

COURSE4 – W3

Vid 1

Object Localization

Bu hafta Object Detection konusunu işleyeceğiz. Computer Vision'ın alanlarından biri olan Object Detection son yıllarda çok popüler ve eskiye göre çok daha iyi çalışıyor.

Object Detection öğrenmeden önce Object Localization'dan bahsetmeliyiz. Önce bazı tanımları yapalım ki kavramlar kafamıza daha iyi otursun.

- **Image Classification:** Bu problemde input image alınır ve algoritma bunu sınıflandırır. (Car/Cat/Bird gibi)
- **Classification with Localization:** Bu problemde ise algoritma inputu sınıflandırmakla kalmıyor, tespit ettiği sınıfın input image'da tam olarak nerede bulunduğunu işaretliyor.
- **Detection:** Bu problemde ise input içinde birden fazla object olur ve algoritma ilgili objeleri detect etmekle kalmayıp her birini localize eder.

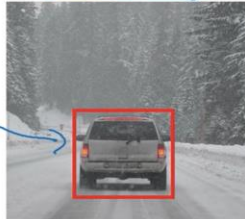
What are localization and detection?

Image classification



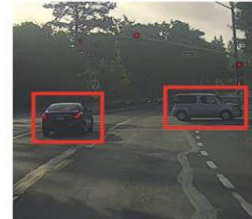
"Car"

Classification with localization



"Car"

Detection



multiple
objects

Andrew Ng

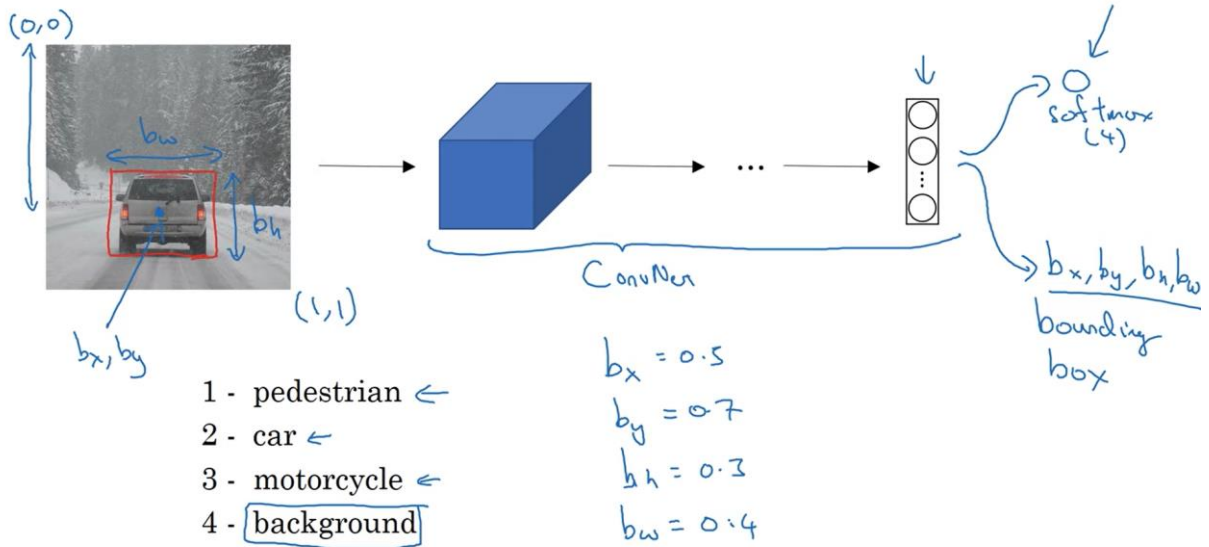
İlk iki tip problemde genelde bir input içinde tek bir object olur ve algoritma bunu sınıflandırmaya ve/veya localize etmeye çalışır. Buna karşın detection probleminde birden fazla obje (tek çeşit veya farklı çeşitlerde – car/pedestrian/motorcycle gibi) olur ve algoritma hepsi için hem sınıflandırma hem localization yapar.

Şimdiye kadar öğrendiğimiz classification bilgileri "classification with localization" problemini anlamamızda yardımcı olacak, daha sonra da bu problemi anlamamız "detection" problemini anlamamızda bize yol gösterecek.

Image classification'ın nasıl yapıldığını geçtiğimiz iki haftanın konusunda zaten gördük.

Hızlıca açıklamak gerekirse, diyelim ki amacımız fotoğrafları 4 farklı class'a ayırmak (Pedestrian/Car/Motorcycle/Background) aşağıdaki gibi bir input image bir ConvNet'e alınır, ve 4 unitli bir softmax layer ile output alınır. Bu yapıda bir ConvNet eğiterek başarılı bir şekilde classification yapılabilir.

Classification with localization



Andrew Ng

Peki ya probleme localization da dahil olursa? Yani hem classification yapılması hem de ilgili objenin etrafına bir bounding box çizilmesi gerekiyorsa?

Bu durumda output layer'a ekstra 4 unit daha ekleyebiliriz. Böylece network'ün class tahmininin yanında, ilgili objenin merkez koordinatları (b_x, b_y) ve height ile width (b_h, b_w) bilgilerini de tahmin etmesini bekleyebiliriz.

Aşağıda target label Y'ın tam olarak nasıl specify edileceğini anlamaya çalışalım.

Sonuçta ilgili objeyi localize edebilmek için bir rectangle koordinatına ihtiyacımız var, bu yüzden de b_x, b_y, b_h, b_w koordinatlarını kullanıyoruz, ilk ikisi rectangle'ın merkezini tarif ediyor, diğer ikisi ise width ve height bilgilerini.

Bu 4 unit output label'a dahil edilecek peki hem classification hem de localization için output label nasıl hazırlanacak?

Aşağıda gördüğümüz gibi hazırlanacak:

- Toplam 8 output unit olacak, 4 tanesi class detection için (Pedestrian/Car/Motorcycle/Background) ve diğer 4 tanesi de localization için.
- Output'un ilk unit'i is there any object sorusuna cevap veriyor yani bir başka deyişle 4. sınıf olan background'ı tespit ediyor. (0 ise background, 1 ise object)
- Eğer bu ilk unit 1 ise demekki bir object detect edildi, o zaman bunun hangi object olduğu ve localization rectangle'ın bilgileri de tahmin ediliyor.
- Ancak bu ilk unit 0 ise ortada localize edilecek bir object yok demektir, yani kalan 7 unit'in çıktısının hiçbir önemi yok.

Defining the target label y

Need to output b_x, b_y, b_h, b_w , class label (1-4)

1 - pedestrian
2 - car
3 - motorcycle
4 - background

$L(\hat{y}, y) = \begin{cases} (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \dots + (\hat{y}_n - y_n)^2 & \text{if } y_1 = 1 \\ (\hat{y}_1 - y_1)^2 & \text{if } y_1 = 0 \end{cases}$

$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$ is there any object?

(x, y)

Andrew Ng

Ne demek bu kalan 7 unit'in çıktısının önemi yok? Ne için önemi yok? Hem prediction sonucu açısından önemi yok hem de loss hesaplarken önemi yok.

- Eğer $y_1=1$ ise yani object localize edildiyse, bir şekilde her unit için loss hesaplanabilir ve bunlar toplanabilir.
- Ancak $y_1=0$ ise yani object yok ise, diğer unitlerinin sonuçları loss'a dahil edilmemeli sadece y_1 loss'a dahil edilmeli!!!

Son olarak yukarıda her unit için squared loss kullanılmış ancak, gerçekte farklı loss'lar kullanılabilir mesela c_1, c_2, c_3 unitleri için log likelihood (softmax) error, rectangle coordinates için squared error, p_c için ise logistic regression loss gibi.

Vid 2

Landmark Detection

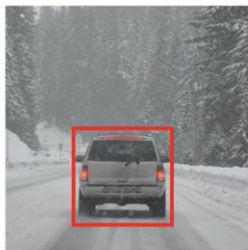
Önceki kısımda, localization problemi için bir NN'in nasıl 4 farklı output unit ile rectangle koordinatları verebileceğini gördük.

Bu yaklaşımı genellersek, bir input image üzerinde önemli olan noktaların X ve Y koordinatlarını output olarak veren NN'ler eğitilebilir. Bu önemli noktalara "lanmarks" da denir.

Aşağıda bu kavramı daha iyi anlamaya çalışalım.

- Car detection için NN b_x, b_y, b_h, b_w şeklinde 4 koordinatı output verecek şekilde eğitiliyordu.
- Bir başka örnek için verilen face image input'ları üzerinde bizim için önemli noktaların işaretlenmesini istediğimizi düşünebiliriz. Yani göz, burun, ağız ve çene sınırlarının koordinatlarının output olarak verilmesini isteyebiliriz.
- İşte bu önemli noktalar landmark olarak adlandırılır, sonuçta aynı car localization örneğinde 4 rectange koordinatı kullandığımız gibi burada da NN'in output layer'ına gerektiği kadar unit ekleyerek istediğimiz işi yapacak bir network eğitebiliriz.

Landmark detection



b_x, b_y, b_h, b_w



$\left. \begin{matrix} l_{1x}, l_{1y}, \\ l_{2x}, l_{2y}, \\ l_{3x}, l_{3y}, \\ l_{4x}, l_{4y}, \\ \vdots \\ l_{64x}, l_{64y} \end{matrix} \right\} X, Y$



$\left. \begin{matrix} l_{1x}, l_{1y}, \\ \vdots \\ l_{32x}, l_{32y} \end{matrix} \right\}$

Andrew Ng

- Bu tür uygulamalar ile yüz sınırları tespit edilebilir ve bu bilgi belki instagram filters gibi uygulamalarda işlevsel olabilir, yani bu noktalara göre mesela kişiye taç eklenebilir vesaire.
- Son olarak aynı mantık belki kişinin posture'unu detect etmek için kullanılabilir, vücut üzerinde bazı key noktalar landmarks olarak belirlenir ve network bu noktaları tespit edecek şekilde eğitilirse, kişinin posture'unu tespit edebiliriz.

