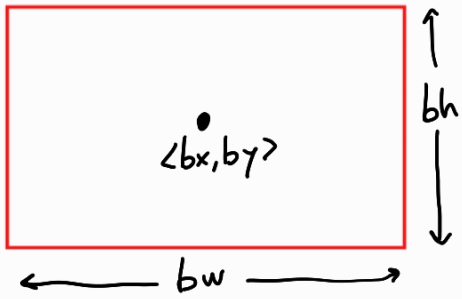
Object localization: specify where a wanted type of object is.

This requires the algorithm to output:

- bx, by ⇒ <x, y> coordinates of the area midpoint.
- bw ⇒ width of the rectangle
- bh ⇒ height of the rectangle



The target label is defined as follows:

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} \in \{0, 1\}$$

p_c → Is there an object? (probability that one of the classes is there ≠ background)

$c_1, c_2, \ldots, c_{n-1} \in \{0, 1\}$

c_n = background

The loss function is, for y = truth, $\hat{y}$ = prediction:

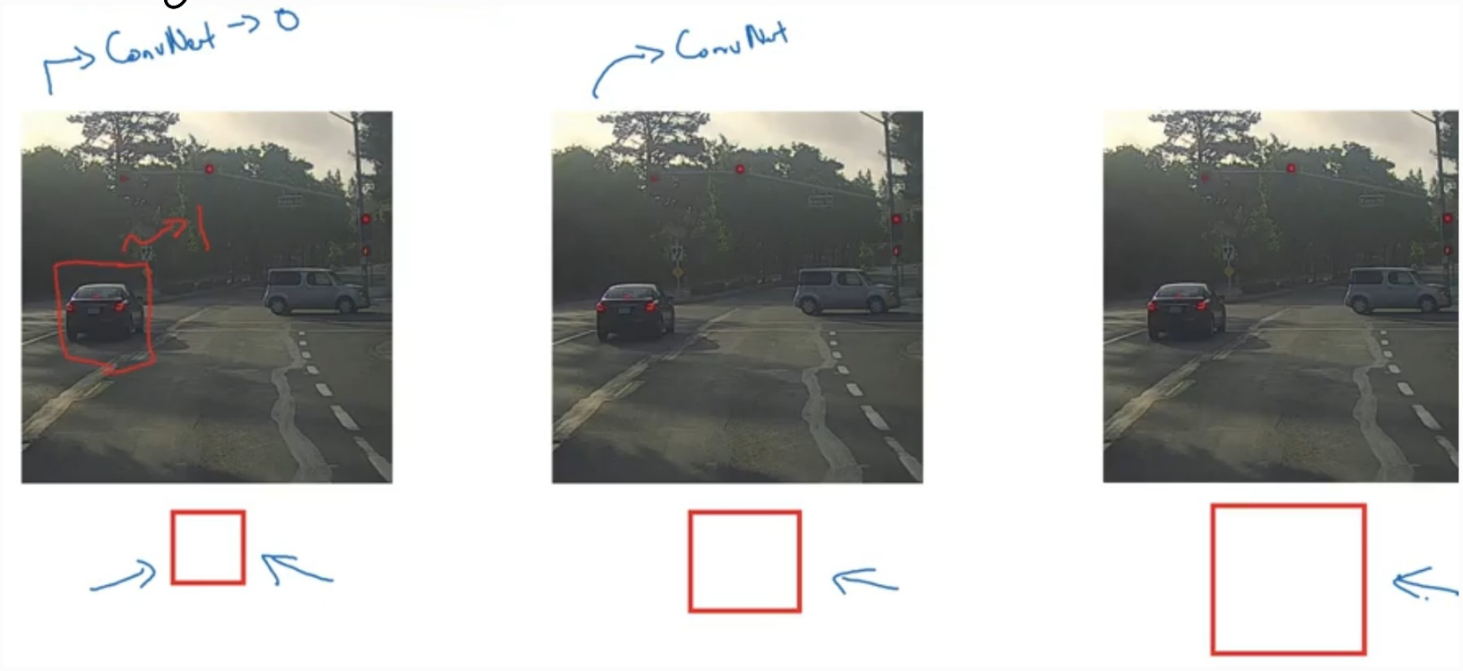
$$\mathcal{L}(\hat{y}, y) = \begin{cases} (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \ldots \\ \quad \ldots + (\hat{y}_m - y_m)^2 & ; \text{ if } y = 1 \\ (\hat{y}_1 - y_1)^2 & ; \text{ if } y = 0 \end{cases}$$

for m components, squared error

This can be generalized to posing or face detection.

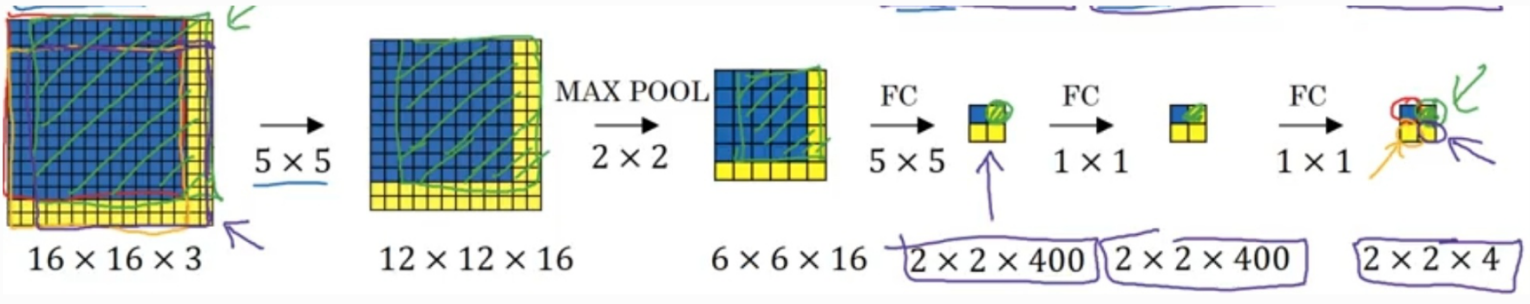
The red dots would be x, y coordinates associated with a wanted part of the body, and then in the case of $y_1 = 1$, there is a vector of $\langle x, y \rangle$ values for each point.

Sliding windows:



Very computationally expensive !!

