

# YOLOv3 for object detection

Alinei-Poiană Tudor

## Cuprins

1) (CNNs) .....	2
2) YOLOv3 development .....	4
3) Mathematical methodology.....	6
3.1) Intersection over union (IoU).....	6
3.2) Non-maximum suppression (not quite mathematical but I leave it here).....	7
3.3) 2D convolution .....	8
3.4) Batch normalization .....	10
3.5) Categorical cross entropy function and binary cross entropy function .....	10
3.6) Leaky ReLU .....	11
3.7) Adam algorithm for optimization.....	12
3.8) Upsampling using transposed convolutions and nearest neighbor approach .....	15
3.8.1) Transposed convolutions .....	15
3.8.2) Nearest neighbor upsampling .....	18
4) YOLOv3 architecture.....	18
4.1) Backbone architecture .....	18
4.2) YOLOv3 architecture.....	20
4.3) Grid cells and Anchor boxes .....	21
4.4) Multi-scale prediction and Skip connections .....	22
4.5) Residual blocks .....	23
4.6) Bounding box prediction.....	24
4.7) Class prediction.....	25
4.7) YOLOv3 loss function .....	26
4.7.1) Regression loss .....	27
4.7.2) Confidence loss .....	27
4.7.3) Class loss .....	28
5) Implementation .....	29
5.1) Used packages .....	29
5.2) Main implemented blocks .....	29

6) Train and infrastructure .....	34
7) Results .....	35
8) Bonus: Lagrange multipliers, SVD visualization for images and formulation for optimization of learning rate.....	38
8.1) SVD visualization for RGB and grayscale image .....	38
8.2) Solving optimization problem with constraints using Lagrange multipliers and Karush-Kuhn-Tucker conditions (MATLAB code) .....	44
8.3) Learning rate optimization .....	46
9) References.....	47

## 1)(CNNs)

Neural networks (NN) have a reach history, the first concept of NN emerged between 1940s and 1950s, but it wasn't until the 1960s that researchers began exploring the idea of hierarchical feature learning.

Hubel and Wiesel conducted research on the visual cortex in the early 1960s, discovering the existence of simple and complex cells that respond to specific visual stimuli.

