**Bounding boxes**

So far, we worked with image classification. In this video, we will talk about bounding boxes and object recognition.

**What is object recognition?**

Object recognition identifies objects in images. Think of a self-driving car. Its systems need to identify the location of all objects on the road such as other cars and pedestrians. This is typically achieved by drawing bounding boxes around the objects. All localized objects must then be identified with their class label. Object recognition is used in many applications such as surveillance, medical diagnosis, traffic management, or sports analytics. We'll be reviewing how to annotate image data with bounding boxes in this video while later videos will cover model evaluation and explore two different model structures for object recognition.

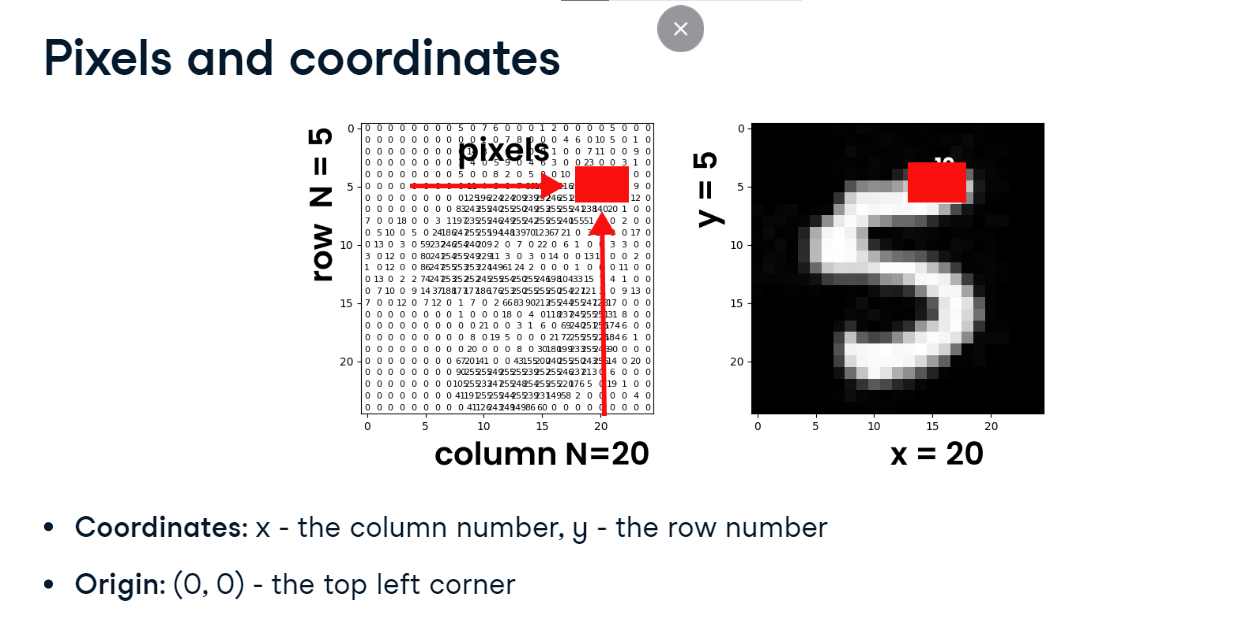
**Bounding box representation**

A bounding box, like this red rectangle around the cat, describes an object's spatial location within the image. Bounding boxes are used for annotating training data. They are also the outputs of object recognition models. A ground truth bounding box precisely outlines the location of an object within an image.

A bounding box is typically described by its top left and bottom right coordinates. These four numbers: x1, y1, x2 and y2 define each bounding box. Sometimes, the one and two are referred to as min and max, respectively, so that x1 is x\_min, x2 is x\_max, and

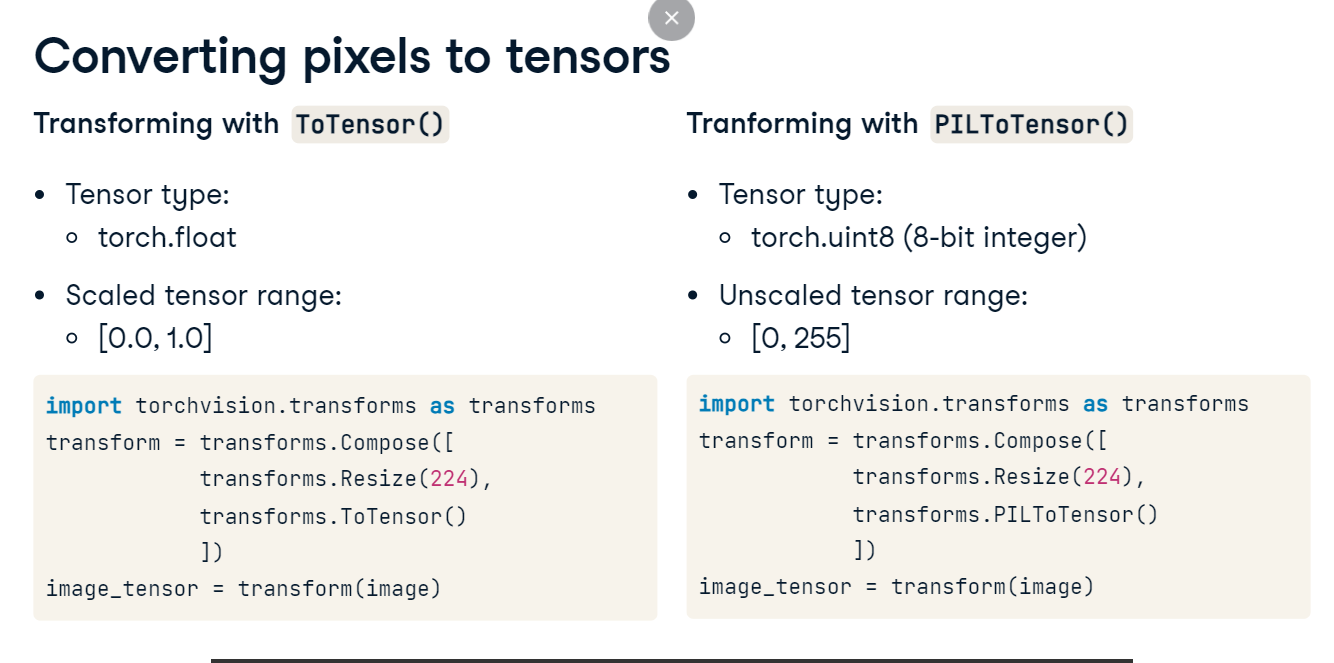
**Pixels and coordinates**

An image consists of pixels. Pixels provide a way to specify the location, size, and object boundaries within an image. Each pixel is defined by the column number or the x-coordinate and the row number or the y-coordinate. The origin of the image (the very first pixel) has the coordinates x equals zero and y equals zero and is at the top-left corner. For example, the pixel in the column twenty and row five corresponds to x and y equal to twenty and five, respectively.



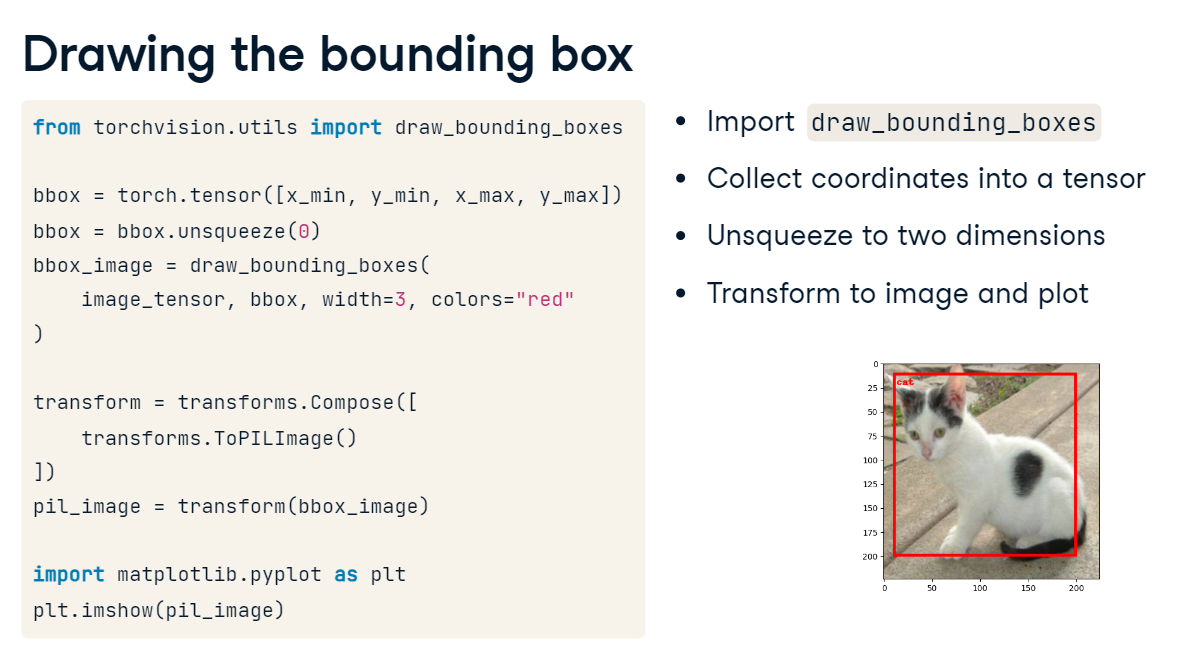
**Converting pixels to tensors**

**To be able to process images in PyTorch, we must convert pixel arrays to tensors.** There are two transforms, ToTensor() and PILToTensor() that produce different output formats. ToTensor converts pixels to float tensors, scaling values from zero to one. We import transforms from torchvision-dot-transforms and use transforms-dot-compose to combine transformations. Let's use transforms-dot-resize to set the image size to 224 and apply transforms-dot-ToTensor to create the float tensors. PILToTensor converts pixels to 8-bit integer tensors. Pixel values remain unscaled from zero to 255. We can apply PILToTensor in the same way, just updating the transform function. The PILToTensor transformation is useful for bounding boxes.



