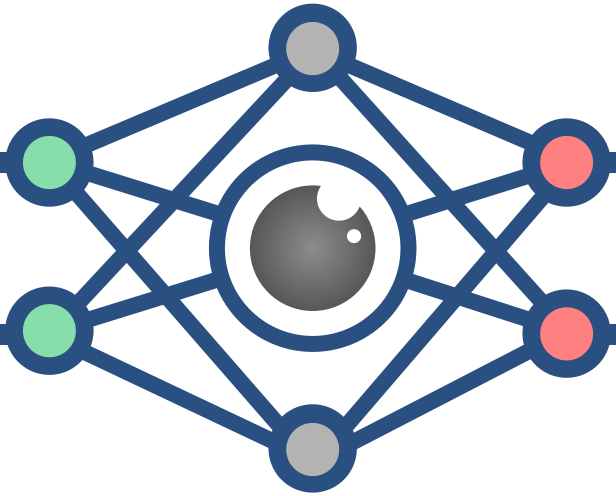
CS396

Deep Learning for Computer Vision



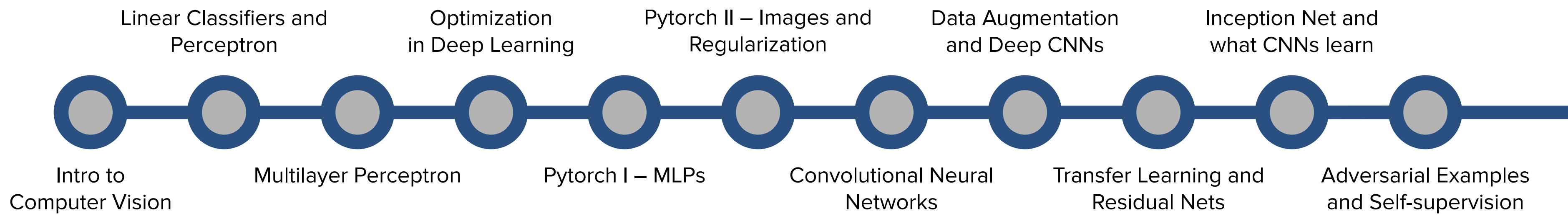
Lec 14: Fast Object Detection

Announcements

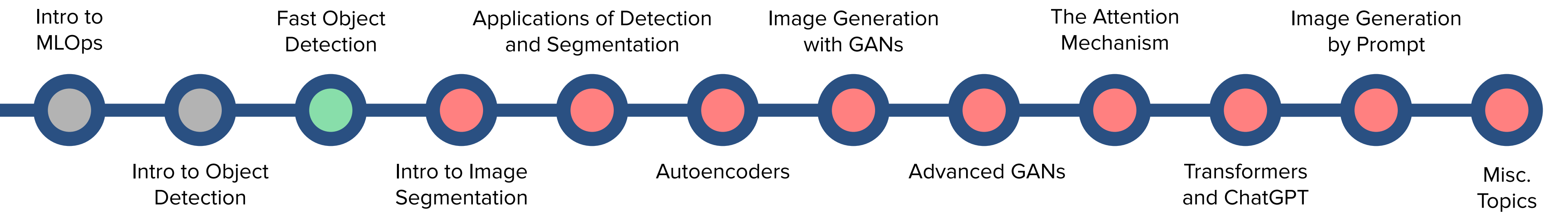
- Quiz today!

(Tentative) Lecture Roadmap

Basics of Deep Learning

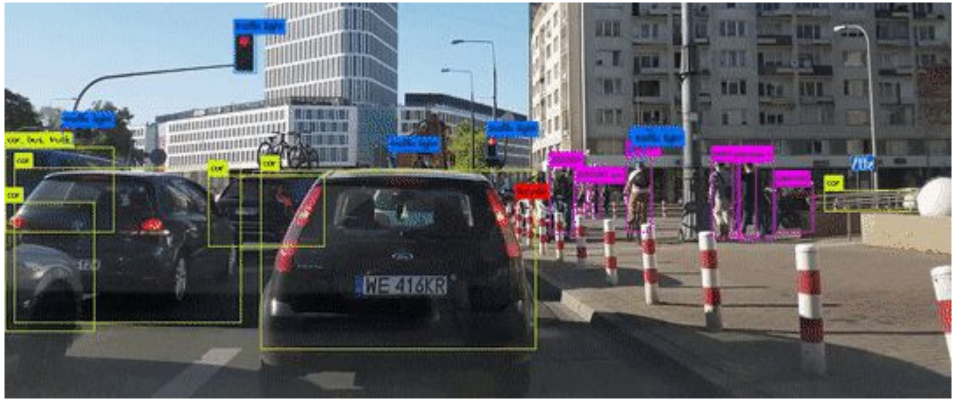

Intro to Computer Vision
Linear Classifiers and Perceptron
Multilayer Perceptron
Optimization in Deep Learning
Pytorch I – MLPs
Pytorch II – Images and Regularization
Convolutional Neural Networks
Data Augmentation and Deep CNNs
Transfer Learning and Residual Nets
Inception Net and what CNNs learn
Adversarial Examples and Self-supervision

Deep Learning and Computer Vision in Practice


Intro to MLOps
Intro to Object Detection
Fast Object Detection
Intro to Image Segmentation
Applications of Detection and Segmentation
Autoencoders
Image Generation with GANs
Advanced GANs
The Attention Mechanism
Transformers and ChatGPT
Image Generation by Prompt
Misc. Topics

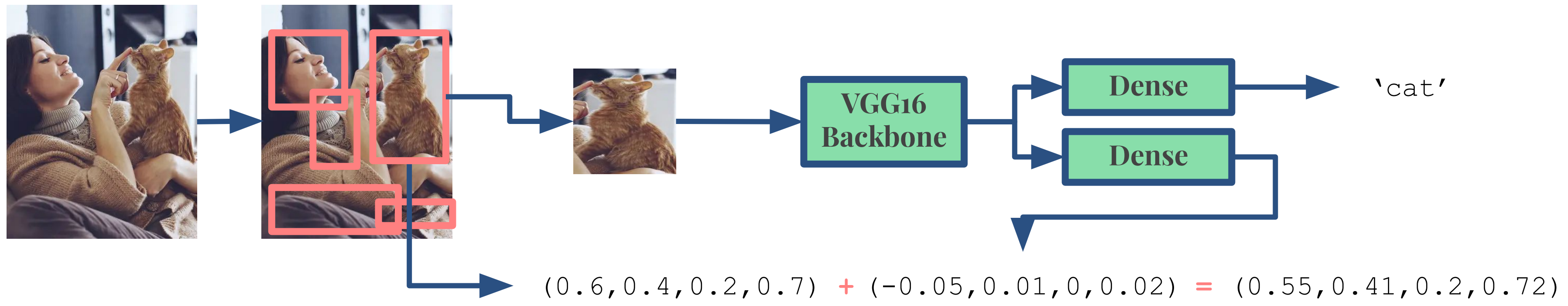
Why Fast Object Detection

- Last time we saw that Object Detection is a pretty useful in many real world scenarios and we use R-CNN to get good detections.
- Many of these scenarios, however, also require not just good, but also **fast detections**.
- Take the example of a self-driving car: it needs to be quick at detecting that there is an object (a person or a another car) in front of it in order to avoid a collision.
- Today we'll see how two strategies used to make R-CNN faster: Fast R-CNN and Faster R-CNN
- Finally, we'll see the most efficient and the *de facto* method used for Object Detection nowadays, called YOLO.



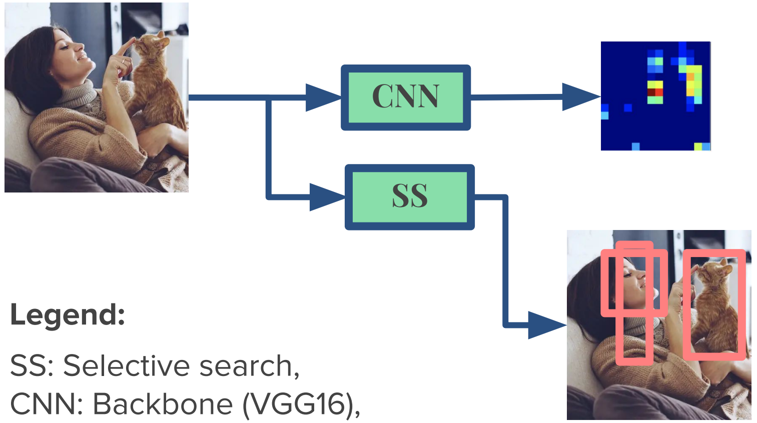
Problems with R-CNN

- In practice, R-CNN is slow for two main reasons:
  - It relies on the Selective Search Algorithm to obtain region proposals, which is slow for large images (such as the ones from self driving cars).
  - It has to wrap/resize each proposal before it goes through the network, which is also a slow procedure.
- In 2014, the same authors of R-CNN published a version of his R-CNN method that makes it faster by avoid the cropping resizing. This new method was named Fast R-CNN.



Fast R-CNN: Step-by-step

- In Fast R-CNN, the input image first goes through:
  a. A CNN backbone (such as VGG16) where its feature map (of a much smaller size, compared to the original image size) are extracted,
  b. A selective search algorithm extracts region proposals from the image.

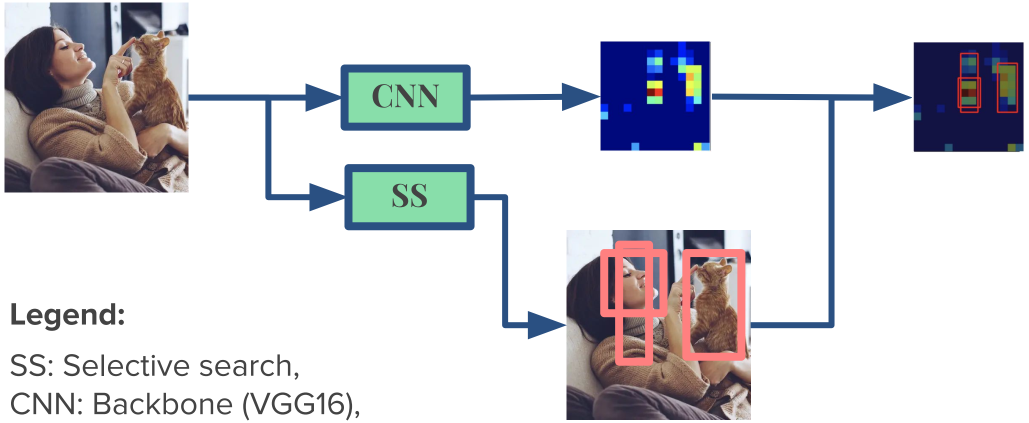


Legend:

SS: Selective search,
CNN: Backbone (VGG16),

Fast R-CNN: Step-by-step

- For each region proposal in the original image, its correspondent location, i.e. the **Region of Interest (RoI)**, on the feature map is computed using simple math. *For example*: after 5 max-pooling operations, a box of size $160{\times}240$ becomes of size $5{\times}7$.*