Vid 3

Object Detection

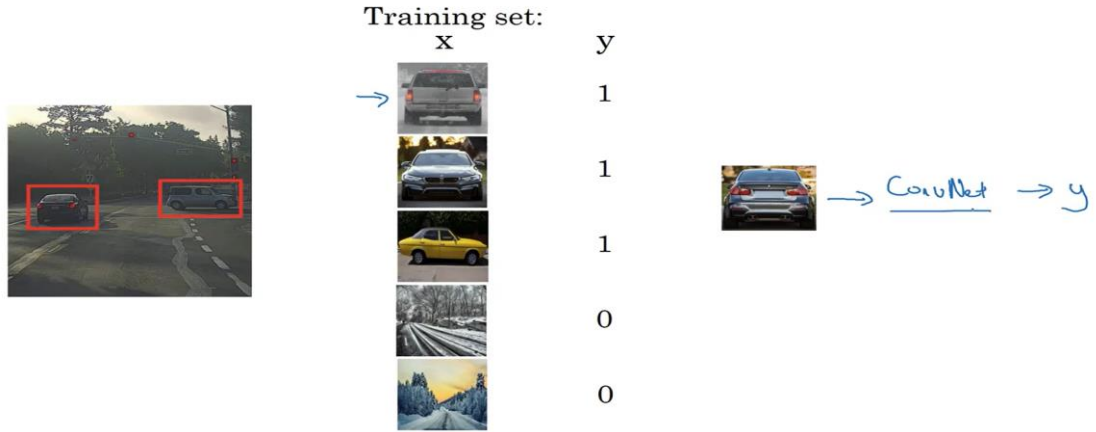
İlk iki kısımda object localization ve landmark detection kavramlarından bahsettik. Bu kısımda ise Object Detection problemine bakalım.

Bu kısımda, "sliding window detection" kavramından yararlanarak ConvNets kullanarak nasıl object detection yapılabileceğinden bahsedilecek.

Diyeelim ki bir car detection algorithm yapmak istiyoruz, şöyle bir yaklaşım geliştirebiliriz:

- Aşağıda görüldüğü gibi closely cropped car examples ile bir training set oluştururuz
- Daha sonra bu training set üzerinde bir ConvNet eğitebiliriz. Bu ConvNet aşağıda en sağda görüldüğü gibi bir inputu alacak ve prediction yapacaktır.

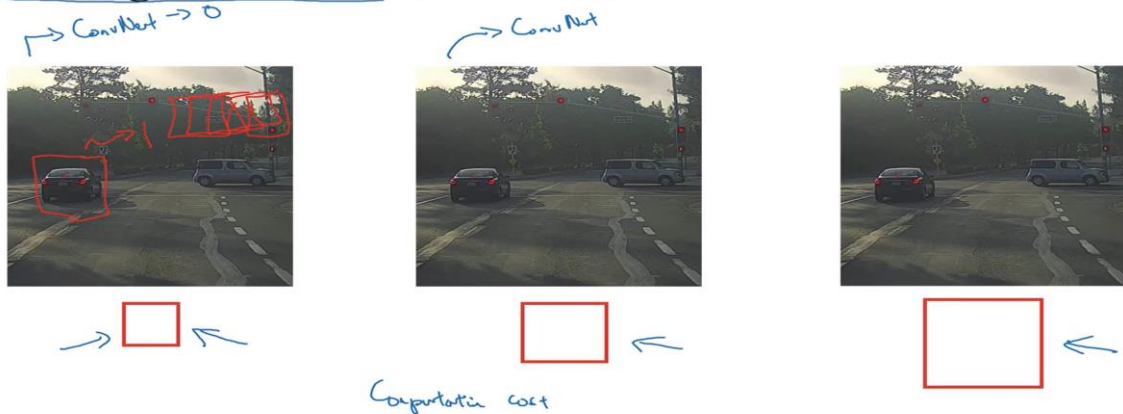
Car detection example



Andrew Ng

- Tam da bu noktada "sliding window detection" kavramı devreye girer.
- Yani diyelim ki alttaki gibi bir input image üzerinde car detection yapmak istiyoruz, bu durumda bir window size belirlenir ve bu input image içinden bu window size kadar ki kısım alınarak, ConvNet'e input olarak verilir ve output'a bakılır.

Sliding windows detection



Andrew Ng

- Sonuçta ConvNet bize, inputun ilgili region'ında bir car var mı yok mu onun cevabını verir.
- Bu window tüm input image boyunca slide edilir ve image'ın her yeri ConvNet'e input olarak verilir, bu işlem sonucunda büyük input image'ın bazı frame'leri için car detection gerçekleşecektir.
- Daha sonra aynı işlem biraz daha büyük bir window size için tekrar edilir.
- 3. Kez daha da büyük bir window size için aynı işlem tekrarlanabilir.

Sonuçta bu yaklaşımla, eğer input image içinde bir yerlerde car var ise, ConvNet bu frame'leri 1 olarak outputlayacaktır, biz de car bulunan lokasyonları kaydetmiş olacağız ve sonuçta car detection gerçekleşmiş olacak.

Ancak bu işlemin çok büyük bir dezavantajı var: Computational Cost.

Sonuçta bir car detect edicem diye, didik didik tüm fotoğrafı farklı window size için cropluyorum ve her birini bir ConvNet'e input olarak veriyorum, ayrıca hassas bir detection için window stride'ın da küçük olması gerekiyor, yani sonuçta çok fazla computation power gerekiyor.

Neural Networks yükselmeden önce, object detection için kullanılan classifiers çok daha basitti, bu yüzden her sliding window ile her crop için ayrı ayrı sonuç elde etmek ve bu yöntemle detection yapmak, computational olarak çok pahalı değildi.

Ancak ConvNet kullanıldığı zaman, tek bir crop için yapılan classification işlemi çok daha computationally expensive.

Sonuçta ConvNets ile sliding window yaklaşımı, bize computation olarak çok pahalıya patlar. Feasible değil.

Ancak neyseki bu problemin de çözümü var, in particular: Sliding window object detection can be implemented convolutionally or much more efficiently.

Vid 4

Convolutional Implementation Sliding Windows

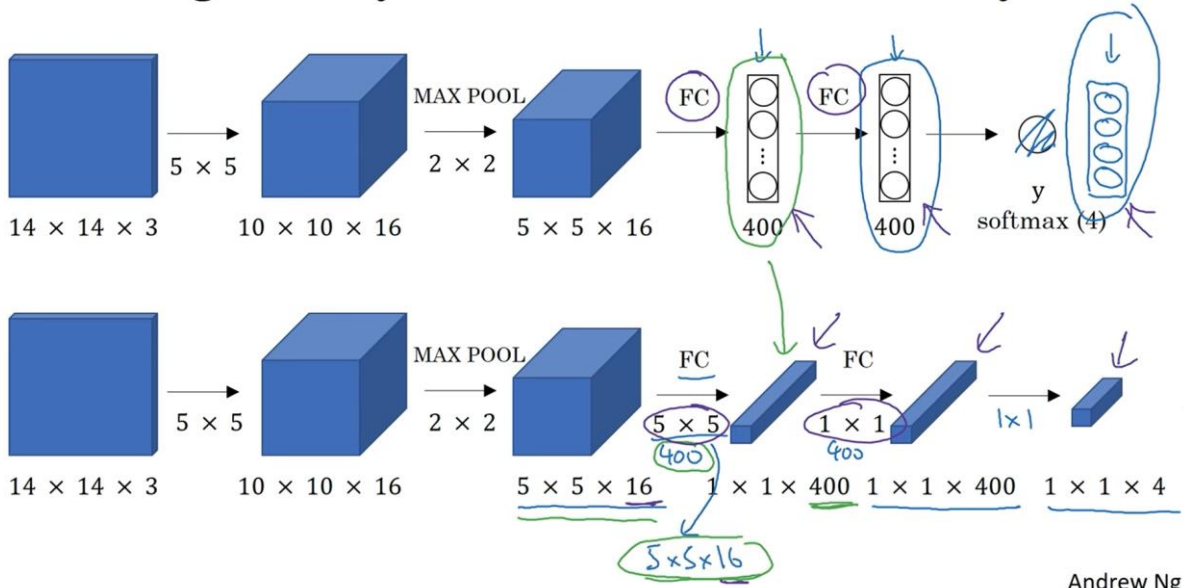
Geçtiğimiz videoda ConvNet kullanarak nasıl sliding window object detection yapabileceğimizden bahsetmiştik, ancak böyle bir algoritmanın çok yavaş olacağını da görmüştük.

Bu kısımda önceki kısımda gördüğümüz sliding window object detection algoritmasını nasıl convolutionally uygulayacağımızı ve böylece algoritmayı hızlandıracağımızı anlayacağız.

Bu yönteme geçmeden önce, fully connected layers'ı nasıl convolutional layers'a çevirebiliriz ondan bahsedelim. Çünkü bu fikri uygulamada kullanacağız.

- Diyelim ki ConvNet'imiz $14 \times 14 \times 3$ 'lük images input olarak alıyor ve 4 class classification yapıyor.
- Aşağıda ilk görülen yapı bu ConvNet'i gösteriyor.

