**Introduction to image segmentation**

**Image segmentation**

Image segmentation is a computer vision task that involves partitioning the image into multiple segments, or areas withing the image, on the pixel level. This means that each pixel in an image is assigned to a particular segment. There exist three types of segmentation: semantic, instance, and panoptic and each of them requires a different model architecture. Let's discuss them one by one.

**Semantic segmentation**

In semantic segmentation, each pixel in the image is classified into a predefined class or category. All pixels belonging to the same class are treated equally, and there is no distinction between different instances of the same class. In a street scene, all pixels belonging to cars are marked as dark blue, all pixels belonging to roads are marked as purple and so on, without distinguishing between individual cars or road sections.



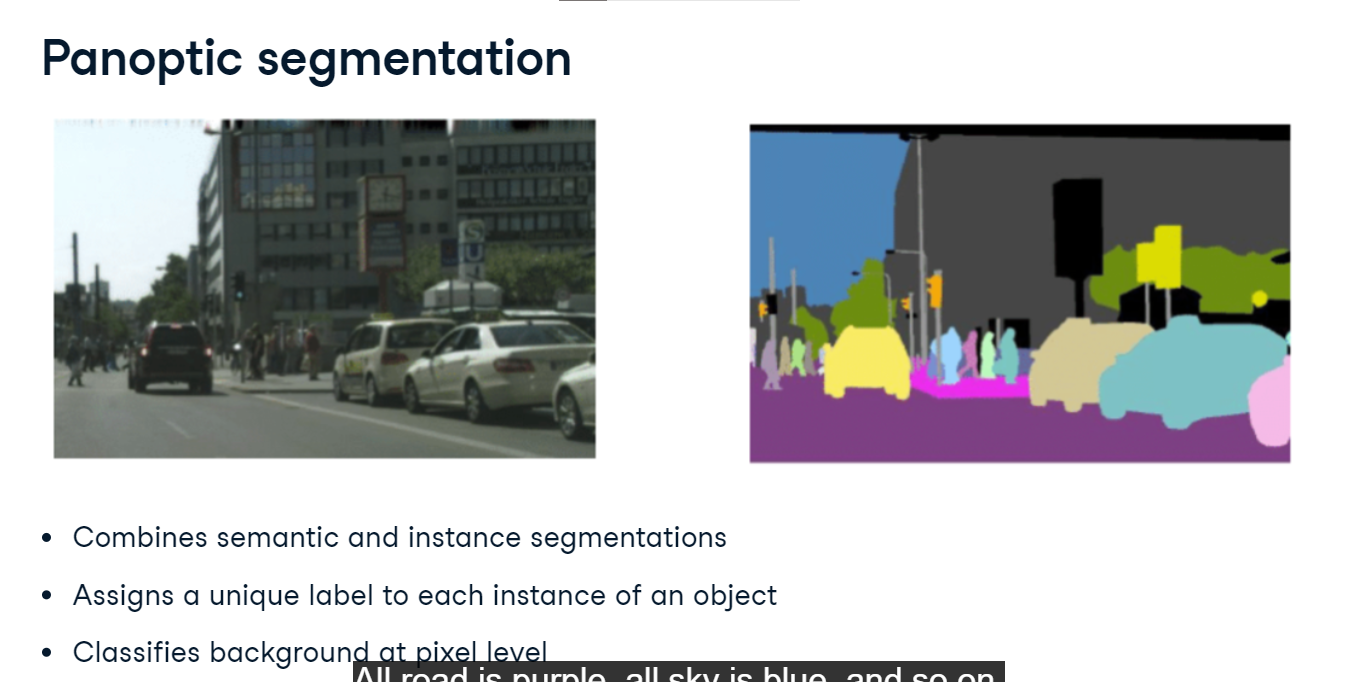
**Instance segmentation**

Instance segmentation goes a step further than semantic segmentation by not only classifying each pixel but also distinguishing between different instances of the same class. In the same street scene, each car would be assigned a unique label shown with a unique color, so that different cars can be differentiated from each other, even though they belong to the same class "car". Since the primary focus of instance segmentation is on identifying and segmenting individual object instances in the image, the background such as road or sky is typically not segmented.



**Panoptic segmentation**

Panoptic segmentation combines the concepts of semantic segmentation and instance segmentation. It assigns a unique label to each instance of an object while also classifying background regions (such as sky, road, or grass) at the pixel level. In the street scene, each car would get a unique label like in instance segmentation (each car is shown in different color). At the same time, areas like the road, sky, and trees are labeled at the pixel level without instance differentiation like in semantic segmentation. All road is purple, all sky is blue, and so on.



**Data annotations**

Let's take a look at data annotations for segmentation tasks. We load two image files. image is the picture of this British Shorthair cat sitting on a sofa. mask is the corresponding data annotation. The mask tells us which pixels are part of the cat, and which are not. Let's convert both PIL images to PyTorch tensors and print their shapes. The image is 333 by 500 pixels and has three color channels. The corresponding annotation is of the same height and width, but has only value for each pixel, describing its segment.



**Understanding the mask**

In the dataset documentation, we read that the annotations can only take three values: 1 for the object, 2 for the background, and 3 for unclassified. But when we print the unique mask values, we see three different numbers! This is because the ToTensor transform has divided the pixel values by 255. In our case, 1 over 255 which equals 0.0039 denotes the foreground, or the object of interest. A similar calculation is done for background and unclassified.