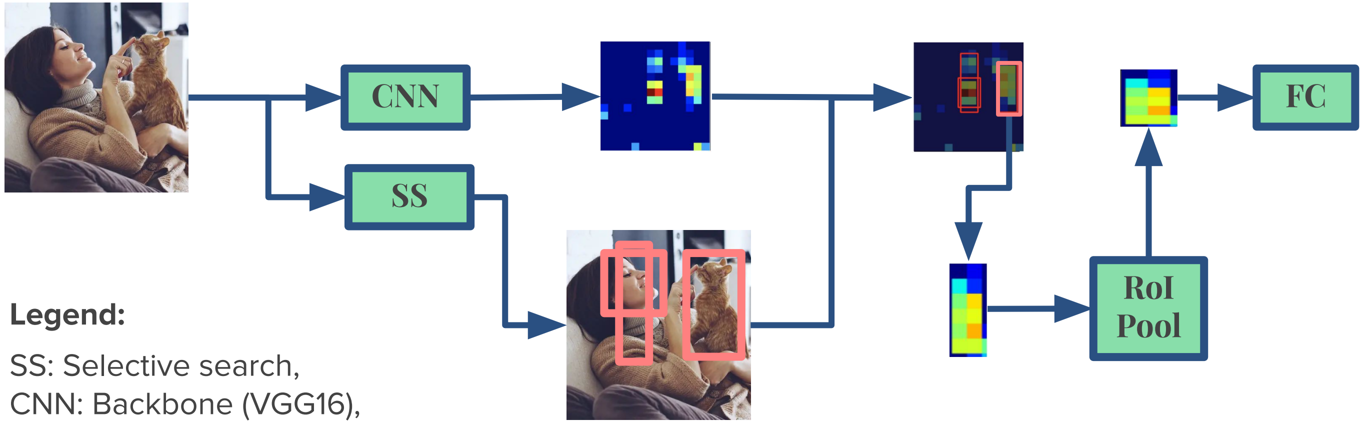
Legend:

SS: Selective search,
CNN: Backbone (VGG16),

* Note that, if one is using **normalized values** for their boxes' `(Cx, Cy, H, W)` vectors, the vectors remain the same in the feature map.

Fast R-CNN: Step-by-step

- One at a time, each region of interest goes through a RoI pooling layer (*more on it later*), whose output is standardized, i.e., it is the same for any RoI shape.
- That output then goes through a sequence of fully connected layers.

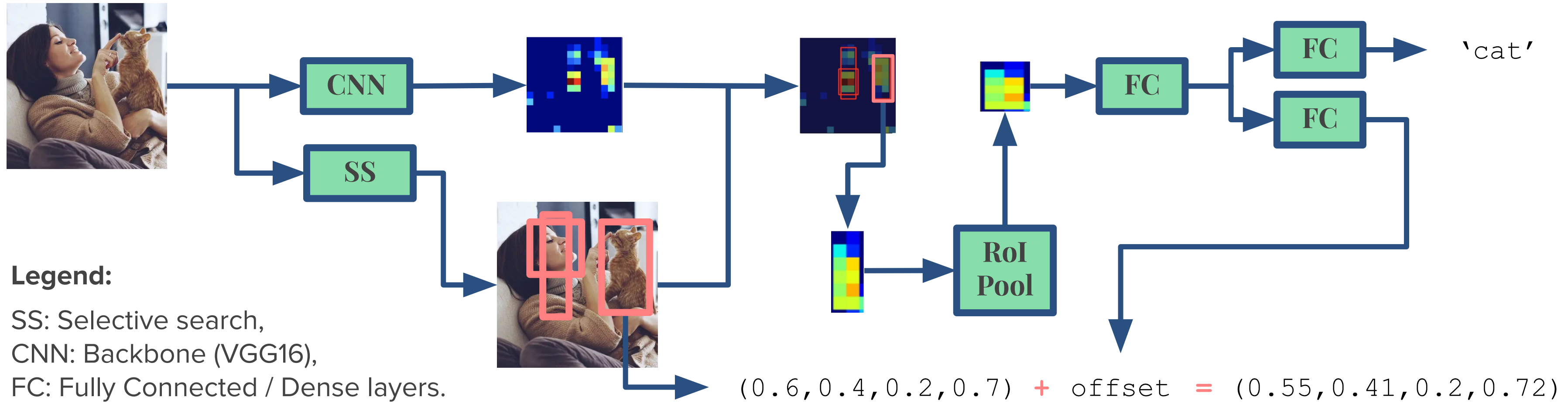


Legend:

SS: Selective search,
CNN: Backbone (VGG16),
FC: Fully Connected / Dense layers.

Fast R-CNN: Step-by-step

- After that FC, the resulting output becomes the input of other two sets of dense layers: one predicts the RoI class and the other, the BB offset on the original image.
- That offset then corrects the initial BB corresponding to the current RoI.



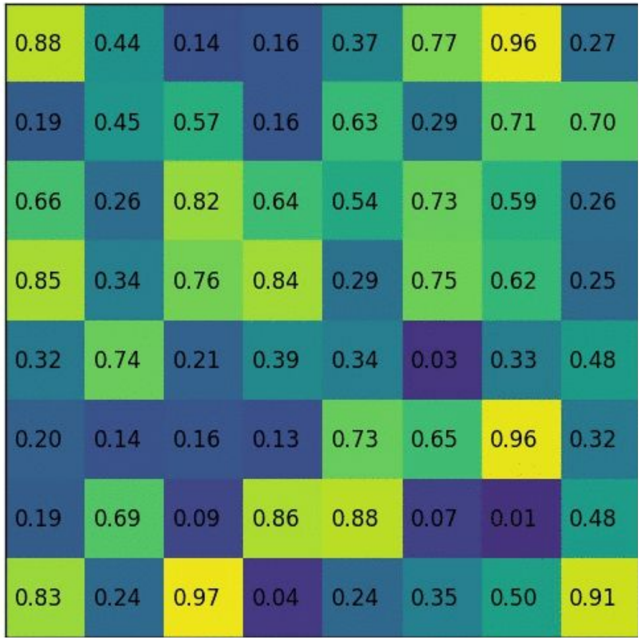
Legend:

SS: Selective search,
CNN: Backbone (VGG16),
FC: Fully Connected / Dense layers.

RoI Pooling Layer

- An important feature of Fast R-CNN is the **RoI pooling layer**, which was introduced in the same paper.
- What does it work? For a given RoI, it takes a section of the input feature map that corresponds to it and scales it to some predefined size (e.g., 2×2).
- The scaling is done by:
  a. Dividing the region proposal into equal-sized sections (whose number is equal to the output dimension). If a given region dimension cannot be divided evenly by the desired integer (like 7 divided by 2), take just the integer parts (like 3 and 4 in that case).
  b. Finding the largest value in each section.
  c. Copying these max values to the output.

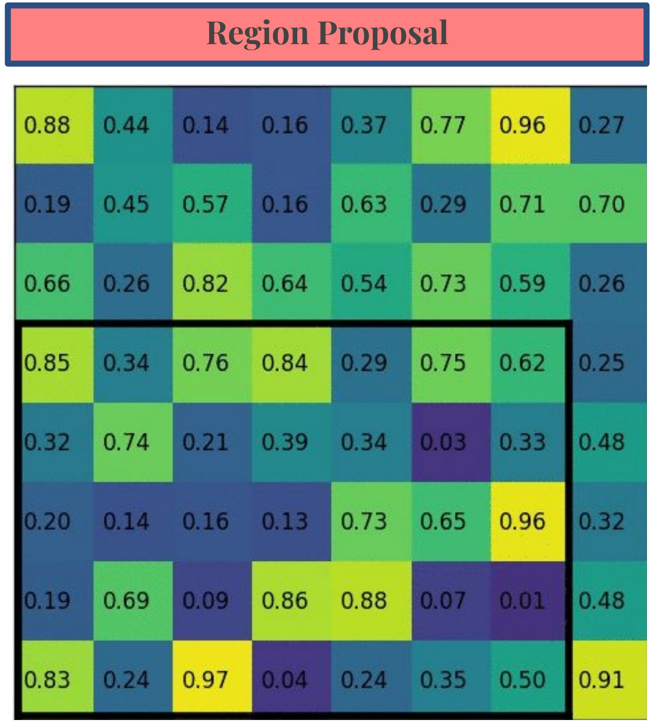
RoI Pooling Input



0.88	0.44	0.14	0.16	0.37	0.77	0.96	0.27
0.19	0.45	0.57	0.16	0.63	0.29	0.71	0.70
0.66	0.26	0.82	0.64	0.54	0.73	0.59	0.26
0.85	0.34	0.76	0.84	0.29	0.75	0.62	0.25
0.32	0.74	0.21	0.39	0.34	0.03	0.33	0.48
0.20	0.14	0.16	0.13	0.73	0.65	0.96	0.32
0.19	0.69	0.09	0.86	0.88	0.07	0.01	0.48
0.83	0.24	0.97	0.04	0.24	0.35	0.50	0.91

RoI Pooling Layer

- An important feature of Fast R-CNN is the **RoI pooling layer**, which was introduced in the same paper.
- What does it work? For a given RoI, it takes a section of the input feature map that corresponds to it and scales it to some predefined size (e.g., 2×2).
- The scaling is done by:
  a. Dividing the region proposal into equal-sized sections (whose number is equal to the output dimension). If a given region dimension cannot be divided evenly by the desired integer (like 7 divided by 2), take just the integer parts (like 3 and 4 in that case).
  b. Finding the largest value in each section.
  c. Copying these max values to the output.

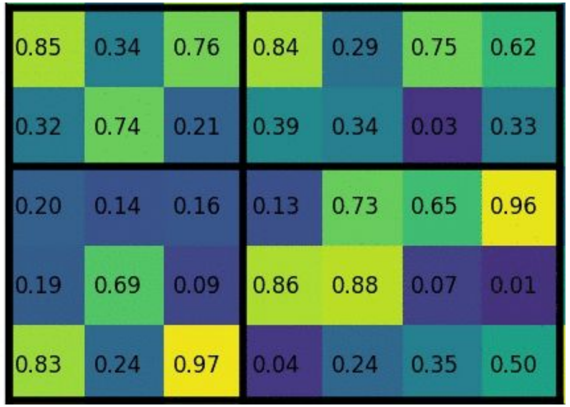


0.88	0.44	0.14	0.16	0.37	0.77	0.96	0.27
0.19	0.45	0.57	0.16	0.63	0.29	0.71	0.70
0.66	0.26	0.82	0.64	0.54	0.73	0.59	0.26
0.85	0.34	0.76	0.84	0.29	0.75	0.62	0.25
0.32	0.74	0.21	0.39	0.34	0.03	0.33	0.48
0.20	0.14	0.16	0.13	0.73	0.65	0.96	0.32
0.19	0.69	0.09	0.86	0.88	0.07	0.01	0.48
0.83	0.24	0.97	0.04	0.24	0.35	0.50	0.91

RoI Pooling Layer

- An important feature of Fast R-CNN is the **RoI pooling layer**, which was introduced in the same paper.
- What does it work? For a given RoI, it takes a section of the input feature map that corresponds to it and scales it to some predefined size (e.g., 2×2).
- The scaling is done by:
  a. Dividing the region proposal into equal-sized sections (whose number is equal to the output dimension). If a given region dimension cannot be divided evenly by the desired integer (like 7 divided by 2), take just the integer parts (like 3 and 4 in that case).
  b. Finding the largest value in each section.
  c. Copying these max values to the output.

