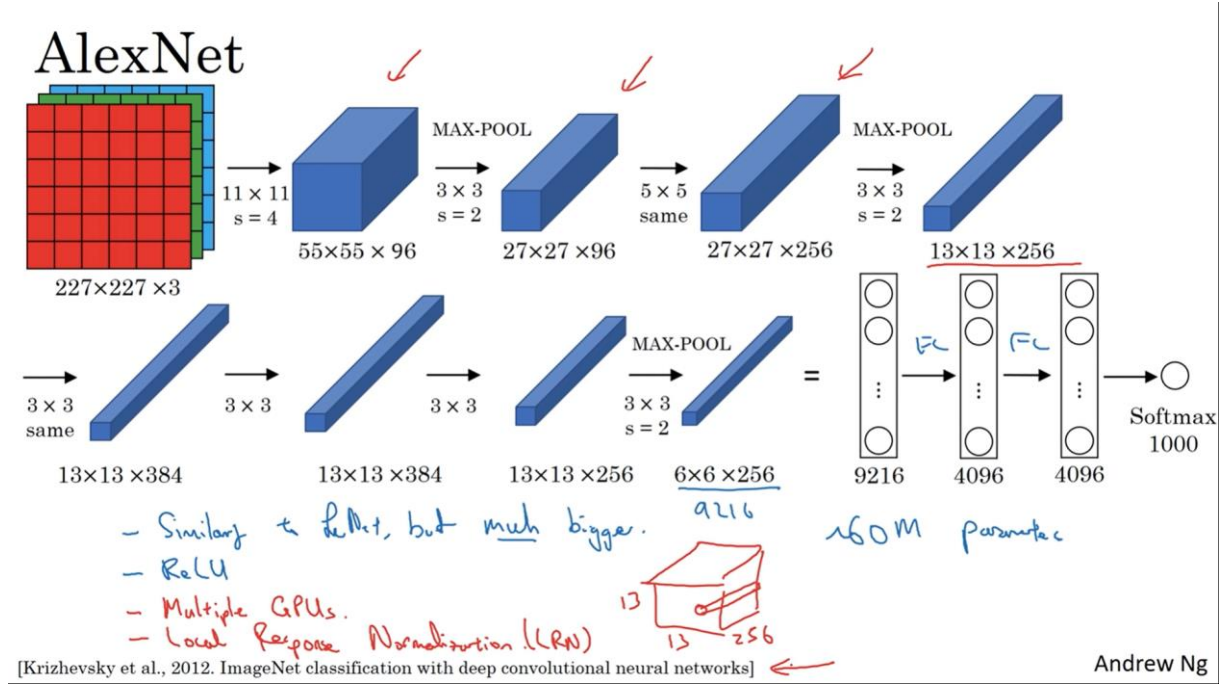
[LeCun et al., 1998. Gradient-based learning applied to document recognition]

Andrew Ng

- Bu networkun 60K parametresi vardı, gayet küçük bir network günümüzdeki networkler 10M – 100M parametre civarında içerirler.
- Bu yapıdan alınacak bir intuition şu: as w ego deeper , nH and nW reduces but nC increases.
- Burada günümüzde de tekrarlanmaya devam eder bir başka pattern şu: CONV → POOL → CONV → POOL → FC → FC → OUTPUT yapısı.
- Farklı olarak bu eski olduğu için activation olarak ReLu kullanmıyor, ayrıca activation'ı bugün yaptığımız gibi CONV→ACTIVATION→POOL değil CONV→POOL→ACTIVATION olarak kullanmış.

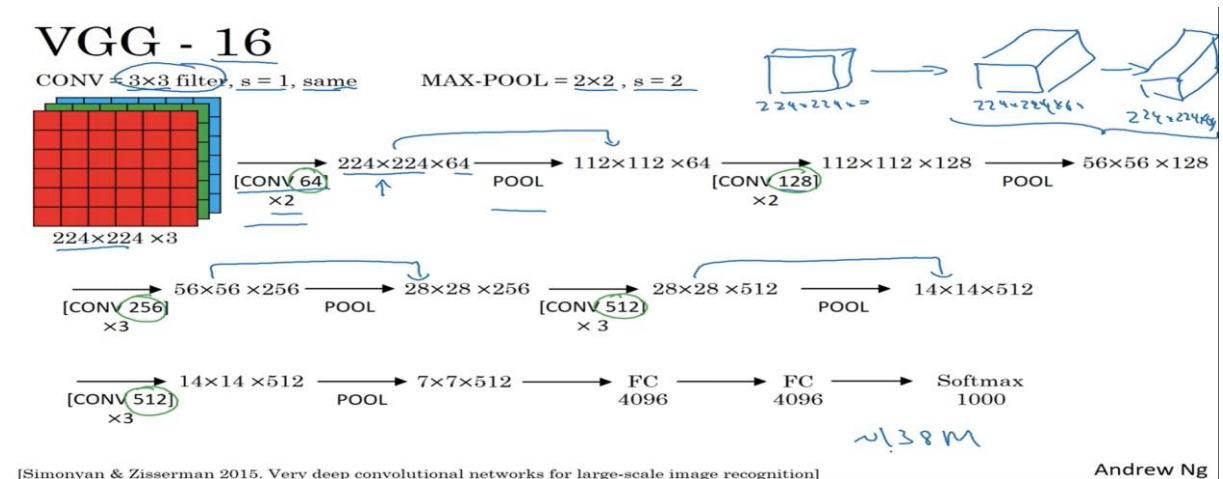
İkinci architecture AlexNet. Bu LeNet'e göre çok daha büyük. Ayrıca ReLu kullanıyor. Eski olduğu için bu ve yukarıdaki paper'ı incelediğimizde bazı computation tricks görürüz, bunlar günümüzde kullanılmayan şeylerdir, mesela bu örnekte multiple GPU's için bir trick yapılmış günümüzde kullanılmayan bir şey, ayrıca AlexNet için bir normalizasyon söz konusu bu da günümüzde kullanılmaz.

Sonuçta yapısı aşağıda görüldüğü gibidir. 60M civarı parametresi vardır.



AlexNet paper'ın önemi şu, insanları deep learning'in computer vision için kullanışlı olduğuna ikna eden bir performans sağlıyordu, böylece deep learning çalışmalarına da aslında büyük katkı sağladı.

Son olarak aşağıda VGG-16 yer alıyor. Aslında bu network AlexNet'e göre daha fazla parametre 138M içerse de, tekrar eden uniform yapıda bir architecture kullandığı için yani sürekli 3x3 filtreler, 2x2 poolingler kullanıldığı için, ve conv layers için sırasıyla 64, 128, 256, 512 filtre kullanıldığı için ilgi çeken bir yapıdır. nH, nW yarı yarıya azalırken, nC iki kat artar.

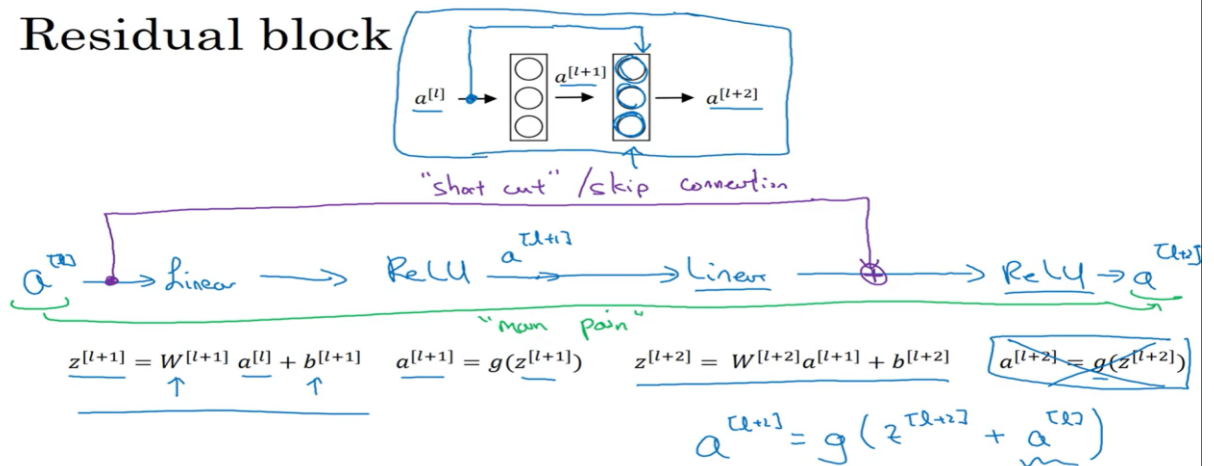


Vid 3

ResNets

- Çok derin NN'leri eğitmek zor çünkü **vanishing/exploding gradients** problemi karşımıza çıkıyor.
- Neydi bu problem?
 - Temelde network'ün derin olmasından dolayı, başlardaki layerlarda bir parametrenin değişiminin, output'a çok büyük veya küçük etkilerinin olması (bu etkinin büyük veya küçük olmasını belirleyen şey weights). Bu etki ne demek, baştaki küçük bir değişim sonucu uçurabilir, yani başlardaki weightlerin türevi inanılmaz yüksek veya düşük çıkabilir bu da eğitimi çok zedeler veya çok yavaşlatır.
- ResNet **"skip connections"** fikri ile bu problemten kurtuluyor.
- Skip connections fikri ile, çok derin ResNets (maybe 100 layers) eğitebiliyoruz.
- ResNets **"residual block"** denen yapılardan oluşur. Öncelikle bunun ne olduğunu anlayalım.
- Öncelikle normal bir NN yapısından bahsedelim, $a[l]$ aktivasyonu aşağıdaki gibi bir layer'a alınsın, $a[l+1]$ çıktı verilsin ve sonra bu çıktı diğer layer'a alınsın ve $a[l+2]$ olarak verilsin.
- Bu $a[l] \rightarrow a[l+2]$ ilişkisi normal bir NN için aşağıda mavi ile görüldüğü gibidir, önce linear işlem yapılır ve $W*a+b$ ile $z[l+1]$ elde edilir . Daha sonra ReLU activation uygulanır, ve işlemler tekrarlanır.
- Bir **Residual Block** için ise $a[l+2]$ elde edilmeden önce işleme short cut ile $a[l]$ de aşağıdaki denklemde görüldüğü gibi dahil olur.