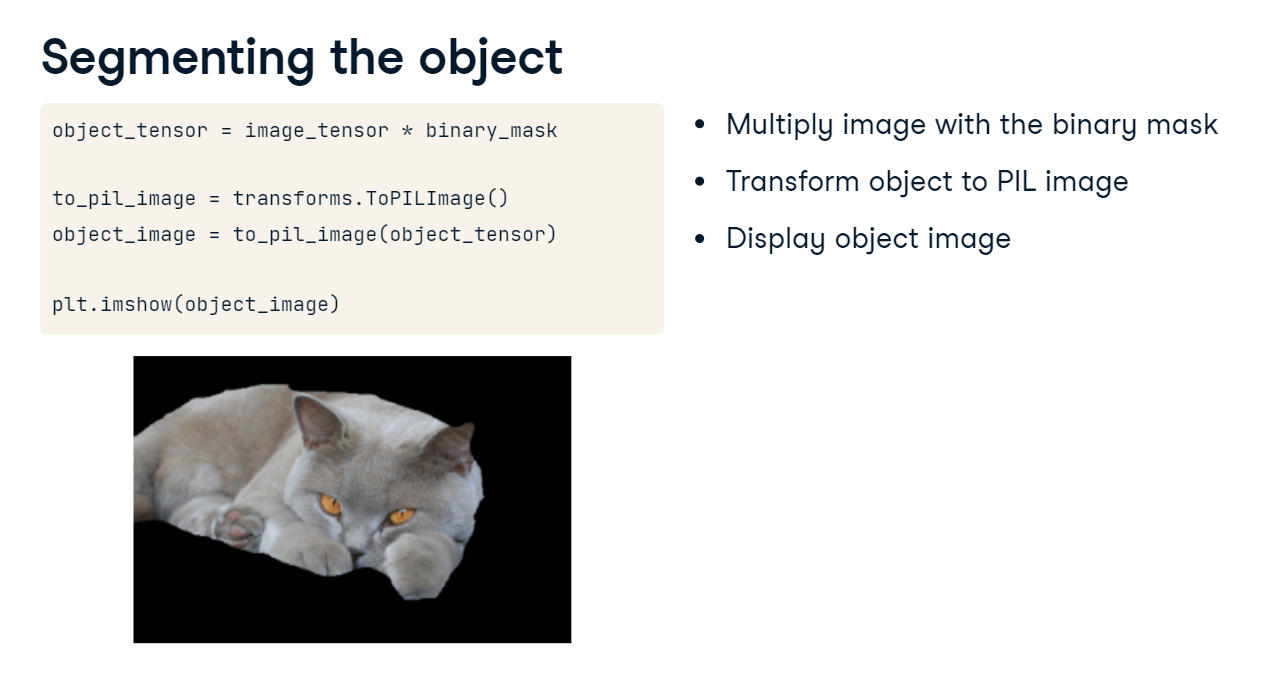
**Creating a binary mask**

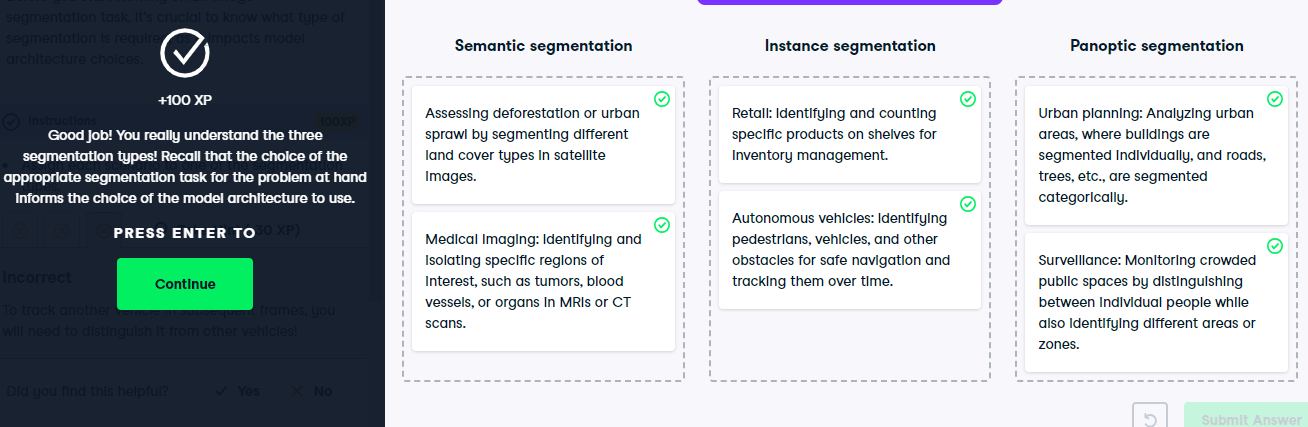
Let's create a binary mask, where 1 corresponds to the object and 0 to everything else. We will use the torch-dot-where function to do so. It takes three arguments. First, the condition to check: whether the pixel value represents the object. Then, the value to use when the condition is met, here 1, followed by the value to use otherwise, here 0. Let's take a look at our binary mask. We convert the mask tensor back to a PIL image and display it. The cat's shape is clearly visible!



**Segmenting the object**

Now, let's segment our cat out of the picture. To create the object tensor, we multiply the image with the binary mask. Next, we proceed just like we did with the mask: we transform the object to a PIL image and display it. The cat has been segmented out and the sofa in the background is gone!