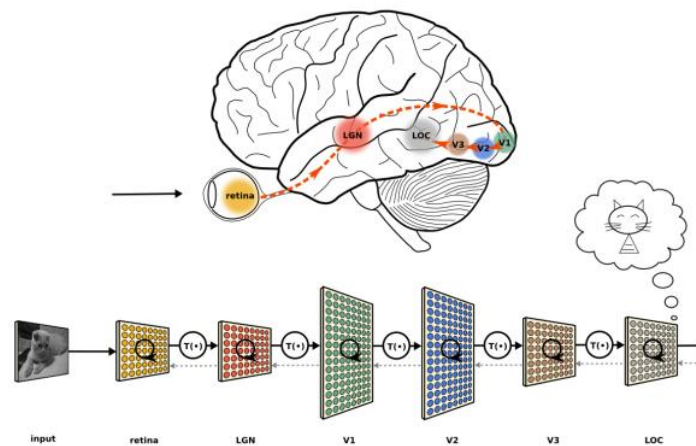


Figure 1: Visual cortex hierarchy

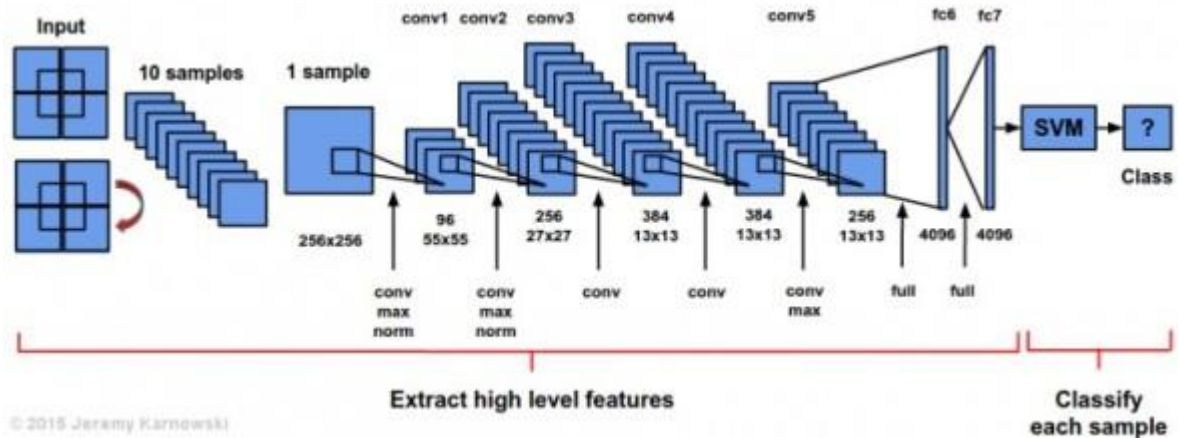


Figure 2: Convolutional neural network architecture

The visual cortex is a part of the brain responsible for processing visual information. In the context of Convolutional Neural Networks (CNNs), the architecture draws inspiration from the hierarchical organization and feature extraction mechanisms observed in the visual cortex.

Visual Cortex	CNN
<ol style="list-style-type: none"> <li>1) The visual cortex is organized hierarchically, with different layers responsible for extracting features at varying levels of abstraction. Information flows through these layers in a sequential manner;</li> <li>2) Neurons in the visual cortex respond to specific features of visual stimuli. Simple cells might detect basic edges and textures, while complex cells combine these features into more complex patterns. This hierarchical feature extraction mirrors the gradual abstraction of information in the visual processing pathway;</li> <li>3) Neurons have receptive fields which are specific regions in the visual field that trigger a response. Simple cells respond to oriented edges within their receptive fields, and complex cells respond to the presence of these edges in a broader context.</li> </ol>	<ol style="list-style-type: none"> <li>1) CNN architectures typically consist of multiple convolutional and pooling layers stacked in a hierarchical manner. Each layer learns to capture increasingly abstract and complex features, similar to the hierarchical organization observed in the visual cortex;</li> <li>2) Convolutional layers mimic the receptive fields of neurons in the visual cortex. These layers apply convolutional filters (kernels) to input images, capturing local patterns like edges and textures;</li> <li>3) Pooling layers, inspired by the concept of spatial summation in the visual cortex, downsample the output of convolutional layers, reducing spatial dimensions while retaining important features. Common pooling operations include max pooling (we will see in the YoloV3 architecture that the downsample can be done by using kernels with stride different from 1);</li> </ol>

	<ol style="list-style-type: none"> <li>4) After several convolutional and pooling layers, CNNs often have fully connected layers for high-level feature aggregation and classification. These layers integrate information across the entire input space;</li> <li>5) Neurons in the visual cortex share parameters and learn to respond to specific features regardless of their location. CNNs employ weight sharing through convolutional filters, allowing the model to learn and recognize patterns invariant to translation</li> </ol>
--	--

## 2) YOLOv3 development

YOLO (You Only Look Once) is a real-time object detection system, and YOLOv1 is its initial architecture. Developed by Joseph Redmon et al. [1], YOLO introduced a groundbreaking approach to object detection by framing it as a regression problem for bounding box coordinates and class probabilities. YOLOv1 was presented in the paper titled "You Only Look Once: Unified, Real-Time Object Detection" in 2016.

YOLOv1 purpose was to process the entire image in a single forward pass through the neural network. Instead of using a sliding window or region proposal network, YOLO divides the input image into a grid and predicts bounding boxes and class probabilities directly for each grid cell. The image is divided into an  $S \times S$  grid, and each grid cell is responsible for predicting a fixed number of bounding boxes. For each bounding box, YOLO predicts the x and y coordinates of the box's center, width and height, confidence score for box existence, and class probabilities.

Compared to other real time detectors, YOLOv1 was capable of both working at high FPS and give satisfactory results in terms of mean Average Precision.