Pooling Sections

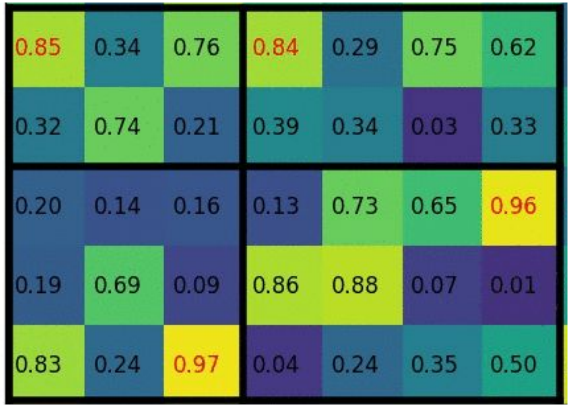


0.85	0.34	0.76	0.84	0.29	0.75	0.62
0.32	0.74	0.21	0.39	0.34	0.03	0.33
0.20	0.14	0.16	0.13	0.73	0.65	0.96
0.19	0.69	0.09	0.86	0.88	0.07	0.01
0.83	0.24	0.97	0.04	0.24	0.35	0.50

RoI Pooling Layer

- An important feature of Fast R-CNN is the **RoI pooling layer**, which was introduced in the same paper.
- What does it work? For a given RoI, it takes a section of the input feature map that corresponds to it and scales it to some predefined size (e.g., 2×2).
- The scaling is done by:
  a. Dividing the region proposal into equal-sized sections (whose number is equal to the output dimension). If a given region dimension cannot be divided evenly by the desired integer (like 7 divided by 2), take just the integer parts (like 3 and 4 in that case).
  b. Finding the largest value in each section.
  c. Copying these max values to the output.

Max Value in each section

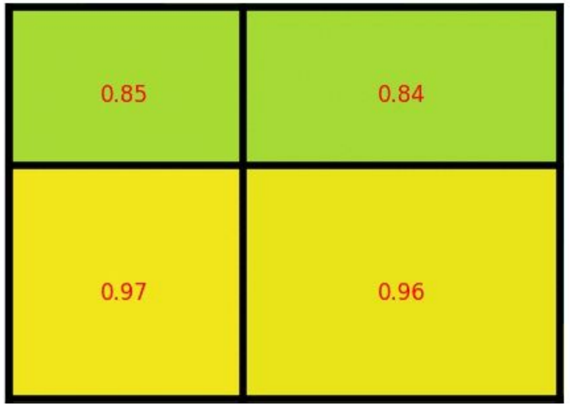


0.85	0.34	0.76	0.84	0.29	0.75	0.62
0.32	0.74	0.21	0.39	0.34	0.03	0.33
0.20	0.14	0.16	0.13	0.73	0.65	0.96
0.19	0.69	0.09	0.86	0.88	0.07	0.01
0.83	0.24	0.97	0.04	0.24	0.35	0.50

RoI Pooling Layer

- An important feature of Fast R-CNN is the **RoI pooling layer**, which was introduced in the same paper.
- What does it work? For a given RoI, it takes a section of the input feature map that corresponds to it and scales it to some predefined size (e.g., $2×2$).
- The scaling is done by:
  a. Dividing the region proposal into equal-sized sections (whose number is equal to the output dimension). If a given region dimension cannot be divided evenly by the desired integer (like 7 divided by 2), take just the integer parts (like 3 and 4 in that case).
  b. Finding the largest value in each section.
  c. Copying these max values to the output.

RoI Pooling Output

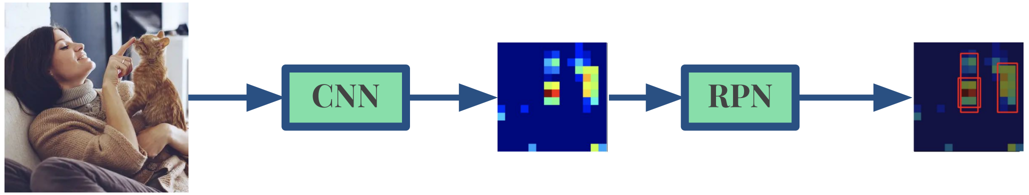


Fast R-CNN vs R-CNN

- Note that the big difference between R-CNN and Fast R-CNN is
  - In R-CNN, we are passing the crops (resized region proposals) through the pretrained model one at a time,
  - In Fast R-CNN, we are cropping the feature map (which is obtained by passing the whole image through a pretrained model) corresponding to each region proposal.
- We thereby avoid the need to pass all resized region proposals through the pretrained model in Fast R-CNN. **We only need to pass the entire image once!**
- Although Fast RCNN overcomes some problems of the R-CNN, the region of proposals are **still calculated via Selective Search** which is run on the CPU, whereas the network usually runs on the GPU.
- In 2015, it was published a paper where R-CNN's authors improved Fast R-CNN that removed that need, by training a network to do the region proposal procedure.
- This new method was named **Faster R-CNN**.

Faster R-CNN: Step-by-step

- In Faster R-CNN, the initial image is initially passed through a backbone CNN like before.
- Now, instead of computing the RoI's from the a selective search algorithm, it predicts the RoI's **from the feature map itself**, using a Region Proposal Network.

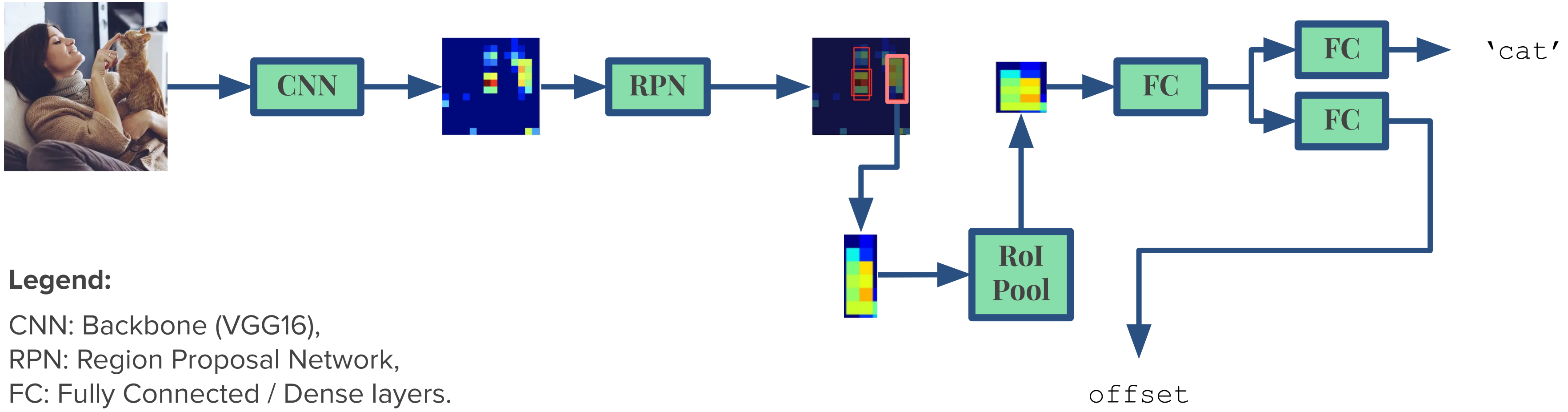


Legend:

CNN: Backbone (VGG16),
RPN: Region Proposal Network,

Faster R-CNN: Step-by-step

- Then as in Fast R-CNN, each proposed RoI goes through a RoI pooling layer, that then becomes the input of a few Fully Connected layers, that eventually predict both the RoI class and the BB offset on the original image.

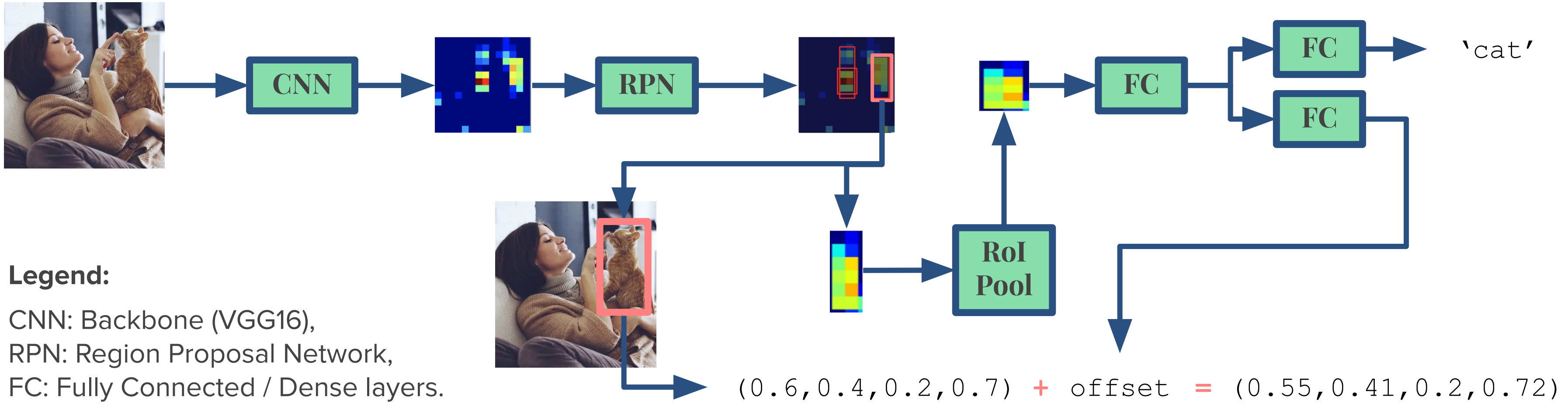


Legend:

CNN: Backbone (VGG16),
RPN: Region Proposal Network,
FC: Fully Connected / Dense layers.

Faster R-CNN: Step-by-step

- Finally, from each RoI prediction, it is possible to recover their locations on the original image using simple math.
- These locations are then corrected by the offsets to become the final BB output.



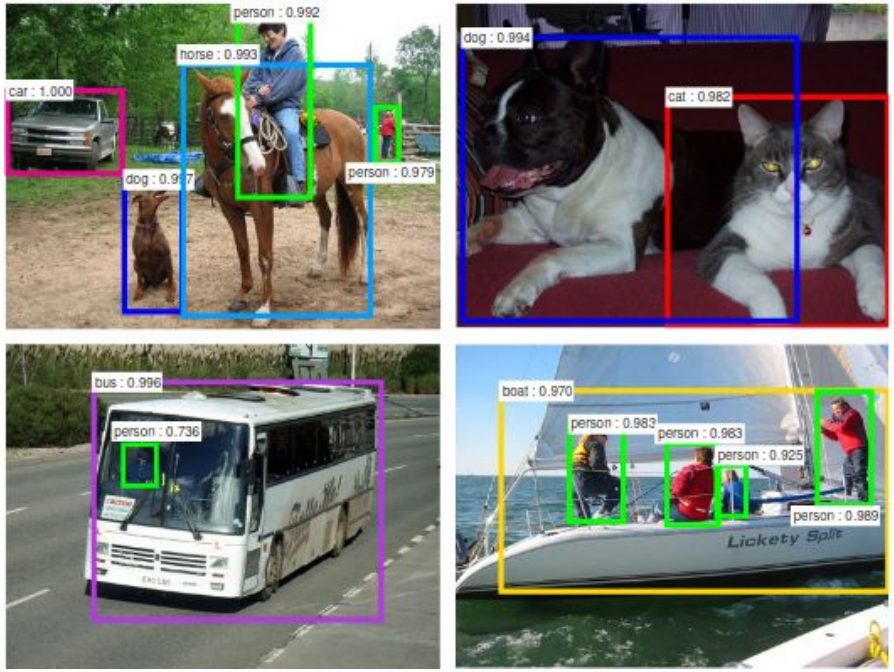
Legend:

CNN: Backbone (VGG16),
RPN: Region Proposal Network,
FC: Fully Connected / Dense layers.