Turning FC layer into convolutional layers



Amacımız, ConvNet içindeki fully connected layerları Convolutional Layers'a çevirmek. Bunu yapabilirsek, bu yöntemi sliding windows object detection'ı hızlandırmak için kullanacağız.

- Peki bunu nasıl yapıyoruz? Yukarıdaki 2. şekil bunu nasıl yaptığımızı açıklıyor.
- $5 \times 5 \times 16$ 'lık volume'u doğrudan açarak 400 unit elde etmek yerine, 400 adet $5 \times 5 \times 16$ filtre ile convolution yaparsak, elimizde $1 \times 1 \times 400$ 'lük bir volume olacaktır.
- Yani fully connected layer yerine convolution kullanıyoruz. Matematiksel olarak aynı işlemi yapmış oluyoruz. Çünkü eğer FC layer kullanırsak, $5 \times 5 \times 16$ 'lık volume'un 400 unitini flatten edip 400 unitlik FC'ye veriyoruz. Conv kullanırsak ise yine her conv. için 400 uniti alıyor, her birini bir weight ile çarpıyor ve sonra activation uyguluyor. FC layer ile bire bir aynı sonucu elde ediyoruz.
- Bir sonraki FC layer yerine yine 400 adet $1 \times 1 \times 400$ filtre ile convolution yapabiliriz.

- Son olarak softmax unit yerine 4 adet $1 \times 1 \times 400$ filtre ile convolution yapıyoruz ve softmax activation uyguluyoruz.

Nihayetinde bir FC layer yerine, Conv. Layer bu şekilde kullanılabilir, aynı matematiksel işlem yapılmış oluyor.

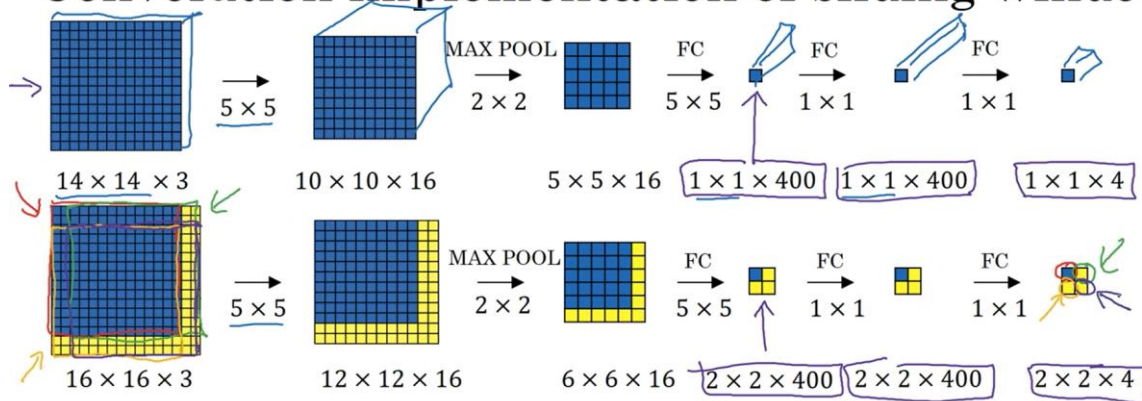
Bu conversion'ı anladığımıza göre, sırada bu fikri sliding window object detection'ı hızlandırmak için kullanmak var.

Şunu tekrar hatırlayalım, sliding window object detection ile yapılan işlem, original image'ı alıp, bunun içerisinde bir window gezdirerek her frame'i bir ConvNet'e input olarak vermek ve sonucunu hesaplamaktır, böylece image'ın her yerinde bir sürü kez ConvNet çalıştırılmış, ve bir obje var ise detect etmiş oluruz. Ancak ConvNet ile yapılan bu tekrarlı işlem computationally expensivedir.

Şimdi bunun yerine convolution'ı kullanacağız ve, input'un her yeri için ayrı ayrı bir ConvNet çalıştırmak yerine, tüm input üzerinde ConvNet çalıştırıp her frame için sonucu tek seferde elde edeceğiz.

- Diyelim ki $14 \times 14 \times 3$ image'lar üzerinde bir aşağıdaki gibi ConvNet eğittik ve bu network classification yapıyor. Bu bizim cropped image ConvNet'imiz. Yani normalde bu ConvNet'i her bir slide için çağırıyorduk.
- Bu network $14 \times 14 \times 3$ cropped input alıyor ve $1 \times 1 \times 4$ softmax output veriyor. Böylece Car/Pedestrian/Motorcycle/Background classification'ı yapabiliyor.

Convolution implementation of sliding windows



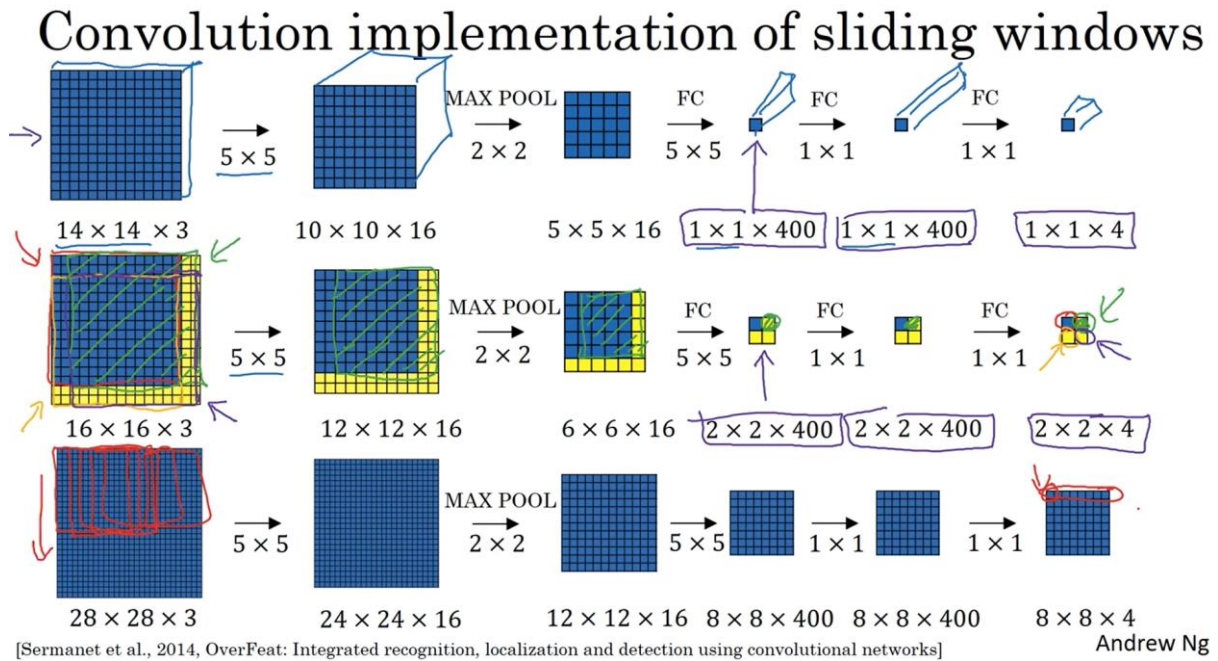
[Sermanet et al., 2014, OverFeat: Integrated recognition, localization and detection using convolutional networks]

Andrew Ng

- Bu ConvNet'i eğittikten sonra ne yapıyorduk, daha büyük image'lar üzerinde 14×14 'lük window gezdirip her frame için bu ConvNet'i çağırıyorduk.
- Diyelim ki Original Image'ımız $16 \times 16 \times 3$ büyüklüğünde yani yukarıda 2. Şekilde görüldüğü gibi olsun.

- Bu durumda eski sliding window'un yaptığı şey 4 tane 14x14 window alıp her biri için output hesaplamak olacaktı ve hangisi için detection yapıldıysa o 14x14'lük kısım işaretlenecekti.
- Yeni sliding window yaklaşımında ise, original image'dan ayrı ayrı 14x14 croplar almak yerine tüm image'ı input olarak kullanırız ve sonucunda, önceden 1x1x4 bir output elde ediyorken bu kez 2x2x4 bir output elde ederiz.
- Güzel kısım şu: Bu 2x2'lik sonucun her biri aslında crop edilmiş input'un sonucuyla aynı. Yani 2x2'lik sonucun sol üstteki kısmı 16x16'lık original image'ın sol üstteki 14x14'lük crop'u için hesaplanmış output ile aynı.
- Böylece convnet'i her crop için ayrı ayrı çalıştırmıyoruz bunun yerine tüm input üzerinde bir kez çalıştırıyoruz ve zaten output farklı croplar üzerinde çalıştırdığımızda elde edeceğimiz sonuçla aynı oluyor.

Şimdi daha büyük bir original image'ımız olduğunu düşünelim ve aynı işlemi uygulayalım:



Yukarıdaki resimde en alttaki network'ü ele alalım 28x28x3'lük bir input veriliyor, bu kez outputumuz 8x8x4 olarak çıkacak bu outputun her bir 1x1x4'lük kısmı aslında original image'ın 14x14x3'lük cropları için elde edilen output'a denk geliyor. Yani 64 farklı crop için convnet çalıştırmak yerine aynı convnet'i bir kez çalıştırdık ve aynı sonuçları tek seferde elde ettik.