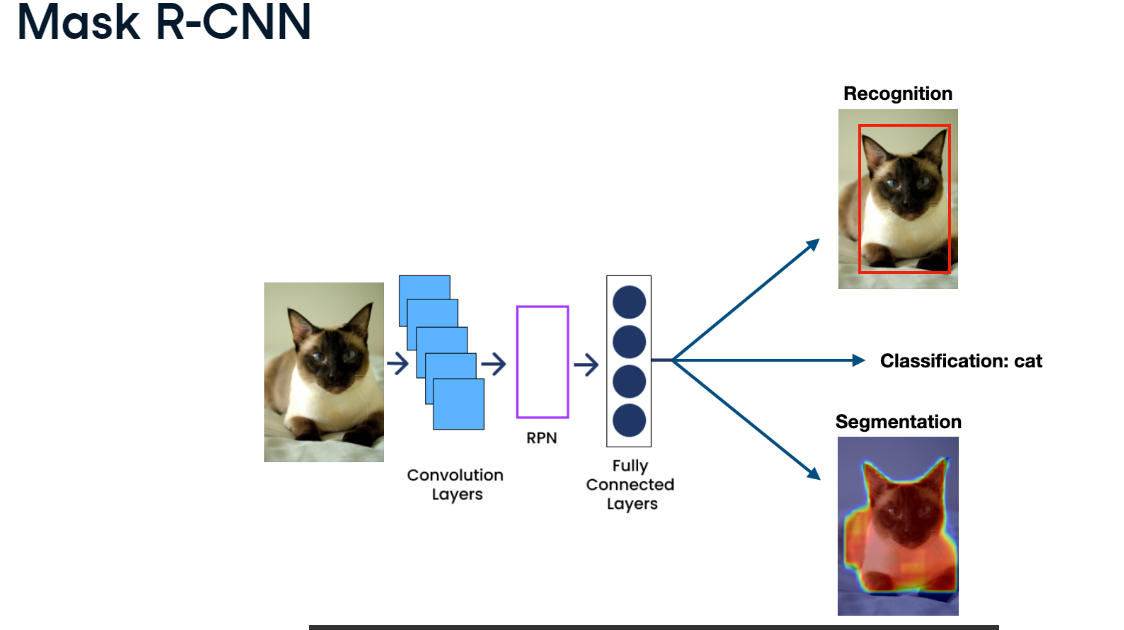
**Instance segmentation with Mask R-CNN**

**Faster R-CNN**

We've previously covered Faster R-CNN for object recognition. Given the image, it would predict its class and the bounding box around the object.

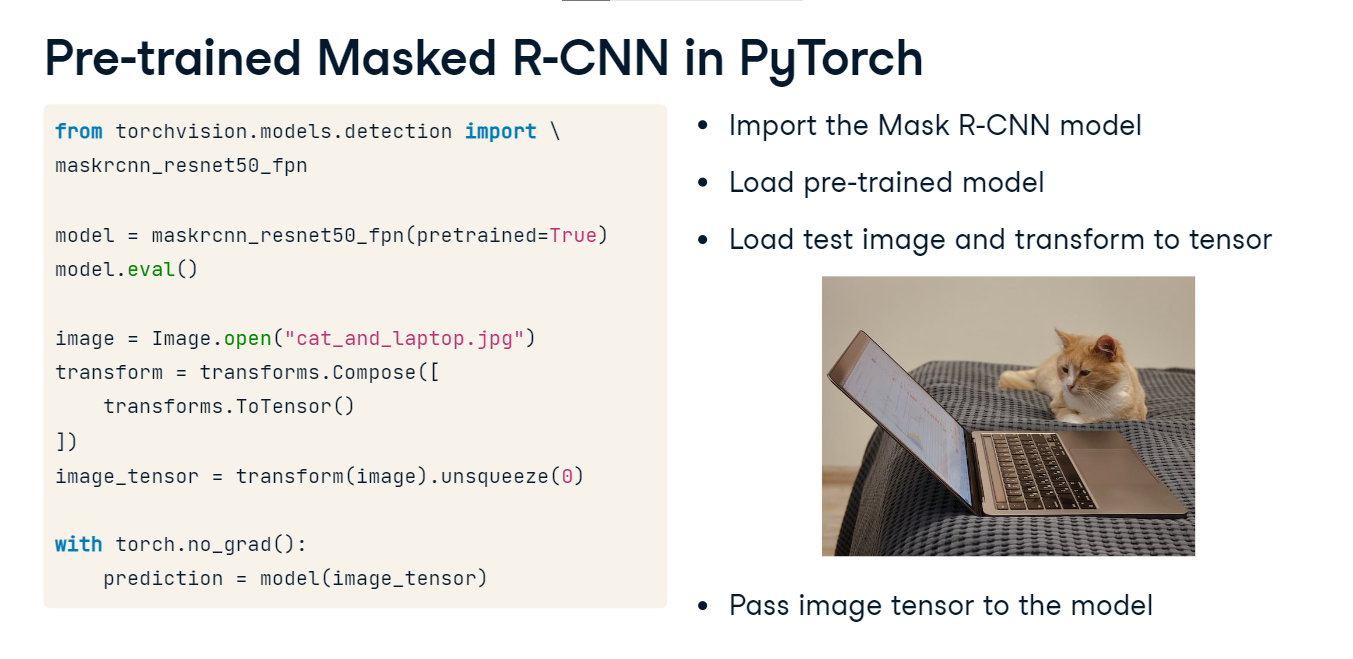
**Mask R-CNN**

Mask R-CNN extends Faster R-CNN by adding instance segmentation, retaining a nearly identical architecture with convolutional layers, a Region Proposal Network, and fully connected layers. Mask R-CNN introduces a third model branch that predicts a pixel-to-pixel segmentation mask. This enables accurate instance segmentation.