**Drawing the bounding box**

Let's see how to draw bounding boxes on top of images! We will use the draw bounding boxes function from torchvision-dot-utils. Assume we know the coordinates, perhaps as predicted by the object recognition model. We collect them into a tensor using the torch-dot-tensor method. Next, we pass the image tensor and the box coordinates to draw bounding boxes, setting line width to three and color to red. To display the box, we convert the tensor to an image using the ToPILImage transform and call imshow from matplotlib. We have a ground truth bounding box around the cat!

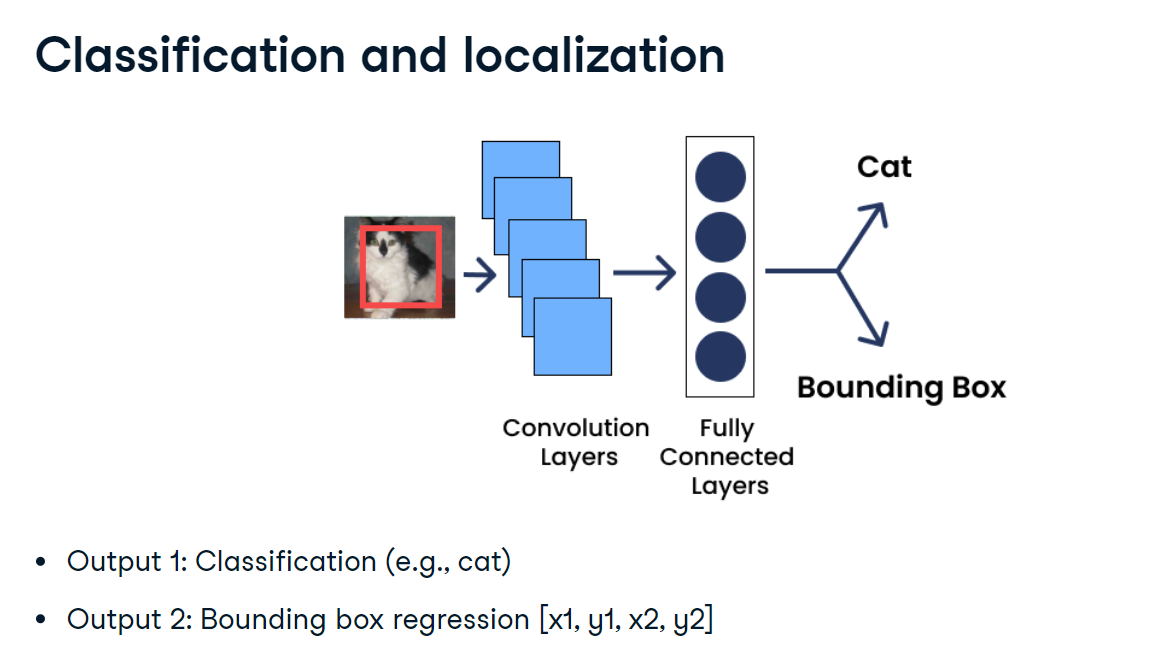


**Evaluating object recognition models**

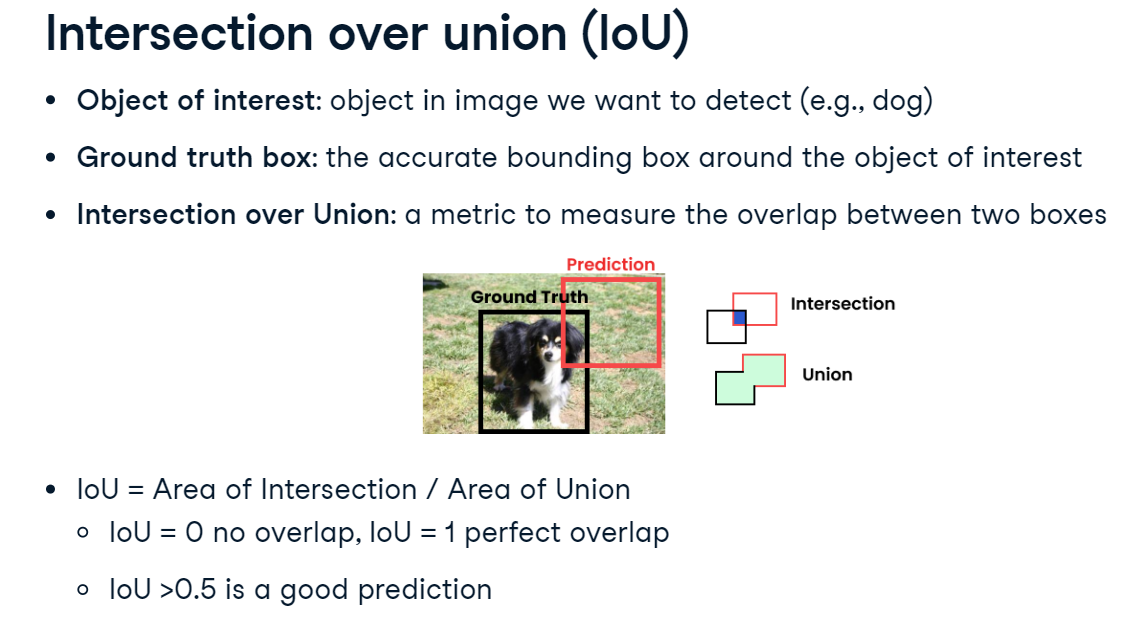
**Classification and localization**

Object recognition requires predicting two things: the class and location of an object in an image. The classification task is similar to the standard image classification we learned earlier.

Additionally, the model learns the bounding box coordinates to properly fit the target object. These coordinates are continuous values making this a regression task.

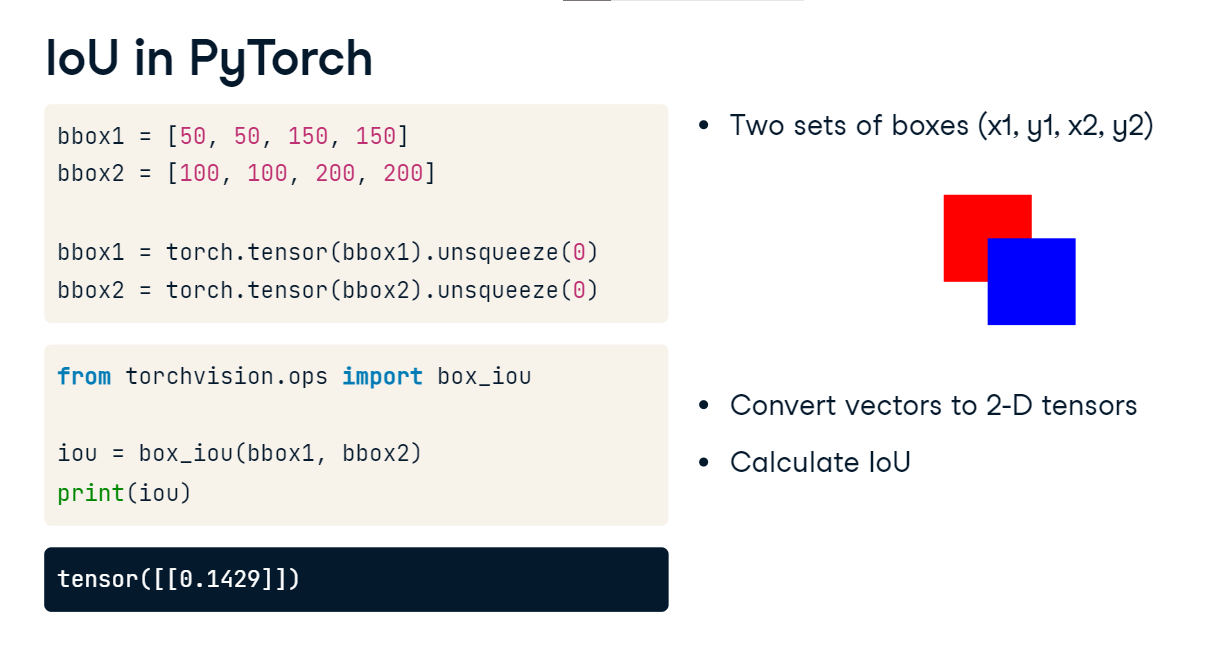


**Intersection over union (IoU)**

Suppose we are interested in detecting dogs - this is our object of interest. We annotated the test dataset with the accurate bounding boxes - these will be our ground truth boxes. Now, our object recognition model predicted a bounding box like this red one for a dog in the image. The ground truth box is colored black. We can see some overlap between these two boxes. How good is our prediction? Intersection over Union, or IoU, is a common metric in object recognition to evaluate the degree of overlap between two boxes. The overlap called an intersection is divided by the area of their union. IoU ranges from zero (no overlap) to one (perfect overlap). The common threshold is point five. Any prediction greater than point five is considered a good prediction.