Residual block



[He et al., 2015. Deep residual networks for image recognition]

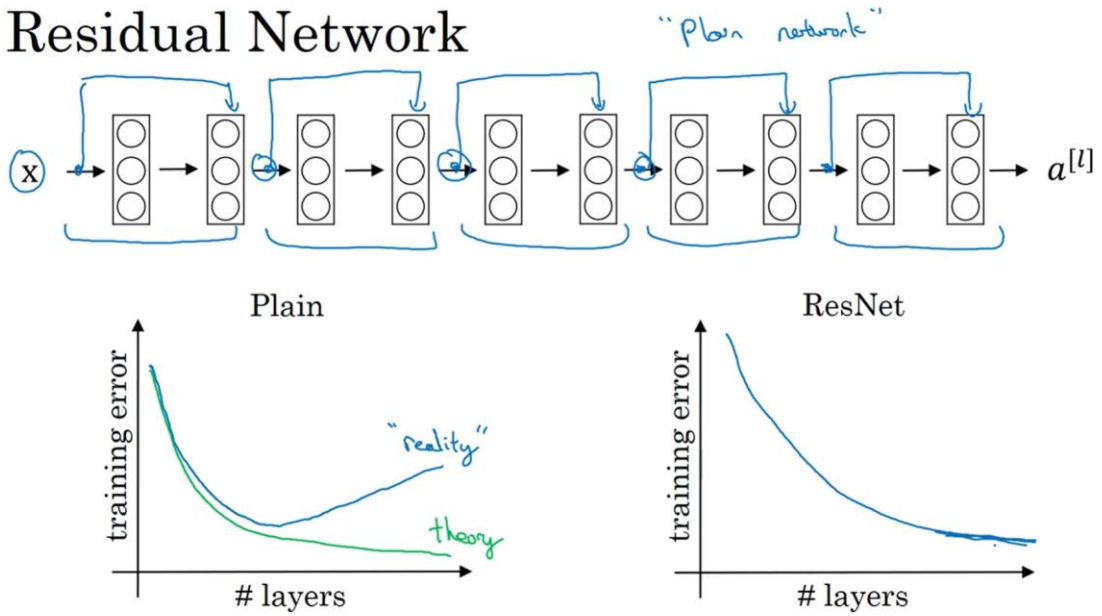
Andrew Ng

$z[l+2] = W*a + b$ şeklinde hesaplandıktan sonra buna $a[l]$ eklenir ve daha sonra activation uygulanarak $a[l+2]$ hesaplanır.

Önemli bir nokta: bunun gerçekleşmesi için $a[l]$ ile $a[l+2]$ 'in boyutlarının aynı olması gerek!

- Anlaşıldı ki işte bu şekilde oluşturulan Residual Blocks kullanınca çok daha derin network'leri problemsiz bir şekilde eğitebiliyoruz.
- Aşağıda 5 residual block'tan oluşan bir Residual Network diagramı görüyoruz, aslında ResNet dediğimiz residual blokcların biraraya getirilmesinden fazla bir şey değil.
- Yukarıda bahsedilen short cut veya skip connections konsepti kullanılmısa, aşağıda görülen network bildiğimiz bir network olacak, buna Plain Network diyebiliriz.

Residual Network



[He et al., 2015. Deep residual networks for image recognition]

Andrew Ng

- It turns out that, if you use a standard optimization algorithm such as gradient descent or other fancy ones to train a plain network(the regular network without the shortcuts) we find that:
 - # of layers arttıkça training error önce düşüyor fakat bir noktadan sonra artmaya başlıyor.
 - Teoride düşününce, bigger network kullanırsak, training error düşmeli çünkü daha complex bir fonksiyon kullanıyoruz, data'ya daha iyi oturmalı, ha elbette overfitting olabileceği için CV error da artar, ama şuan training error'a focuslanalım. Teoride düşmesi gereken training error pratikte bir noktadan sonra artıyor.
 - Anladığım kadarıyla bu training error artışının sorumlusu vanishing/exploding gradients problemi.
 - İşte ResNet bu problemi ortadan kaldırıyor ve derin network'ler eğitebilmemize olanak sağlıyor.

Vid 4

Why ResNets Work

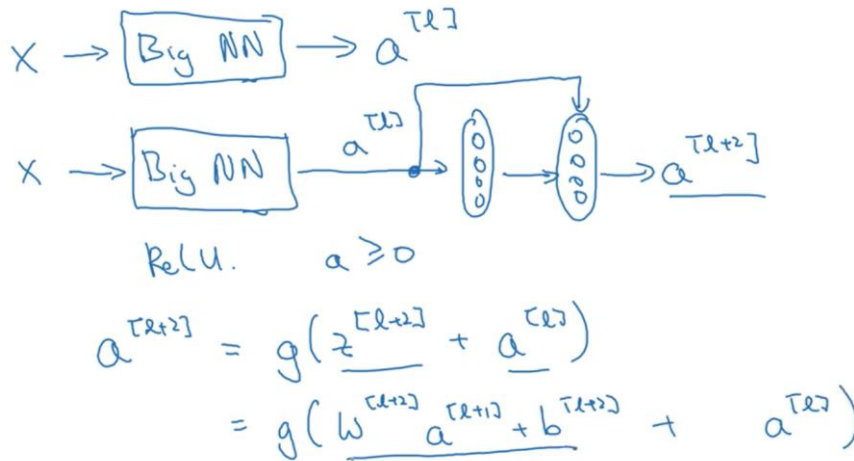
Bu kısımda ResNets'in nasıl olup da network'ün derinleşmesine rağmen performansı zedelememeyi başardığını intuitive bir şekilde anlamaya çalışacağız.

Genelde Training set üzerinde iyi performans elde etmek, CV ve Test set'lerde iyi performansın ön şartıdır. Bu yüzden ResNet'in başardığı şey değerlidir.

Aşağıda bir örnekle residual network'ün büyük network'lerde dahi nasıl performans kaybına uğramadığını anlamaya çalışalım.

- Diyelim ki bir input X 'i büyük bir NN'e besliyoruz ve $a^{[l]}$ elde ediyoruz.
- Bu örnek için bu büyük NN'i alıp modify ettiğimizi ve buna 2 layer daha eklediğimizi düşün, bu 2 layer aslında bir residual block. Sonuçta yeni çıktıya da $a^{[l+2]}$ diyelim.

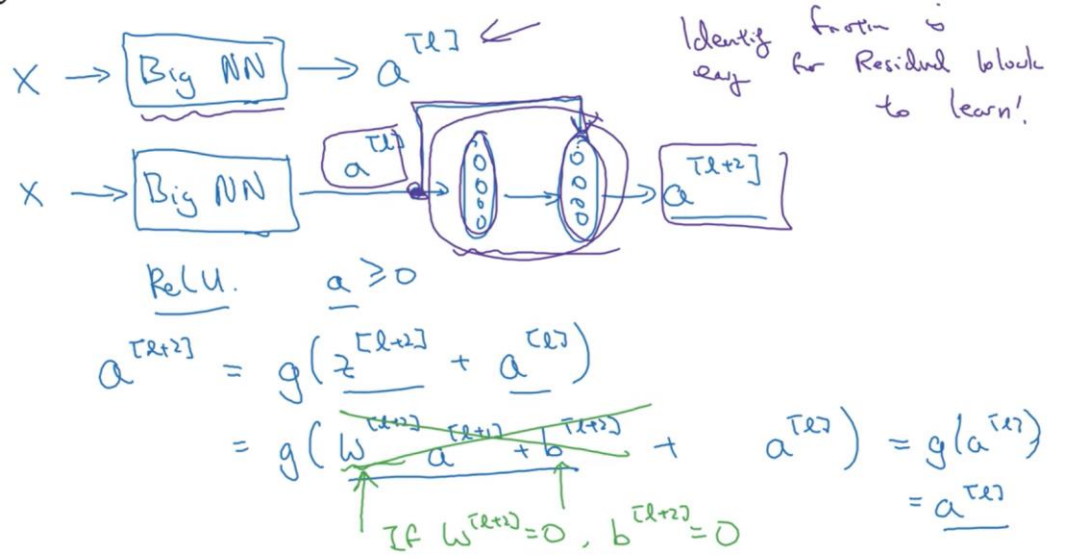
Why do residual networks work?



Andrew Ng

- For the sake of argument diyelim ki network boyunca hep ReLU kullanıldı ve bu yüzden bütün aktivasyonlar 0'dan büyük. ($a \geq 0$)
- Önceki kısımdan bildiğimiz üzere $a^{[l+2]}$ yukarıdaki gibi hesaplanabilir. $a^{[l+1]} \rightarrow \text{Linear} \rightarrow + a^{[l]} \rightarrow \text{ReLU}$ şeklinde.
- Bu noktada şunu düşün, eğer eğitim için L2 regularization kullanıyorsak, values of $W^{[l+2]}$ tend to shrink, genelde b 'ye uygulanmaz ama uygulanırsa $b^{[l+2]}$ de küçülür.
- Eğer $W^{[l+2]}$ ve $b^{[l+2]}$ tamamen yok olursa sonuçta $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$ olur!
- Peki bu ne demek? Residual block için Identity Function'ı öğrenmek kolaydır, yani $a^{[l]}$ 'i $a^{[l+2]}$ 'ye eşitleyen fonksiyon skip connection sayesinde kolayca öğrenilebilir.

Why do residual networks work?