Why is Faster R-CNN faster?

- The RPN network learns how to generate object proposals and is the core component in Faster-RCNN*!
- Because the RPN is also a network, not a separate algorithm, it can be trained in conjunction to the other FC layers in Faster R-CNN.
- That means that Faster R-CNN is a single, unified network for object detection that can be fully trained and inferred in the GPU, making it very fast. (some results are shown on the right).

Object detection with Faster R-CNN



* You can find how it works in our appendix.

Faster R-CNN in PyTorch

- Out of the R-CNN variations, only the pretrained Faster R-CNN is available in PyTorch.
- To use it, we just need to import the model (here with a Resnet50 backbone) and its pretrained weights*:

```
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.models.detection import FasterRCNN_ResNet50_FPN_Weights

model_rcnn = fasterrcnn_resnet50_fpn(weights=FasterRCNN_ResNet50_FPN_Weights.DEFAULT)
```

- If we have an image called `img` (*properly processed*), can simply do:

```
results = model_rcnn(img)
```

where results is a list of dictionaries (one per image, in our case the list will only have one element), each of which contains the BB information, their classes and the classification confidences.

* The FPN on the model name stands for Feature Pyramid Networks, which is at simple trick that improves classification (see this).

Exercise (*In pairs*)

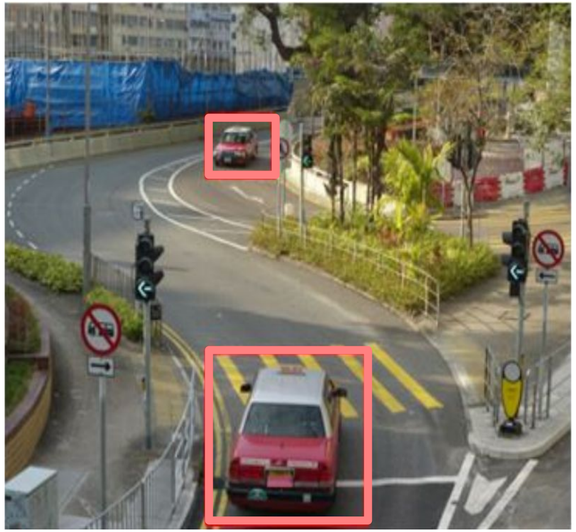
- When considering training and inference, what do you think is a bottleneck of Faster R-CNN in terms of speed? *Hint*: what kind of neural network has the most weights to be learned?

You Only Look Once

- Despite the efforts from the R-CNN variants attain in real time detection, it was a totally different approach that attained it.
- YOLO (You Only Look Once) was originally published in 2015 and since then **its strategy has become the standard for Object Detection**.
- The main idea in YOLO is its smart way to collect detection data, which avoids selective search and only pass the image through the network only once.
- It furthermore enables to use of simpler and faster network architectures.

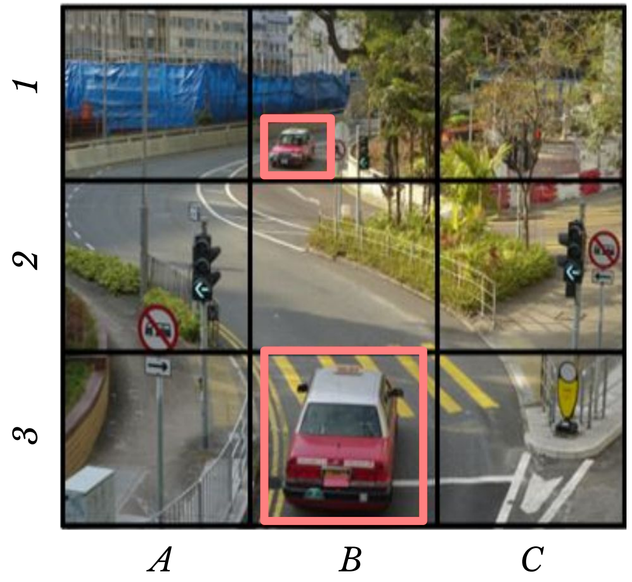
YOLO in Practice

- For training, consider the following detection ground truth image, with two detected car objects. Assume that we only have three possible classes: people, cars and animals.



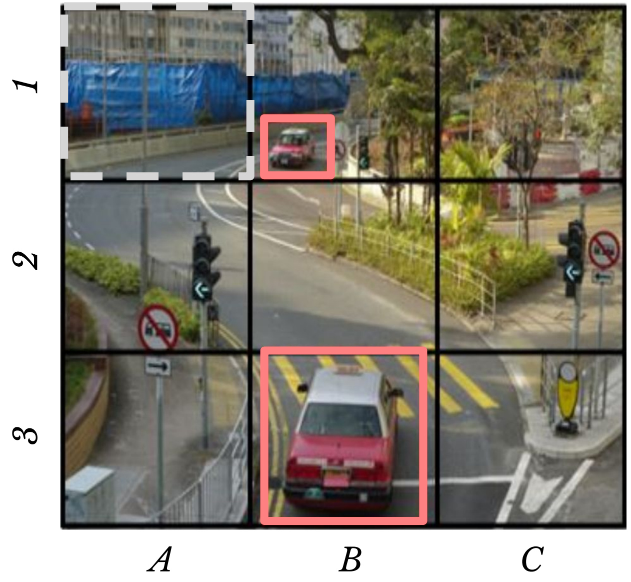
YOLO in Practice

- The first step in YOLO when preparing this data for training consists is to divide the image in a $Nc \times Nc$ grid cells (let's say $Nc = 3$ in our case):



YOLO in Practice

- For each grid cell, we create a vector with numbers that check whether there's an object there and, if so, its characteristics:



Cells: **A1**

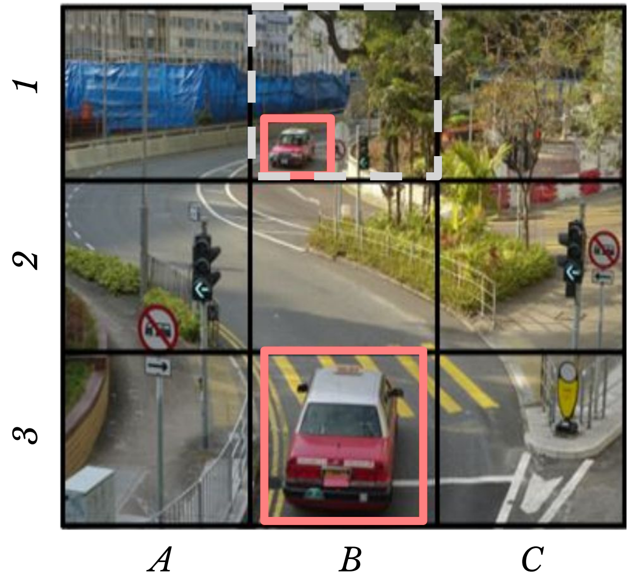
$$\begin{array}{rl} o = & \boxed{0} \\ b_x = & \boxed{?} \\ b_y = & \boxed{?} \\ b_w = & \boxed{?} \\ b_h = & \boxed{?} \\ c_1 = & \boxed{?} \\ c_2 = & \boxed{?} \\ c_3 = & \boxed{?} \end{array}$$

The first number (o) simply checks if there is an object in that cell (0 for yes, 1 for no).

If $o = 0$, the other vector numbers become meaningless and they can take any value.

YOLO in Practice

- For each grid cell, we create a vector with numbers that check whether there's an object there and, if so, its characteristics:

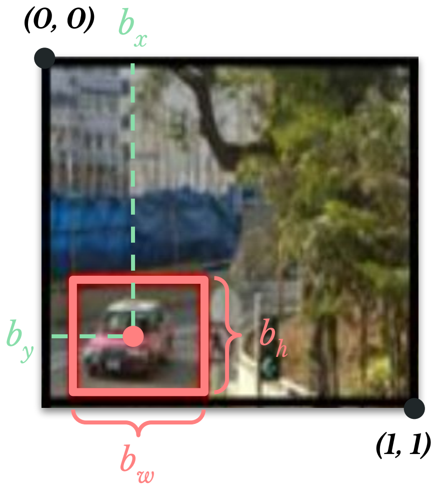


Cells:	A1	B1
o =	0	1
b_x =	?	.2
b_y =	?	.8
b_w =	?	.3
b_h =	?	.2
c_1 =	?	0
c_2 =	?	1
c_3 =	?	0

If $o = 1$, it means that there is an object in that cell.

The following 4 numbers correspond to the bounding box: (b_x, b_y) are the normalized coordinates of its center, (b_w, b_h) are its normalized width and height*.

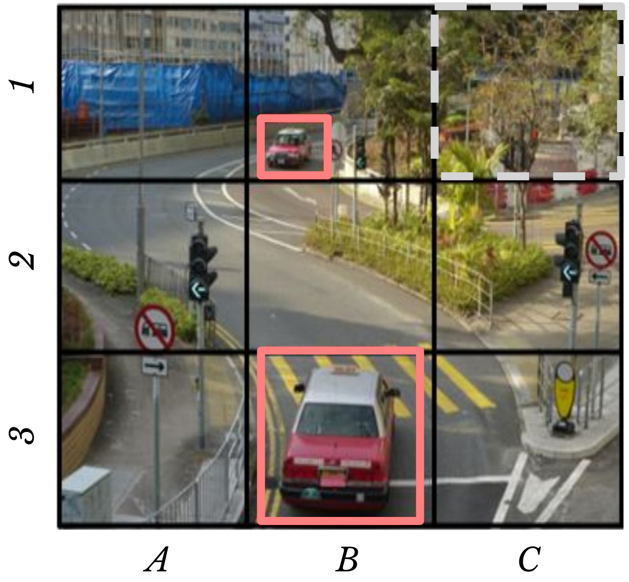
The last three numbers correspond to the class of the object (say c_2 is for cars).



* The values for (b_w, b_h) can be both greater than 1 if their objects span multiples cells.

YOLO in Practice

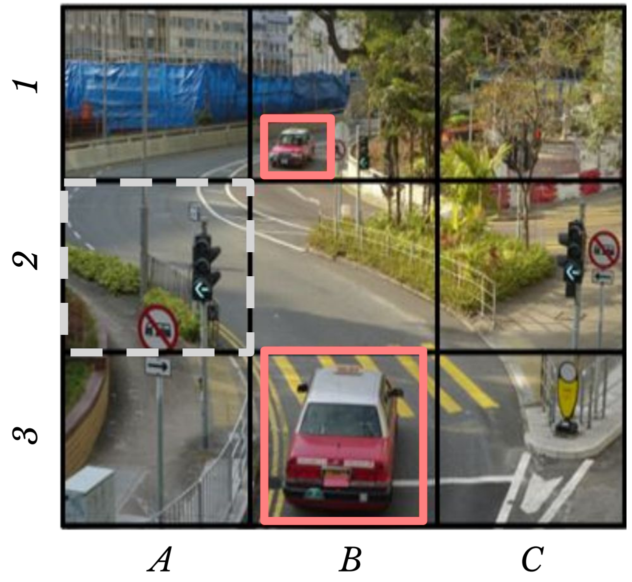
- For each grid cell, we create a vector with numbers that check whether there's an object there and, if so, its characteristics:



Cells:	A1	B1	C1
$o =$	0	1	0
$b_x =$	?	$.8$	?
$b_y =$	?	$.2$	?
$b_w =$	?	$.3$	?
$b_h =$	?	$.1$	?
$c_1 =$	?	0	?
$c_2 =$	?	1	?
$c_3 =$	?	0	?