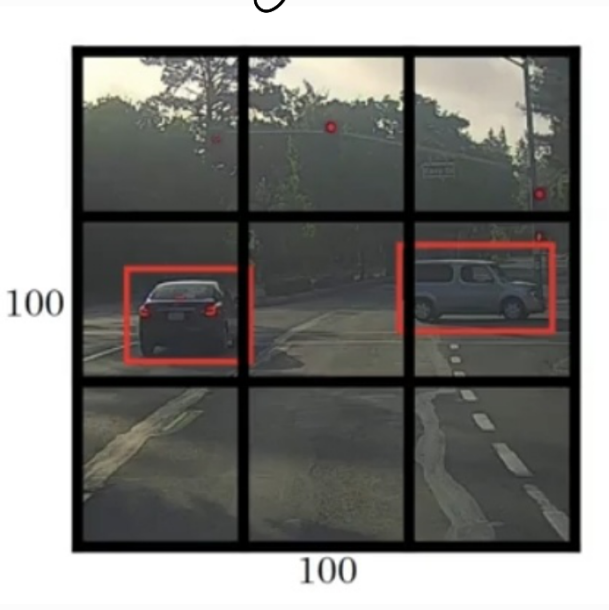
Solution: with convolutions



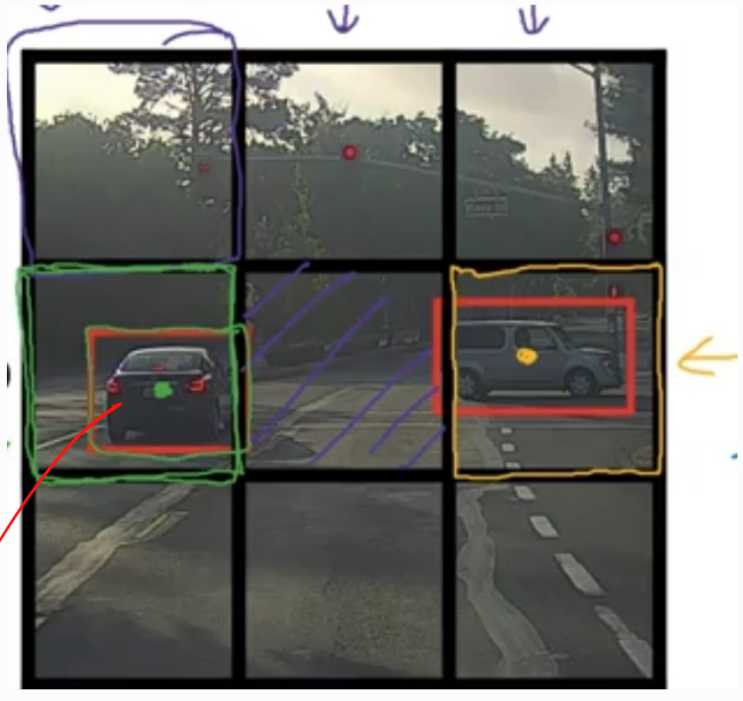
This way, 4 filters are checked in 1 forward propagation step!! But which position of bounding boxes to use?

Bounding box predictions: YOLO algorithm

Create a grid:



Apply the localization algorithm to each cell:

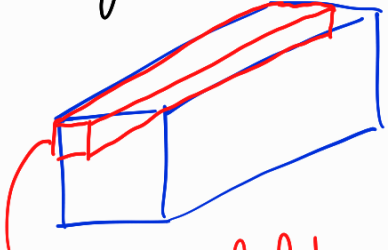


As the midpoint of the car is there, the cell is assigned $\hat{y}_4 = [\underset{\downarrow}{1}\ b_x\ b_y\ b_w\ b_h\ 0\ \underset{\downarrow}{1}\ 0]\ (\dots)$

p_c c_2

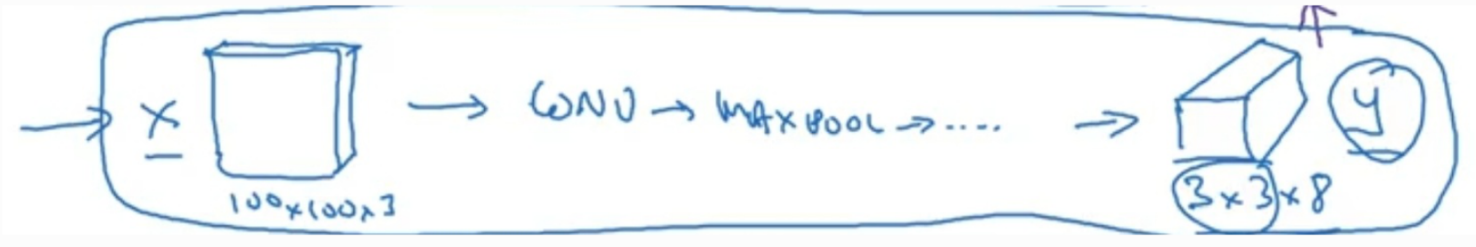
The target output is:

grid_cell_h x grid_cell_w x target_vector_size

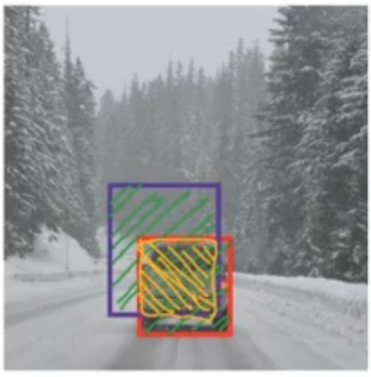


→ upper left cell output vector.

So, the neural network is outputing precise bounding boxes!



Evaluating object localization



Intersection over Union (IoU)

$$= \frac{\text{size of } \square}{\text{size of } \square}$$

"Correct" if IoU ≥ 0.5

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

But, what if my algorithm detects the same object more than once? → Non-max supression: get the maximum p_c vector, delete the rest.