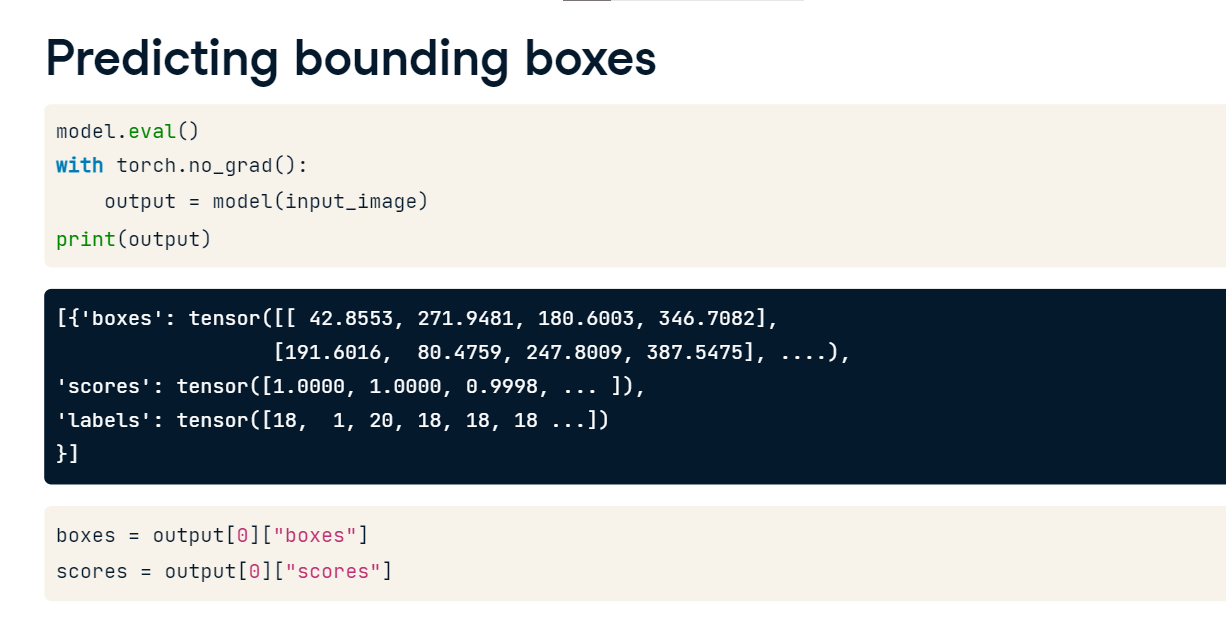
**IoU in PyTorch**

Let's see how we can calculate IoU with PyTorch. Consider this ground truth red box and the predicted blue box. We store their coordinates in two lists called bbox1 and bbox2. To calculate the IoU, we convert them into tensors and reshape them using the unsqueeze method. We import the box iou function from the torchvision-dot-ops module and pass our tensors as arguments. The result is point 14 which is less than the threshold of point five. So the predicted box is not very accurate.



**Predicting bounding boxes**

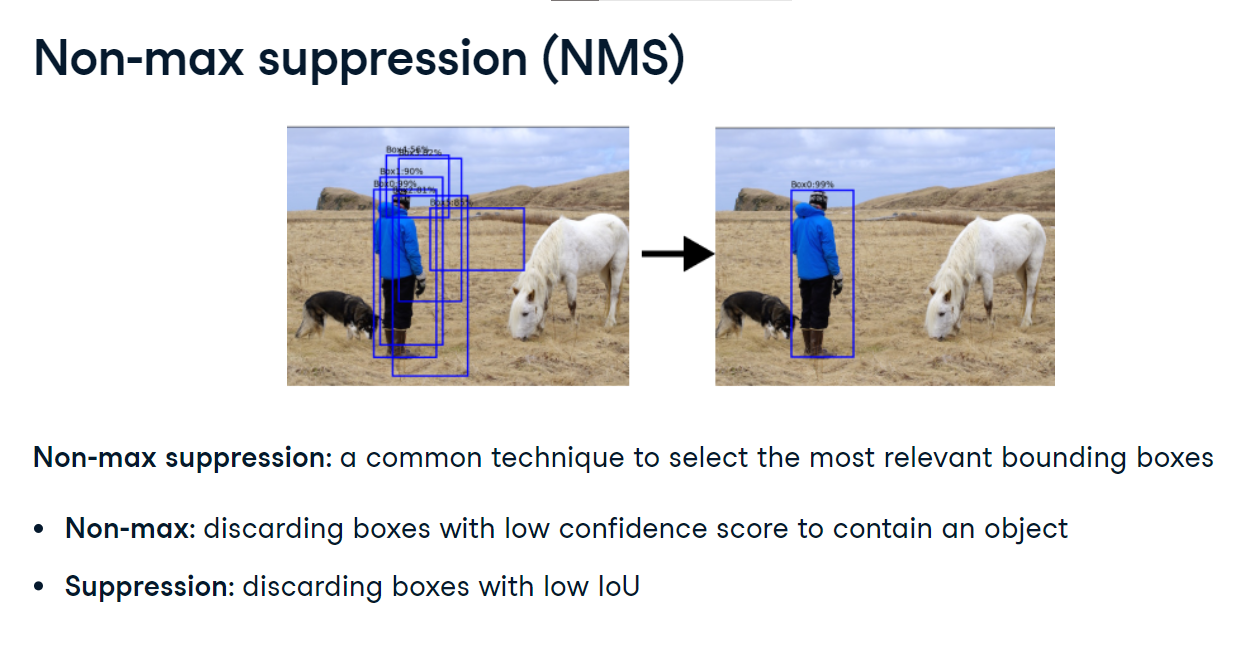
Let's see how to use a trained recognition model to predict bounding boxes. We will look at the model architecture in the next video. We switch the model to evaluation mode to use it for prediction and disable gradients calculation. Then, we pass the input image through the model to get the output predictions. The output is often a list of dictionaries with tensors containing bounding box coordinates of multiple boxes, their associated confidence scores indicating how confident the model is about each box, and predicted class labels for each box. Let's create a variable named boxes and extract coordinates from the output by accessing the first dictionary in the list with the key word boxes. Next, we create a variable named scores and extract confidence scores with the key word scores.