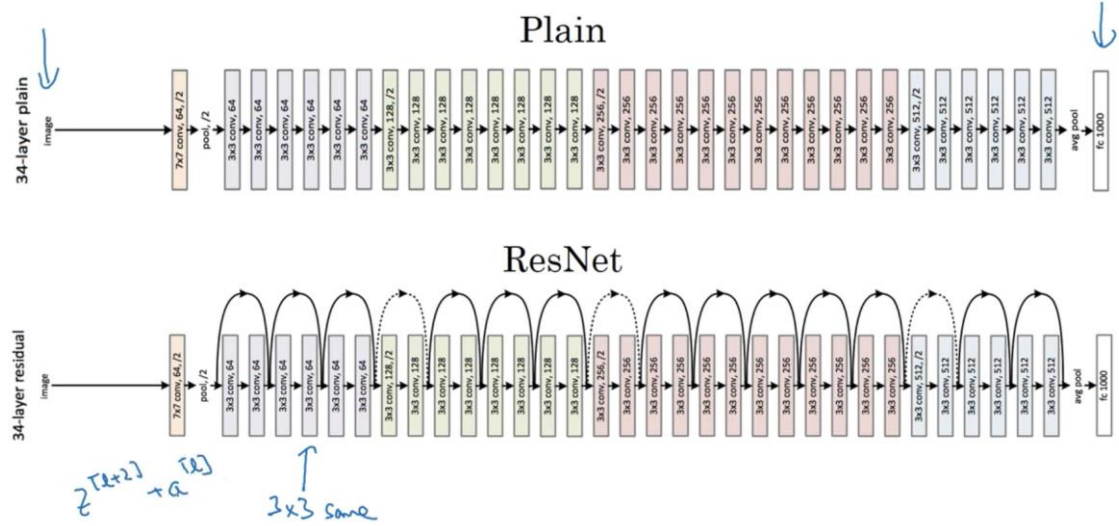


Andrew Ng

- Peki bu ne demek? Yeni eklenen 2 layer bir residual block olduğu için performansı düşürmez, çünkü yeni NN en kötü ihtimalle Big NN'in çıktısını üretebilir, bunu yapmak onun için çok kolay.
- Yani yeni eklenen layer'ın performans düşürmesi çok zor, en kötü ihtimalle network gidecek $a^{[L]}$ 'i çıktı olarak verecek, bunun üzerine bu iki layer ile yeni bazı şeyler öğrenmiş olmak da mümkün bu sebeple, çok derin yapılarda da dahi vanishing/exploding gradients sebebiyle problem yaşamıyoruz.
- Skip connections fikri kullanılmazsa, yani plain networks ile derin yapılar oluşturulursa, network'ün identity function'ı öğrenecek parametreleri bulması dahi çok zorlaşıyor bu sebeple layerların çoğu sonucu iyileştirmek yerine kötüleştiriyor.
- Sonuç olarak residual networks ile yeni eklenen layer'ların identity function'ı öğrenmesi çok kolay böylece eklenen block eğer performansa katkı sağlayamıyorsa en azından inputunu direkt output olarak verir böylece overall network'e zarar da vermiş olmaz, ResNet yapısı ile çok derin network'ler eğitmek bu sebeple mümkün.
- Residual Block yapısı için $a^{[L]}$ ile $z^{[L+2]}$ 'nin dolayısıyla $a^{[L+2]}$ 'nin boyutları aynı olmalı ki shortcut işlemi yapılabilir. Bu sebeple ResNet yapılarında sıklıkla same convolution kullanılır, böylece $a^{[L]}$ ile $a^{[L+2]}$ 'nin boyutlarının aynı olması gözetilir.
- Ancak boyutlar aynı olmasa dahi $a^{[L+2]} = g(W^{[L+1]} * a^{[L+1]} + b^{[L+2]} + a^{[L]})$ işleminde $a^{[L]}$ 'in başına bir W_s matrisi eklenebilir, bu matris ile amaç $a^{[L+2]}$ ile $W_s * a^{[L]}$ 'in boyutlarını aynı formata getirmektir. Bu matris padding veya başka bir işlem yapabilir ama, parametreler öğrenilebilir veya fix olabilir amaç boyutları denkleme.

Aşağıda da örnek birer Plain ve ResNet görölüyor.

ResNet



[He et al., 2015. Deep residual networks for image recognition]

Andrew Ng

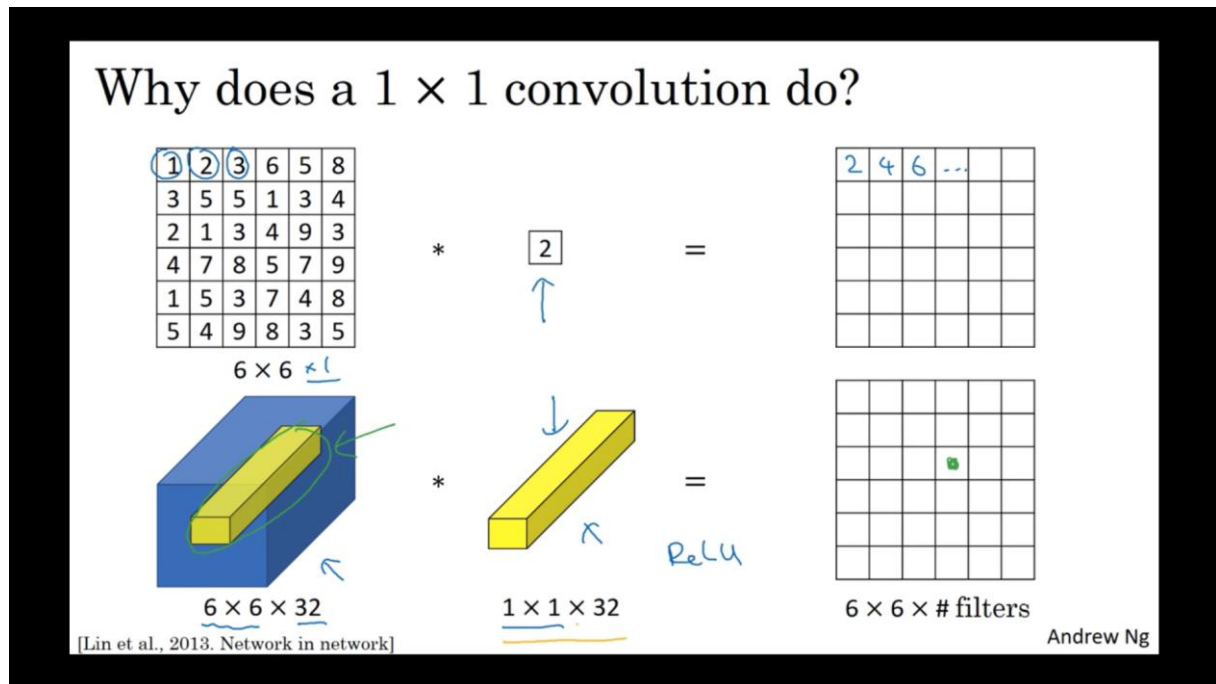
Genelde ResNet'in gerektirdiği $z[l+2] + a[l]$ işleminin yapılabilmesi için gereken boyut benzerliğini sağlamak için same convolution kullanılır, ancak bunun olmadığı veya pooling uygulanan durumlarda boyutlar uyuşmaz işte bu durumlarda da yukarıda bahsedilen Ws matrisi ile dimension adjustment yapılır.

Vid 5

Network In Network

ConvNets tasarlarken kullanılan fikirlerden bir tanesi 1×1 filtre kullanmaktır. İlk bakışta bu saçma gelebilir çünkü 2D düşünülünce 1×1 filtre kullanmak demek, matrisi bir constant ile çarpmaktan farksızdır, asıl olay dimension artınca anlaşılır.

- Aşağıdaki ilk durum için $6 \times 6 \times 1$ 'lik bir input image alınmış, buna 1×1 filtre uygulamak anlamsız çünkü, bu filtrenin yapacağı işlem tüm matrisi bir constant ile çarpmaktan farksızdır.
- Ancak aşağıdaki ikinci örneğe bakarsak, yani input'un $6 \times 6 \times 32$ olduğu örneğe, burada 1×1 filtre basit bir çarpımın ötesine geçer.
- Böyle bir input için 1×1 filtre kullanmak demek aslında $1 \times 1 \times 32$ filtre kullanmak demektir. Çünkü ancak böyle convolve edilebilir. Burada filtrenin 32 weight'inin aynı olma şartı olmadığını unutma!



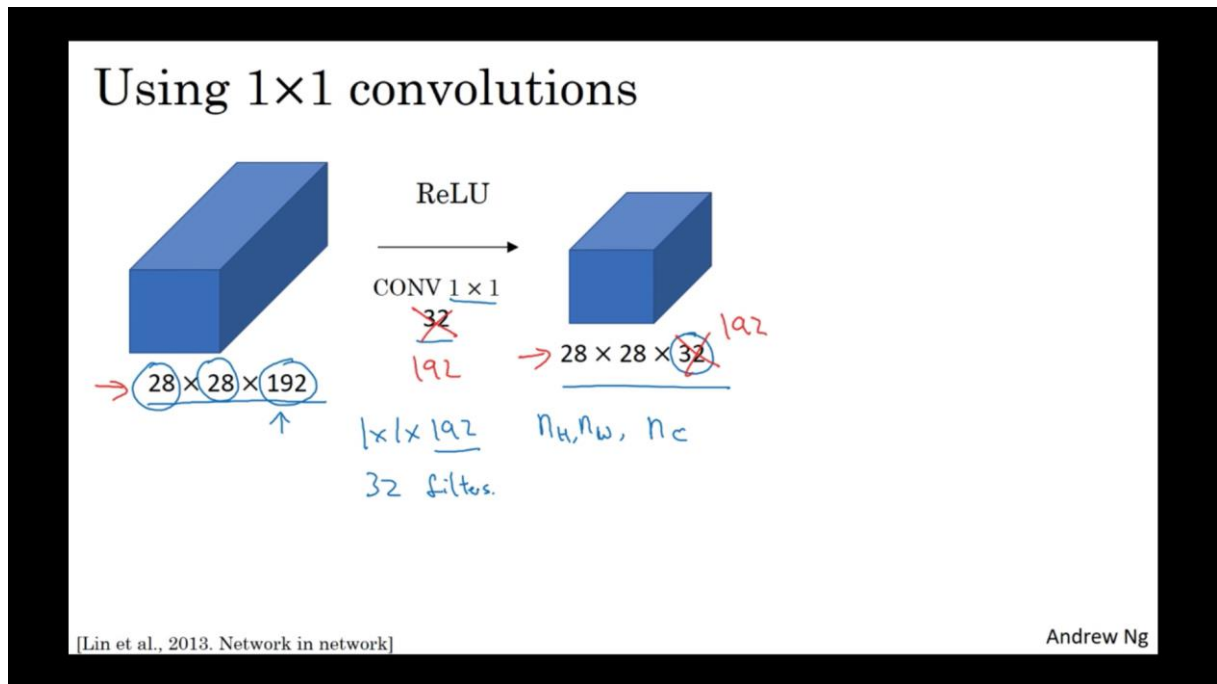
- Yani yukarıdaki örnekte 1x1 convolution, inputun 1x1x32'lik bir section'ı ile 1x1x32'lik filtreyi elementwise çarpar ve ReLU uyguladıktan sonra sonucu map üzerinde ilgili noktaya yerleştirir.
- Tabi birden fazla 1x1x32'lik filtre uygulamamız durumunda output size'ımız da 6x6x#filters şeklinde belirlenecektir.