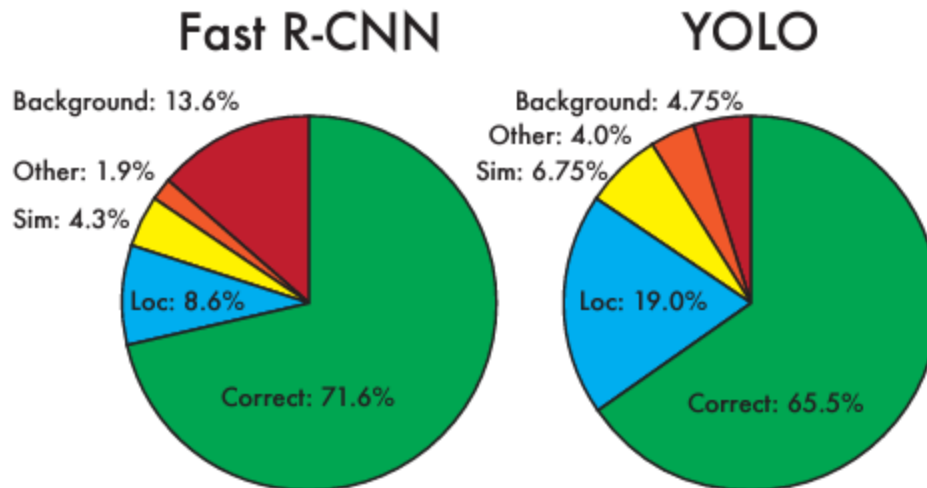


Figure 3: Error comparison between Fast R-CNN and YOLO [1]

Although the result for YOLOv1 were promising, it also had certain limitations like:

- YOLOv1 struggles with handling small objects or many objects crowded in a grid cell.
- The fixed number of bounding boxes per grid cell may lead to imprecise localization for objects with different scales.
- Not being able to predict and understand complex scenes (as a matter of fact, the YOLOv1 used 64x64 grid cells on 448x448 images, so it was capable of only predicting a maximum of 49 objects inside a picture)

The evolution of the YOLO architecture was an iterative process. Following the YOLOv1 we have YOLO9000 (YOLOv2) which extended the capabilities of v1 to over 9000 classes and also introduced the concept of anchor boxes (which will be described in the context of YOLOv3). YOLOv3 was first introduced in 2018 having the following key features:

- Feature pyramid network (FPN): YOLOv3 detects objects at three different scales which makes it useful for handling both small and large objects;
- Darknet-53 Backbone;
- Objectness Score and Class Scores: it introduced a separate objectness score and class score for each bounding box prediction in the loss function, improving the model's ability to distinguish between background and objects

In the next chapter we will go through the whole structure and methodology that stands behind YOLOv3 architecture.

### 3) Mathematical methodology

#### 3.1) Intersection over union (IoU)

Mathematically, as the name suggests, IoU between two bounding boxes is the ratio of the area of intersection between two bounding boxes to the area of the union of two bounding boxes. A more intuitive formulation can be observed in Fig. 4.

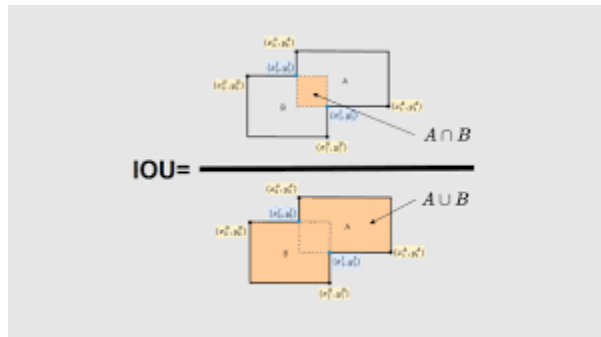


Figure 4: Intuitive representation of IoU ([Machine learning space](#))

IoU is a measure which tells us how close is the bounding box prediction for a detected object to the real bounding box of the object. The closer to 1 this ratio is, the more accurate is the prediction.

How can we compute each of those two areas for IoU? Well, let's take a look at Fig. 5.

For the intersection area we have to find two pairs of points corresponding to the top left corner of the intersection and the bottom right corner of the intersection. Knowing the coordinates of each

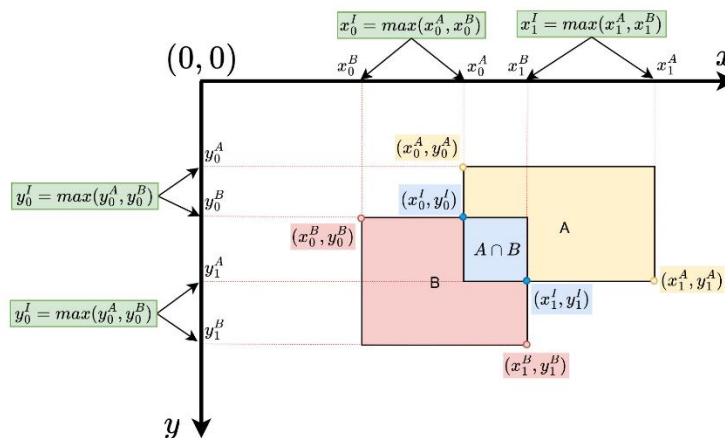


Figure 5: IoU areas

bounding box top left corner and width and height of the bounding box, we can find the bottom right corner of each box. Then, we need to find the maximum between the top left corners for x coordinate and the maximum between the top left y coordinate in order to determine the top left corner coordinates for the intersection area  $(x_0, y_0)$ . Analog, we can apply the same reasoning for

the bottom right corner coordinates of each A and B area, but instead of using the maximum we take the minimum between each pair, to find the bottom right corner of the intersection area  $(x_1, y_1)$ .