Each output prediction is:

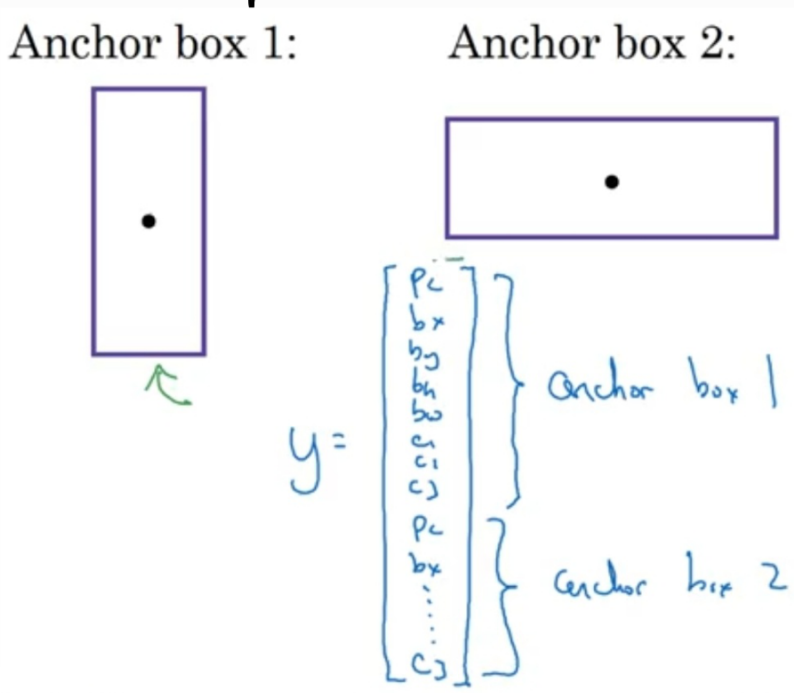
$$\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \end{bmatrix} \quad \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

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 IoU ≥ 0.5 with the box output in the previous step

What if a grid cell wants to detect multiple objects (overlapping)? Anchor boxes: Repeat the output vector.

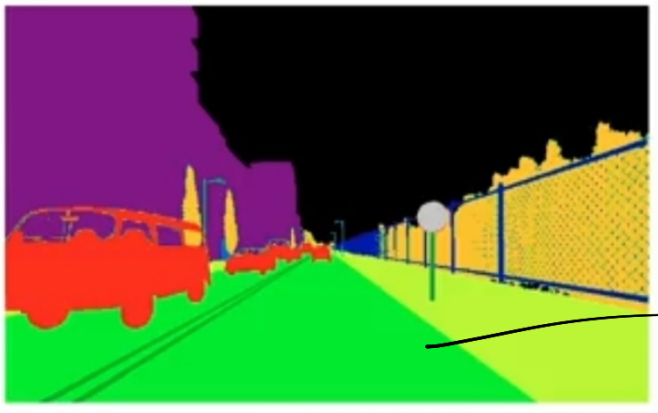


With two anchor boxes:

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.

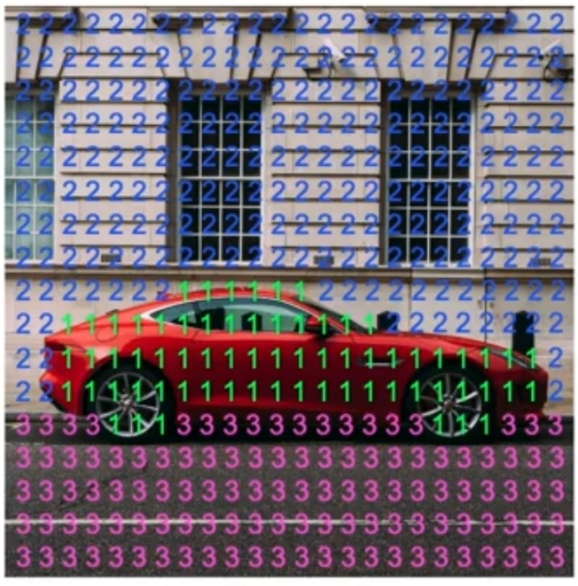
How many to choose? K-means.

What if I want to learn to recognize everything in an image? Semantic segmentation