Bu 1x1 convolution fikri bazen “Network in Network” olarak da isimlendirilir.

Bu fikir bir sonraki derste göreceğimiz **Inception Network** dahil olmak üzere bir çok network'ü etkilemiştir.

1x1 convolution'ın useful olduğu yerlerin birine aşağıda bir örnek var.

- Diyelim ki $28 \times 28 \times 192$ 'lik bir inputumuz var, eğer amacımız height and width shrink etmek ise, bunu pooling layer ile yapabileceğimizi biliyoruz.
- Peki ya n_c 'yi yani number of channels'ı shrink etmek istiyorsak bunu nasıl yapabiliriz?
- 32 tane $1 \times 1 \times 192$ filtre kullanarak convolution layer uygularız ve sonuçta $28 \times 28 \times 32$ 'lik output'umuzu elde edebiliriz.
- Bu işlemle yapılan şey şu elimizde 192 tane filtreden elde edilmiş feature detection sonucu var, bunları alıyorum tek bir filtre ile birleştiriyorum yani bir nevi bu 192 feature'dan tek bir feature'a geçiş yapmış oluyorum. Bunu aynı 192 input'u bir ReLU unit'e bağılıyormuşuz gibi düşün, weights filter weights olmuş oluyor.



- 192 tane $1 \times 1 \times 192$ filtre kullanarak output depth'i sabit de tutabiliriz, bu durumda da her bir yeni feature 192 feature'ın mixture'u gibi düşün, yani her bir yeni feature artık daha complex.

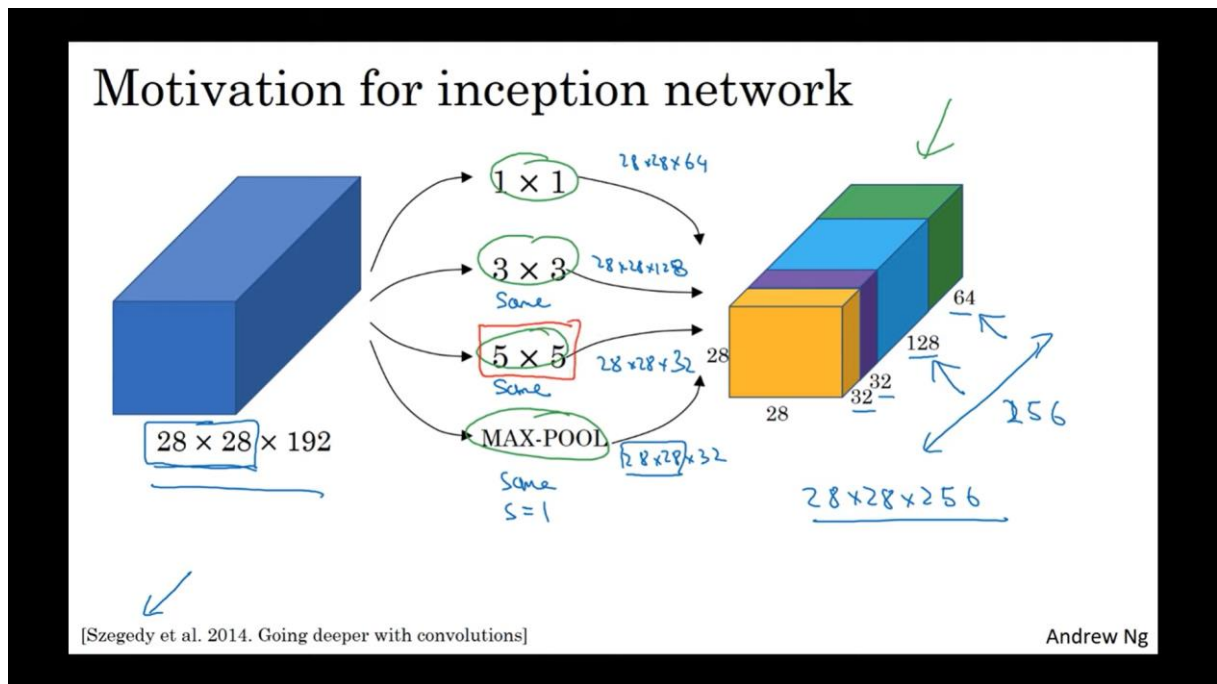
Vid 6

Inception Network Motivation

Bir ConvNet design ederken, filter boyutunu veya pooling layer koyup koymayacağımızı seçmemiz gerekir. Inception Network'un buna yaklaşımı ise şu: Neden hepsini yapmıyoruz? Tabii hepsini yaptığımızda, network architecture becomes much more complicated ama aynı zamanda works remarkably well.

Bir örnekle bunu anlamaya çalışalım, diyelim ki aşağıdaki gibi $28 \times 28 \times 192$ 'lik bir inputumuz var, inception layer şunu der, filtre boyutuna karar vermek yerine hatta conv layer mı pooling layer mı kullanacağımıza karar vermek yerine hepsini birden yapalım. Yani hem 1×1 , hem 3×3 hem 5×5 filtre kullanalım, ayrıca pooling de yapalım, bütün sonuçları bir volume olarak output alalım ve bu output ile yola devam edelim, hangisinin değerli olduğuna network karar versin.

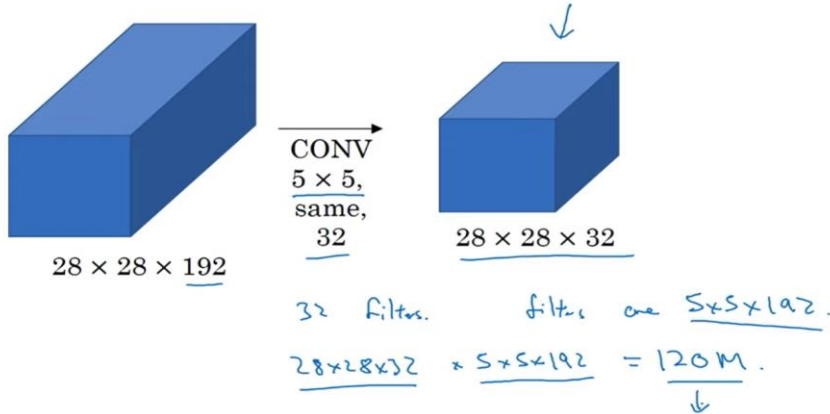
- $1 \times 1 \times 192$ filtre için diyelim ki 64 filtre kullanıldı ve sonuç $28 \times 28 \times 64$ olacaktır.
- $3 \times 3 \times 192$ filtreden 128 tane kullanalım, same convolution uygularsak sonuç $28 \times 28 \times 128$.
- $5 \times 5 \times 192$ filtreden 32 tane kullanırsak, same conv ile sonuç $28 \times 28 \times 32$ olur.
- Ayrıca belki conv layer değil pooling layer istiyoruz bunu da uygulayalım, ancak buradaki pooling height ve width azaltmayacak bunun için padding'li pooling uyguluyoruz, amaç sağ altta görülen volume'u elde edebilmek bu yüzden her sonuç $28 \times 28 \times K$ şeklinde olmalı, K herhangi bir sayı olabilir.



Sonuçta olay şu, filtre boyutunu veya pooling mi conv mu uygulayacağımızı seçmek yerine, hepsini uyguluyoruz ve network hangisinin değerli olduğuna kendi karar veriyor.

Yalnız bu yapının şöyle bir problemi var: Computational Cost. Bir sonraki slide'da 5×5 'lik filtre ile conv. işlemi için ne kadar computation gerektiğini anlamaya çalışalım, yani yukarıdaki mavi $28 \times 28 \times 192$ input'dan mor $28 \times 28 \times 32$ 'lik output'un elde edilme aşamasına odaklanalım.

The problem of computational cost



Andrew Ng

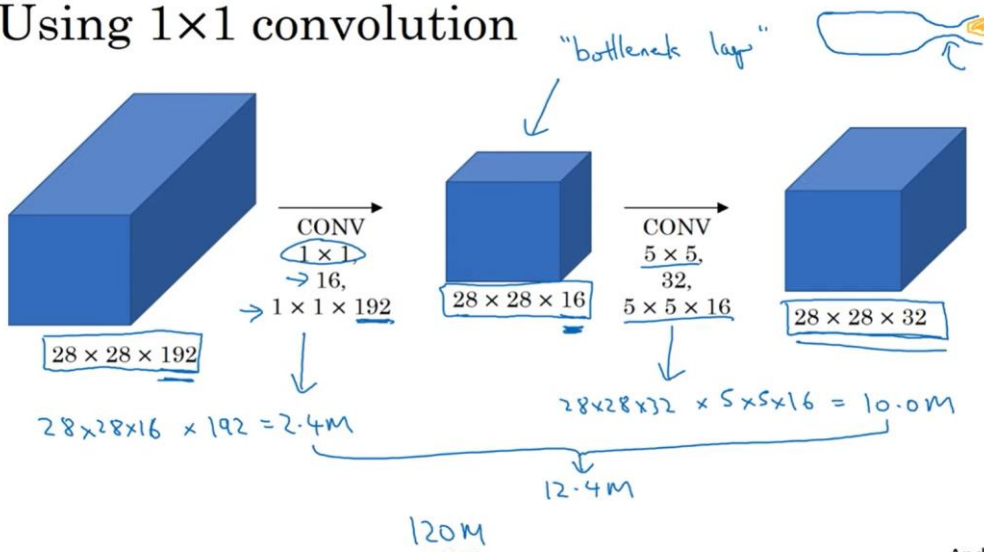
Yukarıdaki convolution'a yakından bakalım $28 \times 28 \times 192$ input'a $5 \times 5 \times 192$ 32 filtre ile same conv uyguluyoruz, sonuç $28 \times 28 \times 32$ çıkacak.

Bu işlem için computational cost'a focuslanalım:

- Elimizde $28 \times 28 \times 32$ tane output var. Her output için filtre ile convolution yapıldı.
- Yani her output için $5 \times 5 \times 192$ 'lik filtre ile aynı volume'a sahip bir input kısmı arasında elementwise multiplication oldu.
- Özete $28 \times 28 \times 32$ output'un her biri için $5 \times 5 \times 192$ tane multiplication yapıldı bu da totalde 120M civarı multiplication anlamına gelir.
- 120M multiplication is pretty expensive operation.