**Pre-trained Masked R-CNN in PyTorch**

Let's explore using a pre-trained Mask R-CNN model for instance segmentation. We start by importing the maskrcnn-resnet50-fpn from torchvision-dot-models-dot-detection. Next, we load the model with pre-trained weights. Then, we load and convert the test image into a tensor. We will use a photograph of a cat sitting next to a laptop, and we want to detect these two objects. Since the model is pre-trained on the COCO dataset, which includes common objects like animals and computers, it should detect our objects without requiring fine-tuning. Finally, we pass the image tensor to the model to run the inference, saving the result in a variable called "prediction".



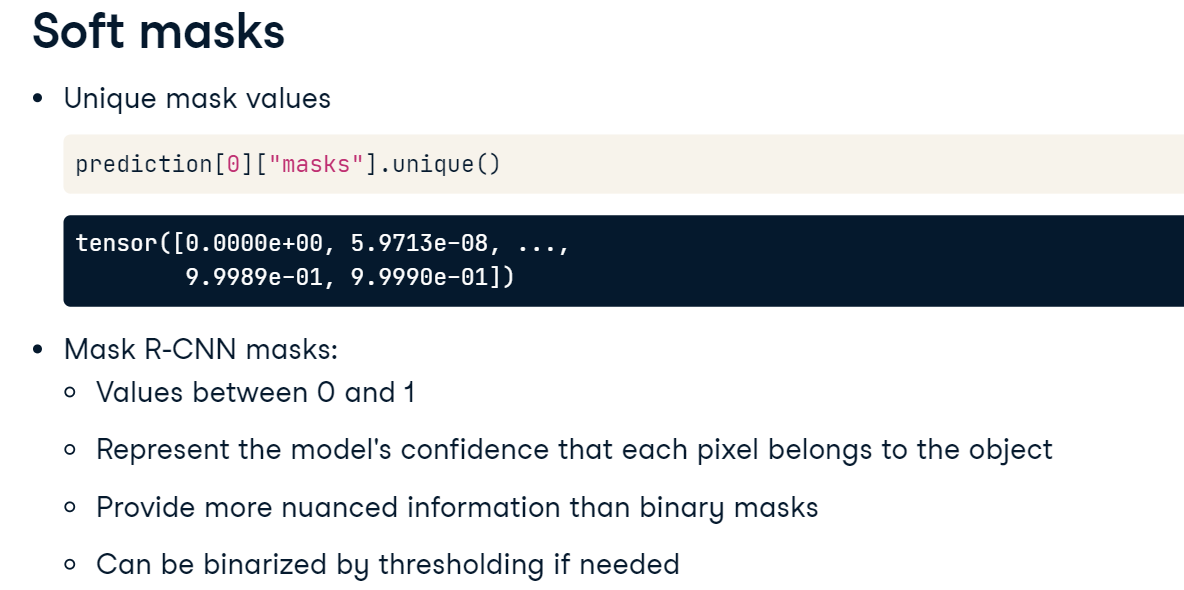
**Model outputs**

Examining the Mask R-CNN outputs: "prediction" is a list of length one since we only passed one image to the model. This single list element is a dictionary with a couple of keys. "labels" contains the class IDs of recognized objects. These IDs correspond to the COCO dataset classes which we have stored in the variable class\_names. These class names are available on the COCO dataset website. We can see that the top two predicted classes with indices 17 and 73 correspond to a cat and a laptop, respectively. The scores key stores the class probabilities. We can see that the cat was detected with a probability larger than 99% - that's the first value in the tensor - and the laptop with more than 96%. The following values correspond to other, less probable classes. Finally,the masks key stores instance segmentation masks which we will look at next. Additionally, the Mask R-CNN prediction also contains bounding boxes, but we are not interested in them when discussing segmentation.