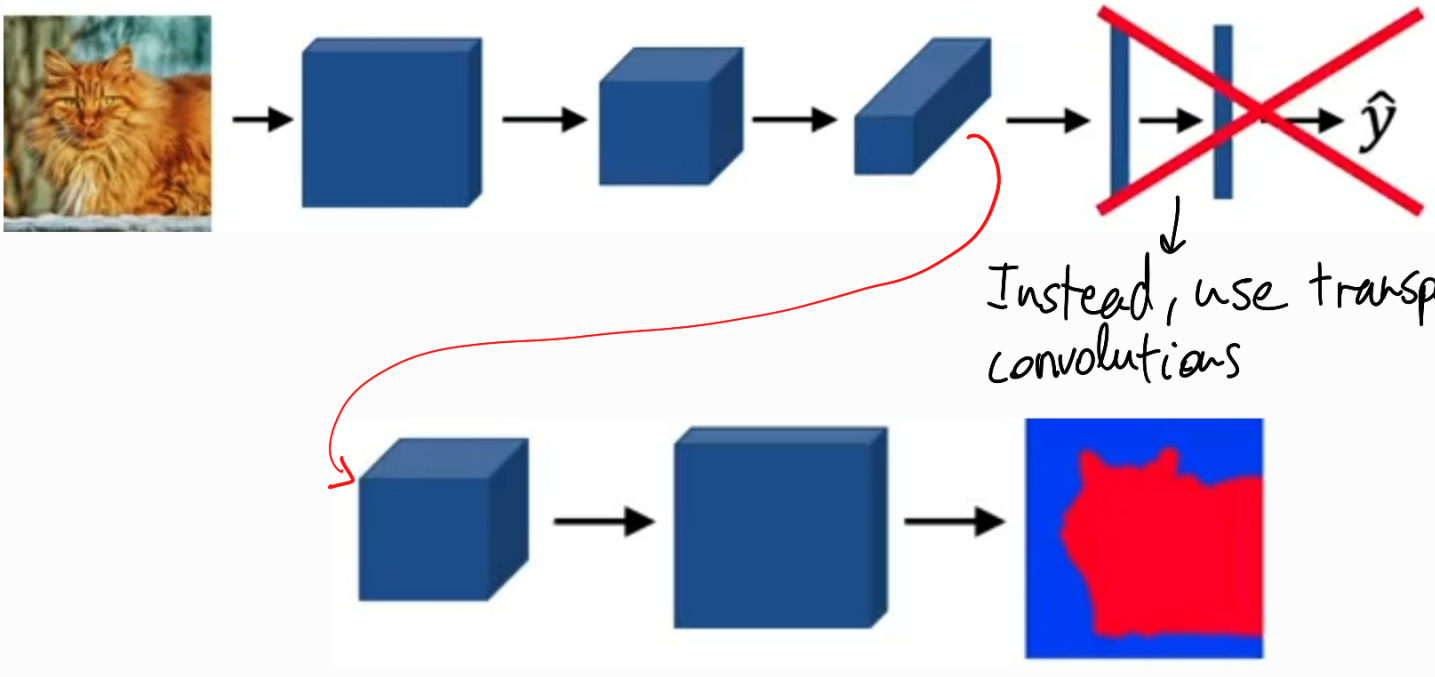
→ green path ⇒ drivable

Per-pixel class labels

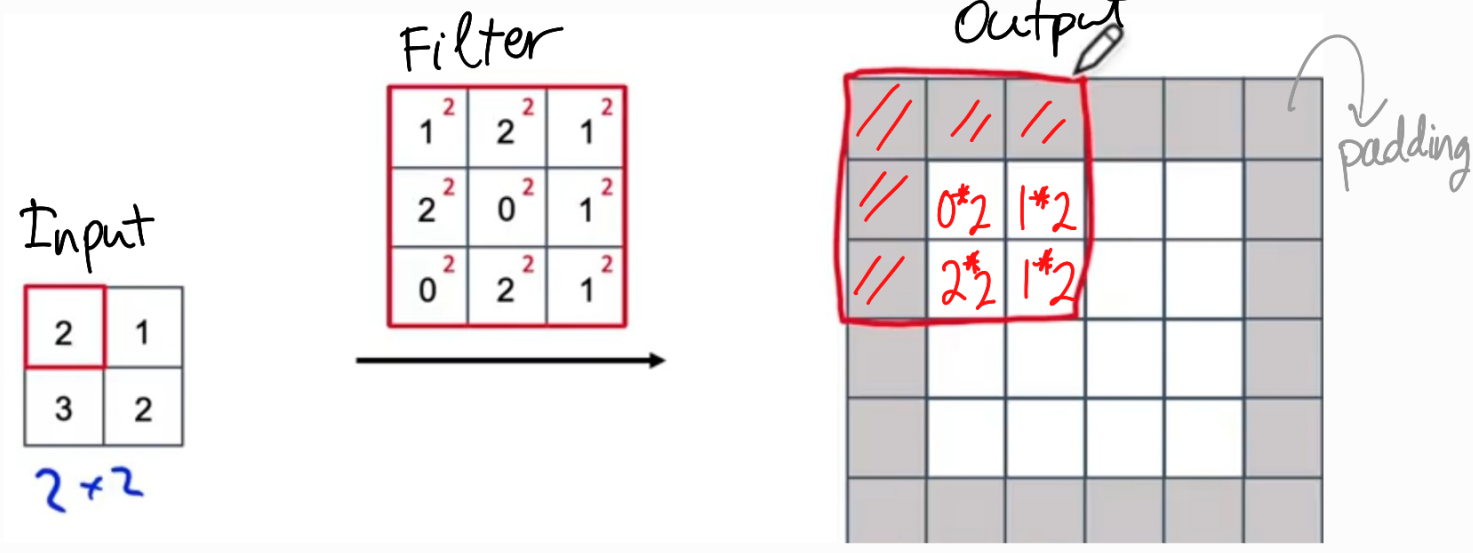


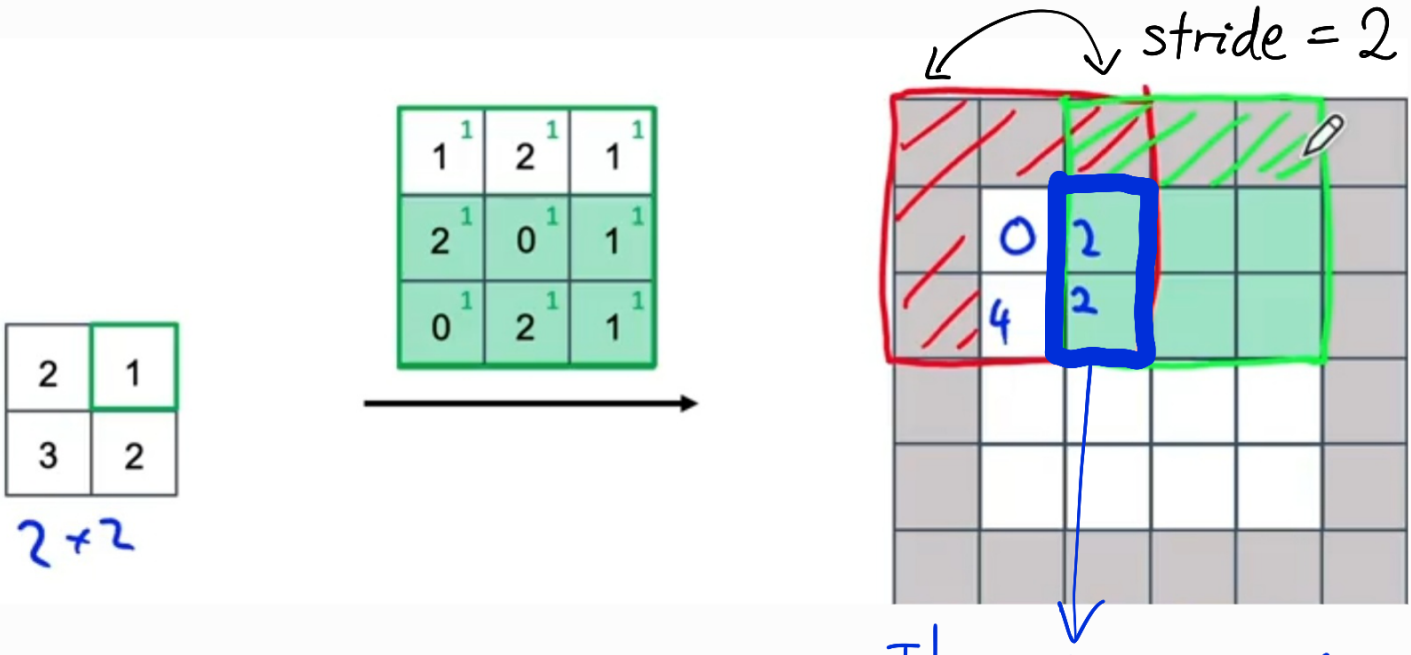
1. Car
2. Building
3. Road

Segmentation Map

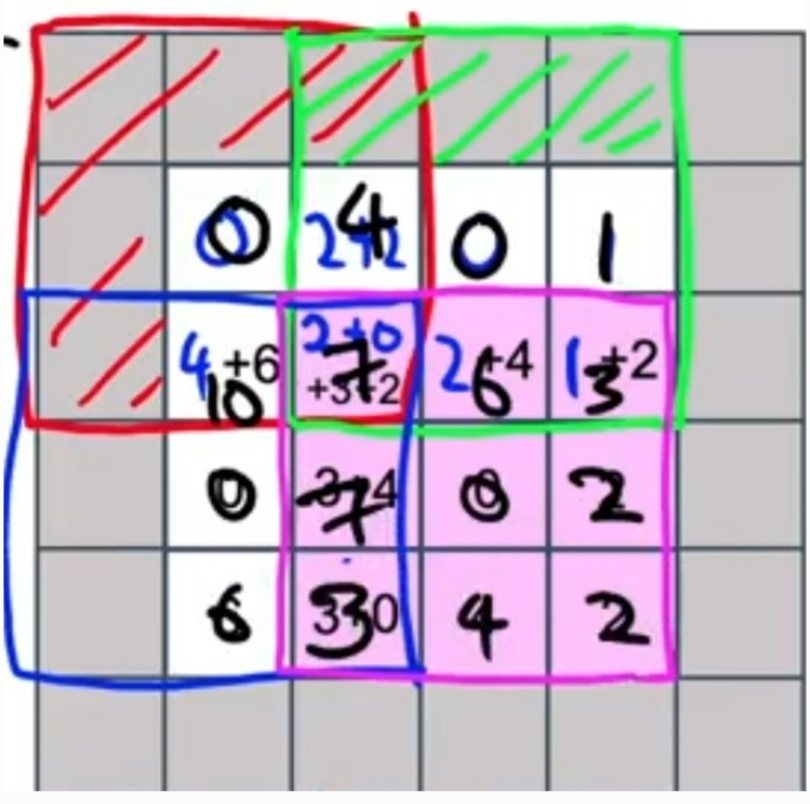


Transpose convolution: you place the filter in the OUTPUT!!

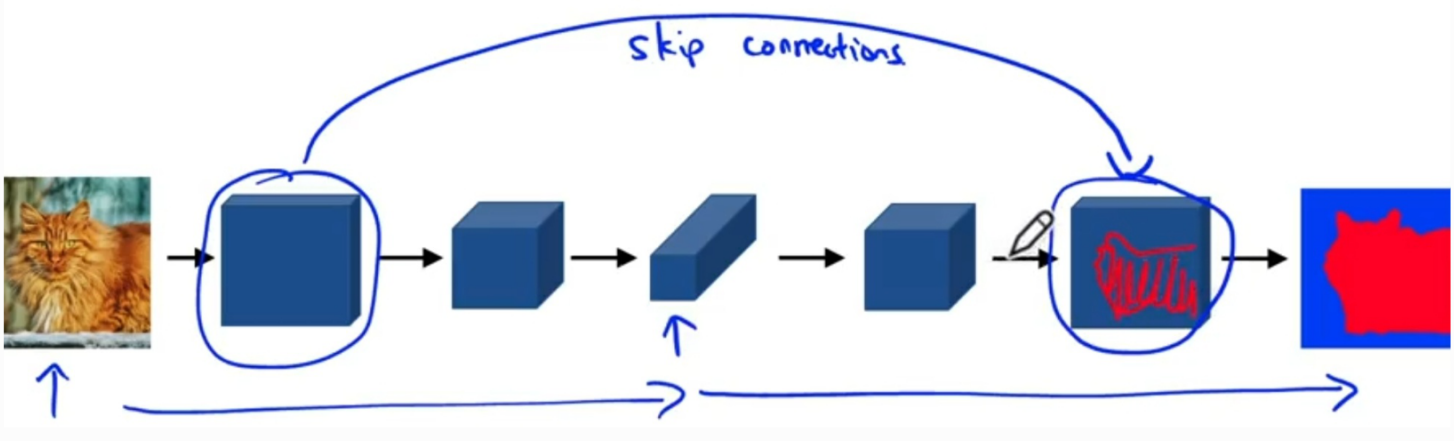




(...)



U-Net : adds skip connections (ResNets blocks):



Notebooks explanations: YOLO

What you should remember:

- YOLO is a state-of-the-art object detection model that is fast and accurate
- It runs an input image through a CNN, which outputs a 19x19x5x85 dimensional volume.
- The encoding can be seen as a grid where each of the 19x19 cells contains information about 5 boxes.
- You filter through all the boxes using non-max suppression. Specifically:
  - Score thresholding on the probability of detecting a class to keep only accurate (high probability) boxes
  - Intersection over Union (IoU) thresholding to eliminate overlapping boxes
- Because training a YOLO model from randomly initialized weights is non-trivial and requires a large dataset as well as lot of computation, previously trained model parameters were used in this exercise. If you wish, you can also try fine-tuning the YOLO model with your own dataset, though this would be a fairly non-trivial exercise.