Bu noktada geçen kısımda öğrendiğimiz 1×1 convolution devreye giriyor ve bu computation cost'u 10 kat azaltıyor.

Using 1×1 convolution



Andrew Ng

1x1 convolution kavramının, inception network için gerekli computation cost'u nasıl remarkably düşürdüğünü yukarıda görüyoruz bunu anlamaya çalışalım:

- 28x28x192 input'dan doğrudan 5x5 filters ile 28x28x32 output elde etmek için 120M computation gerekiyordu, bundan kurtulmak için araya bir 1x1 convolution adımı ekleriz.
- 1x1x192 boyutunda 16 tane filtre ile convolution yaparsak arasonucun boyutu 28x28x16 olur, daha sonra bu ara sonucu 5x5 filtreler ile convolution uygularız.
- Bunu yapınca 120M yerine yaklaşık 12M multiplication ile işi çözeriz.

Yani 28x28x192'lik input'u alıp harmanladık ve shrink ettirdik, bu yüzden bu layer'a bazen bottleneck layer da denir.

Özetle inception network dediğimizde bir layer'da tek bir boyutta filtre ile convolution veya pooling uygulanması yerine hepsinin birden yapıldığı network'ten bahsederiz, böylece daha kompleks bir yapı elde ederiz, ancak iyi bir performans da elde ederiz. Network hangi filtre boyutunun vb. önemli olduğuna kendi karar verir.

Bu işlemin computational cost'u büyük bir dezavantajdır, bu computational cost'u significantly reduce etme noktasında 1x1 convolution fikri devreye girer. Performansı etkilemeden computation cost'u bariz şekilde azaltır.

Yukarıda görülen 3 adımlık kısım bir Inception Module olarak adlandırılabilir, bunlar biraraya geldiğinde Inception Network elde etmiş oluruz, bir sonraki kısımda bunu göreceğiz.

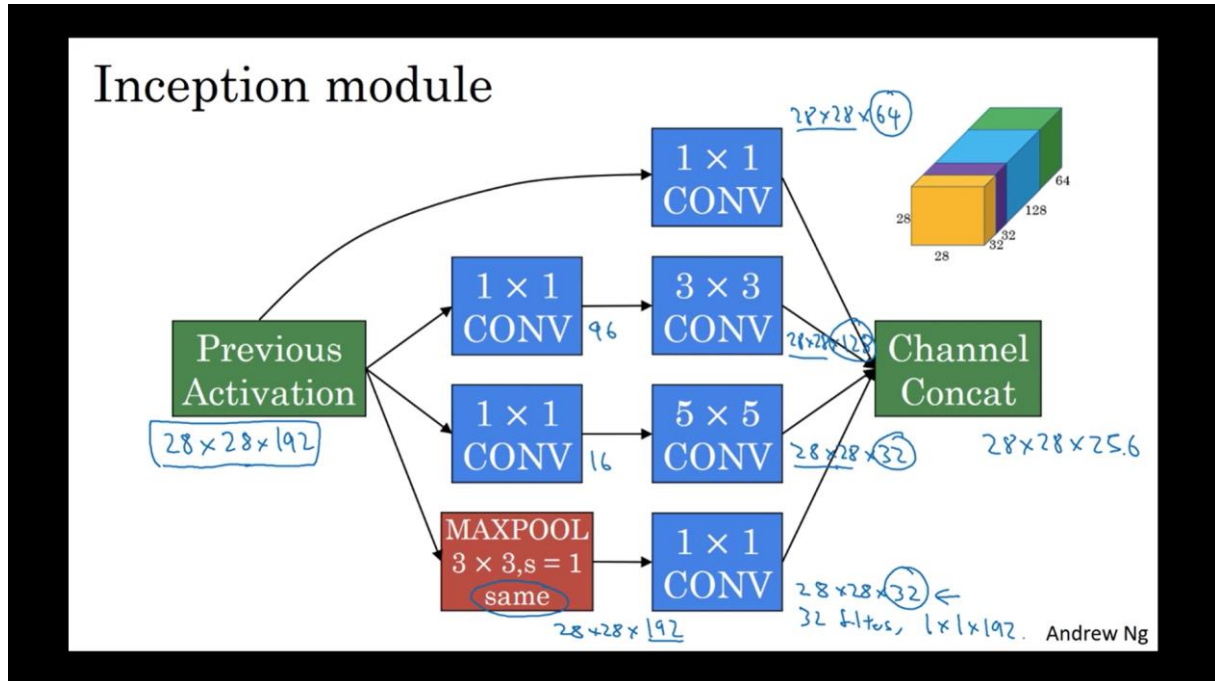
Vid 7

Inception Network

Önceki kısımda inception module'u yaratan fikirleri anlamıştık, bu kısımda da bir inception network nasıl oluşur ona bakacağız.

Aşağıda bir inception modüle yapısını görüyoruz:

- Inputumuz $28 \times 28 \times 192$ olsun, bunun için farklı filtreler kullanıp output'ları birleştiriyorduk.
- Örneğin 1×1 filtre kullanabiliriz bu durumda sonucu direkt elde ederiz.
- Veya 3×3 filtre ile conv. Yapabiliriz, bu durumda computation'dan kısmak için önce 1×1 convolution yapıyoruz
- Benzer şekilde 5×5 filtre için de aynı işlem yapılabilir.

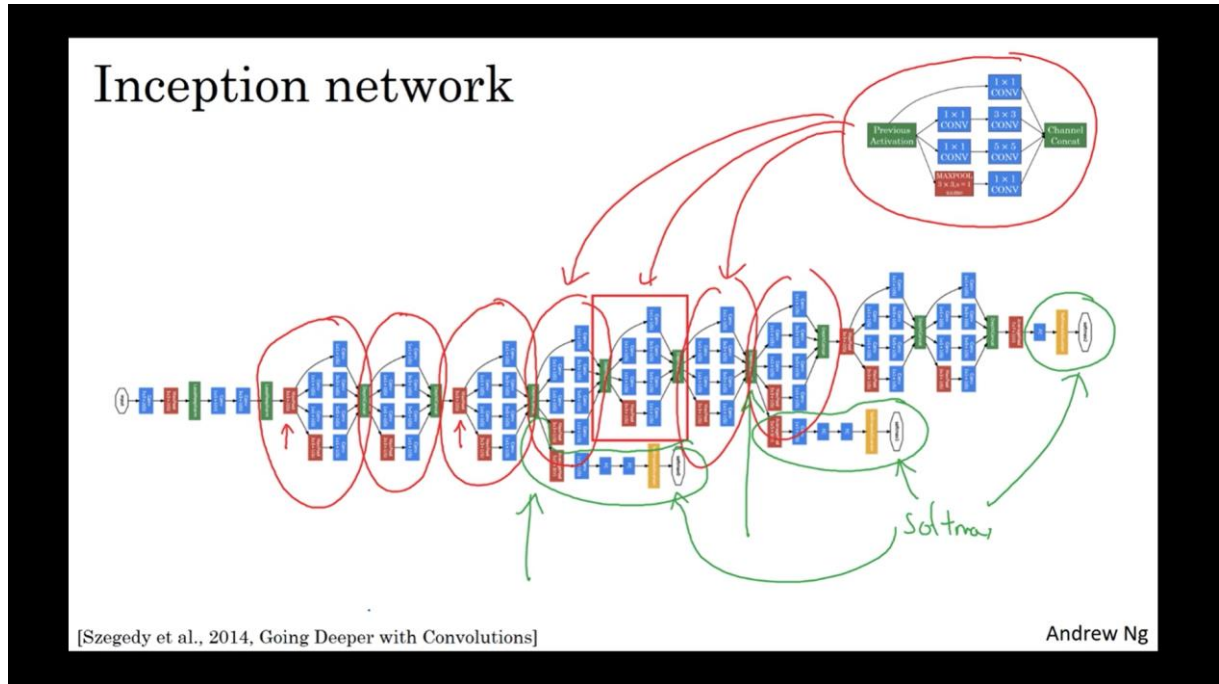


- Son olarak pooling de kullanırız ancak burada kullanılan pooling biraz farklı, öncelikle output'ları concatenate etmek istediğimiz için pooling'in dimension reduction etkisini istemiyoruz bu yüzden normalde yapılmayan padding ile pooling yapıyoruz.
- Stride 1 ve same padding ile 3×3 pooling yaparsak $28 \times 28 \times 192$ 'lik bir output elde edebiliriz.
- Ancak elde edilen bu pooling sonucunun 192 tane channel'ı var, bu çok fazla, işte bu noktada 1×1 convolution tekrar devreye girer ve bu kez de 1×1 convolution'ı channel reduction için kullanırız.
- Sonuçta $28 \times 28 \times 32$ 'lik bir pooling output elde edebiliriz.

Nihayetinde tüm bu output'lar concatenate edilip $28 \times 28 \times 256$ 'lık bir output'a ulaşılır. İşte burada görülen yapı bir Inception Module olarak isimlendirilebilir.

Inception Network'ün ise yaptığı temelde bu modülleri bir araya getirmekten çok da farklı değil.

- Aşağıda görülen her kırmızı kısım bir inception block/module.
- Output için ise tahmin edebileceğimiz gibi fully connected layer ile softmax layer kullanılmış.
- Ayrıca original paper'a bakarsak inception network'de olan bir şka yapı yeşil ile gösterilen side branches.
- Side branch'in yaptığı şey ise hidden layer activations'ı alıp predictions yapmak, bu yüzden burada da fully connected layers ve softmax kullanılıyor. Bu işlemi bir detay gibi düşün, yani arada hesaplanan feature'lar ile output'un predict edilemediği control ediliyor bir nevi regularization sağlanmış oluyor.



Bu yapıya inception network denmesinin sebebi ise, authorların çıkış noktasının aşağıdaki meme olması, filmi izleyen için anlaşılır, Leo'nun zaman kazanmak için bir sonraki katmana geçmeleri gerektiğini belirttiği sahne.