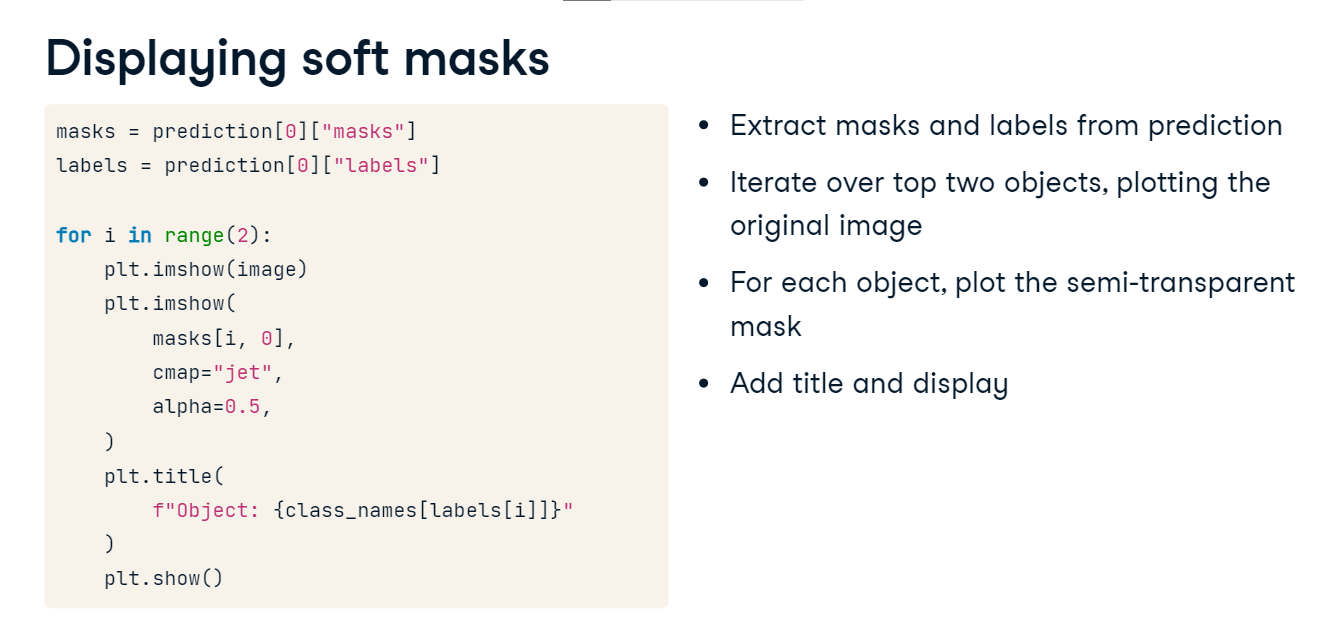
**Soft masks**

Let's print the unique values of the predicted masks. The values in the segmentation masks produced by Mask R-CNN are not binary (0s and 1s) but are instead floating-point values ranging from 0 to 1. These values represent the model's confidence that each pixel belongs to the object being segmented. These continuous values produce what is known as a "soft mask". Soft masks can provide more nuanced information than binary masks, especially at the boundaries of objects where there might be ambiguity. If we need a binary mask, we can apply a threshold to the soft mask. For example, we might decide that any value above 0-point-5 should be considered part of the object (set to 1), and any value below 0-point-5 should be considered background (set to 0).



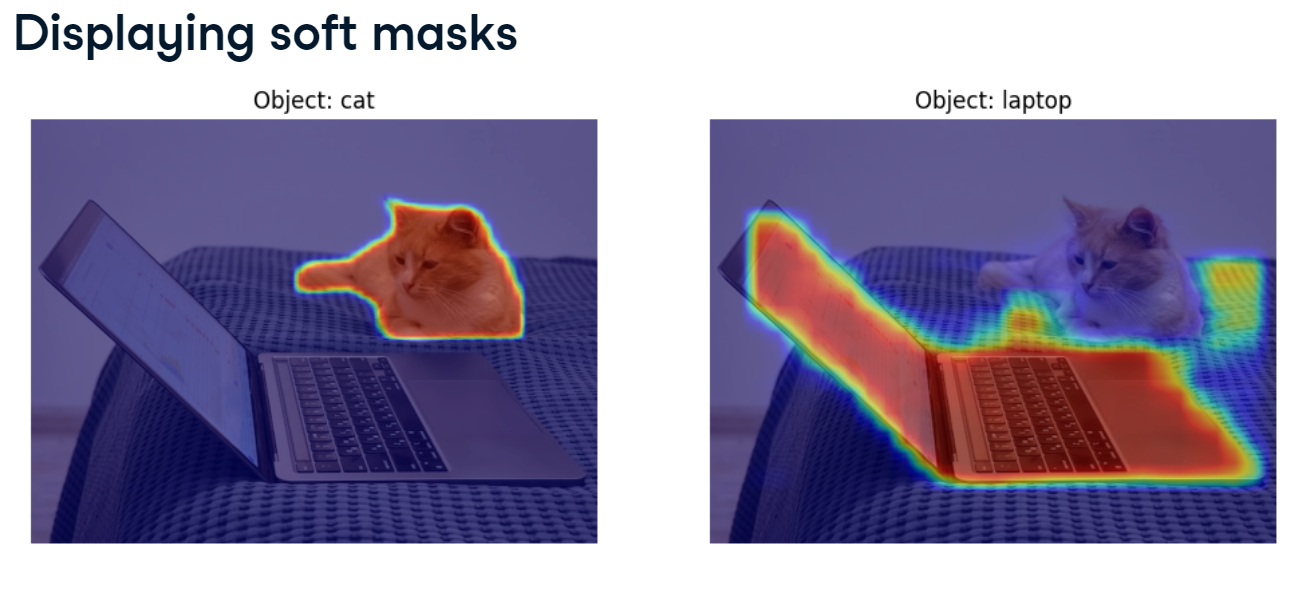
**Displaying soft masks**

Lets' see how to display a soft mask overlaid on top of the image. We first extract the masks and labels from the prediction. Next, we iterate over the top two predicted objects. For each, we display the original image, and then the mask, setting the color map to "jet" and alpha to 0-point-5 to make the mask semi-transparent so that it does not obscure the image. Finally, we add class labels to the title and display.



**Displaying soft masks**

The cat mask is very accurate. The one for laptop slightly less so, although the high-confidence red regions are still pretty good.



**Semantic segmentation with U-Net**

Great work so far! It's time to learn about semantic segmentation!