YOLO in Practice

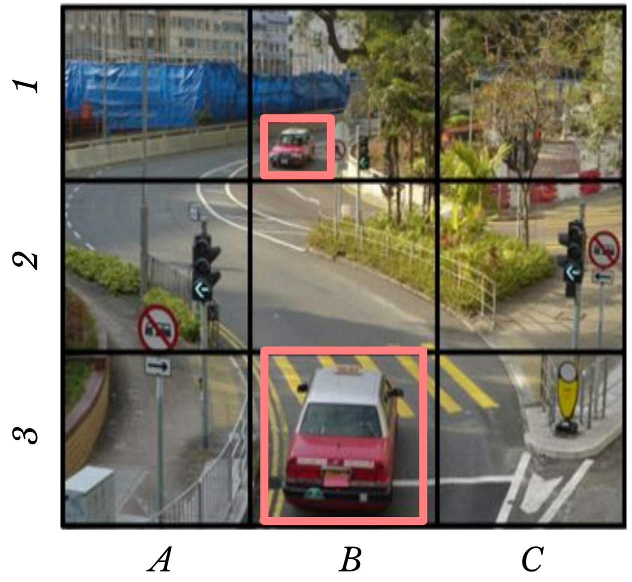
- For each grid cell, we create a vector with numbers that check whether there's an object there and, if so, its characteristics:



Cells:	A1	B1	C1	A2
o =	0	1	0	0
b_x =	?	.8	?	?
b_y =	?	.2	?	?
b_w =	?	.3	?	?
b_h =	?	.1	?	?
c_1 =	?	0	?	?
c_2 =	?	1	?	?
c_3 =	?	0	?	?

YOLO in Practice

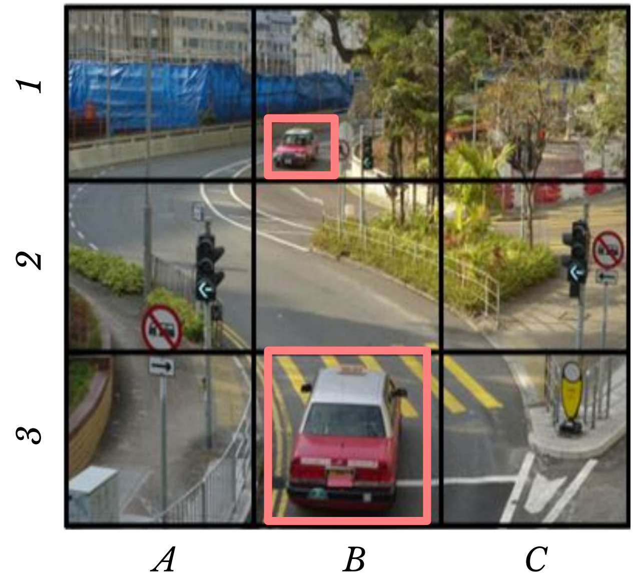
- For each grid cell, we create a vector with numbers that check whether there's an object there and, if so, its characteristics:



Cells:	A1	B1	C1	A2	B2	C2	A3	B3	C3
o =	0	1	0	0	0	0	0	1	0
b_x =	?	.8	?	?	?	?	?	.5	?
b_y =	?	.2	?	?	?	?	?	.5	?
b_w =	?	.3	?	?	?	?	?	.9	?
b_h =	?	.1	?	?	?	?	?	1.1	?
c_1 =	?	0	?	?	?	?	?	0	?
c_2 =	?	1	?	?	?	?	?	1	?
c_3 =	?	0	?	?	?	?	?	0	?

YOLO in Practice

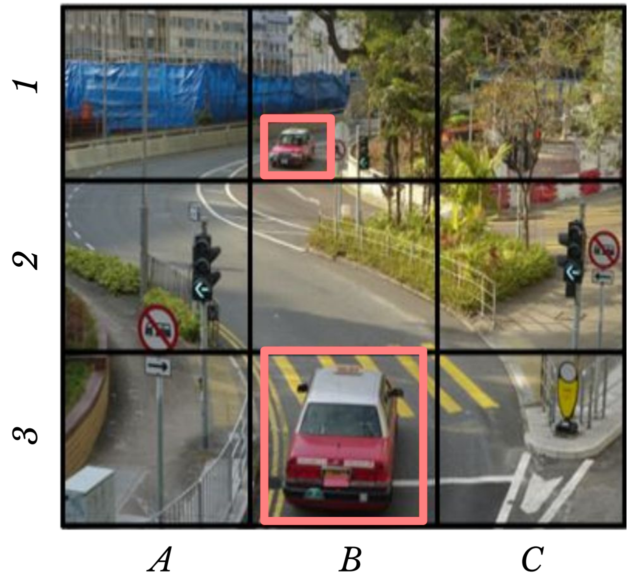
- For training, we can set the numbers that can take any value (when there is no object in a cell) as any value (for instance, they can be set *1*):



Cells:	A1	B1	C1	A2	B2	C2	A3	B3	C3
o =	0	1	0	0	0	0	0	1	0
b_x =	1	.8	1	1	1	1	1	.5	1
b_y =	1	.2	1	1	1	1	1	.3	1
b_w =	1	.3	1	1	1	1	1	.5	1
b_h =	1	.1	1	1	1	1	1	.2	1
c_1 =	1	0	1	1	1	1	1	0	1
c_2 =	1	1	1	1	1	1	1	1	1
c_3 =	1	0	1	1	1	1	1	0	1

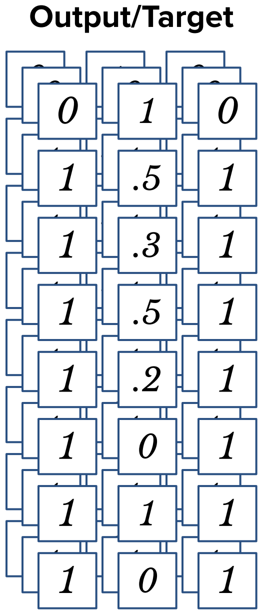
YOLO in Practice

- After computing the vectors for each cell, we can rearrange them as a tensor of shape *(Nc, Nc, 5 + K)*, where K is the number of classes in our dataset.



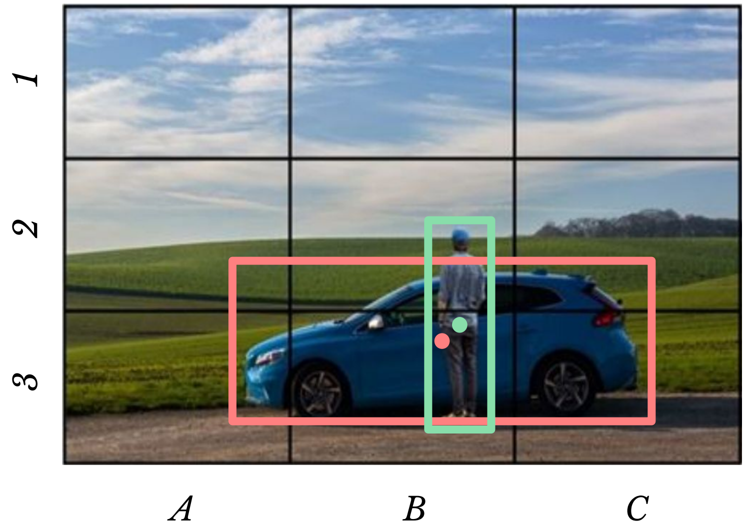
Cells:	A1	B1	C1	A2	B2	C2	A3	B3	C3
o =	0	1	0	0	0	0	0	1	0
b_x =	1	.8	1	1	1	1	1	.5	1
b_y =	1	.2	1	1	1	1	1	.3	1
b_w =	1	.3	1	1	1	1	1	.5	1
b_h =	1	.1	1	1	1	1	1	.2	1
c_1 =	1	0	1	1	1	1	1	0	1
c_2 =	1	1	1	1	1	1	1	1	1
c_3 =	1	0	1	1	1	1	1	0	1

=



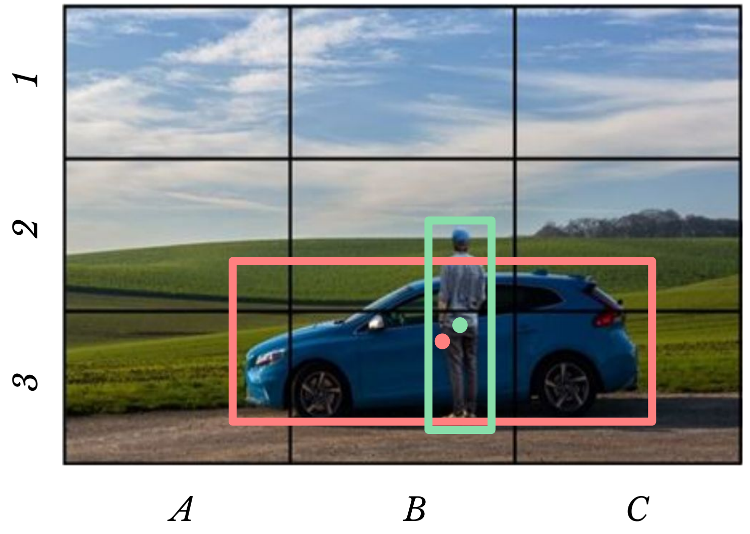
When There are Multiple Objects in a Cell

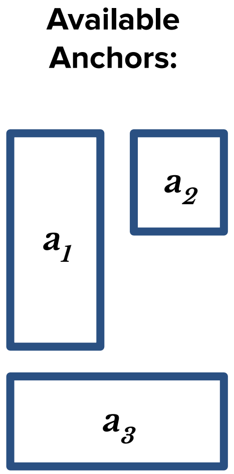
- One big problem with this method is that it doesn’t take into account objects whose centers are in the same cell. For example:



When There are Multiple Objects in a Cell

- To handle multiple objects in the same cell, we can (again) use a set of available **Anchors***.