Vid 8

Using Open Source Implementation

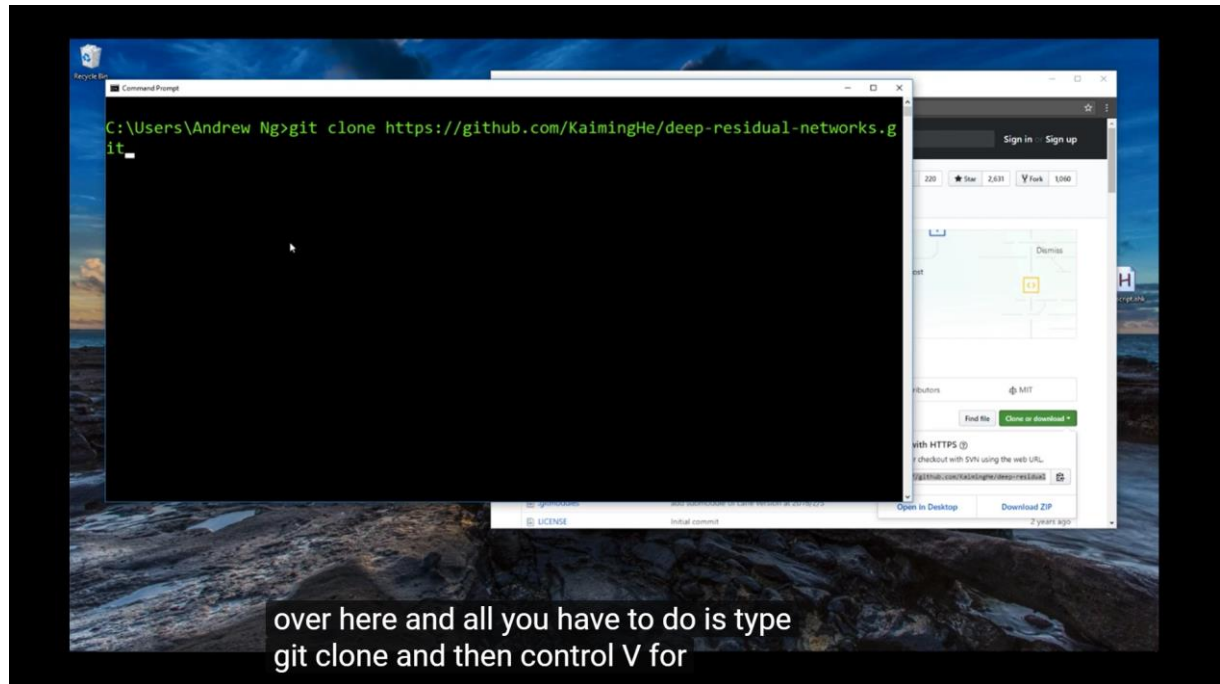
Artık bir çok highly effective NN ve ConvNet architectures öğrendik. Bu haftanın kalan konularında bunları nasıl kullanacağımıza dair practical advices verilecek.

Bu kısımda open source uygulamaları kullanmaktan bahsedeceğiz. Gördüğümüz uygulamalar bir çok hyperparameter tuning işlemi gerektirir ve en iyi üniversitelerde PhD. Öğrencileri için bile sadece research paper'a bakarak uygulama yapmak gerçekten zor olabilir. İşte bu noktada kodlarını github'da paylaşan deep learning uzmanlarının işlerinden faydalanabiliriz.

Yani bir research paper'ı implement etmeden önce github'a bir bak diyor, zaten implement edilmiş hali veya bir benzeri olabilir, onu alıp üzerine devam etmek büyük zaman kazandırır.

Kalan kısımda github'dan nasıl repository indireceğini gösteriyor.

- Repo'ya git.
- Clone or download.
- Copy the URL
- Git clone ile URL'den repoyu çek.



Zaten bildiğin bir süreç.

Vid 9

Transfer Learning

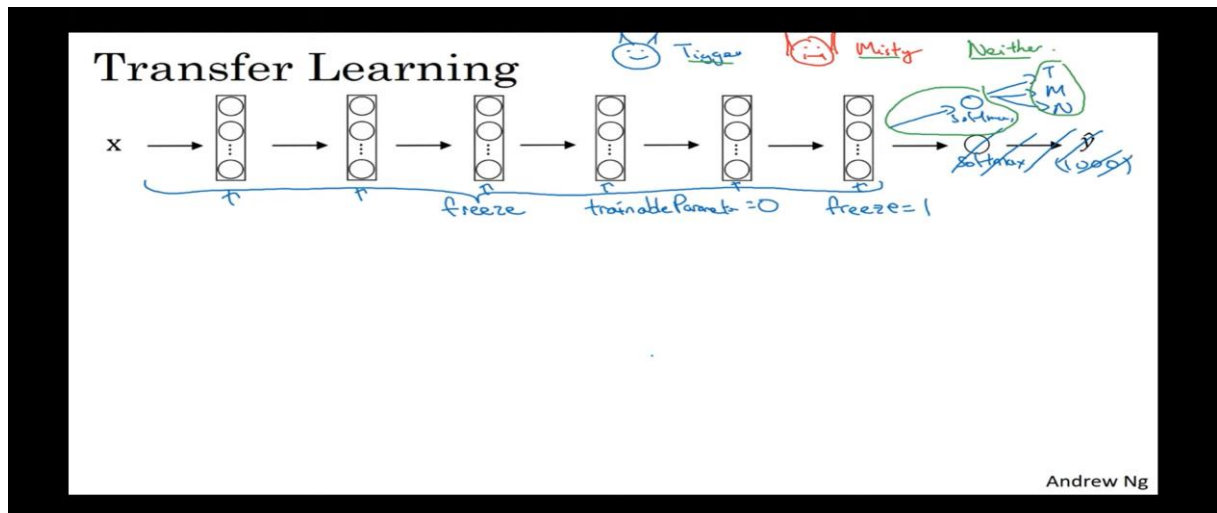
Bir Computer Vision application build ederken, weightleri random initialize edip sıfırdan eğitmektense, daha önce birinin benzer bir task için eğittiği bir architecture'ı alıp bu weightleri pre-training weight olarak kabul ederiz ve modeli yeni task için tekrar eğitebiliriz (fine-tuning), bu süreç çok daha hızlı olacaktır.

Course3W2'de bu sürecin transfer learning olarak adlandırıldığını zaten görmüştük.

Bazen bir modeli eğitmek haftalar veya aylar alabilir, hem de çok iyi bir computation power ile bu noktada bu süreci baştan yaşamaktansa, eğitilen modelin weightlerini kullanarak çok büyük bir yükten kurtulabiliriz.

Bir örneğe bakalım. Diyelim ki bir cat classifier eğitmek istiyoruz, "Tigger tipi kedi" – "Misty tipi kedi" veya "Hiçbiri" şeklinde 3 çıkışı olacak.

- Böyle bir örnek için büyük ihtimalle elimizde çok büyük bir training set olmayacaktır, çünkü tigger ve misty ve neither etiketi ile data bulmak kolay değil.
- E bu durumda ne yapacağız? Yeterli datamız olmazsa muhtemelen modelimiz kısıtlı dataya overfit edecek. Bu problemi çözmemiz lazım.
- İnternette başka bir Computer Vision problemi için eğitilmiş network'u alırız. Çıkış layer'ını atarız ve kendi kullanımımıza göre yenisini koyarız.
- Mesela 1000 class için eğitilen bir modeli alıp, son softmax layer'ı yerine kendi 3 class'lı softmax layer'ımızı yerleştirebiliriz.
- Daha sonra, önceki layer'ları dondururuz yani weightlerini sabit tutarız sadece yeni eklenen çıkış layer'ının weightlerini eğitiriz
- Böylece amacımız transfer learning ile önceki modelin elde ettiği low level intuition'ları ve bir image'dan features çıkarabilme becerisini koruyarak sadece bu feature'ları başka bir amaç için kullanmak.
- Böylece bizim yapacağımız uygulama için görece küçük bir datasetimiz olsa dahi bu data set tek bir layer'ı eğitmek için yeterli olacaktır, önceki layer'ların eğitimi zaten başka bir data ile yapılmış şekilde elimize geldi.
- Bu tip bir operasyonu bir çok Deep Learning framework destekler, bahsedilen weight freezing işlemi için "trainable=0" "freeze=1" gibi bir parametre set etmek yeterli olacaktır.



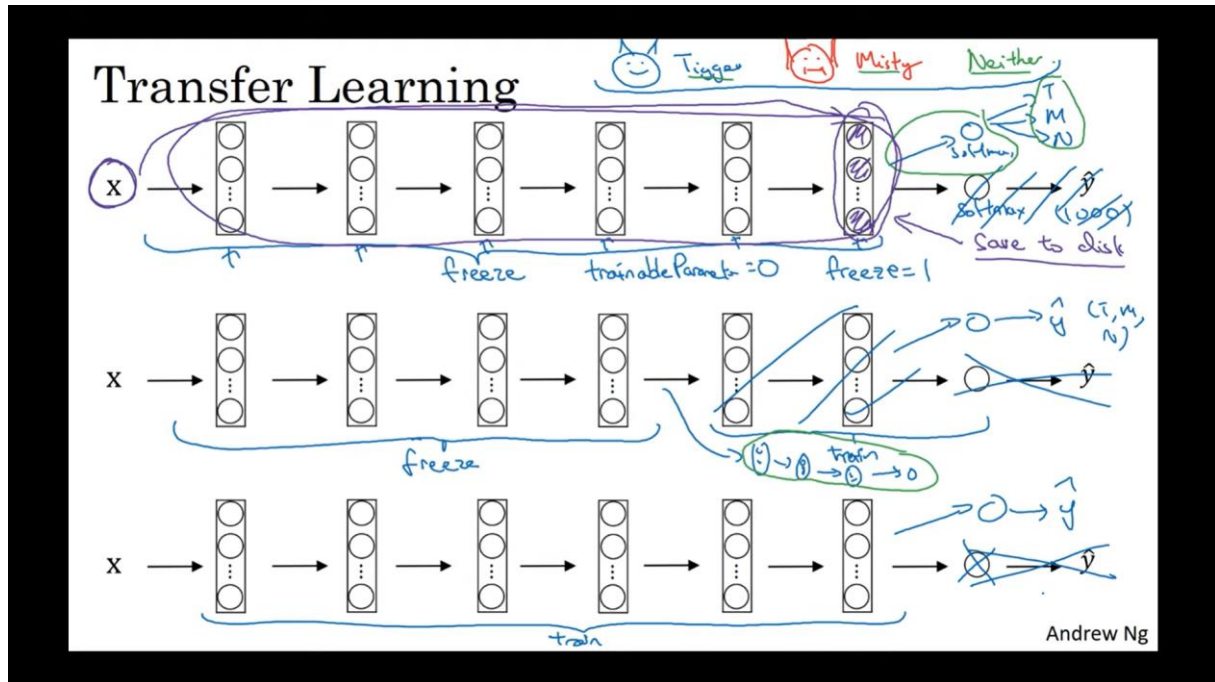
Yukarıdaki örnek için computation'ı hızlandırmak için uygulanan bir trick de şu:

- Önceki weightleri freeze edip tüm network'u eğitmektense önce freeze edilen kısmı kullanarak tüm inputlar için outputs bulunur.
- Daha sonra bu outputs kullanarak sadece softmax layer eğitilir.
- Bu yöntem pratikte daha hızlı oluyor.