**Semantic segmentation**

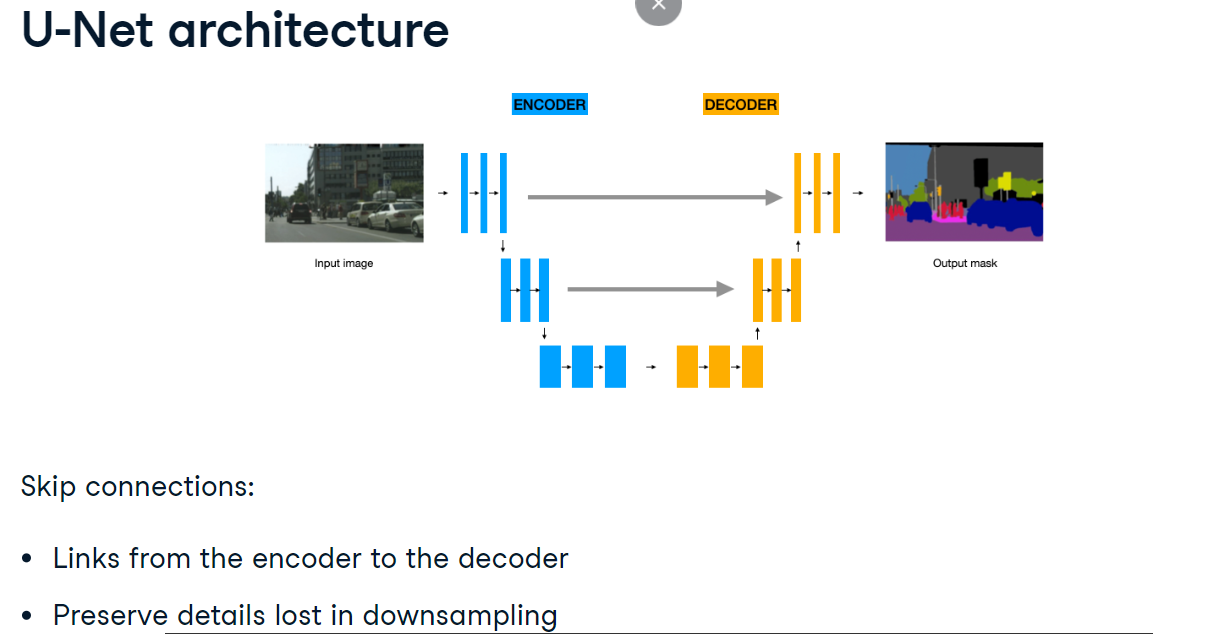
In semantic segmentation, we don't distinguish between instances of the same class. This approach to segmentation is useful for medical imaging or satellite image analysis, among others. A common neural network architecture for semantic segmentation tasks is the U-Net.

**U-Net architecture**

The U-Net, initially designed for biomedical image segmentation, is named after its architecture's U shape. It consists two parts: an encoder and a decoder. The encoder, shown in blue, captures the image context through a series of convolutional and pooling layers, reducing height and width while increasing depth of the feature maps. This process is referred to as downsampling.

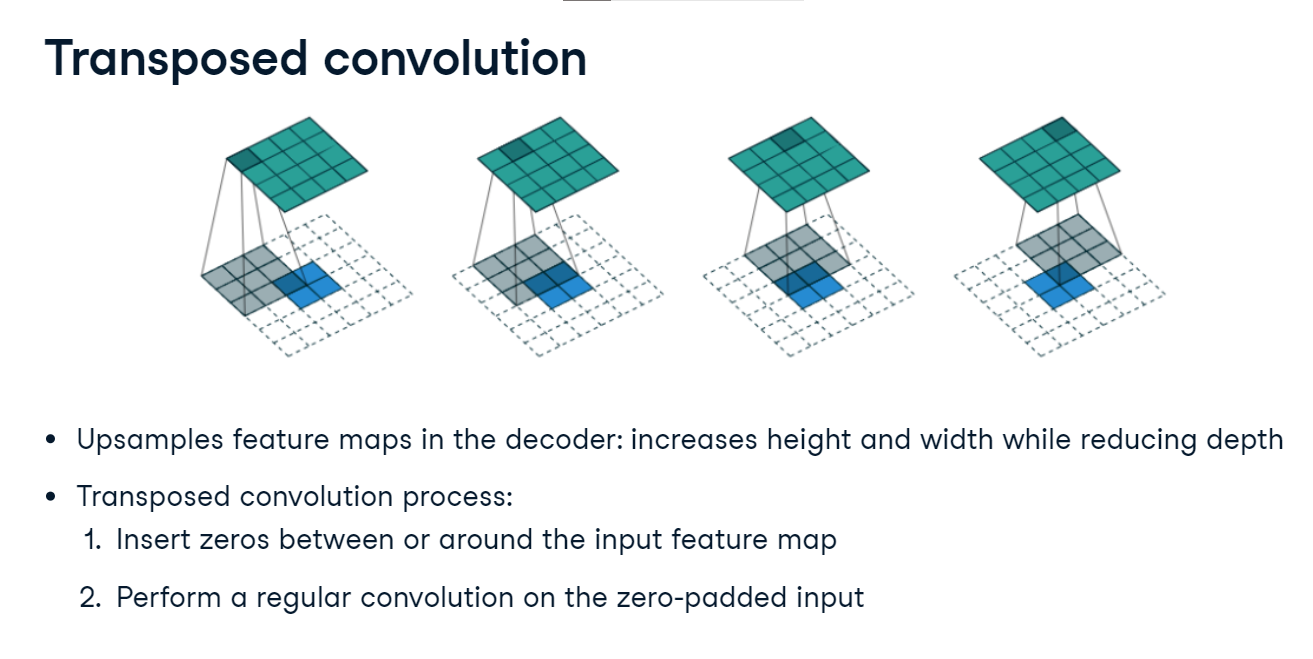
The decoder, shown in orange, mirrors the encoder. It gradually upsamples the feature maps using transposed convolutional layers which we will discuss shortly, making feature maps higher and wider but shallower. This leads to the output of the same spatial dimensions as the input, allowing us predict the class for each pixel in the form of a mask.