Ancak burada çok önemli bir nokta var unutma: Evet belki ayrı ayrı 64 crop için convnet çalıştırmak yerine tek bir seferde çalıştırdık ve computational efficiency sağladık ancak lokalizasyon çok başarılı olmayacaktır, yani bounding box çok hassas yerleştirelemeyecek çünkü stride söz konusu. Burada max pool 2x2 olduğu için stride 2, yani original image için her olası pozisyon croplanamıyor. Sliding window'un stride'ı büyük olursa, ilgili objenin lokalizasyonunu yapmamız da zorlaşacaktır, çünkü belki en iyi window bile öyle bir denk gelecekli objenin sadece yarısını kendine dahil ediyor, bu durumda lokalizasyon başarısız olacaktır. Bir sonraki başlıkta nasıl başarılı yapabileceğimizi göreceğiz

Vid 5

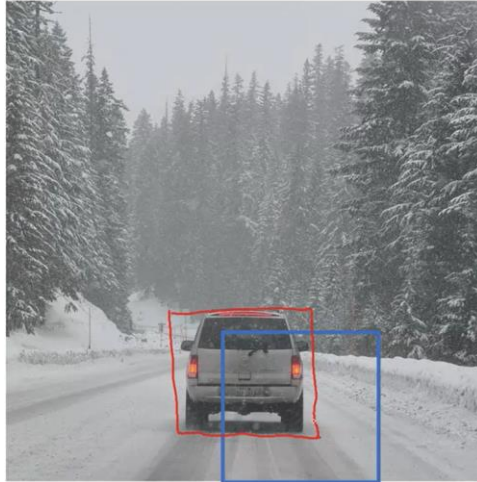
Bounding Box Prediction

Geçtiğimiz kısımda, sliding windows yaklaşımının convolution ile uygulamasını gördük, bu computationally efficient idi. Ancak hala accurate bir bounding box tahmininin nasıl yapılacağını görmedik.

Bu videoda accurate bir şekilde nasıl bounding box prediction yapabileceğimizi göreceğiz.

Başarılı lokalizasyon yapamamızın bazı sebepleri var, mesela stride olduğu için, belki original image içinde gezen window (mavi) çok büyük aralıklarla geziyor ve arabayı yakaladığı en iyi frame aşağıda görülen frame, burada araba detect ediyor ancak biz bu sınırı alırsak, ground truth ile çok daha farklı olacak.

Output accurate bounding boxes



Andrew Ng

Bun yanında belki ground truth tam olarak square olmayabilir, ancak bizim croplarımız hep square, yani böyle croplarla arabayı yakalayacak bir stride'imiz olsa bile, tam olarak ground truth'u tutturamayız çünkü window square ama ground truth rectangle

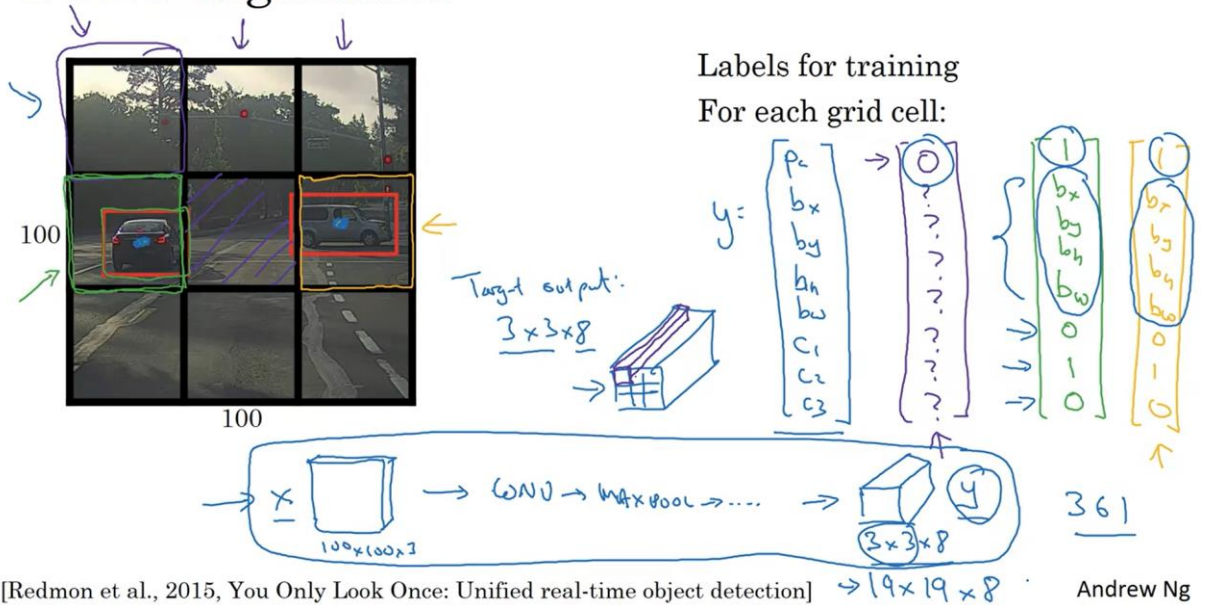
Peki accurate bounding boxes nasıl elde edilecek

→ More accurate bounding boxes elde etmek için YOLO (You Only Look Once) algorithm kullanılabilir.

→ Bu videoda YOLO algorithm'i anlamaya çalışalım.

- Diyelim ki aşağıdaki gibi 100x100px bir input image'imiz var.
- Bu input image'a bir grid yerleştiriyoruz, bu videoda basit olması için 3x3 bir grid yerleştiriyoruz, normalde daha fine bir grid kullanılır, mesela 19x19.
- Daha sonra yapılacak şey ilk kısımda gördüğümüz image classification and localization algoritmasını her bir grid için uygulamak. Ancak bunu her bir grid için ayrı ayrı tek tek uygulamıyoruz da convolutinally uyguluyoruz bu yüzden YOLO çok hızlı çalışır, real time işlemlerde bile kullanılır.
- Peki training set nasıl olacak? Classification&Localization için tek bir image'a 8 elemanlı bir vektör output label olarak atanıyordu. Burada ise her input image için grid sayısı kadar 8 elemanlı vektör output label olarak atanacak. Yani aşağıdaki gibi 9 gridli bir input'un label'ında 3x3x8 tane eleman bulunacak, her bir grid için [prob. of object, bx,by,bh,bw,pedestrian, car, motorcylce]
- Mesela sol üstteki grid için label, aşağıda morla görüldüğü gibi olacak, $P_c=0$ diğerleri don't care. Benzer şekilde yeşil ve turuncu gridler için ise car detect edilecek, ve ilgili car'ın bx,by,bh,bw koordinatları da label olarak verilmiş.
- YOLO her objeyi, tek bir grid cell'e assign eder, yani bu şu demek training set'e bakarsak, sadece yeşil grid cell için $P_c=1$ 'dir. Eğer gridler daha küçük olsaydı da sadece tek bir grid cell için $P_c=1$ olacaktı, yani arabanın merkezi o grid cell olacaktı.

YOLO algorithm



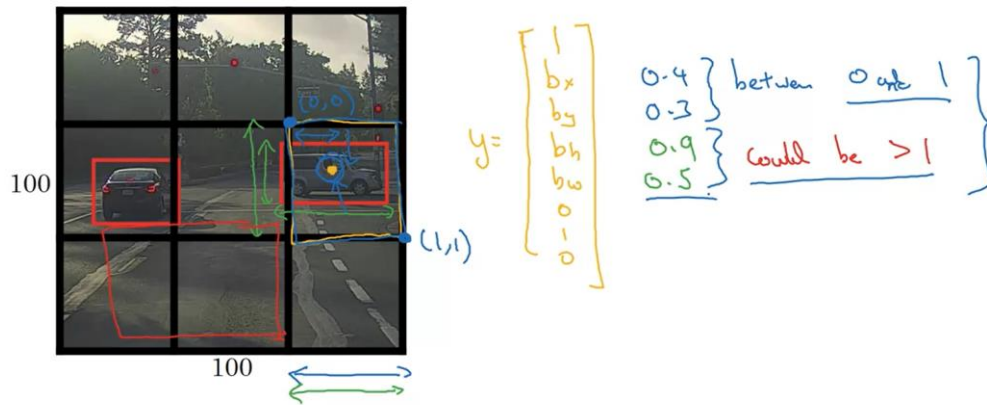
- Nihayetinde YOLO'ya 100x100x3'lük bir input girecek, 3x3x8'lik bir volume çıkacak, bu volume training set ile karşılaştırılarak backpropagation ile eğitim gerçekleştirilecek.
- Yani her grid için ayrı ayrı bir classification&localization çalışmıyor, convolution ile hepsi bir anda yapılıyor.
- Burada bana tuhaf gelen şey şu: Eğer 19x19 gibi finer bir grid kullansaydık, algoritma atıyorum arabanın kaportasının görüldüğü bir grid'i alacak, sadece buna bakarak hem classification yapacak hem de localization. Yani bir başka deyişle modelimiz arabanın tamamına değil, küçük parçalarına bakarak hem classification hem de localization yapmayı öğreniyor. Bu zor ve verimsiz görünüyor ama anladığım kadarıyla böyle yapıyor ve gayet iyi çalışıyor.

En nihayetinde bu algoritma, ile real time olarak accurate lokalizasyon yapabiliyorum, bir önceki kısımda gördüğümüz convolutional sliding windows object detection ile burada olduğu gibi hassas bir lokalizasyon yapamıyorduk, stride bu duruma engel oluyordu.