Sonuçta yukarıdaki cat classification örneği için elimizde az data vardı ve sıfırdan bir model eğitmemiz zordu bu yüzden, daha fazla data ile eğitilen bir başka computer vision modelini baz olarak kullandık bu modelin feature öğrenbilme yeteneğinden yararlanarak modeli modifiye ettik ve kendi uygulamamıza uyarladık.

Peki ya yapacağımız uygulama için elimizde yeterince datamız var ise?

- Bu durumda dataset büyüklüğüne bağlı olarak yapılabilecek birkaç şey var.
- İlki şu: bu kez alınan modelin sadece son layer'ını değil sondan birkaç layer'ını koparırsınız ve yerine randomly initialized kendi layerlarınızı koyarsınız, bunları eğitiriz.
- Bunu şöyle düşün data sayısı arttığı için artık tek bir layer yerine birden fazla layer eğitiyoruz.
- Buna benzer olarak sondan birkaç layer'ı kesmek yerine sadece son layer'ı keseriz, ancak ondan önceki birkaç layer'ı da freeze etmeyiz ama randomly initialize de etmeyiz, transfer edilen weightler ile initialize edip eğitiriz.



- Son olarak eğer elimizde yeterli dataset'in var olduğunu düşünüyorsak tüm layerları eğitebiliriz, burada transfer etmemizin amacı sadece modele bir pretraining sağlamak olmuş oluyor, bu pre-training üzerine fine-tuning yapıyoruz.

Vid 10

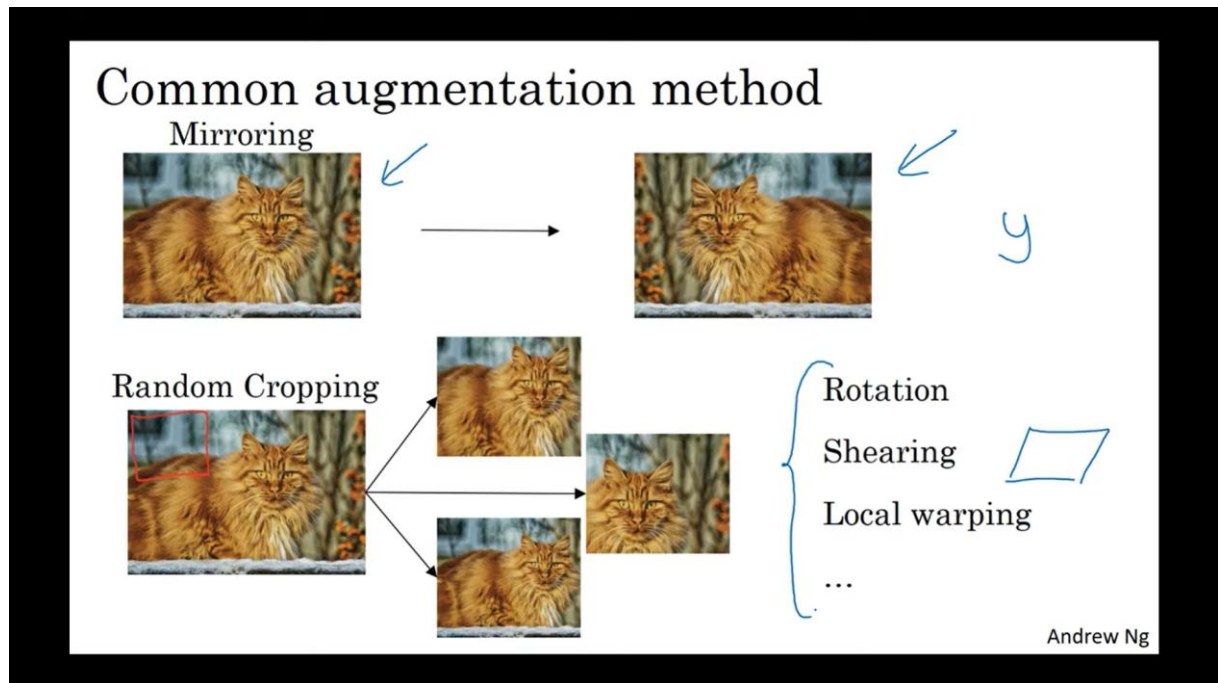
Data Augmentation

Computer Vision uygulamalarının birçoğu daha fazla data'ya hayır demez. Her zaman data'ya açtılar. Daha fazla data neredeyse her zaman performansı iyileştirir. Bazı diğer domainler için data açlığı bu kadar fazla olmayabilir bir noktadan sonra yeterli veriye ulaştığımızı söyleyebiliriz.

Ancak computer vision oldukça karmaşık bir iş, bilgisayarın pixellere bakarak fotoğrafta ne olduğunu söylemesi için inanılmaz kompleks bir fonksiyon öğrenmesi gerek. İşte bu yüzden neredeyse her zaman daha fazla data performansı iyileştirir.

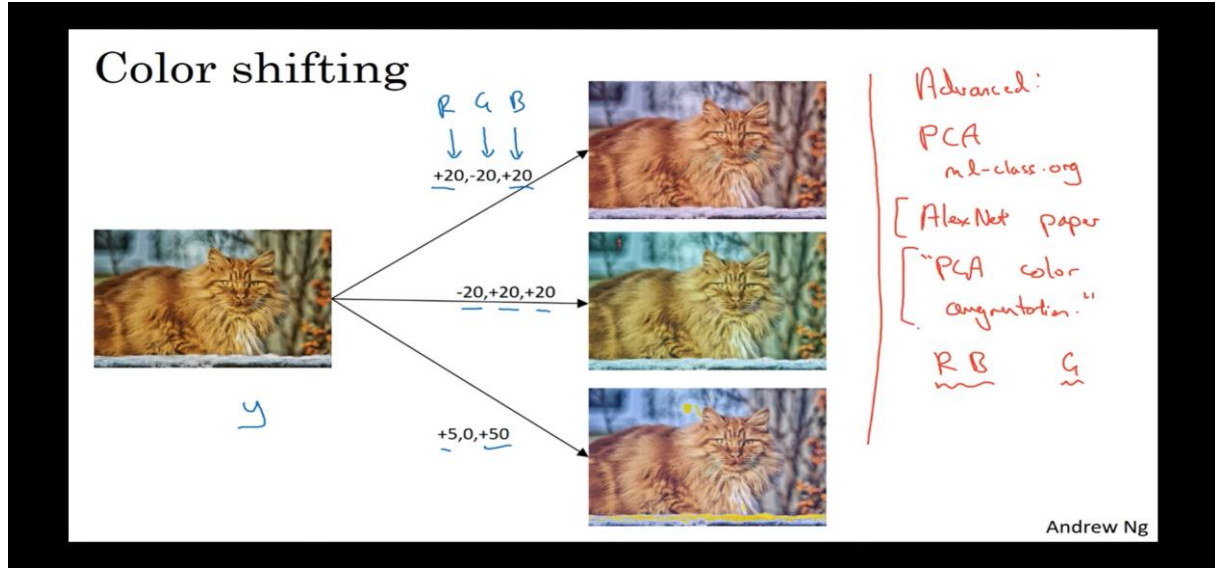
Data Augmentation da temelde modelin data açlığını gidererek performansı iyileştirmeye yönelik sıkça kullanılan bir teknik. Bu kısımda sık kullanılan data augmentation techniques'e bakacağız.

- En popüler data augmentation yöntemlerinden biri vertical mirroring.
- Bir diğer yöntem de random cropping'dir. Aşağıda görüldüğü gibi orijinal image'dan random parçalar kesilir.
 - Ancak bu yöntem çok da iyi bir yöntem olmayabilir, çünkü ya gidipte aşağıda kırmızı ile görülen kedinin sırt kısmını kroplarsak ne olacak? Öyle yada böyle bu method da pratikte işe yarıyor.
- Bunların yanında rotation, shearing, local warping gibi yöntemler de data augmantation için kullanılabilir.



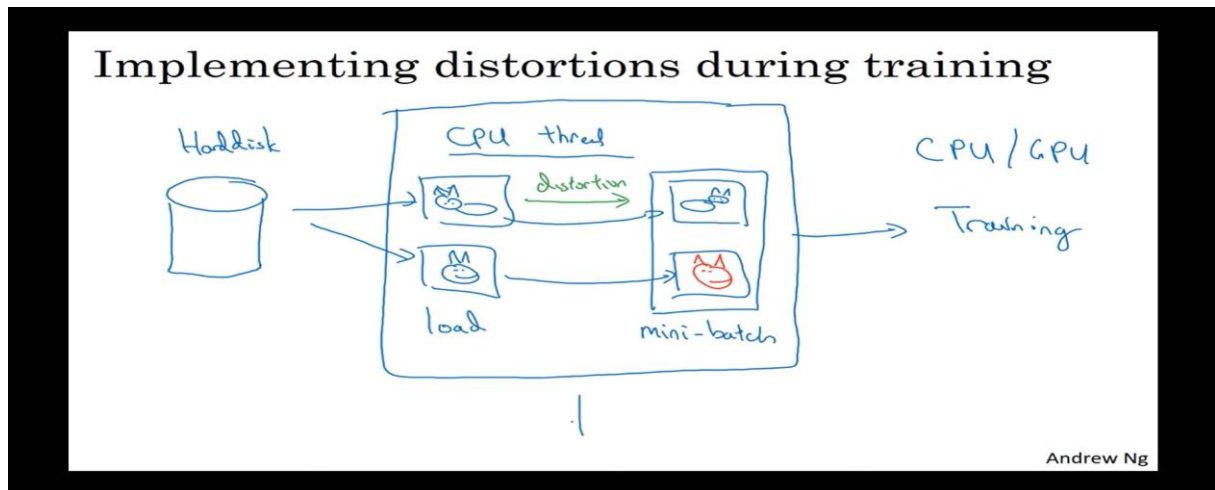
Bir diğ er data augmentation tipi ise Color Shifting'tir.