2 - YOLO

"You Only Look Once" (YOLO) is a popular algorithm because it achieves high accuracy while also being able to run in real time. This algorithm "only looks once" at the image in the sense that it requires only one forward propagation pass through the network to make predictions. After non-max suppression, it then outputs recognized objects together with the bounding boxes.

2.1 - Model Details

Inputs and outputs

- The **input** is a batch of images, and each image has the shape (m, 608, 608, 3)
- The **output** is a list of bounding boxes along with the recognized classes. Each bounding box is represented by 6 numbers $(p_c, b_x, b_y, b_h, b_w, c)$ as explained above. If you expand c into an 80-dimensional vector, each bounding box is then represented by 85 numbers.

Anchor Boxes

- Anchor boxes are chosen by exploring the training data to choose reasonable height/width ratios that represent the different classes. For this assignment, 5 anchor boxes were chosen for you (to cover the 80 classes), and stored in the file './model_data/yolo_anchors.txt'
- The dimension for anchor boxes is the second to last dimension in the encoding: $(m, n_H, n_W, anchors, classes)$.
- The YOLO architecture is: IMAGE (m, 608, 608, 3) -> DEEP CNN -> ENCODING (m, 19, 19, 5, 85).

Encoding

Let's look in greater detail at what this encoding represents.