**Non-max suppression (NMS)**

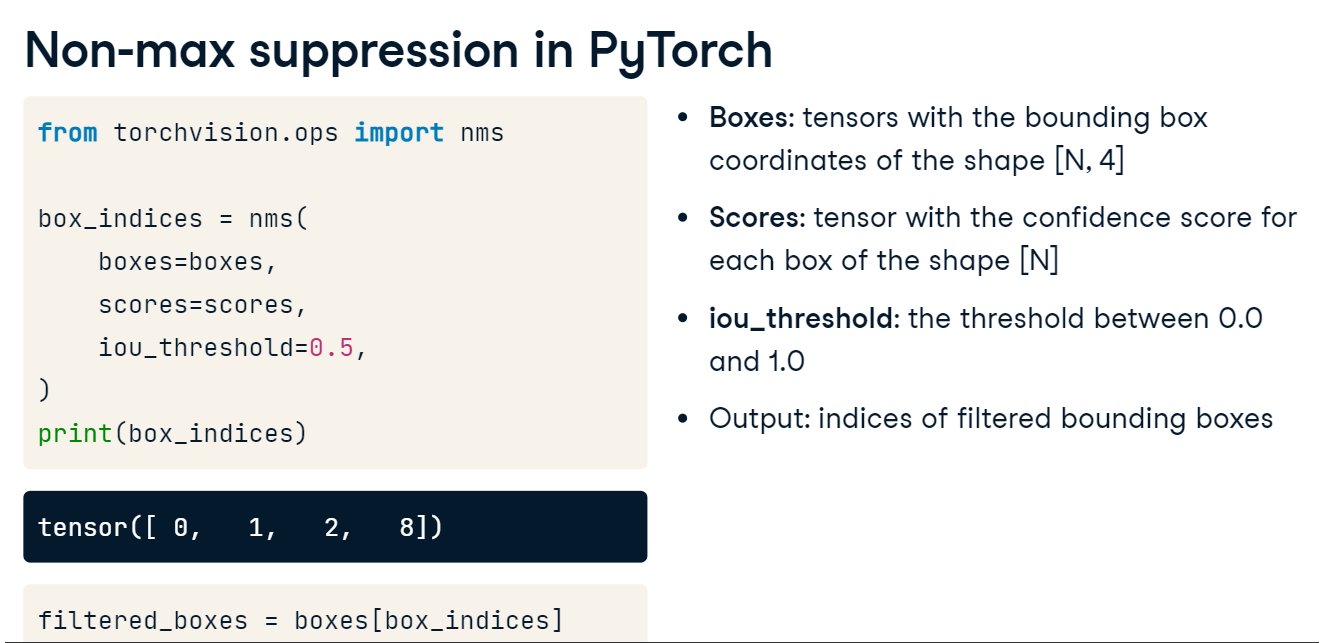
As we just saw, object recognition models may generate many bounding boxes and some of them may be overlapping near-duplicates.

Our goal is to discard unnecessary boxes. Non-max suppression, or NMS, is a common technique in object recognition to select the most relevant box for our object of interest by discarding boxes with low confidence scores and below the IoU threshold.



**Non-max suppression in PyTorch**

Let's apply nms in PyTorch. We start with importing the nms function from torchvision-dot-ops. We then pass it three arguments: boxes, a two-dimensional tensor with the four bounding box coordinates for N boxes, scores, a one-dimensional tensor with a confidence score for each box, and the iou threshold, which we set to point five here. As a result, we get a list of the most relevant bounding boxes with no overlapping duplicates. The output of the nms function is a tensor containing the indices of the filtered boxes after non-maximum suppression. We can use these indices to filter our bounding boxes by retaining only the selected boxes.



**Object detection using R-CNN**

In this video, we will apply what we have learned about bounding boxes and detect objects using R-CNN models.

**Region-based CNN family: R-CNN**

R-CNN is a family of region-based convolutional models for object detection. These models consist of three modules. First, the R-CNN model generates many region proposals. These are potential bounding boxes that might contain objects.

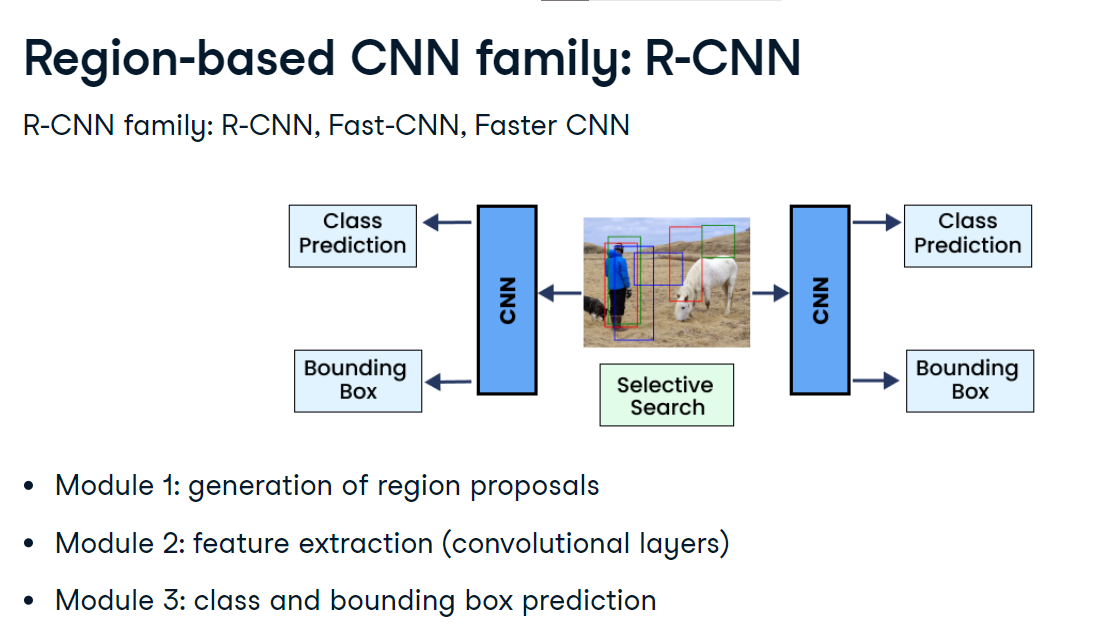
1. 1 Citation: Jason Brownlee. 2019. Deep Learning for Computer Vision.

The second module uses a convolutional neural network to extract features from each region.

1. 1 Citation: Jason Brownlee. 2019. Deep Learning for Computer Vision.

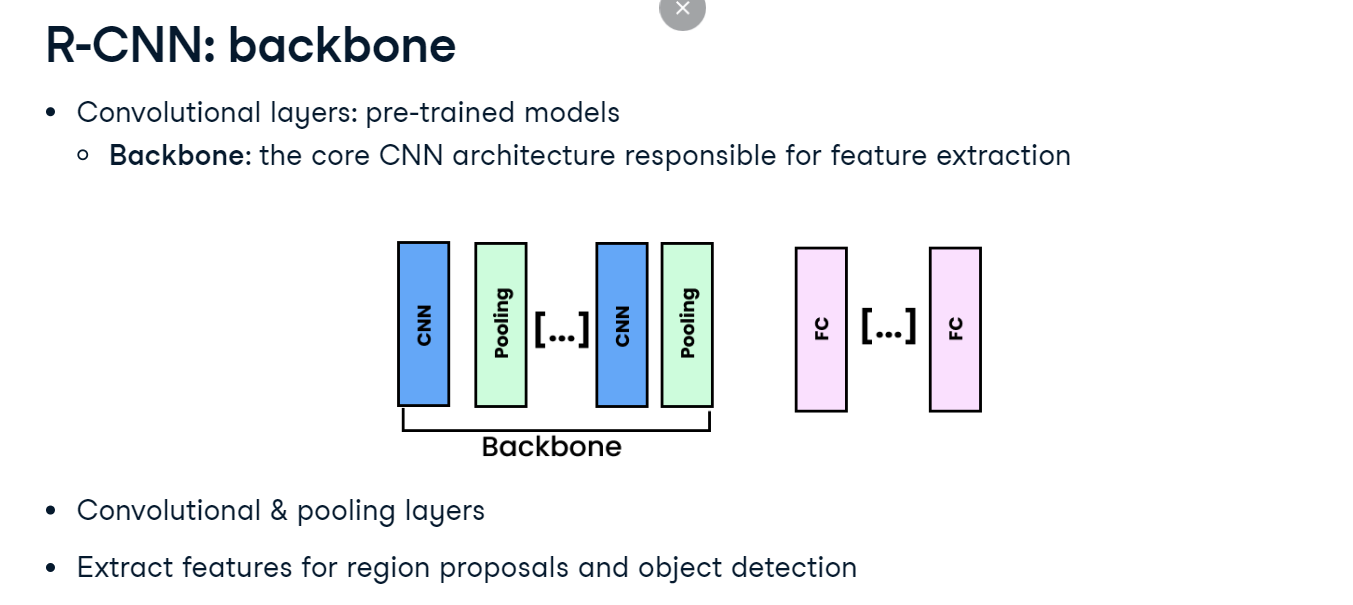
In the third module, features from each region proposal are used to predict the class and bounding box for that region.

1. 1 Citation: Jason Brownlee. 2019. Deep Learning for Computer Vision.