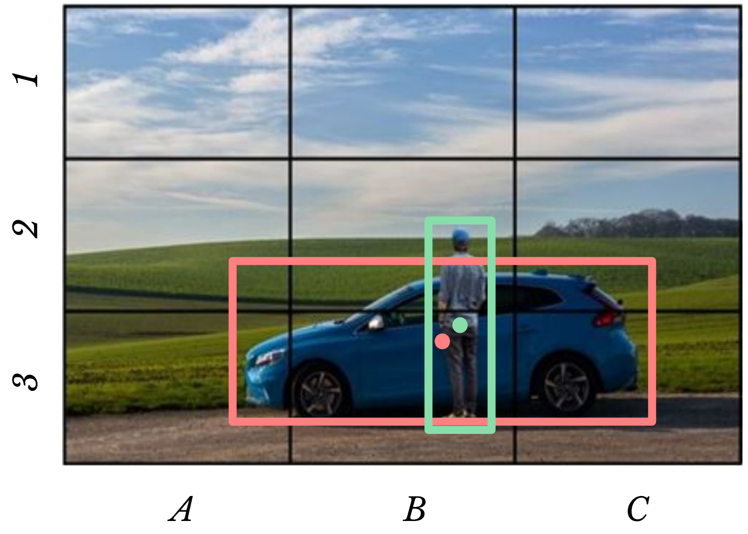




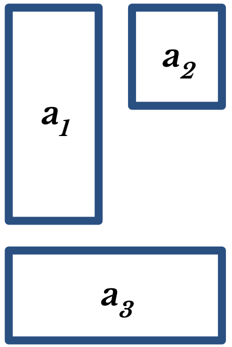
* Anchor boxes were introduced in the YOLO framework with YOLOv2.

When There are Multiple Objects in a Cell

- To handle multiple objects in the same cell, we can (again) use a set of available **Anchors**.



Available Anchors:



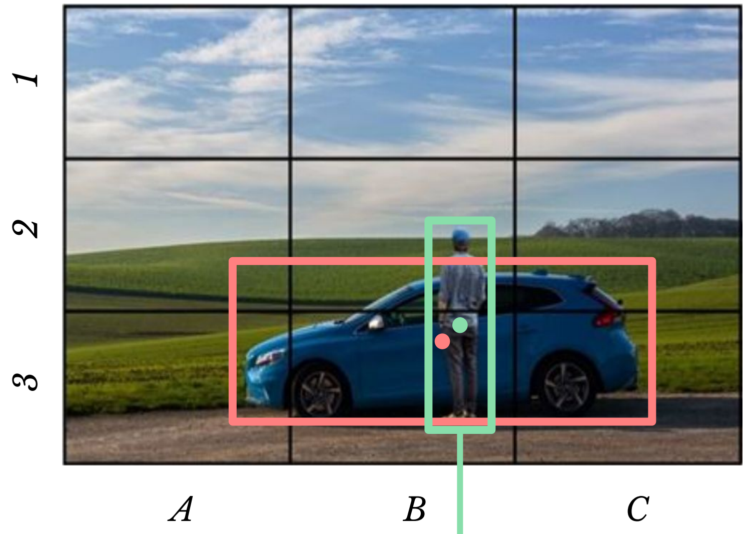
For each cell and for each anchor, we'll compute a vector similar to what was done in Faster RCNN.

Vectors for cell B3 (one for each anchor):

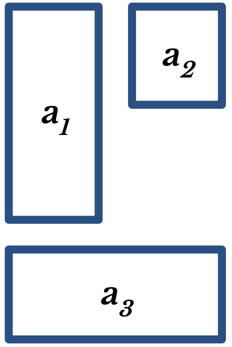
	a_1	a_2	a_3
$o =$			
$b_x =$			
$b_y =$			
$b_w =$			
$b_h =$			
$c_1 =$			
$c_2 =$			
$c_3 =$			

When There are Multiple Objects in a Cell

- To handle multiple objects in the same cell, we can (again) use a set of available **Anchors**.



Available Anchors:



Now for each object in the image, we check which anchor has the most similar shape to it.

In the case of the person on the right, it is anchor a_1.

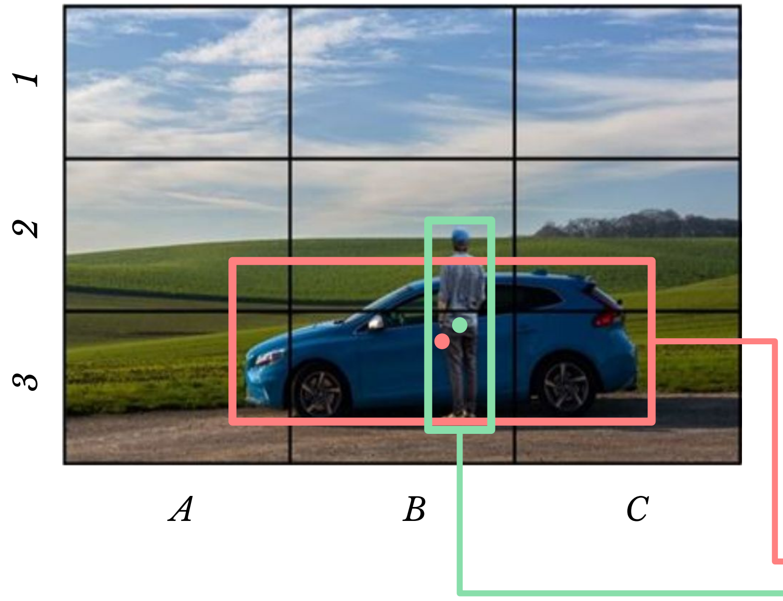
Having decided that, we fill in the vector respective to the anchor, making sure that o = 1 in it.

Vectors for cell B3 (one for each anchor):

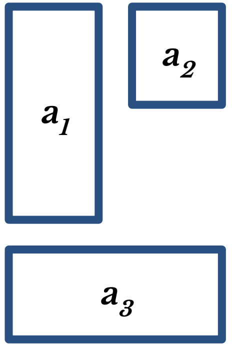
	a_1	a_2	a_3
$o =$	1		
$b_x =$	$.81$		
$b_y =$	$.03$		
$b_w =$	$.2$		
$b_h =$	1.5		
$c_1 =$	1		
$c_2 =$	0		
$c_3 =$	0		

When There are Multiple Objects in a Cell

- To handle multiple objects in the same cell, we can (again) use a set of available **Anchors**.



Available Anchors:



Now for each object in the image, we check which anchor has the most similar shape to it.

In the case of the car on the right, it is anchor a_3.

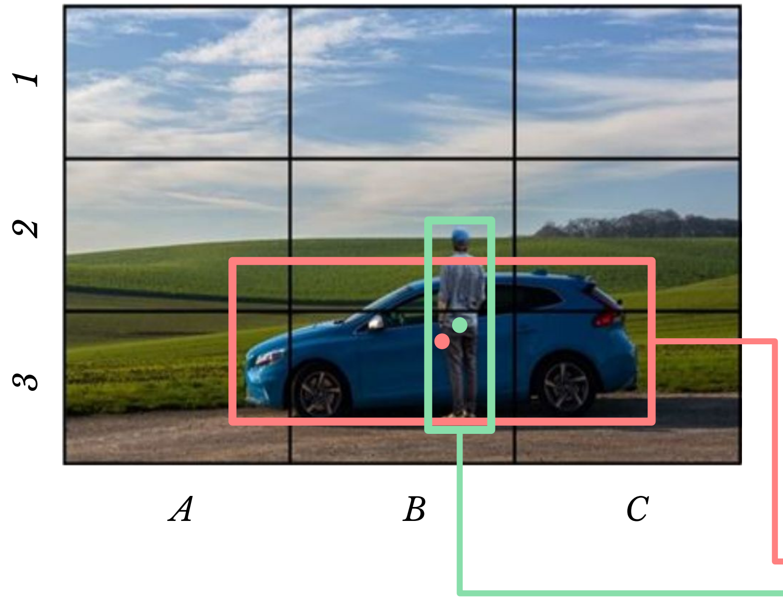
Having decided that, we fill in the vector respective to the anchor, making sure that $o = 1$ in it.

Vectors for cell B3 (one for each anchor):

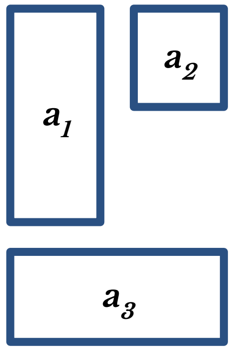
	a_1	a_2	a_3
$o =$	1		1
$b_x =$	.81		.78
$b_y =$	.03		.05
$b_w =$	.2		2.1
$b_h =$	1.5		1.2
$c_1 =$	1		0
$c_2 =$	0		1
$c_3 =$	0		0

When There are Multiple Objects in a Cell

- To handle multiple objects in the same cell, we can (again) use a set of available **Anchors**.



Available Anchors:



Now for each object in the image, we check which anchor has the most similar shape to it.

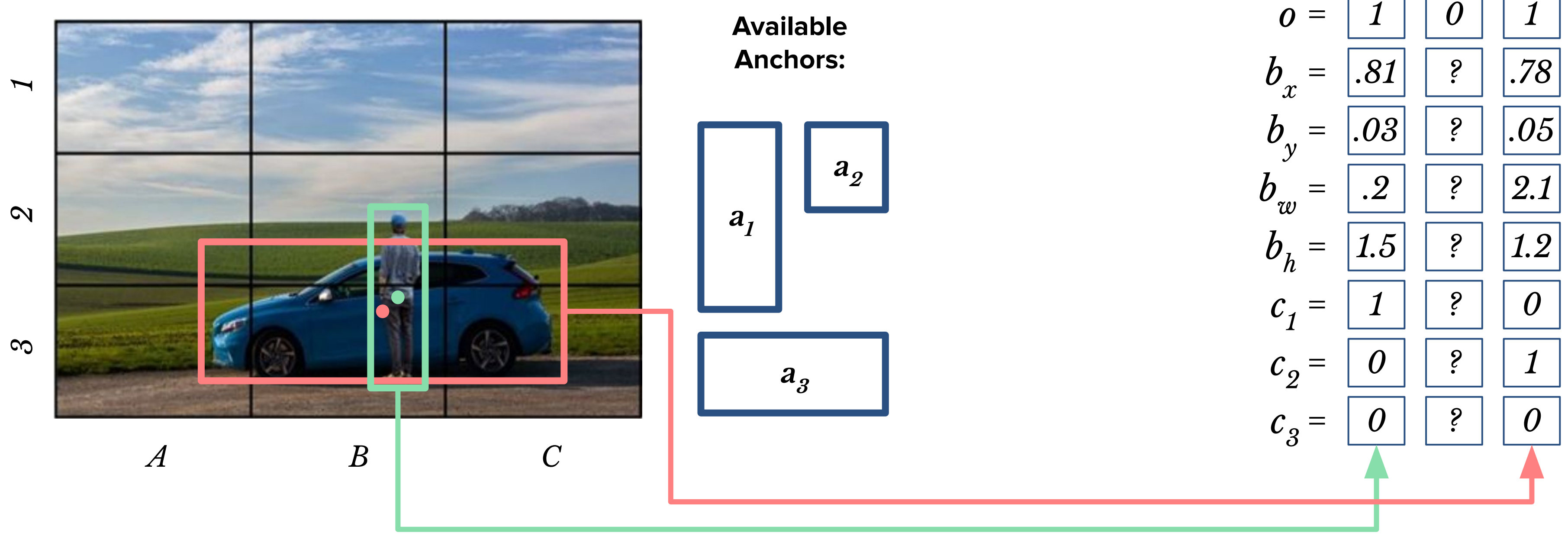
If there is an anchor that wasn't chosen by any object, we make $o = 0$ in its respective vector.