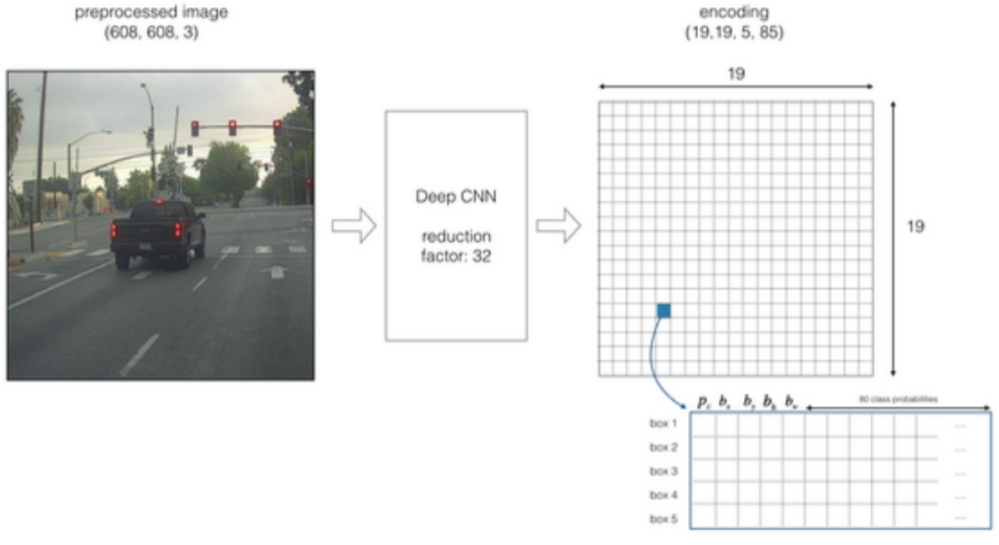


Figure 2 : Encoding architecture for YOLO

If the center/midpoint of an object falls into a grid cell, that grid cell is responsible for detecting that object.

Since you're using 5 anchor boxes, each of the 19 x19 cells thus encodes information about 5 boxes. Anchor boxes are defined only by their width and height.

For simplicity, you'll flatten the last two dimensions of the shape (19, 19, 5, 85) encoding, so the output of the Deep CNN is (19, 19, 425).

For simplicity, you'll flatten the last two dimensions of the shape (19, 19, 5, 85) encoding, so the output of the Deep CNN is (19, 19, 425).

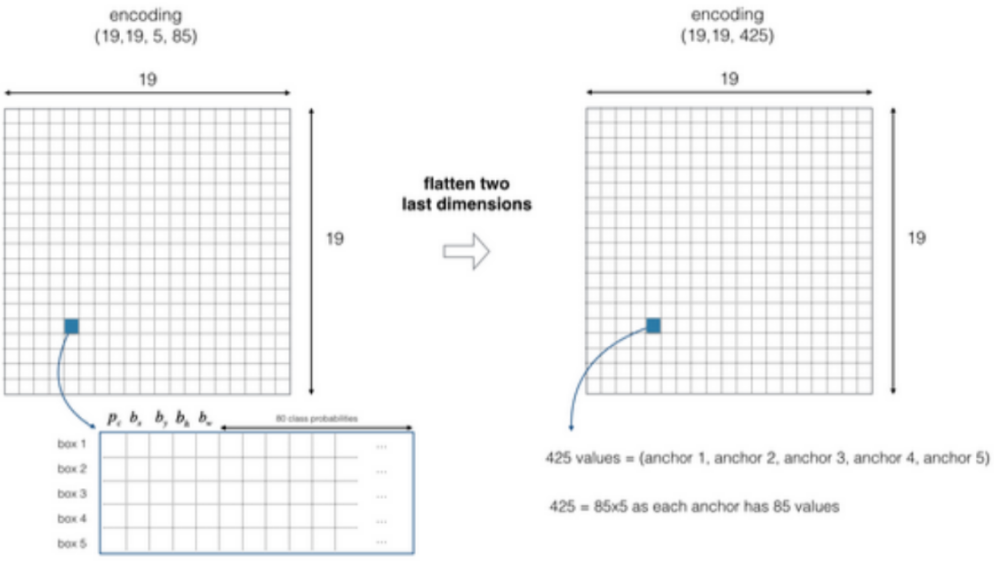


Figure 3 : Flattening the last two last dimensions

Class score

Now, for each box (of each cell) you'll compute the following element-wise product and extract a probability that the box contains a certain class.
The class score is $score_{c,i} = p_c \times c_i$: the probability that there is an object p_c times the probability that the object is a certain class c_i.

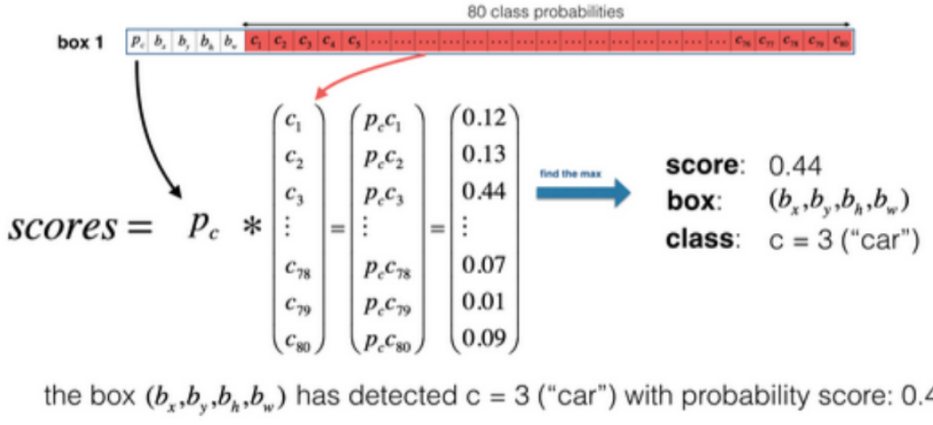


Figure 4: Find the class detected by each box

Example of figure 4

- In figure 4, let's say for box 1 (cell 1), the probability that an object exists is $p_1 = 0.60$. So there's a 60% chance that an object exists in box 1 (cell 1).
- The probability that the object is the class "category 3 (a car)" is $c_3 = 0.73$.
- The score for box 1 and for category "3" is $score_{1,3} = 0.60 \times 0.73 = 0.44$.

Visualizing classes

Here's one way to visualize what YOLO is predicting on an image:

- For each of the 19x19 grid cells, find the maximum of the probability scores (taking a max across the 80 classes, one maximum for each of the 5 anchor boxes).
- Color that grid cell according to what object that grid cell considers the most likely.

Doing this results in this picture:

Figure 5: Each one of the 19x19 grid cells is colored according to which class has the largest predicted probability in that cell.

Note that this visualization isn't a core part of the YOLO algorithm itself for making predictions; it's just a nice way of visualizing an intermediate result of the algorithm.

Visualizing bounding boxes

Another way to visualize YOLO's output is to plot the bounding boxes that it outputs. Doing that results in a visualization like this:

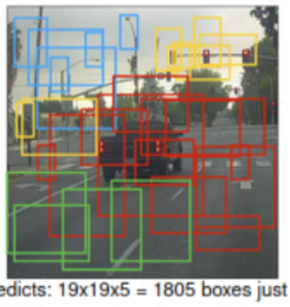


Figure 6: Each cell gives you 5 boxes. In total, the model predicts: 19x19x5 = 1805 boxes just by looking once at the image (one forward pass through the network)! Different colors denote different classes.