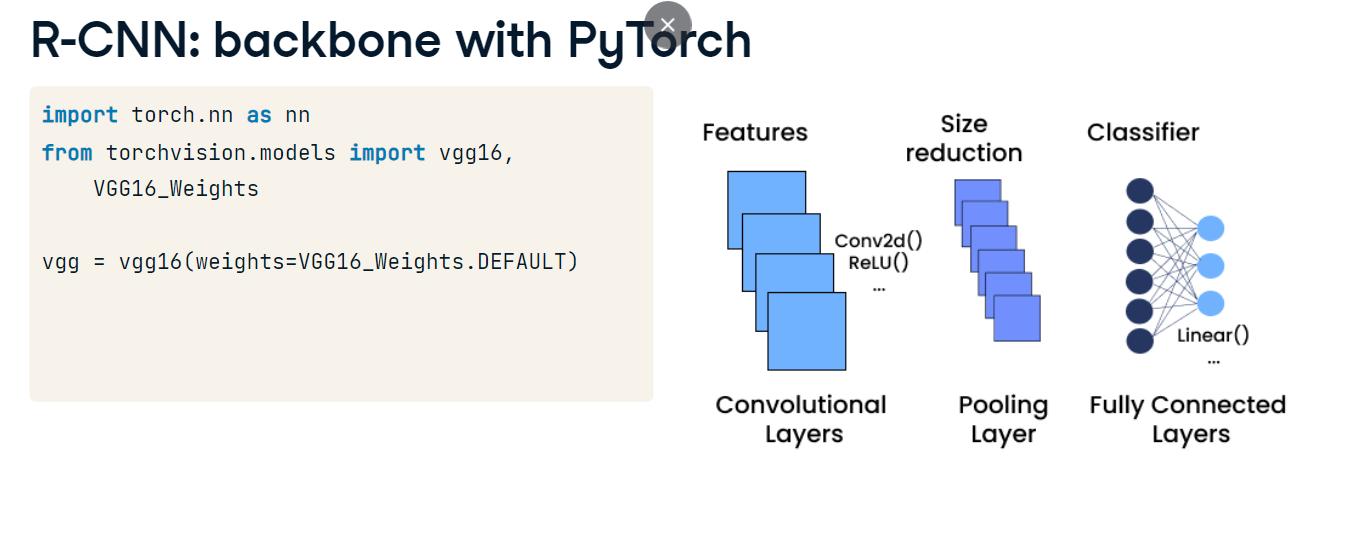
**R-CNN: backbone**

Using a pre-trained model as the backbone is a common strategy. The term backbone refers to the core CNN architecture responsible for feature extraction. The backbone consists of multiple layers of convolutions and pooling operations. These layers extract features for region proposal and object detection.



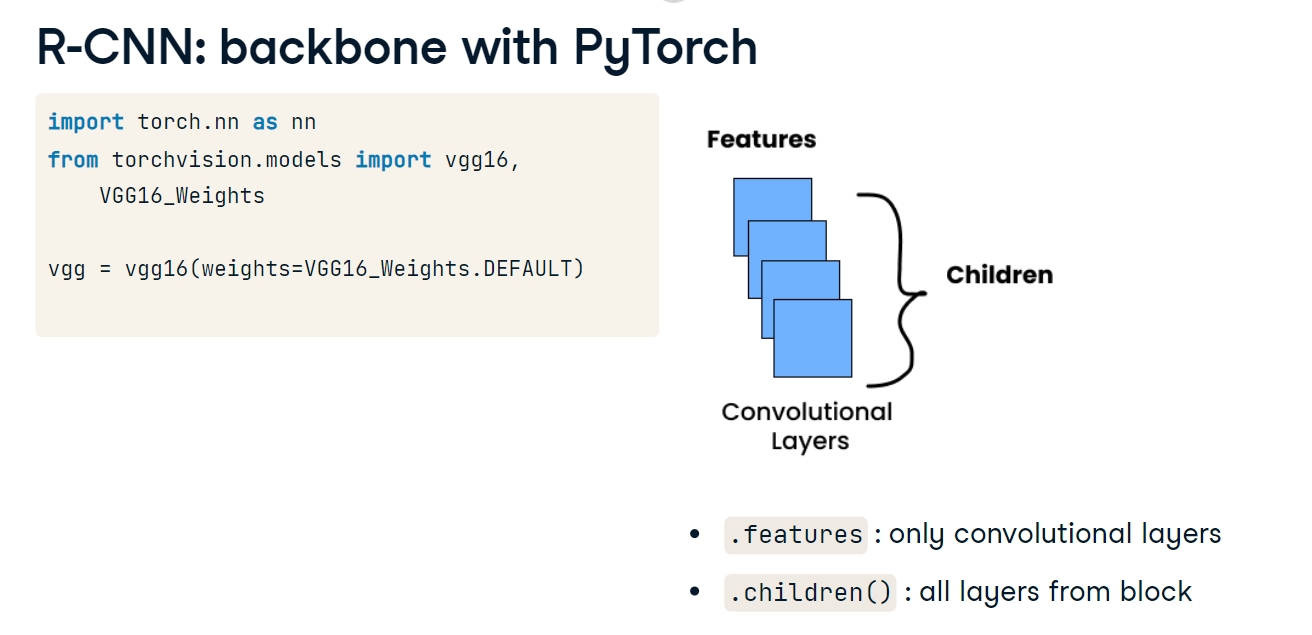
**R-CNN: backbone with PyTorch**

Let's use the backbone from the pre-trained classification model called VGG16. We import torch-dot-nn and the VGG16 model together with weights from torchvision-dot-models, and initialize the model with default pre-trained weights. The VGG model has a features block, a pooling layer for size reduction, and a classifier block with fully connected layers.

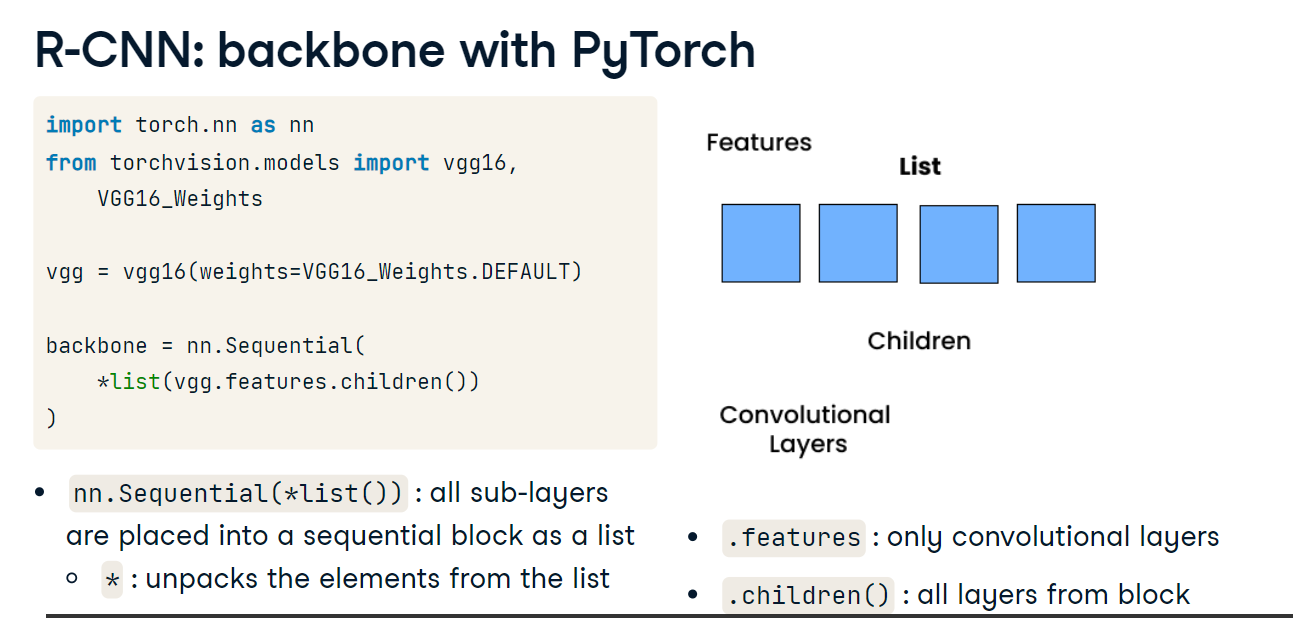


Our goal is to re-use only the features block from the pre-trained VGG model. The dot-features attribute provides access to these convolutional layers.

The children method returns all layers of the features block.



To extract the backbone, we convert all layers from the features block into a list and pass to a new sequential block.