Vectors for cell B3 (one for each anchor):

	a_1	a_2	a_3
$o =$	1	0	1
$b_x =$	.81	?	.78
$b_y =$	.03	?	.05
$b_w =$	.2	?	2.1
$b_h =$	1.5	?	1.2
$c_1 =$	1	?	0
$c_2 =$	0	?	1
$c_3 =$	0	?	0

When There are Multiple Objects in a Cell

- If we stack all anchor vectors per cell, our output is a tensor of shape *(Nc, Nc, (5 + K)×Na)*, where *Na* is the number of available anchors.



Vectors for cell B3 (one for each anchor):

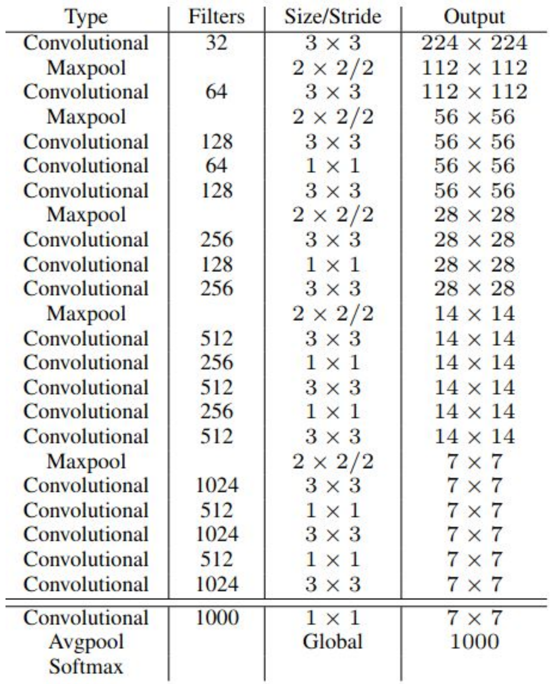
	a_1	a_2	a_3
$o =$	1	0	1
$b_x =$	.81	?	.78
$b_y =$	.03	?	.05
$b_w =$	.2	?	2.1
$b_h =$	1.5	?	1.2
$c_1 =$	1	?	0
$c_2 =$	0	?	1
$c_3 =$	0	?	0

Training of YOLO and Comparisons

- Since the input (an image) and output (the tensor from the previous slide) of YOLO are tensors, it can be trained using a **Fully Convolutional Network (FCN)**.
- An FCN is a network made of only ConvLayers and no Fully Connected / Dense Layers, which are training and inference-wise expensive.
- YOLO (with its network called Darknet-19) was able to reach Real Time* Detection and was **much faster** than R-CNN.
  - Fast R-CNN: 0.5 Frames Per Second (FPS).
  - Faster R-CNN : 7 FPS (VGG16), 5 FPS (ResNet)
  - YOLO (no FCN): 45 FPS.
  - YOLO (FCN): 67 FPS (for 416×416 images).

* This was introduced with YOLOv2. The original YOLO conventionally made use of FC layers.
** Real time performance is usually considered to be attained at 60 FPS.

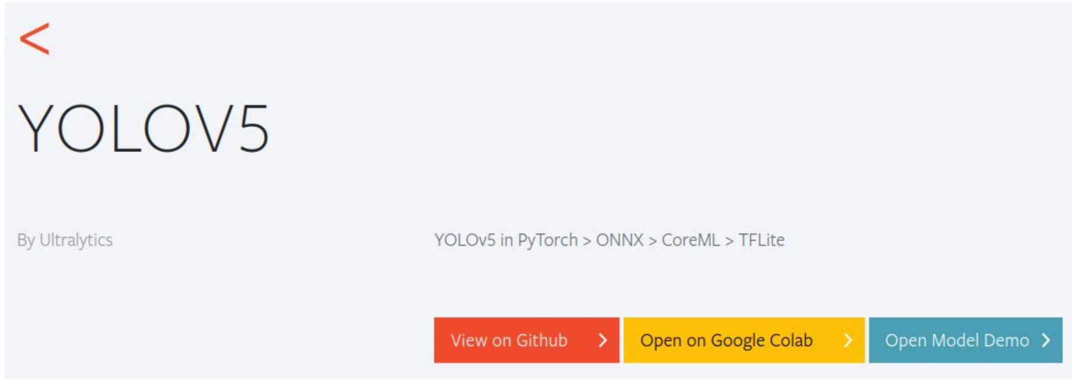
Darknet-19 Architecture



Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Versions of YOLO

- There were many improvements on the original YOLO model (YOLOv2, YOLO9000, YOLOv3, YOLOv4+, etc.) which improved its detection of small objects, overall accuracy and speed.
- One of the latest update in YOLO (YOLOv5) was introduced in 2020 (in a blog post!) and it does predictions **at 140 FPS**!
- You can easily access YOLOv5 it and used in you PyTorch code via TorchHub.
- There you also find YOLOP, for Panoptic Driving vision, i.e., it comprises Object (people, car, etc.) and Lane Detection.



YOLOV5

By Ultralytics

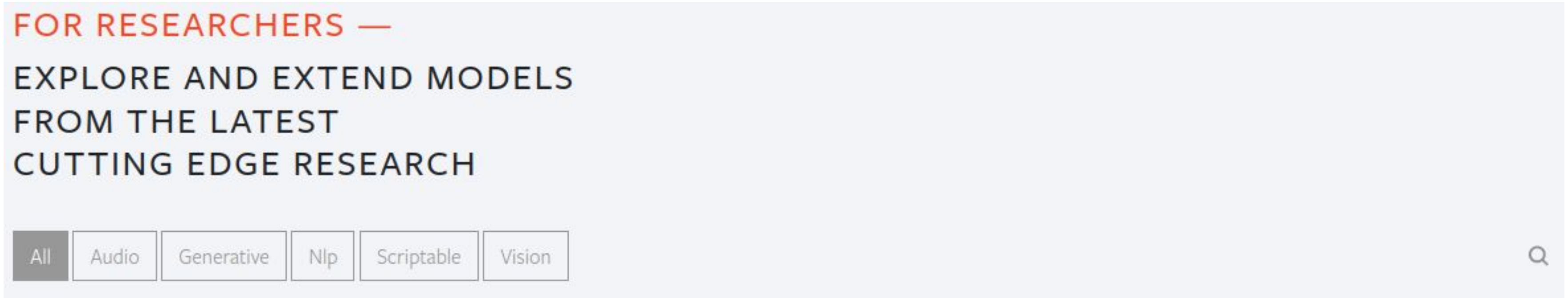
YOLOv5 in PyTorch > ONNX > CoreML > TFLite

PyTorch Hub

- In PyTorch Hub you find many more easily accessible pretrained DL models in PyTorch for Vision, NLP, Audio, etc. **It's worth checking out!**

PYTORCH
HUB

Discover and publish models to a pre-trained model repository designed for research exploration.

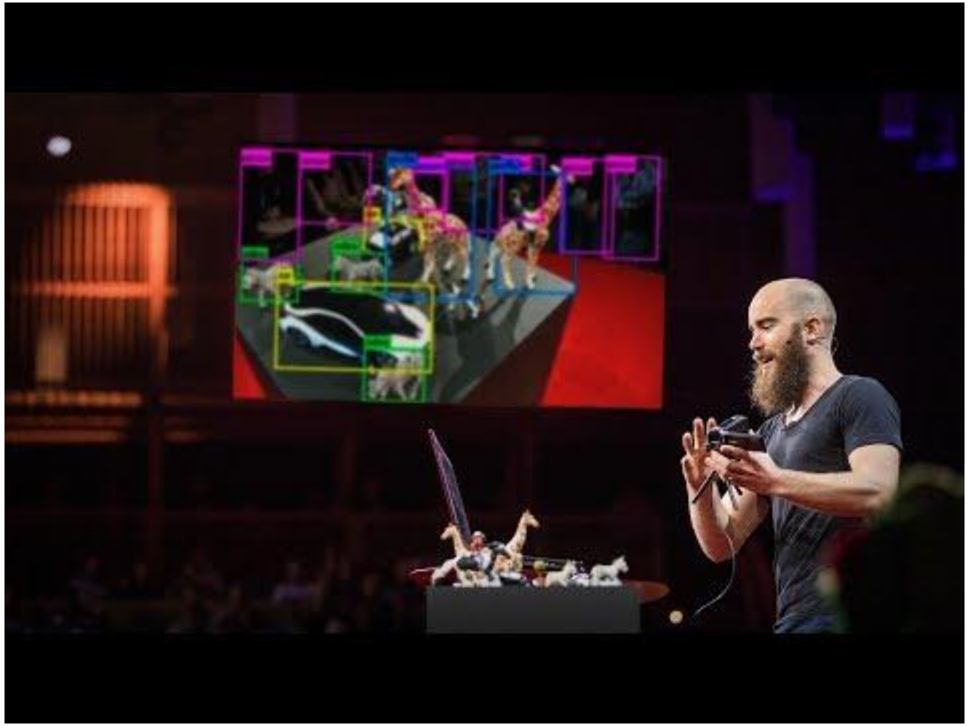

FOR RESEARCHERS —

EXPLORE AND EXTEND MODELS FROM THE LATEST CUTTING EDGE RESEARCH

Exercise (*In pairs*)

- Say you have a fully trained Yolo model and you have one image to which you want to do Object Detection (inference). How do you proceed?

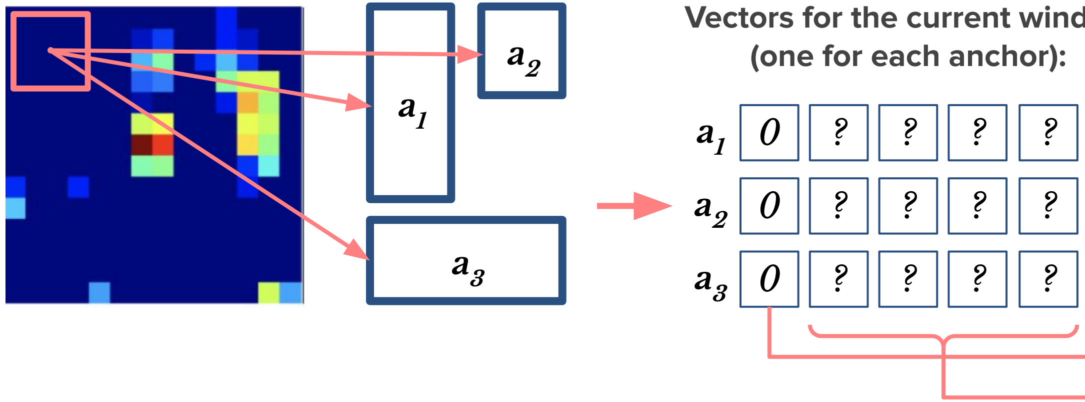
Video: *You Only Look Once*



Appendix: Region Proposal Network

- The RPN network predicts the object proposals using Deep Learning!
- In this network, we aim at predicting a vector of 5 dimensions for each window of the output and for each available **Anchor** (which are some predefined rectangular shapes).
- With this, we see if a given window has an object of size similar to an anchor and its BB.