Son olarak bu b_x, b_y, b_h, b_w 'nin nasıl tanımlandığına değinelim:

- Bu değerler, ilgili grid box'a göre belirlenir. Örneğin aşağıda 2. Arabaya bakalım, ilgili arabanın merkezi oraya denk düşen grid'in ortalarında olduğu için b_x ve b_y 0.4 ve 0.3 olarak konumlandırılabilir.
- b_x ve b_y değerleri her zaman 0 ile 1 arasında olmak zorunda, çünkü merkez o grid'in içinde olmalı.
- Ancak b_h ve b_w ise 1'den büyük olabilir, eğer 19x19 grid kullanıyor olsaydık, bir gridlerden biri için yine 0-1 arasında b_x, b_y olacaktı ancak, boundary o grid'e sığmayacağı için, 1'den büyük b_h ve b_w ile karşılaşacaktık.

Specify the bounding boxes



[Redmon et al., 2015, You Only Look Once: Unified real-time object detection]

Andrew Ng

ÖNÜMÜZDEKİ BİR KAÇ VIDEO BOYUNCA DA YOLO ALGORİTMASINI GELİŞTİRMEK İÇİN BAZI BAŞKA FİKİRLERDEN BAHSEDİLECEK.

Vid 6

Intersection Over Union

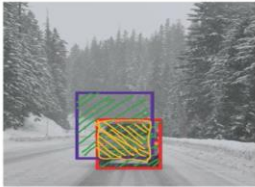
Artık, tek seferde tek image üzerinde hangi regionda hangi objectlerin detect edildiğini söyleyen bir algoritmamız var. Peki detection sonucunda elde edilen boundary ne kadar doğru, bu sorunun cevabını nasıl verebiliriz?

Bu kısımda Intersection Over Union fonksiyonu ile böylece object detection algoritmamızı evaluate edebileceğiz. Ayrıca aynı fonksiyonu bir sonraki kısımda object detection algoritma'ya bir başka component ekleyerek algoritmayı geliştirmek için kullanacağız.

Object detection için amaç, tespit edilen objeyi localize etmek. Yani aşağıdaki resimde ground truth kırmızı boundary olsun, ve bizim algoritmamız ise sonuçta mavi rectangle'ı output olarak verdi, bu durumda algoritma başarılı mı değil mi nasıl evaluate edeceğiz?

- Classification olsa evaluation için basitçe accuracy hesaplayabiliriz. Kaç tane doğru classification yapıldıysa yüzdesini veririz.
- Ancak bu durumda iki rectangle'ın uyumunu evaluate etmeliyiz işte bu amaç için Intersection Over Union (IoU) function kullanılır.
- Temelde bu fonksiyon, ground truth ile prediction rectangle'ın kesişim alanını, birleşim alanına böler. Sonuç 1'e ne kadar yakın ise o kadar doğru kabul edilir.
- Eğer bir örnek için sonuç 0.5'ten büyükse detection'ın başarılı olduğunu, söyleyebiliriz. Yani bu örnek için başarı elde edildi gibi düşün, sanki doğru classification yapılmış gibi sayabiliriz.
- 0.5 yerine 0.6, 0.7 gibi değerler de seçilebilir.
- Sonuçta bu fonksiyon kullanılarak, eğitilen modelin başarısı test edilebilir. Yani kaç example için 0.6 üzeri IoU hesaplanıyor onu buluruz ve böylece accuracy sonucu elde ederiz.

Evaluating object localization


$$\text{Intersection over Union (IoU)} = \frac{\text{size of } \text{[yellow box]}}{\text{size of } \text{[green box]}}$$

"Correct" if $\text{IoU} \geq 0.5$ ←
0.6 ←

More generally, IoU is a measure of the overlap between two bounding boxes.

Andrew Ng

Yani IoU fonksiyonunu, object detect eden algoritmamı evaluate edebilmek için kullanabilirim. Ancak daha genel bir yaklaşımla IoU function, iki bounding box'ın ne kadar overlap ettiğinin ölçüsüdür bu kavramı bir sonraki başlıkta object detection algoritmamızı geliştirmek için tekrar kullanacağız.

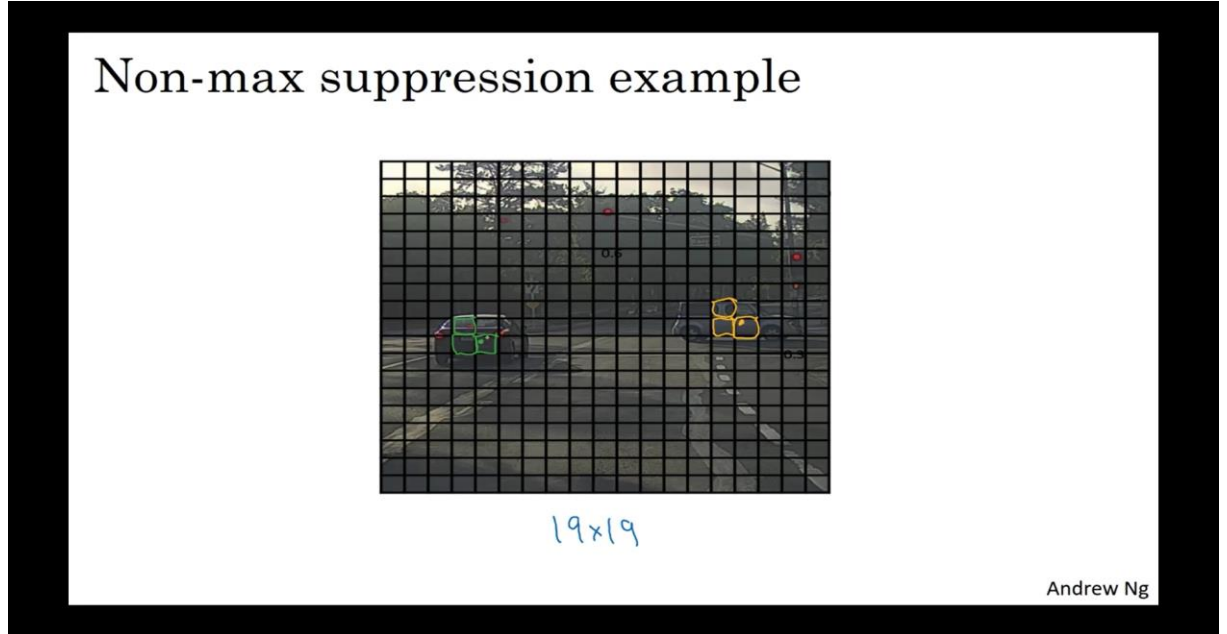
Vid 7

NonMax Suppression

YOLO ile object detection algoritmasının bir problemi var: Algoritmamız aynı objeyi birden fazla kez detect edebilir. NonMax Suppression ile algoritmamızın her objeyi yalnızca bir kez detect ettiğinden emin olabiliriz.

Diyelim ki bir algoritma eğittik ve aşağıdaki gibi input image'ı alıyor, her grid için (toplam 19x19 grid var) YOLO ile classification ve localization yapıyor. Yani algoritma her grid için önce hangi class'a ait olabileceğini söylüyor ve sadece bu grid'e bakarak bir bounding box koordinatı output ediyor.

Burada her gridin çok pixelli olabileceğini unutma, 1 grid demek orijinal image'ın crop edilmiş bir parçası demek. Her grid input olarak alınınca 5 adet output çıkıyor: [Pc yani is there a car, bx,by,bh,bw]. Burada diğer sınıfları şimdilik ayırmadığımızı kabul et.



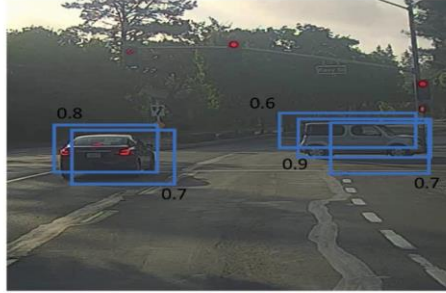
Sonuçta $19 \times 19 = 361$ tane grid için algoritma çalıştığında, bunlardan sadece iki tanesi değil, birçoğu car detect edebilir ve kendine göre bir boundary verebilir. Ki muhtemelen böyle olacaktır.

Yolo ile atıyorum 1000x1000'lik bir inputtan bana 19x19x5'lik output çıkıyor. Her bir grid için 5 output [Is there a car?, Center position x, Center position y, Height, Width]. Her bir grid için, sorular sorular, bu grid bir objenin merkezi mi? Eğer öyleyse hangi obje? (Burada sadece araba ayırt ediliyor.) Bu objenin sınırları nedir?

En nihayetinde birden fazla grid için car tespit edildi ve, dolayısıyla her car için birden fazla boundary bulundu. Ancak biz tek bir car için tek bir boudary elde etmek istiyoruz.

İşte bu noktada Non-max suppression devreye giriyor. Aşağıda görüldüğü gibi bir car için 2 diğer car için 3 boundary tespit edilmiş olsun, biz bu noktadan her car için tek bir boundary elde edeceğiz.

Non-max suppression example



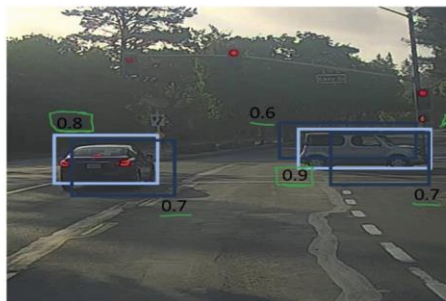
Andrew Ng

Her boundary aslında bir grid'in 5 outputundan bir tanesi, yani yukarıda toplam 5 tane grid için car detect edilmiş ve 5 farklı boundary verilmiş, ayrıca her grid için P_c değeri yani is this a car probability'si verilmiş (0.6,0.7,0.9 gibi değerler.)