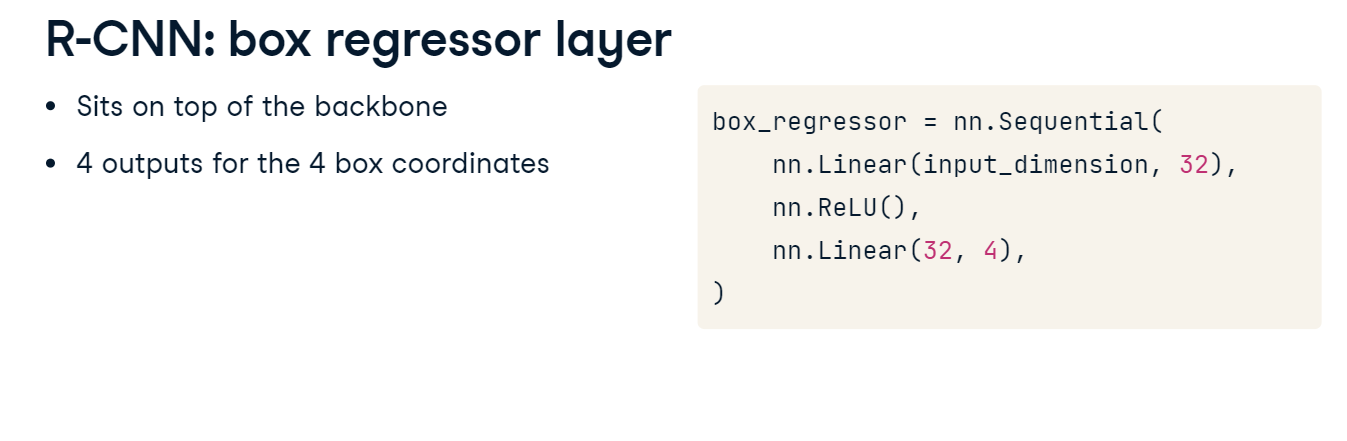
**R-CNN: classifier layer**

Let's define the classifier layer. It comes on top of the backbone, so its input size must match the backbone's output size. To extract the output size of the VGG backbone, we create a list of all layers in model's original classifier block. We extract the first layer from the list using the index zero and dot-in-features and store this value as input\_dimension. Now, we define a new classifier sequential block with two linear layers and the relu activation. The first layer's input dimension is what we defined earlier. The last linear layer has a number of classes as output size.



**R-CNN: box regressor layer**

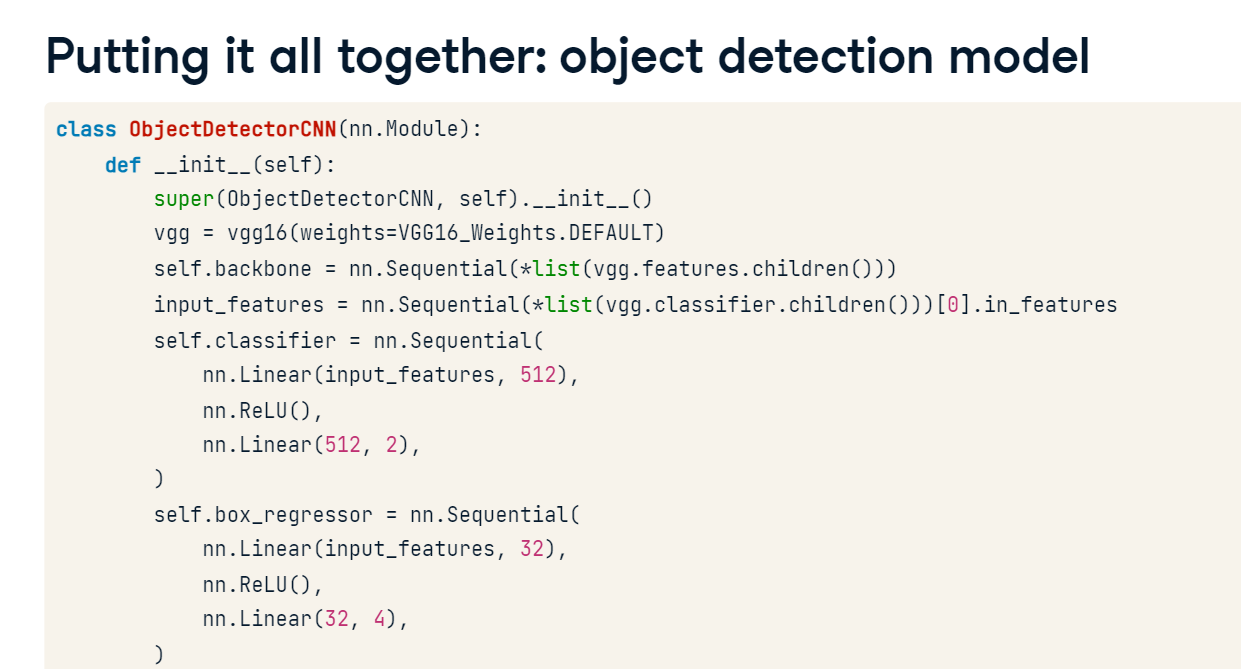
Finally, let's define regressor to predict bounding box coordinates. Similarly to the classifier, it also sits on top of the backbone, so we use the same input size. We define a sequential block with two linear layers and the relu activation. In features is set to the input dimension from the backbone. The second linear layer has an output equal to four, representing the four coordinates to predict.



**Putting it all together: object detection model**

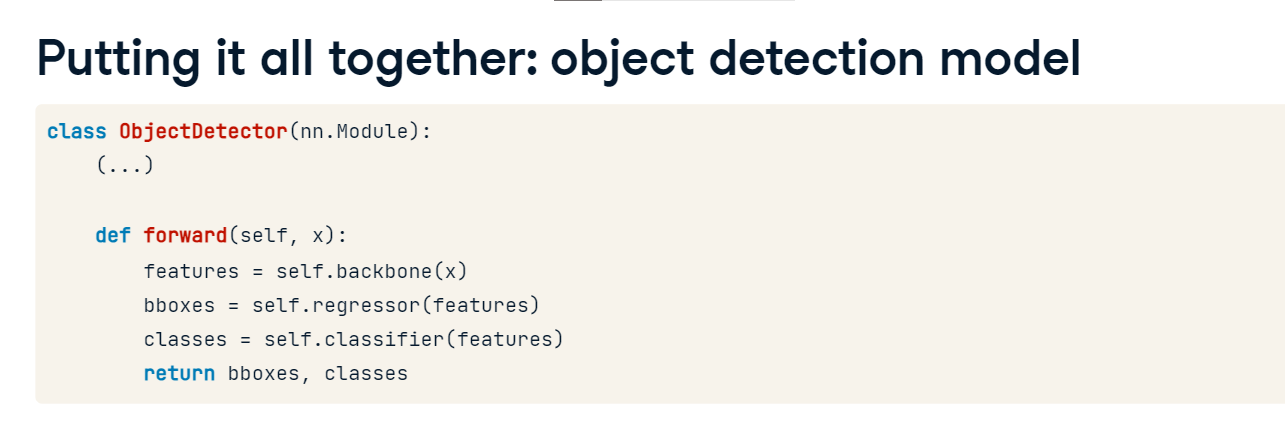
Let's put the backbone, classifier and box regressor together into a one model called ObjectDetectorCNN. In the init method, we extract the VGG16 backbone and assign it to self-dot-backbone. Next, we identify the input shape required for the classifier and regressor, and define both of them just like we have seen before.

We also define the forward method that passes the input through the backbone to extract features. It then processes features separately using the classifier and the bounding box regressor to obtain the two outputs.



**Running object recognition**

With the model at hand, let's recap how to run object recognition for an image. We start by loading and transforming an image to a tensor. Remember to unsqueeze it in order to add the batch dimension. Next, we pass the image tensor to the model and run non-max suppression over model's output Finally, we can draw the bounding box on top of the image.



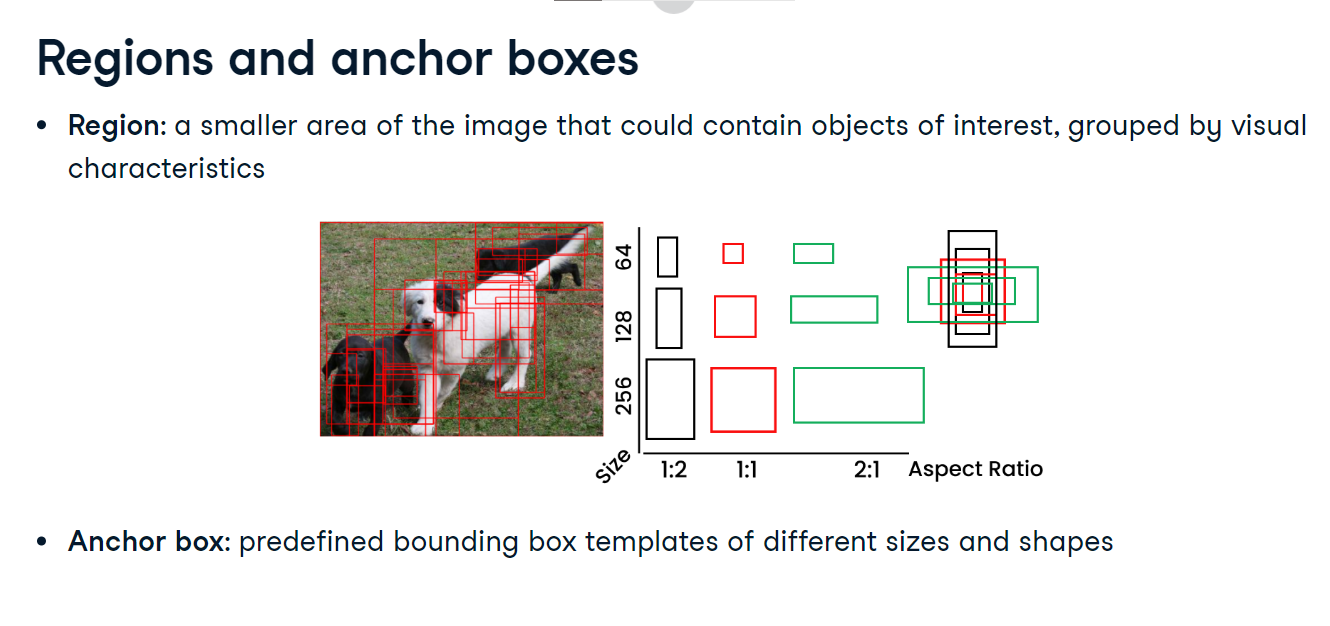
**Region network proposals with Faster R-CNN**

we will cover region proposal networks and the Faster R-CNN model.