Region Network Proposal with 3 anchors

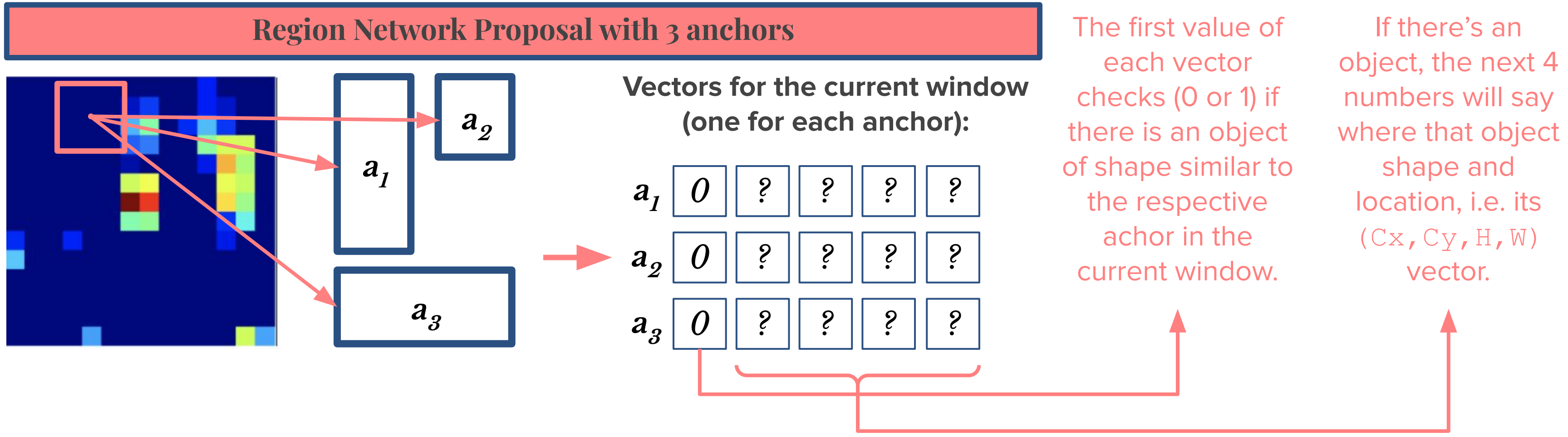


The first value of each vector checks (0 or 1) if there is an object of shape similar to the respective achor in the current window.

If there isn't an object in the current window can take any number (*more on it later*).

Appendix: Region Proposal Network

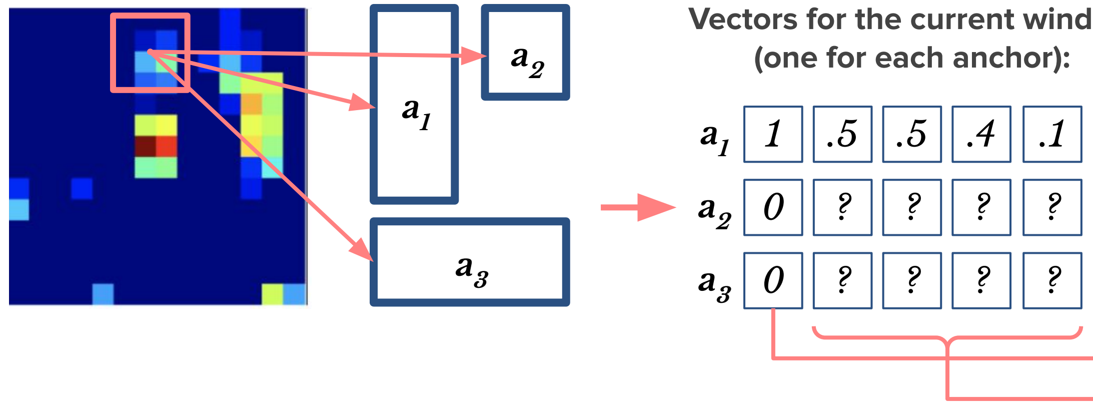
- The RPN network predicts the object proposals using Deep Learning!
- In this network, we aim at predicting a vector of 5 dimensions for each window of the output and for each available **Anchor** (which are some predefined rectangular shapes).
- With this, we see if a given window has an object of size similar to an anchor and its BB.


Region Network Proposal with 3 anchors
a_1
a_2
a_3
Vectors for the current window (one for each anchor):
a_1 0 ? ? ? ?
a_2 0 ? ? ? ?
a_3 0 ? ? ? ?
The first value of each vector checks (0 or 1) if there is an object of shape similar to the respective achor in the current window.
If there's an object, the next 4 numbers will say where that object shape and location, i.e. its (Cx,Cy,H,W) vector.

Appendix: Region Proposal Network

- The RPN network predicts the object proposals using Deep Learning!
- In this network, we aim at predicting a vector of 5 dimensions for each window of the output and for each available **Anchor** (which are some predefined rectangular shapes).
- With this, we see if a given window has an object of size similar to an anchor and its BB.

Region Network Proposal with 3 anchors



Vectors for the current window (one for each anchor):

a_1	1	.5	.5	.4	.1
a_2	0	?	?	?	?
a_3	0	?	?	?	?

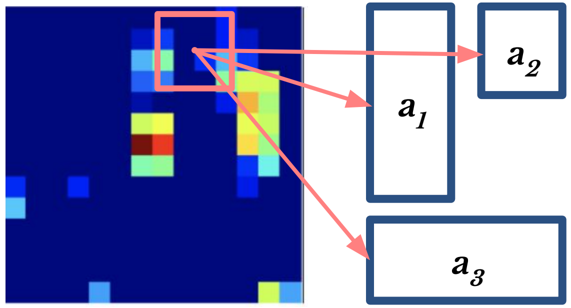
The first value of each vector checks (0 or 1) if there is an object of shape similar to the respective achor in the current window.

If there's an object, the next 4 numbers will say where that object shape and location, i.e. its `(Cx,Cy,H,W)` vector.

Appendix: Region Proposal Network

- The RPN network predicts the object proposals using Deep Learning!
- In this network, we aim at predicting a vector of 5 dimensions for each window of the output and for each available **Anchor** (which are some predefined rectangular shapes).
- With this, we see if a given window has an object of size similar to an anchor and its BB.

Region Network Proposal with 3 anchors



Vectors for the current window (one for each anchor):

a_1	1	.5	.5	.4	.1
a_2	1	.7	.2	.3	.3
a_3	0	?	?	?	?

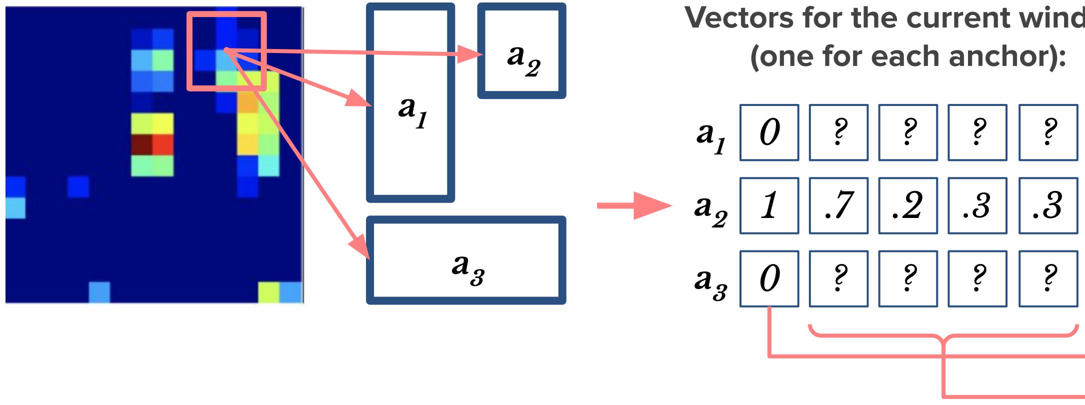
The first value of each vector checks (0 or 1) if there is an object of shape similar to the respective achor in the current window.

If there's an object, the next 4 numbers will say where that object shape and location, i.e. its `(Cx,Cy,H,W)` vector.

Appendix: Region Proposal Network

- The RPN network predicts the object proposals using Deep Learning!
- In this network, we aim at predicting a vector of 5 dimensions for each window of the output and for each available **Anchor** (which are some predefined rectangular shapes).
- With this, we see if a given window has an object of size similar to an anchor and its BB.

Region Network Proposal with 3 anchors



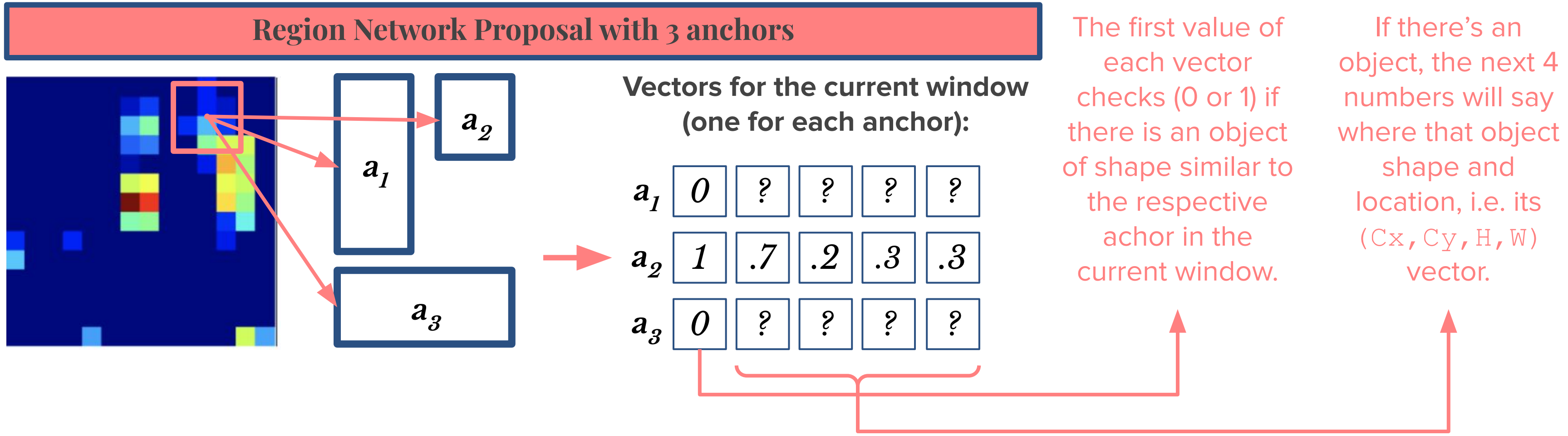
a_1	0	?	?	?	?
a_2	1	.7	.2	.3	.3
a_3	0	?	?	?	?

The first value of each vector checks (0 or 1) if there is an object of shape similar to the respective achor in the current window.

If there's an object, the next 4 numbers will say where that object shape and location, i.e. its `(Cx,Cy,H,W)` vector.

Appendix: Region Proposal Network

- After training, we pass a feature map through the RPN and check the network's output.
- If, for a given window, the first value of the vector (the *confidence* that there is an object) is close *1*, we consider the other values in it and say we detected an object in that window.
- The paper uses 9 anchors: 3 different shapes (2 rectangles and 1 square) in 3 scales.


Region Network Proposal with 3 anchors
a_1
a_2
a_3
Vectors for the current window (one for each anchor):
The first value of each vector checks (0 or 1) if there is an object of shape similar to the respective achor in the current window.
If there's an object, the next 4 numbers will say where that object shape and location, i.e. its (Cx,Cy,H,W) vector.

a_1	*0*	*?*	*?*	*?*	*?*
a_2	*1*	*.7*	*.2*	*.3*	*.3*
a_3	*0*	*?*	*?*	*?*	*?*