- Ařağıda verilen input'un RGB channellarına ekleme veya  ıkarma yapılırsa sağıda g r len farklı renklerde yeni images oluřturur.
- B ylece eğıtilen model, inputlarda g r len color difference i in more robust olur.
- Bu iřlem i in PCA Color Augmentation kullanılabilir, detayına istersen bakabilirsin.



Eğıer elimizde  ok b y k bir training set var ise, bu data augmentation iřlemi genelde thread ile eğıtime paralel řekilde yapılır.

- Datanın harddiskte tutulduğıunu varsayalım.
- Bir CPU thread ile harddiskten s rekli datalar alınıyor, bunlara data augmentation yapılıyor ve bir mini-batch oluřturuluyor.



- Aynı anda bir bařka thread ile de training ger ekleřiyor. Eğıtim CPU ya da GPU'da yapılabilir, dataset b y kse genelde GPU'da yapılır.

Sonuçta data augmentation genelde bu řekilde paralel thread'ler ile uygulanır.

Data augmentation i in de bazı hyperparameter vardır bu y zden bařkasının implementationından yararlanmak mantıklı olabilir.

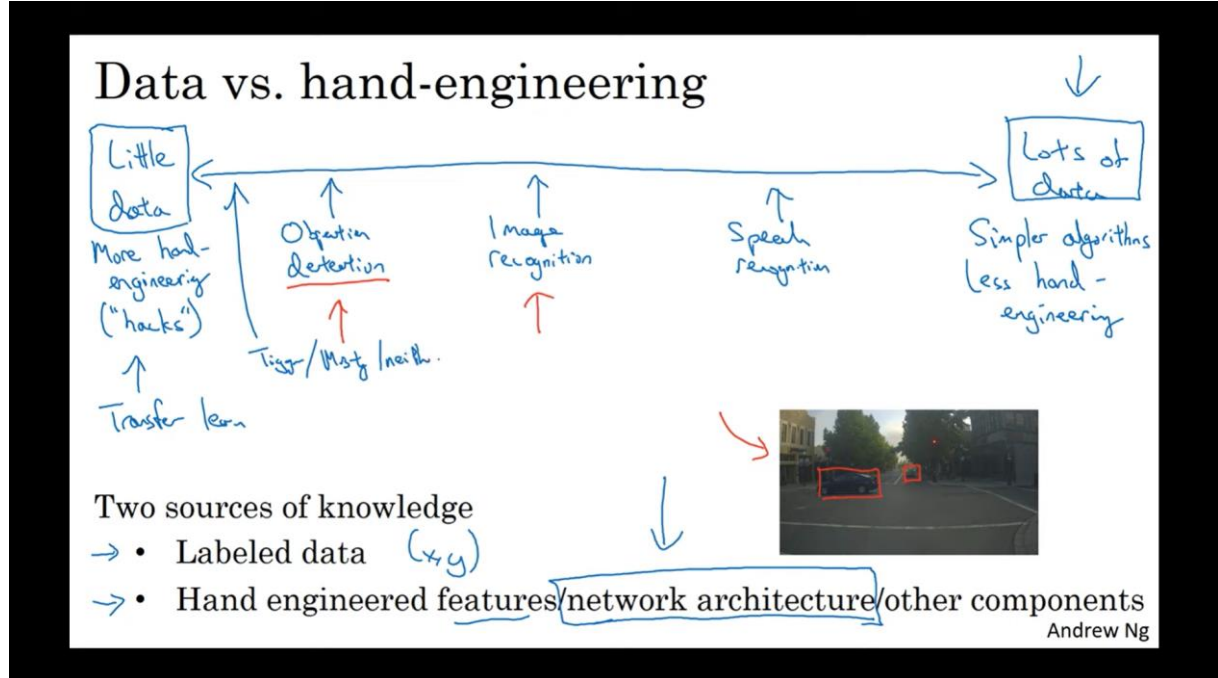
Vid 11

State of Computer Vision

Bildiğimiz gibi DL birçok farklı alana rahatça uygulanabiliyor. Bu kısımda anlayışımızı geliştirmek adına bazı bilgiler verilecek.

ML problemleri aşağıda görülen skalaya yerleştirilebilir. Skalanın sol tarafı ilgili problem için yeterli data'nın olmadığını temsil ederken sağ tarafı ise yeterli datanın olduğunu gösteriyor.

- Örneğin speech recognition problemi kendi complexity'si hesaba katıldığında sağ uca yakın bir kısımda yer alır, bu konuda yeterli dataya sahibiz.
- Günümüzde Image recognition problemi için tonlarca data olsa da problem karmaşık olduğu için hala daha fazla data isteriz, bu yüzden skalada sol tarafa yakın bir konumda.
- Bu bağlamda object detection ise daha solda kalacaktır, çünkü object detection örneği image recognition'dan daha zor. Fark şu: object detection için datasette ground truth'u gösteren rectangles olmalı, böyle örnekleri toplamak cat/not cat gibi örnekleri toplamaktan daha zor.
- Eğer problem için elimizde yeterli data varsa, görüyoruz ki daha basit algoritmalar kullanarak ve daha az hand-engineering ile iyi performanslar sağlayabiliyoruz.
- Buna karşın ilgili problem için elimizde yeterli veri yoksa, bu durumda da problemi çözebilmek için çok daha fazla hand-engineering kullanmak durumunda kalıyoruz.



Bir modelin öğrenebileceği iki kaynak var.

- Birincisi labeled data.
- İkincisi ise, hand engineering. Yani burada kastedilen, mühendisin kendi deneyimleri ve/veya insightlarını modele yansıtması, mesela features seçmesi, mesela probleme uygun architecture kullanması vb.

Eğer elimizde yeterli data varsa, düzgün bir model mühendisin ekstra bir çabasına gerek kalmadan, istenilen performansı sağlayacaktır. Ancak eğer, elimizdeki veri yeterli değilse ki computer vision için genelde yeterli olmaz, işte bu noktada mühendisin devreye girmesi gerekir ve öğrenme sürecine başlamadan veya başladıktan sonra müdahalelerle bir nevi modele kendi bildiklerini öğretmesi gerekir.

Neyse ki elimizde az data olduğunda yardımımıza koşan kavramlardan biri Transfer Learning. Böylece ilgili problem için elimizde az data olsa bile, benzer başka bir dataset üzerinde elde edilen modelin kazandığı bilgiyi kendi modelimize aktarabiliyoruz.

Piyasada bir çok insanın standardlaşmış benchmark dataset üzerinde iyi performans sergilemek ve yarışma kazanmak konusunda çok tutkulu olduğunu görebiliriz.

Genelde insanlar buna ilgi duyar çünkü, eğer bu standardlaşmış benchmark dataset üzerinde iyi performans sergileyen bir model geliştirirsek, yayın yapman kolaylaşır yani insanlar seni dikkate alır.

Tips for doing well on benchmarks/winning competitions

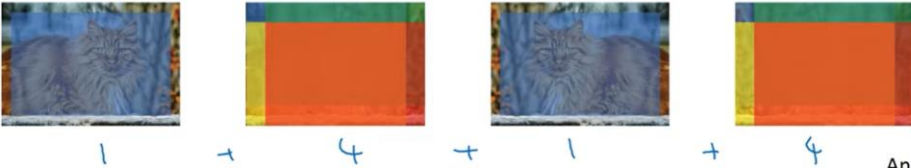
Ensembling 3-15 networks → \hat{y}

- Train several networks independently and average their outputs

Multi-crop at test time

- Run classifier on multiple versions of test images and average results

10-crop



Andrew Ng

Ancak bu takıntının bir dezavantajı var, insanlar bu veriler üzerinde iyi sonuçlar elde etmek adına bazı trickler yaparlar ve bu trickler, pratikte gerçek bir uygulama yapacağımızda işe yaramaz.

Şimdi burada bu tiplerden bahsedeceğiz, Andrew bunları kendi pratik uygulamaları için kullanmadığını söylüyor ancak, yarışma kazanmak için kullanılan bazı methodlar şunlar:

- **Birincisi "Ensembling"**: Bunu basitçe çoğunluğun gücü olarak düşünebiliriz, aynı problem için birden fazla (3-15) farklı network eğitiriz ve hepsinin tahminlerinin ortalamasını alırız, genelde tek bir modele göre daha iyi sonuçlar elde ederiz.
- Ancak her network için ayrı ayrı tahmin almamızı gerektirdiği için, uygulamanın network sayısı ile orantılı olarak yavaşladığını unutma. Bu yüzden pratikte kullanmak

pek mantıklı olmayabilir, belki performans %1-2 iyileşektir ancak tek bir örneğin sonucunu hesaplamak için çok fazla computational cost gerekecektir.

- Bir diğer trick ise “Multi-crop at test time”, sonucu elde ederken input image üzerinde farklı crop'lar için sonuç hesaplanır ve ortalamaları alınır. 5 crop test image'in kendisinde 5 crop ise mirror'ında alınır ve sonuç hesaplanır.
- Yine biraz daha iyi sonuç elde edebiliriz ancak, crop sayısı kadar daha fazla computation power gerekeceğini de unutma.

Ayrıca aşağıdaki maddeler de önemli.

- Bir problem için özelleşmiş computer vision modeli genelde bir başka problem için de gayet iyi çalışır, bu yüzden transfer learning kavramı çok değerli, kullanmayı unutma.
- Eğer bir problem için model geliştirmek istiyorsak, önce literatürde yayınlanan paperlara bakabiliriz.
- Daha sonra bunların open source uygulamalarına bakabiliriz.
- Bir model bulunca da, kendi problemimiz için elimizdeki datanın boyutuna göre, ya modeli pretrained model olarak kullanıp kendi problemimiz için tekrar eğitiriz, veya datamız az ise modelin output layer'ını belki birkaç layer'ını daha keseriz ve kendi layerımızı yerleştirerek, sadece yeni layerları eğitebiliriz.

Use open source code

- Use architectures of networks published in the literature
- Use open source implementations if possible
- Use pretrained models and fine-tune on your dataset

Andrew Ng