The U-Net uses skip connections, shown as gray horizontal arrows in the diagram. Skip connections are direct links from the encoder to the decoder, ensuring the preservation of details lost during downsampling. Notice how the first encoder block is linked to the last decoder block, the second encoder block to the penultimate decoder block, and so on. The input to each decoder block consists of concatenated outputs of the previous decoder block, and the corresponding encoder block.



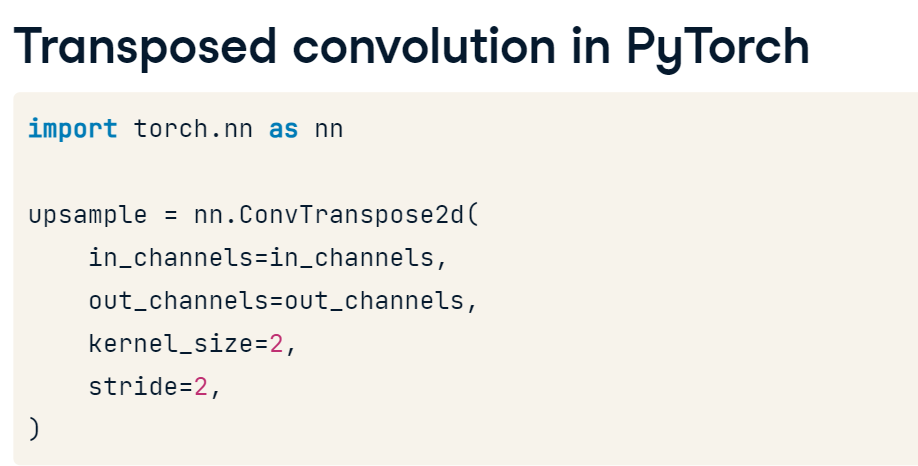
**Transposed convolution**

Transposed convolution is used to upsample feature maps in the decoder part of the U-Net. It increases their height and width, while reducing depth. The process involves inserting zeros between or around the feature map input pixels. Here, the blue two-by-two feature map is padded with white zeros. Next, a regular convolution operation is performed on the zero-padded input, resulting in an upsampled feature map with enlarged spatial dimensions.