Şimdi gereksiz boundary'leri elimine etmek için:

- Önce en yüksek P_c değerli çıktı ele alınır. Yukarıdaki örnekte bu 0.9'luk çıktı.
- Daha sonra bu seçilen çıktının boundary'si ile fazla kesişen diğer bütün çıktılar elimine edilir.
- Peki fazla kesişmenin kararını nasıl vereceğiz? Elbette IoU function ile!
- Daha sonra, kalan rectangles içinde highest P_c Değerli olan alınır ve aynı işlem tekrarlanır.
- Sonuçta en iyi tahminler dışındakiler surpass edilmiş olur.

Non-max suppression example

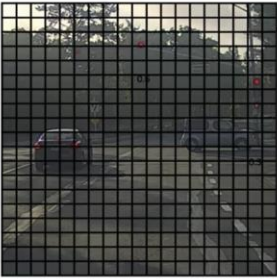


Andrew Ng

Algoritmayı daha detaylı anlatalım:

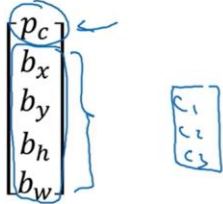
- Original image input olarak alınıyor, ve 19x19'luk her grid için bir output elde ediyoruz.
- Her output, bu grid bir araba'ya mı ait? Eğer aitse, ilgili arabanın sınırları neresidir, sorusunun cevabını çıktı olarak veriyor.
- Elbette, bunu birden fazla class için de yapabildik bu durumda 5 değil 8 output olacaktı, aynı daha önce classification and localization örneğindeki gibi. Şimdilik sadece car için problemi anlayalım.

Non-max suppression algorithm



19x 19

Each output prediction is:



Discard all boxes with $p_c \leq 0.6$

→ While there are any remaining boxes:

- Pick the box with the largest p_c
Output that as a prediction.
- Discard any remaining box with $\text{IoU} \geq 0.5$ with the box output in the previous step

Andrew Ng

Şimdi Non-max suppression'ın detayına bakalım:

- Öncelikle $P_c < 0.6$ olan tüm grid çıktılarını bi eyleyim. Car detection için en az 0.6 output versin.
- Daha sonra hala elimizde outputlar olabilir, ve bunlar aynı arabayı birden fazla kez işaretlemiş olabilir.
- Bunun için aynı arabayı işaretleyen gereksiz output'lardan kurtulmam gerek.
- En büyük P_c 'li tahmini alıyorum, bu tahminin rectangle'ıyla kesişen diğer bütün outputları discard ediyorum.
- Bu kesişimi de $\text{IoU} \geq 0.5$ ile anlıyorum.
- Seçilen ve elenen rectangle'lar dışında kalanlar için aynı işlemi tekrarlıyorum.
- Taa ki daha fazla output kalmayana kadar.
- Non-max suppression'ımızı hazır, afiyetle yiyin.

Burada tek bir class classification ve localization örneği kullanıldı, eğer birden fazla class olsaydı, her bir class için ayrı ayrı nonmax suppression çalıştırıyoruz!

Vid 8

Anchor Boxes

YOLO ile object detection'ın bir diğer problemi ise şu: bir grid cell sadece tek bir obje detect edebilir.

Peki ya bir grid cell'a bakıp birden fazla object detect etmek istiyorsak, yani bir başka deyişle ya aynı grid cell'e birden fazla object assign edilmesi gerekiyorsa? Çünkü iki farklı objenin merkezi bu grid cell'e denk düşüyorsa?

İşte bu durumda Anchor Boxes fikrinden yararlanabiliriz.

- Diyelim ki aşağıdaki gibi bir image'ımız var ve bu örnek için 3x3 grid kullanmaya devam edelim.
- Resimdeki pedestrian ve car'ın midpoint'leri neredeyse aynı nokta ve bu yüzden ikisinin de aynı grid cell'e atanması gerekiyor.
- Ancak bildiğimiz kadarıyla YOLO kullanırsak, her bir grid için resmin hemen altında görüldüğü gibi 8 elemanlı bir output kullanılır, bu durumda bu grid 8 eleman ile hem insanı hem de arabayı represent edemeyiz, birini seçmek zorundayız.

Overlapping objects:

Anchor box 1:

Anchor box 2:

$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$

Anchor box 1: $\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$

Anchor box 2: $\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$

[Redmon et al., 2015, You Only Look Once: Unified real-time object detection]

Andrew Ng

Ancak anchor boxes fikriyle yapacağımız şey şu:

- 2 farklı anchor box'ı predefine ederiz.
- Böylece iki farklı tahmini bu iki anchor box ile ilişkilendirebiliriz.
- In general we can use more than two anchor boxes, ancak burada anlaşılması için iki anchor boxes kullanılacak.
- Artık target label'ı 16 eleman olarak oluşturabiliriz, ilk 8'i anchor box 1 ile ilişik çıktılar, diğer 8'i ise anchor box 2 ile ilişkili çıktılar.

- İlgili örnekte pedestrian'ın şekli ile anchor box 1'in şekli birbirine daha benzer olduğu için, labeling'in ilk 8 elemanı ile pedestrian'ı encode ederiz, diğer 8 elemanla da anchor 2 'ye benzeyen car'ı encode ederiz.

Özetlemek gerekirse:

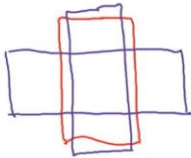
- Anchor box kullanmadan önce, training image içindeki her obje objenin merkez noktasını içeren tek bir grid cell'e assign ediliyordu.
- Ancak artık, training image içindeki her obje objenin merkez noktasını içeren grid cell'e ve o grid cell içindeki ilgili anchor box'a atanıyor. Objenin şekli hangi anchor box ile daha benzer ise obje o anchor box'a atanmış oluyor.
- Bu kısımdaki örnek için her grid cell içerisinde 2 anchor box yer almış oluyor.

Anchor box algorithm

Previously:

Each object in training image is assigned to grid cell that contains that object's midpoint.

Output y:
 $3 \times 3 \times 8$



With two anchor boxes:

Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU.

Output y:
 $3 \times 3 \times 16$
 $3 \times 3 \times 2 \times 8$

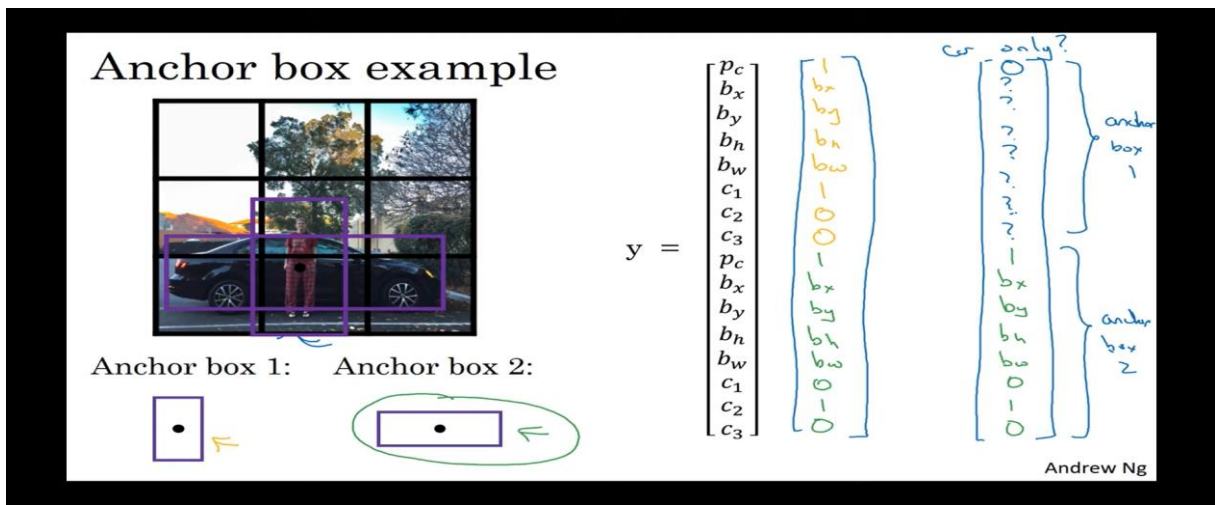
(grid cell, anchor box)

Andrew Ng

- Yani önceden 9 grid ve her grid için 8 output olduğu için $3 \times 3 \times 8$ olan output size, artık $3 \times 3 \times 16$ oldu, çünkü her grid'e artık 2 anchor box bağlı ve, her obje hem bir grid'e atanıyor hem de grid içindeki bir anchor box'a.
- Anchor box atama işlemi için IoU kavramından yararlanıyoruz, objenin shape'i ile anchor box shape'lerinin kesişimi bizim için önemli oluyor.

Şimdi bir training example'ın tek bir grid'i için output label'ın nasıl olacağına bakalım. Unutma tek bir training example için output label 3x3x16 olacak, bu kısımda sadece 1 grid için output'u inceleyeceğiz:

- Orta sıranın en altındaki grid iki farklı objeye atanmalı, çünkü iki objenin merkezi de bu grid'e düşüyor.
- Bu sebeple anchor box kavramını devreye sokuyoruz. İnsan'ın şekli anchor box 1 ile daha uyumlu olduğu için grid output'un ilk 8 elemanı sarıyla görüldüğü gibi olacak, insanı temsil edecek.
- Diğer 8 eleman anchor box 2 ile ilgili, arabayı temsil edecek.
- Eğer bir gridde sadece araba olsaydı ve insan olmasaydı, bu arabanın şekli de anchor box 2 ile uyumlu olsaydı bu durumda target label aşağıda en sağda görüldüğü gibi olacaktı.