**Regions and anchor boxes**

Regions are smaller areas of an image that might contain objects of interest. They are grouped by visual characteristics like color and shape. Detecting regions aids object detection by narrowing down search areas.

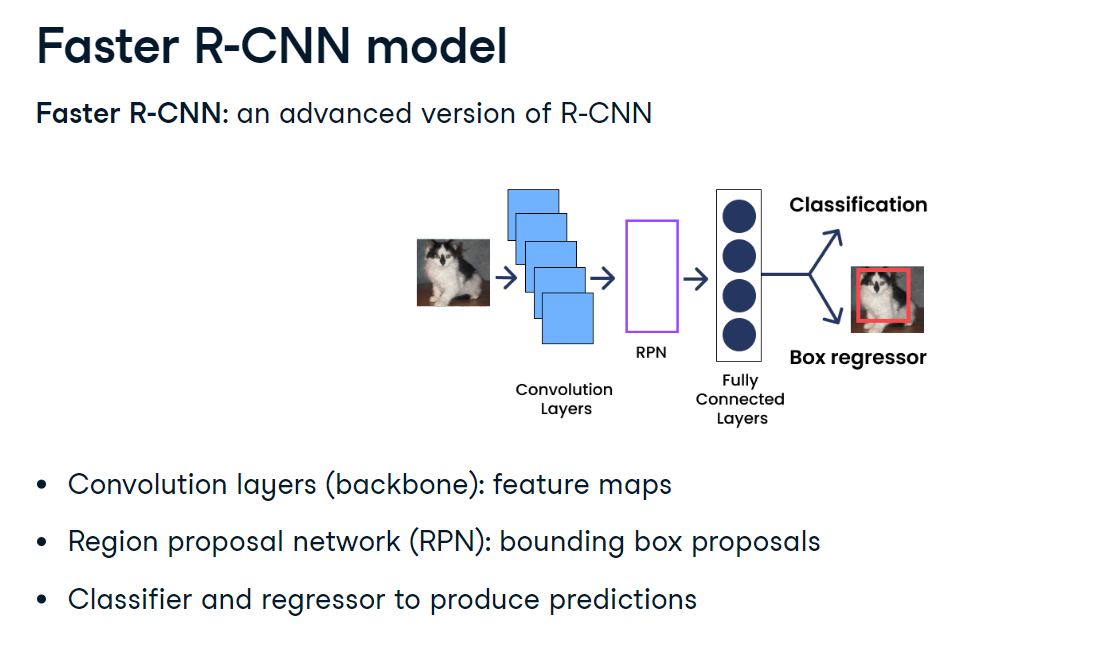
Anchor boxes are often used to help generate the regions. They are pre-defined frames of different sizes and aspect ratios.

****

**Faster R-CNN model**

Faster R-CNN is an advanced version of the previously discussed R-CNN. It consists of 3 modules: a backbone with pre-trained convolutional layers, a region proposal network, or RPN, to generate bounding boxes, and the classifier and regressor like the ones in the regular R-CNN.

1. 1 Edward Raff. 2022. Inside Deep Learning.

****

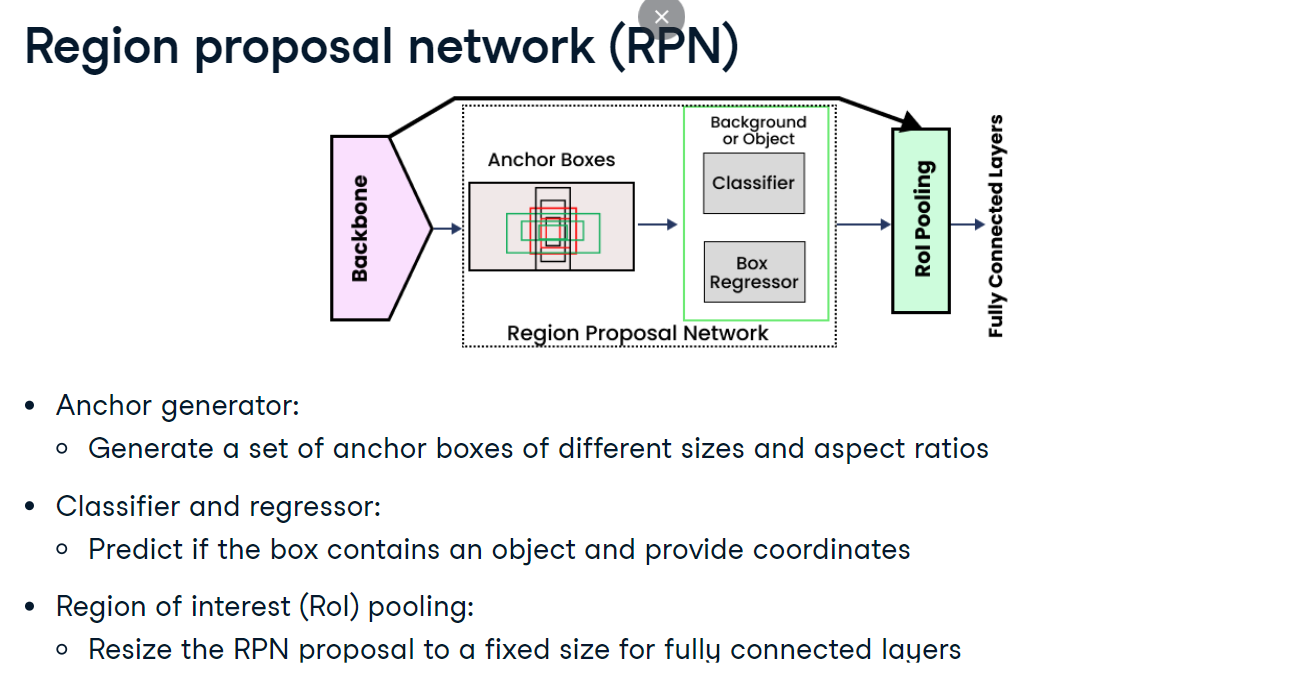
**Region proposal network (RPN)**

The backbone processes the input image and extracts feature maps for the RPN.

The Region Proposal Network starts by generating region proposals. It is faster than the original R-CNN, and trainable end-to-end. It generates multiple anchor boxes of different sizes and aspect ratios on top of the backbone's output.

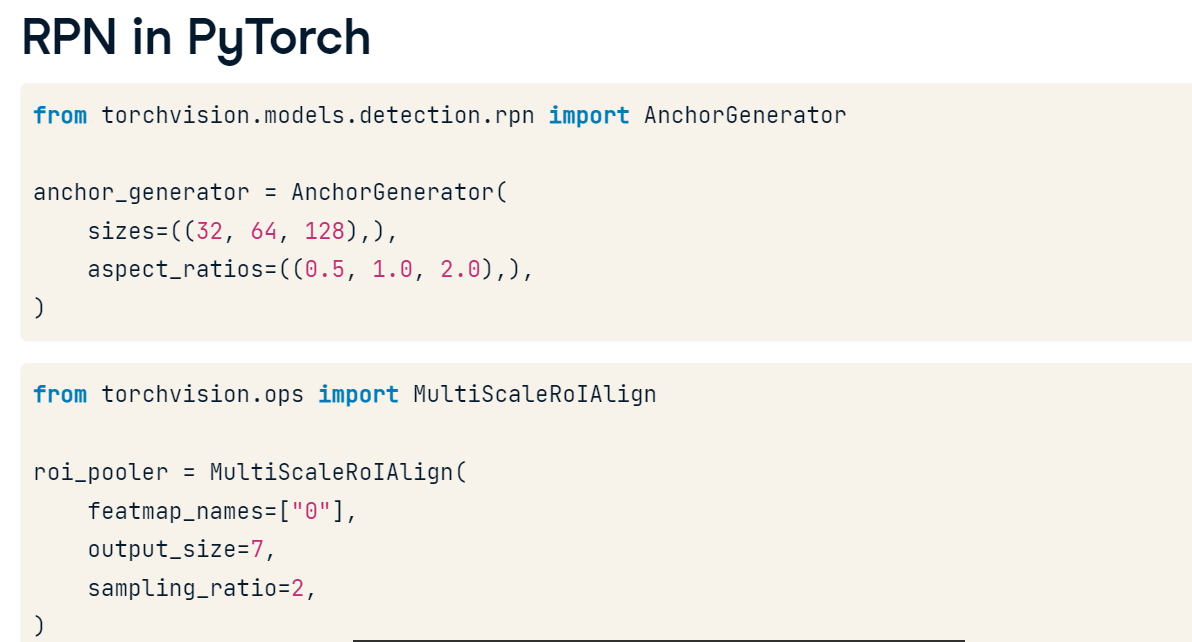
Then, the RPN predicts whether each box contains any object, as well as the box coordinates.