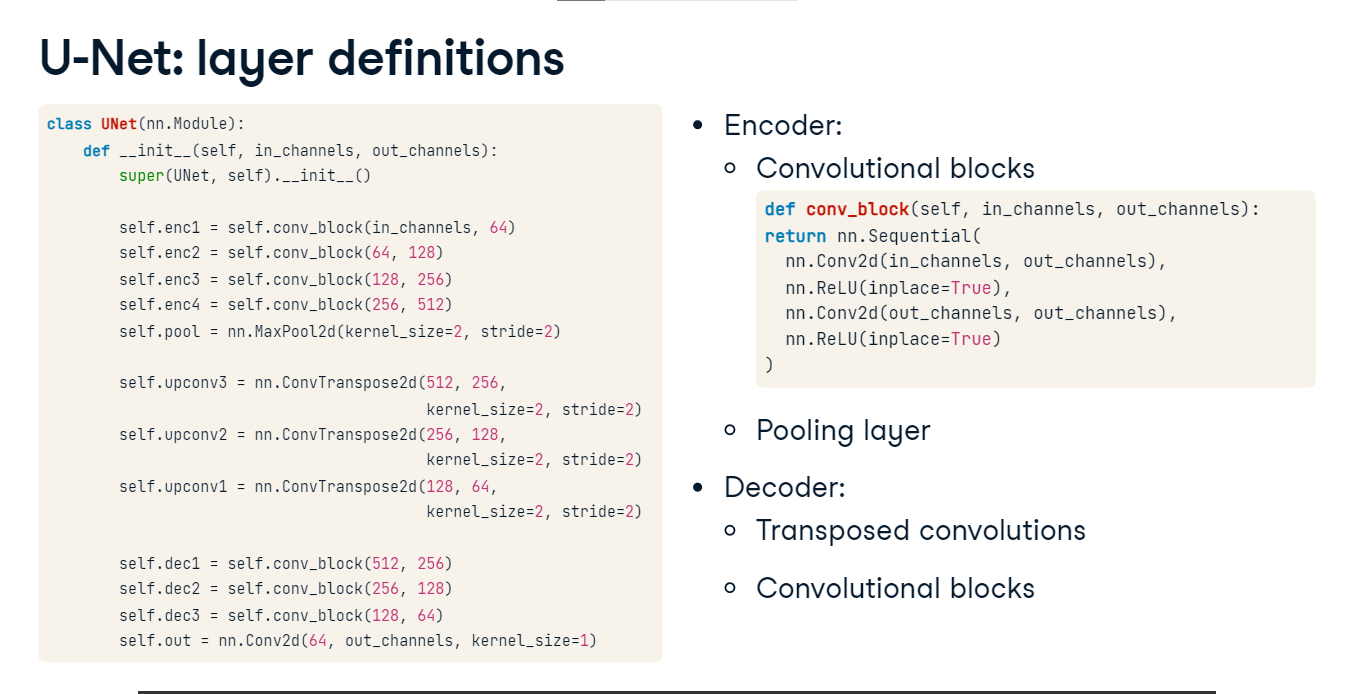
**Transposed convolution in PyTorch**

The transposed convolutional layer is available from torch-dot-nn as ConvTranspose2d. It accepts parameters similar to regular convolutional layers, including input and output channel numbers, kernel size, and stride. In our U-Net architecture, we'll set both the kernel size and stride to two.

****

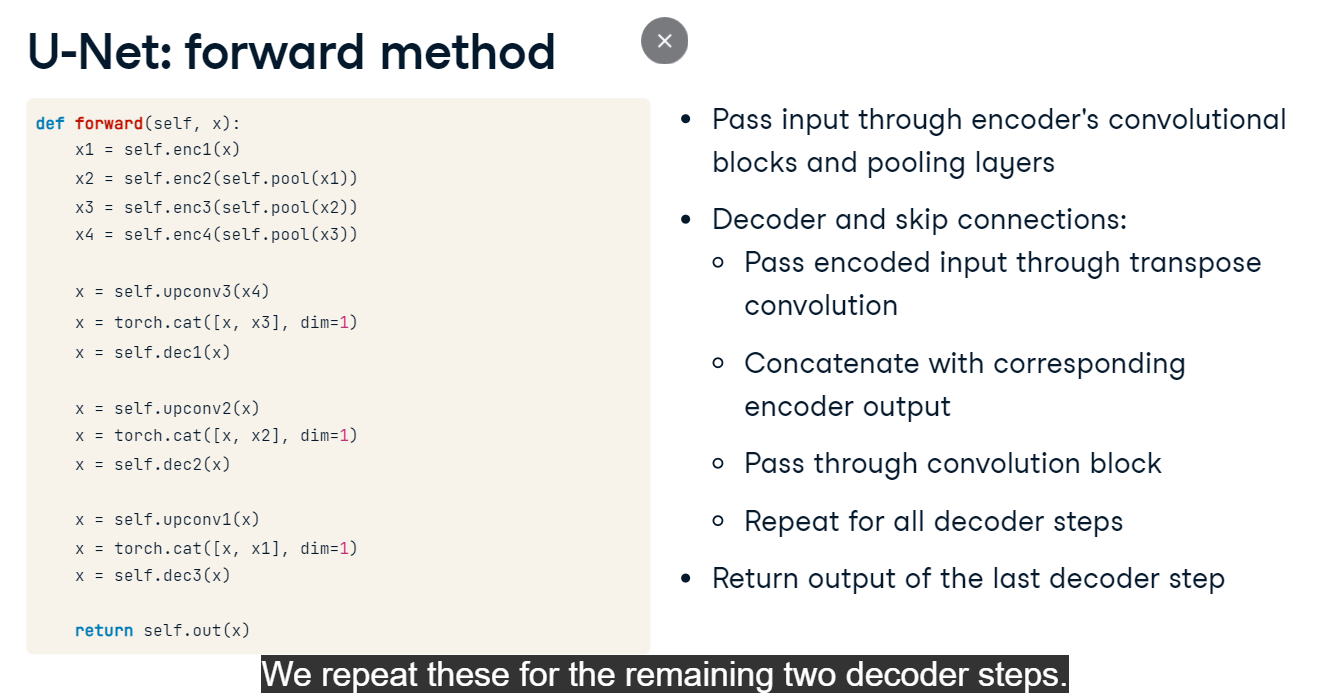
**U-Net: layer definitions**

Let's build a U-Net! We start with the init method where we define the model's layers and layer blocks. First, the encoder layers consist of four convolutional blocks with ReLU activations, defined using the custom helper function conv\_block. This function will be provided for you in the exercises. We assign the encoder blocks to enc1, enc2, and so on. Notice how each subsequent encoder block increases feature maps' depth. We also define a max pooling layer here. For the decoder, we define three transpose convolutions decreasing feature maps' depth, assigning them to upconv3, upconv2 and upconv1, respectively. Finally, we define the convolutional blocks to be used in the decoder.



**U-Net: forward method**

Now, we can use the layers we have defined in the init method to construct the forward method. It receives the input x. First, the input is passed through the encoder's convolutional blocks while applying the pooling layer before each block. Next, we implement the decoder part and the skip connections. Since we have defined three upsampling layers, the decoder will consists of three steps. First, we pass the encoded input to the transposed convolution. Next, we concatenate it with the corresponding encoder output using torch-dot-cat. Finally, we pass the result through a decoder convolution block. We repeat these for the remaining two decoder steps. Finally, we return the output of the last decoder step.



**Running inference**

To wrap up, let's use a trained U-Net to produce segmentation masks. We load the model and put it in evaluation mode. Next, we load this car image and convert it to a tensor. Then, we pass it to the model for inference. The prediction in this case is two masks, one for the background and one for the foreground. Let's display the latter.