Non-Max suppression

In the figure above, the only boxes plotted are ones for which the model had assigned a high probability, but this is still too many boxes. You'd like to reduce the algorithm's output to a much smaller number of detected objects.

To do so, you'll use **non-max suppression**. Specifically, you'll carry out these steps:

- Get rid of boxes with a low score. Meaning, the box is not very confident about detecting a class, either due to the low probability of any object, or low probability of this particular class.
- Select only one box when several boxes overlap with each other and detect the same object.

2.3 - Non-max Suppression

Even after filtering by thresholding over the class scores, you still end up with a lot of overlapping boxes. A second filter for selecting the right boxes is called non-maximum suppression (NMS).

Figure 7 : In this example, the model has predicted 3 cars, but it's actually 3 predictions of the same car. Running non-max suppression (NMS) will select only the most accurate (highest probability) of the 3 boxes.

Non-max suppression uses the very important function called **"Intersection over Union"**, or IoU.

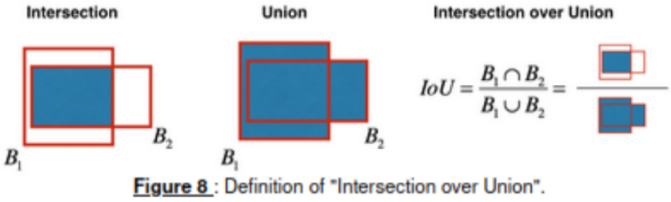


Figure 8 : Definition of "Intersection over Union".

Exercise 2 - iou

Implement `iou()`

Some hints:

- This code uses the convention that (0,0) is the top-left corner of an image, (1,0) is the upper-right corner, and (1,1) is the lower-right corner. In other words, the (0,0) origin starts at the top left corner of the image. As x increases, you move to the right. As y increases, you move down.
- For this exercise, a box is defined using its two corners: upper left (x_1, y_1) and lower right (x_2, y_2), instead of using the midpoint, height and width. This makes it a bit easier to calculate the intersection.
- To calculate the area of a rectangle, multiply its height $(y_2 - y_1)$ by its width $(x_2 - x_1)$. Since (x_1, y_1) is the top left and x_2, y_2 are the bottom right, these differences should be non-negative.
- To find the **intersection** of the two boxes (xi_1, yi_1, xi_2, yi_2):
  - Feel free to draw some examples on paper to clarify this conceptually.
  - The top left corner of the intersection (xi_1, yi_1) is found by comparing the top left corners (x_1, y_1) of the two boxes and finding a vertex that has an x-coordinate that is closer to the right, and y-coordinate that is closer to the bottom.
  - The bottom right corner of the intersection (xi_2, yi_2) is found by comparing the bottom right corners (x_2, y_2) of the two boxes and finding a vertex whose x-coordinate is closer to the left, and the y-coordinate that is closer to the top.
  - The two boxes **may have no intersection**. You can detect this if the intersection coordinates you calculate end up being the top right and/or bottom left corners of an intersection box. Another way to think of this is if you calculate the height $(y_2 - y_1)$ or width $(x_2 - x_1)$ and find that at least one of these lengths is negative, then there is no intersection (intersection area is zero).

2.4 - YOLO Non-max Suppression

You are now ready to implement non-max suppression. The key steps are:

1. Select the box that has the highest score.
2. Compute the overlap of this box with all other boxes, and remove boxes that overlap significantly (iou >= `iou_threshold`).
3. Go back to step 1 and iterate until there are no more boxes with a lower score than the currently selected box.

This will remove all boxes that have a large overlap with the selected boxes. Only the "best" boxes remain.

Notebooks explanations: U-Net

What you should remember:

- Semantic image segmentation predicts a label for every single pixel in an image
- U-Net uses an equal number of convolutional blocks and transposed convolutions for downsampling and upsampling
- Skip connections are used to prevent border pixel information loss and overfitting in U-Net

3.1 - Model Details

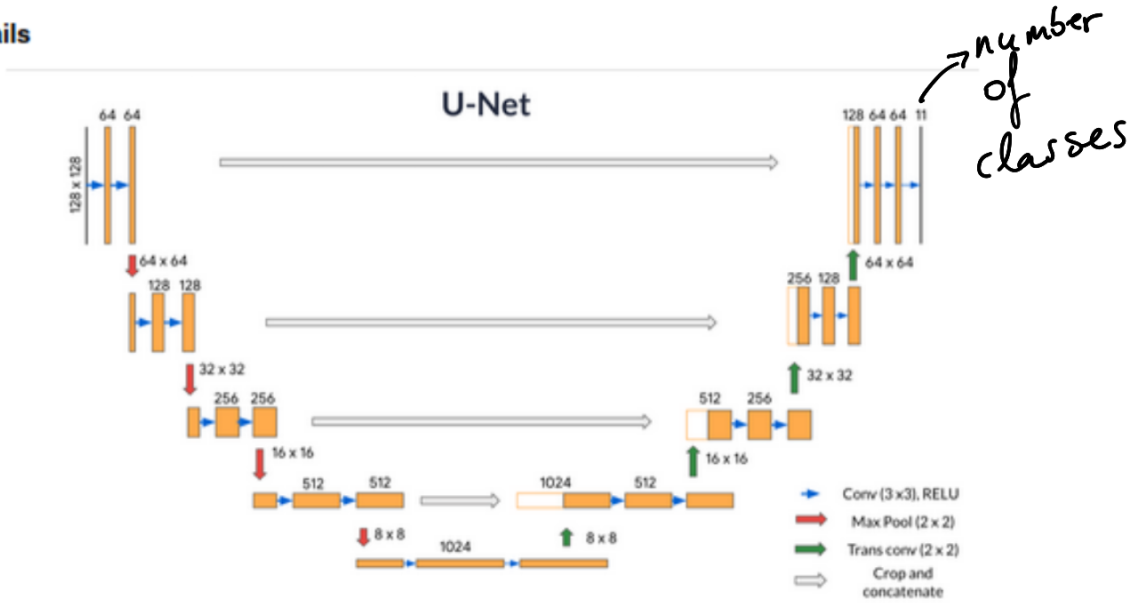


Figure 2 : U-Net Architecture

Contracting path (Encoder containing downsampling steps):

Images are first fed through several convolutional layers which reduce height and width, while growing the number of channels.