Finally, the proposed regions from the RPN are resized to a fixed size using a process called Region of Interest (RoI) pooling. This allows the regions to be processed by fully connected layers regardless of their original size. These layers then determine the object class and refine bounding box coordinates.



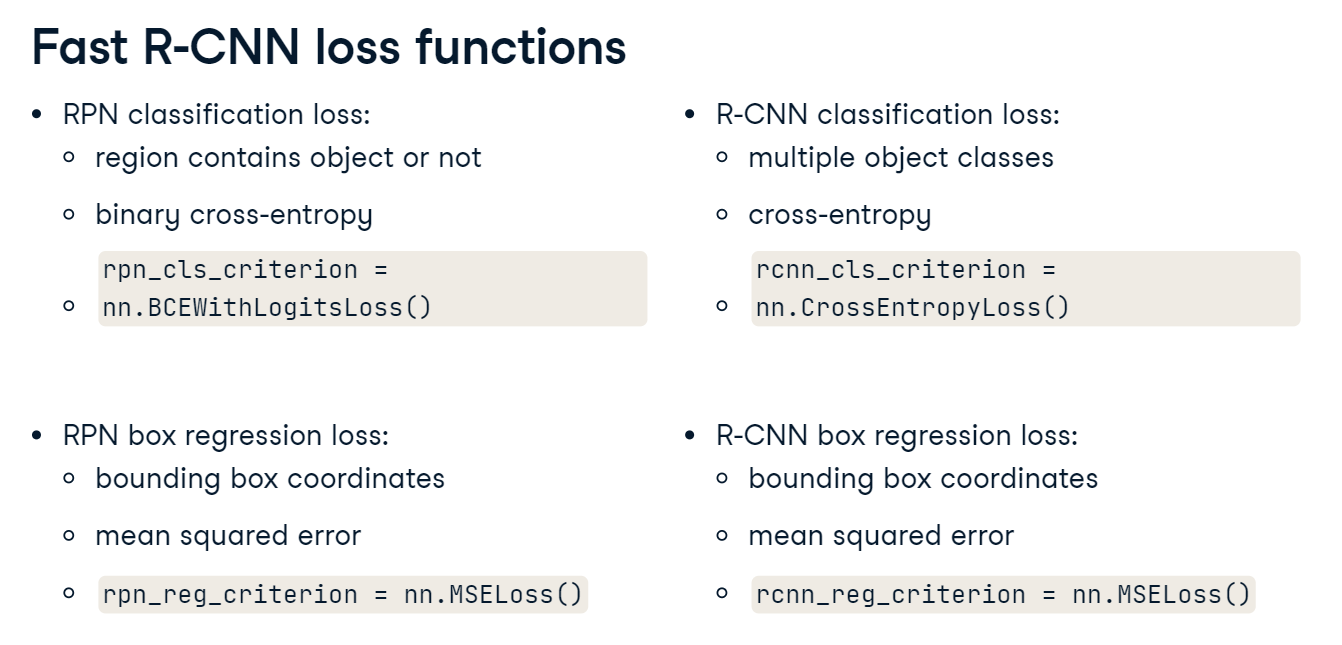
**RPN in PyTorch**

Let's build an RPN in PyTorch! We start by importing anchorgenerator from torchvision-dot-models-detection-rpn. We instantiate the anchor generator and specify sizes and aspect ratios for the boxes. Faster R-CNN typically uses three scales and three aspect ratios, resulting in nine anchor boxes. For RoI pooling, we import the MultiScaleRoIAlign class module from torchvision-dot-ops. We create a pooler by specifying the backbone layer name. Here we choose the first layer labeled zero in our backbone architecture. We also pass two other parameters. Output size determines the size of the output after pooling, while sampling ratio specifies how many samples are taken from each bin when pooling. We will set them to 7 and 2, respectively.



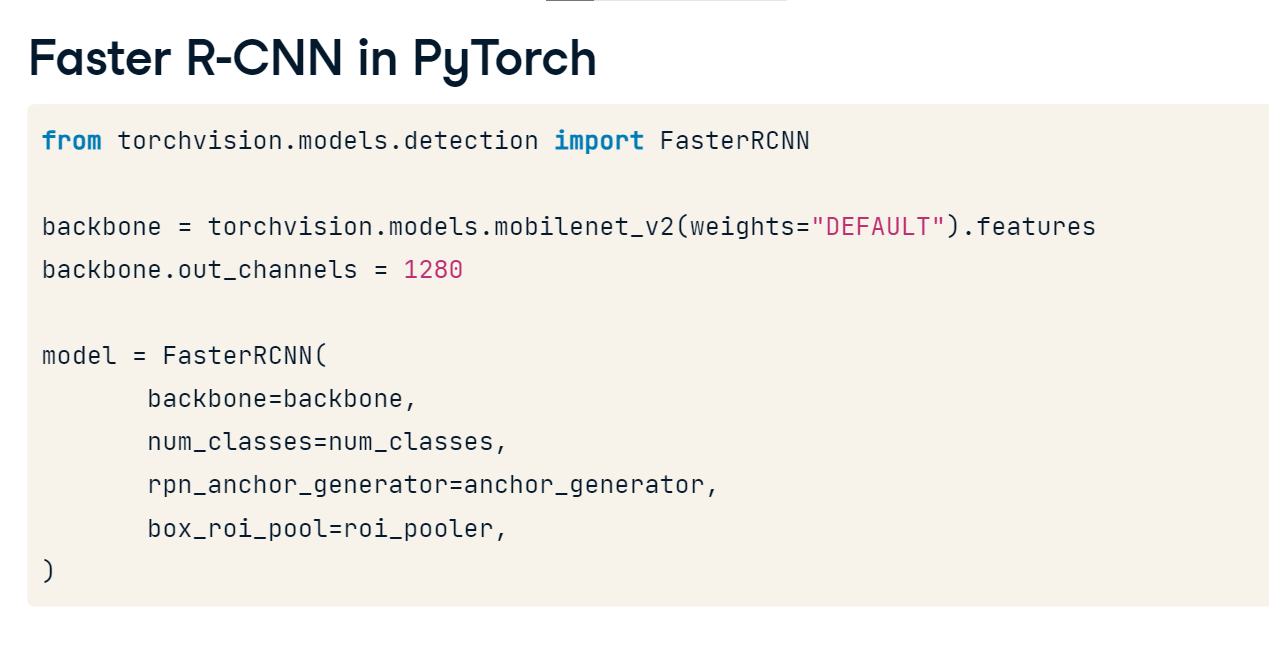
**Fast R-CNN loss functions**

The region proposal network uses two loss functions. For the RPN classifier, we use binary cross-entropy available as nn-dot-BCEWithLogitsLoss, since this is a binary classifier indicating whether a proposed region contains an object. For the RPN box regressor, we use the mean squared error loss available as nn-dot-MSEloss. For the final R-CNN classification, we apply nn-dot-crossentropyloss, since we may have many classes. For the R-CNN box regressor, we use nn-dot-MSEloss again.



**Faster R-CNN in PyTorch**

FasterRCNN model is available from torchvison. We choose the backbone, here: a small mobilenet model with default pre-trained weights. We extract its backbone using the dot-features attribute. FasterRCNN model requires setting out channels in the backbone. We could print the model architecture to check it. Here we already know the value as 1280. To create the FasterRCNN model, provide the backbone, the number of object classes, and the previously defined anchor generator and RoI pooling module.



**Faster R-CNN in PyTorch**

We can also use a pre-trained Faster R-CNN without manually extracting a backbone from a different model. We import FastRCNNPredictor from torchvision-dot-models-dot-detection-dot-faster-rcnn and load a pre-trained Faster R-CNN model, this time with resnet50 as a backbone and its default weights. We set the number of classes to two for our binary classification problem of detecting cats and dogs. Next, we extract the number of input features to the classifier head of the Faster R-CNN model and store it as in-features. Finally, we replace the default box predictor of the model with a new one that has the desired number of output classes. The model is ready to use!