Bazı ekstra detaylardan bahsedelim:

- Ya 2 anchor box belirlediysek, ancak ilgili grid cell'e 3 object düşüyorsa ne olacak? Bu algoritma 3. objeyi yakalayamaz, birini dışarıda bırakmak durumunda.
- Benzer şekilde eğer iki farklı object aynı grid cell'e ve düşüyorsa ancak ikisi de aynı anchor box ile uyuşuyorsa ne olacak? Yine bu algoritma bu durumu iyi handle edemez.

Şunu da unutmamak lazım, anchor box kavramıyla aynı grid cell'e birden fazla obje ataması yapabiliyoruz tamam ama zaten pratikte aynı grid cell'e çok nadiren birden fazla object düşer, çünkü fine grid cells kullanılır.

Anchor box'ın bir başka motivasyonu da şu: it allows our learning algorithm to specialize better, yani bazı outputlar uzun ince objeleri detect etmek için uzmanlaşırken, diğer outputlar geniş kısa objeleri detect etmek için uzmanlaşır.

Peki anchor boxes neye göre seçilir? Önceden elle seçilirdi, 5-10 tane sık görülen obje tiplerini temsilen seçilirdi. Daha iyi bir yolu ise, K-means ile en sık görülen obje şekillerini gruplamak, böylece representative olan 5-10 tane obje tipi için anchor boxes seçilebilir.

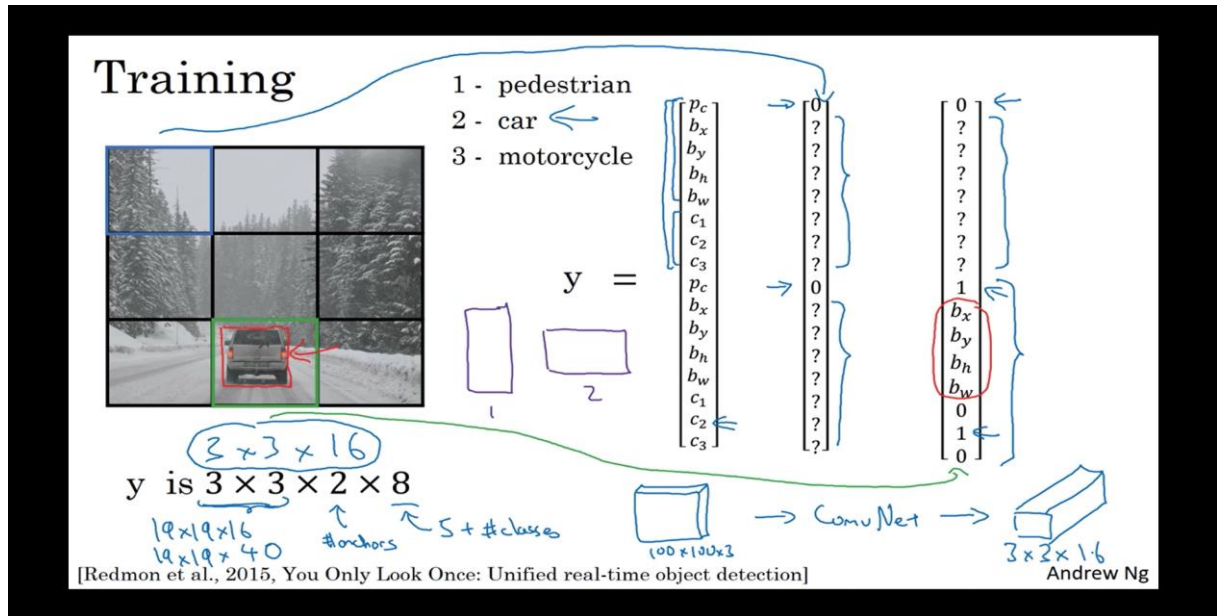
Vid 9

YOLO Algorithm

Object detection'ın birçok componentini gördük ve anladık. Şimdi bu başlık altında gördüğümüz tüm componentleri biraraya getireceğiz ve YOLO Object Detection Algorithm'i oluşturacağız.

Önce training set'i nasıl construct ettiğimize bakalım:

- Background hariç 3 class'ı classify ediyoruz.
- 3x3 grid kullandığımızı varsayalım.
- Her grid için de 2 anchor box kullanıldığını varsayalım.
- Bu durumda tek bir grid için 2x8=16 tane output olacak bu da bir image için 3x3x16 tane output olacağı anlamına gelir.
- Original images 100x100x3 boyutunda ise, output 3x3x16 boyutunda olacak, her bir grid-anchor için 8 adet çıkış var. [Pc: prob. of object, coordinates, classes]

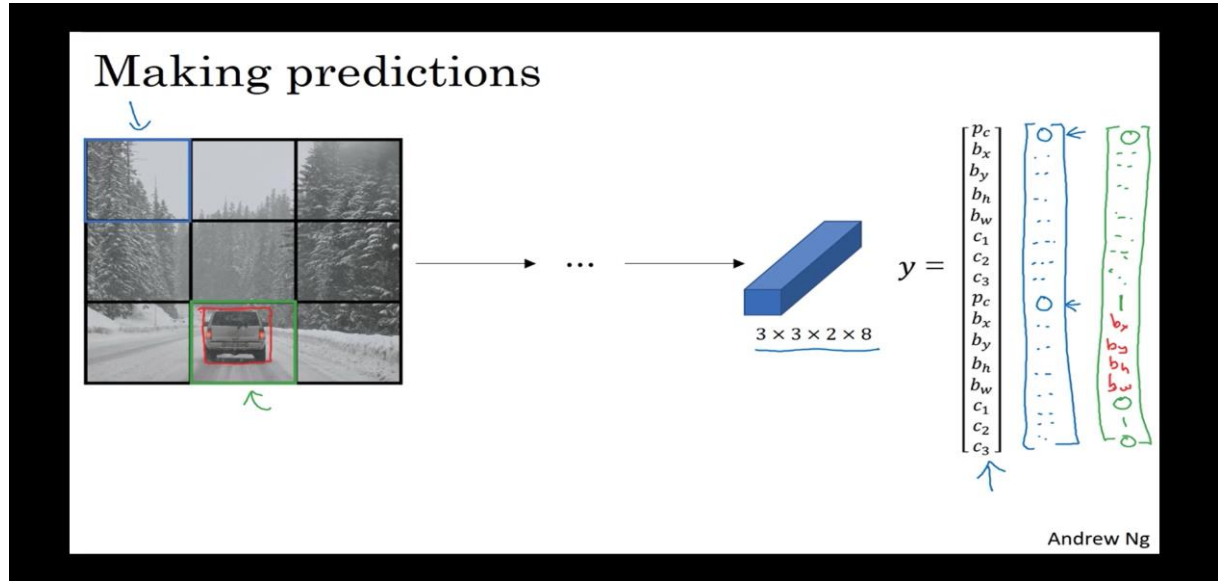


- Bu bağlamda, yukarıdak image için target label'ı yine yukarıda görüldüğü gibi hazırlarız.
- Örneğin ilk grid için hiç bir obje olmadığı için Pc1 ve Pc2 = 0 olur diğerleri de don't care.
- Fakat yeşil grid için Pc1=0 iken Pc2=1 olur, çünkü arabanın kırmızı ile görülen bounding box'ı anchor 2'ye daha uygun, çünkü width'ı height'ından biraz daha geniş, bu yüzden bu obje ilgili grid'in ikinci anchor'ına assign edilir.
- Anladığım kadarıyla YOLO için training set'i hazırlamadan önce elimizdeki örneklerde zaten objelerin detect edilmiş ve rectangle içine alınmış olması lazım, sonra muhtemelen YOLO için böyle bir training set için label hazırlıyoruz.

En nihayetinde 100x100x3 input YOLO ile ConvNet'e alınıyor her grid için 16 çıkış olmak üzere 3x3x16'lık bir output volume elde ediliyor.

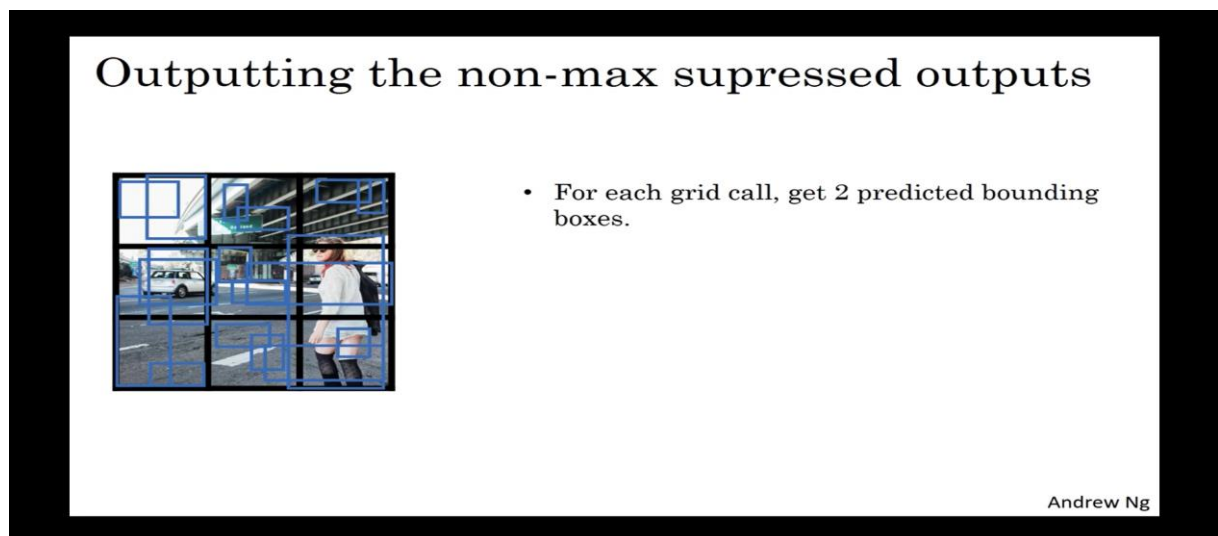
Şimdi de algoritmanın nasıl prediction yaptığına bakalım:

- Aşağıdaki gibi bir image eğitilmiş algoritmaya verildiği zaman, YOLO algoritmamız $3 \times 3 \times 16$ ya da $3 \times 3 \times 2 \times 8$ şeklimde bir output verecek.
- Her bir grid cell için 8 output.
- Mesela sol üstteki grid cell için hopefully our algorithm will give an output like the blue one shown below: Yani 0 0 Sonuçta bu gridde bir obje olmadığı için P_c ve P_c 'nin 0 olarak verilmesini isteriz, bu durumda diğer elemanların değeri önemli değil.
- Benzer şekilde yeşil grid'in 2. Anchor'ı için araba tespit edilmesini bekleriz.



Son olarak, Nonmax suppression algoritmasını kullanmak kalıyor böylece her obje için tek bir boundary elde etmiş olacağız. Şimdi bağımsız bir örnekte bunu inceleyelim.

- Eğer iki anchor box kullanıyorsak, her grid cell için YOLO NN'imiz 2 farklı boundary box'ı output olarak verecek. Obje detect etse de etmese de sonuçta 2 boundary box output vermek zorunda.



- Elbette bazı boundary boxes için P_c değeri küçük olacak, yani aslında burada bir obje detect edilmemiş olacak. İşte bu boundary boxes'tan kurtuluruz.
- Geriye yalnızca $P_c > 0.5$ olan grid-anchor predictions kalır. Bir başka deyişle muhtemel obje boundaryleri.

Outputting the non-max suppressed outputs

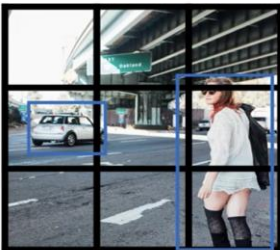


- For each grid cell, get 2 predicted bounding boxes.
- Get rid of low probability predictions.

Andrew Ng

- Ancak hala tek bir obje için birden fazla boundary atanmış olabilir. Yani belki aynı objeye hem aynı gridin 2 anchor'ı predict etti, veya iki farklı grid de aynı objeyi predict etti sonuçta yukarıdaki gibi aynı obje için elimizde birden fazla boundary var.
- İşte bu noktada her bir class için ayrı ayrı non-max suppression'dan yararlanılır.
- Yani ne yaparız, mesela önce insan için tahmin edilen tüm boundary'leri alırız, max P_c 'li olanı buluruz, daha sonra bu boundary ile kesişen yani IoU'su 0.5 üstü olan tüm diğer boundary'lerden kurtuluruz.
- Başka bir insan olsaydı, bu işlemi devam ettirecektik.
- Daha sonra aynı işlemi cars ve motorcycles içinde yaparız ve sonuçta boundaries aşağıdaki gibi kalır:

Outputting the non-max suppressed outputs



- For each grid cell, get 2 predicted bounding boxes.
- Get rid of low probability predictions.
- For each class (pedestrian, car, motorcycle) use non-max suppression to generate final predictions.

Andrew Ng

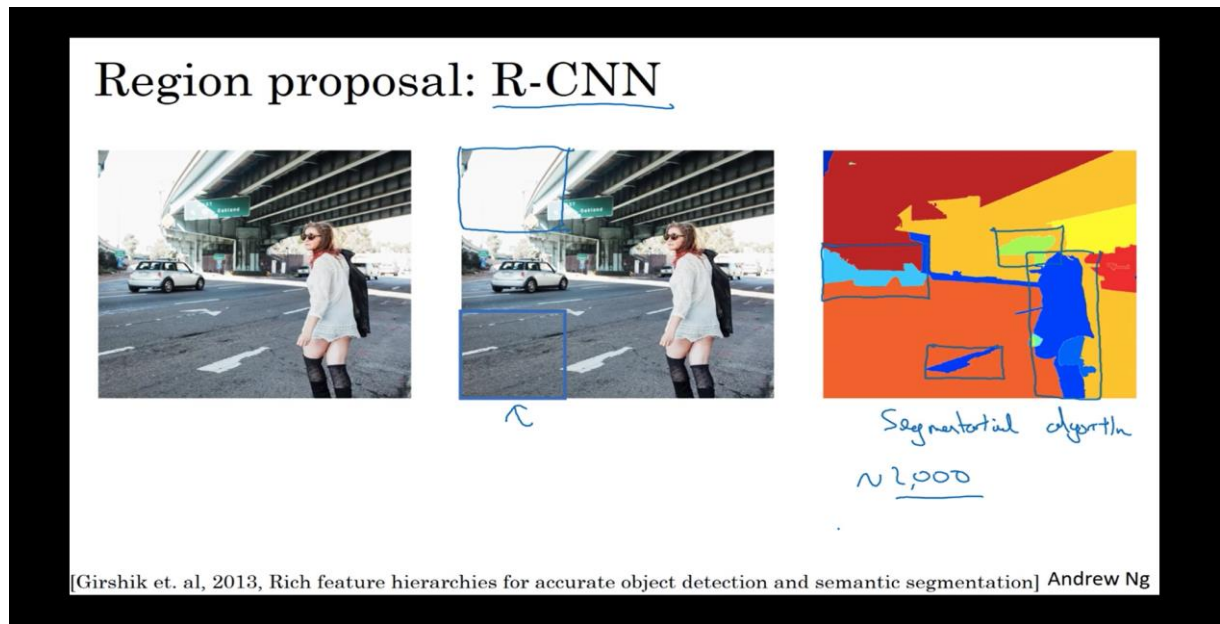
Vid 10

Region Proposals

Bu opsiyonel kısımda object detection literature’da yer eden bir kavram olan region proposals dan bahsedilecek.

Sliding window yaklaşımını hatırlarsak, bir classifier eğitip, bir image’ın farklı framelerinde bu ConvNet’i çalıştırıyorduk, işlemi daha hızlı yapmak için convolutional sliding window kavramından yararlanabilsek de bu algoritmanın bir dezavantajı şu, image’ın belki %90’lık bir kısmında hiçbir obje olmamasına rağmen algoritma image’ın her yerinde çalışmak zorunda kalıyor.

İşte R-CNN fikri yani Regions with CNN bu noktada devreye giriyor. Bu fikrin temeli şu: ConvNet classifier’ı input image’ın tamamında çalıştırmak yerine, önce image üzerinde object olma ihtimali olan regionlar belirleniyor ve daha sonra ConvNet bu regionlar üzerinde çalıştırılıyor. Böylece stride’ı düşürebiliriz ve algoritma çok yavaşlamadan hassas bir şekilde detection yapabilir.



Peki region proposals nasıl yapılıyor?

- Segmentation algorithm çalıştırılır ve input image üzerinde hangi regionların bir object olabileceği belirlenir.
- Yukarıda 3. image bunu anlatıyor, buradaki renk farklılıklarına göre belirli regionlar seçilir.
- Sonuçta ConvNet classifier sadece segmentation ile seçilen regionlar üzerinde çalıştırılır.
- Segmentation ile mesela 2000 tane potansiyel object region bulabiliriz, bunu yapmasak sliding window ile ConvNet’i çok daha fazla kısımda çalıştırmak zorunda kalabilirdik.

Ancak görünüŖe göre R-CNN algoritması hala yavaŖ sayılır. Bu yüzden bu algoritmayı hızlandırmak için bir çok çalıŖma yapıldı.

Basic R-CNN algoritması:

- Önce segmentation ile birsürü region propose edilir ve her bir region için CNN çalıŖtırılır, classification yapılır ve bunun yanında bounding box da tahmin edilir. Yani sadece propose edilen region'ı bounding box olarak almaz, kendi tahminini yapar.
- Bu algoritmanın bir dezavantajı hala çok yavaŖ olmasıdır.

Bundan daha hızlı olan Fast R-CNN algoritması da vardır:

- Bunun temel farkı region proposalı tek tek deęil de convolution ile tek seferde yapmasıdır.
- Böylece algoritma hızlanır.

Bir başka algoritma ise Faster R-CNN'dir:

- Bu da region proposal için CNN kullanmıŖtır.
- Böylece hız kazanmıŖtır.

Faster algorithms

→ R-CNN: Propose regions. Classify proposed regions one at a time. Output label + bounding box. ←

Fast R-CNN: Propose regions. Use convolution implementation of sliding windows to classify all the proposed regions. ←

Faster R-CNN: Use convolutional network to propose regions.

[Girshik et. al, 2013. Rich feature hierarchies for accurate object detection and semantic segmentation]

[Girshik, 2015. Fast R-CNN]

[Ren et. al, 2016. Faster R-CNN: Towards real-time object detection with region proposal networks]

Andrew Ng

TÜM BUNLARA RAĞMEN FASTER R-CNN DAHI YOLO'YA GÖRE ÇOK DAHA YAVAŞ ÇALIŞIR.