The contracting path follows a regular CNN architecture, with convolutional layers, their activations, and pooling layers to downsample the image and extract its features. In detail, it consists of the repeated application of two 3 x 3 unpadded convolutions, each followed by a rectified linear unit (ReLU) and a 2 x 2 max pooling operation with stride 2 for downsampling. At each downsampling step, the number of feature channels is doubled.

Crop function: This step crops the image from the contracting path and concatenates it to the current image on the expanding path to create a skip connection.

Expanding path (Decoder containing upsampling steps):

The expanding path performs the opposite operation of the contracting path, growing the image back to its original size, while shrinking the channels gradually.

In detail, each step in the expanding path upsamples the feature map, followed by a 2 x 2 convolution (the transposed convolution). This transposed convolution halves the number of feature channels, while growing the height and width of the image.

Next is a concatenation with the correspondingly cropped feature map from the contracting path, and two 3 x 3 convolutions, each followed by a ReLU. You need to perform cropping to handle the loss of border pixels in every convolution.

Final Feature Mapping Block: In the final layer, a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. The channel dimensions from the previous layer correspond to the number of filters used, so when you use 1x1 convolutions, you can transform that dimension by choosing an appropriate number of 1x1 filters. When this idea is applied to the last layer, you can reduce the channel dimensions to have one layer per class.

The U-Net network has 23 convolutional layers in total.

Important Note:

The figures shown in the assignment for the U-Net architecture depict the layer dimensions and filter sizes as per the original paper on U-Net with smaller images. However, due to computational constraints for this assignment, you will code only half of those filters. The purpose of showing you the original dimensions is to give you the flavour of the original U-Net architecture. The important takeaway is that you multiply by 2 the number of filters used in the previous step. The notebook includes all of the necessary instructions and hints to help you code the U-Net architecture needed for this assignment.

3.2 - Encoder (Downsampling Block)

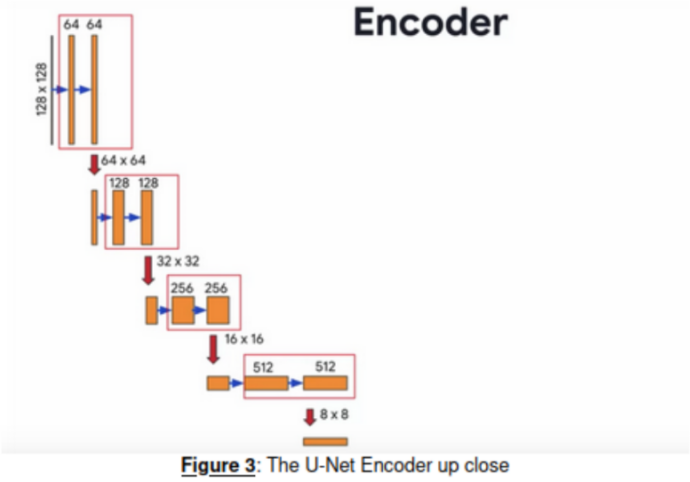


Figure 3: The U-Net Encoder up close

The encoder is a stack of various conv_blocks:

Each `conv_block()` is composed of 2 **Conv2D** layers with ReLU activations. We will apply **Dropout**, and **MaxPooling2D** to some conv_blocks, as you will verify in the following sections, specifically to the last two blocks of the downsampling.

The function will return two tensors:

- `next_layer` : That will go into the next block.
- `skip_connection` : That will go into the corresponding decoding block.

Note: If `max_pooling=True` , the `next_layer` will be the output of the MaxPooling2D layer, but the `skip_connection` will be the output of the previously applied layer(Conv2D or Dropout, depending on the case). Else, both results will be identical.

3.3 - Decoder (Upsampling Block)

The decoder, or upsampling block, upsamples the features back to the original image size. At each upsampling level, you'll take the output of the corresponding encoder block and concatenate it before feeding to the next decoder block.

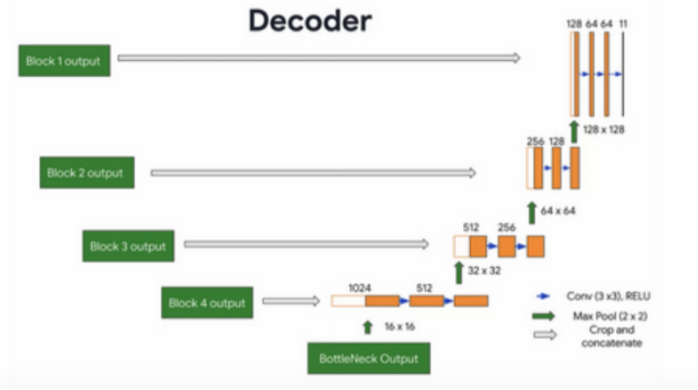


Figure 4: The U-Net Decoder up close

There are two new components in the decoder: `up` and `merge` . These are the transpose convolution and the skip connections. In addition, there are two more convolutional layers set to the same parameters as in the encoder.

Here you'll encounter the `Conv2DTranspose` layer, which performs the inverse of the `Conv2D` layer. You can read more about it here.

Exercise 2 - upsampling_block

Implement `upsampling_block(...)` .

For the function `upsampling_block` :

- Takes the arguments `expansive_input` (which is the input tensor from the previous layer) and `contractive_input` (the input tensor from the previous skip layer)
- The number of filters here is the same as in the downsampling block you completed previously
- Your `Conv2DTranspose` layer will take `n_filters` with shape (3,3) and a stride of (2,2), with padding set to `same` . It's applied to `expansive_input` , or the input tensor from the previous layer.

This block is also where you'll concatenate the outputs from the encoder blocks, creating skip connections.

- Concatenate your Conv2DTranspose layer output to the contractive input, with an `axis` of 3. In general, you can concatenate the tensors in the order that you prefer. But for the grader, it is important that you use `[up, contractive_input]`

For the final component, set the parameters for two Conv2D layers to the same values that you set for the two Conv2D layers in the encoder (ReLU activation, He normal initializer, `same` padding).

3.6 - Loss Function

In semantic segmentation, you need as many masks as you have object classes. In the dataset you're using, each pixel in every mask has been assigned a single integer probability that it belongs to a certain class, from 0 to num_classes-1. The correct class is the layer with the higher probability.

This is different from categorical crossentropy, where the labels should be one-hot encoded (just 0s and 1s). Here, you'll use sparse categorical crossentropy as your loss function, to perform pixel-wise multiclass prediction. Sparse categorical crossentropy is more efficient than other loss functions when you're dealing with lots of classes.