We can now compute the area of intersection:  $(x_1 - x_0) * (y_1 - y_0)$ .

The area of union can be computed by subtracting from the sum of the two areas A and B the area of union computed above.

Although it seems simple, we have to point out that this computation does not take into account the possibility of no intersection between the two areas.

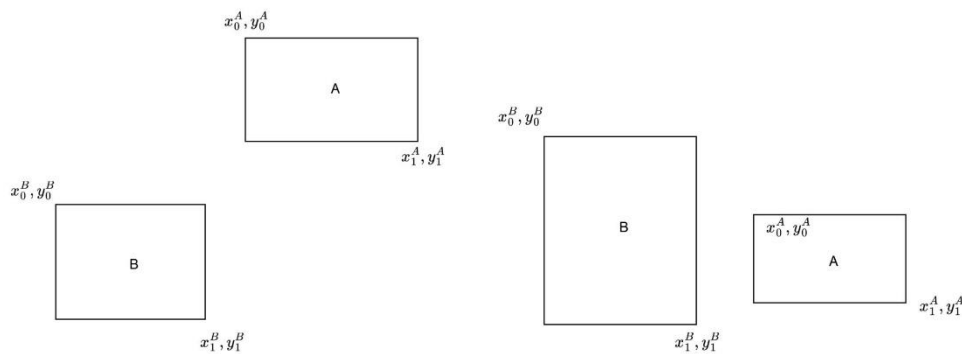


Figure 6: Special cases of false IoU

In the case presented in the left image, by using the above presented algorithm, the width  $(x_1 - x_0)$  and the height of the intersection box  $(y_1 - y_0)$  will have negative values. This thing tells us that there is no intersection between the two boxes.

For the image on the left, we can apply the same principle, the only difference being made in the conclusion, namely, that if either the width or the height have negative values, then there is no intersection between the two bounding boxes.

### 3.2) Non-maximum suppression (not quite mathematical but I leave it here)

Non-maximum suppression is a technique used in computer vision and object detection to eliminate overlapping or redundant predicted bounding boxes. In general, multiple potential bounding boxes are predicted around multiple objects, the goal being to retain only the most relevant and accurate bounding boxes.

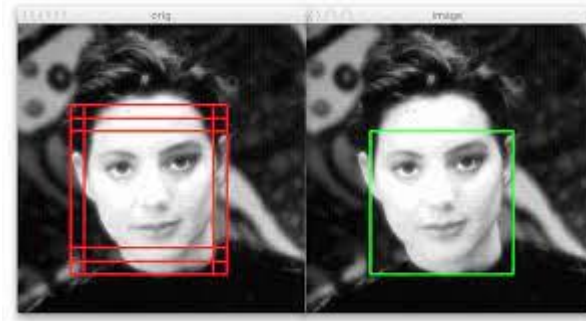


Figure 7: Image before and after non-maximum suppression

The algorithm has the following steps, considering we have the predicted boxes  $B$ , corresponding confidence scores  $S$  and an IoU threshold score,  $N$  for our predicted boxes:

1. We select the predicted box with the highest confidence score (usually this is done by first sorting the boxes in descending order according to their confidence score).
2. We compare the above selected box (let's call it  $A$ ) with all the other boxes– we calculate the IoU of this prediction with every other prediction ( $X$ ) and if this IoU score is higher than  $N$ , we remove  $X$  from  $B$ .
3. We add  $A$  to a list of remaining proposed bounding boxes and remove  $A$  from  $B$ .
4. After that, we take the next prediction with the highest confidence and repeat step (2) and (3).
5. This is repeated until there are no left predictions in  $B$ .

In Fig. 8 we have a pseudocode for NMS algorithm.

---

**Algorithm 1** Non-Max Suppression

---

```

1: procedure NMS( $B, c$ )
2:    $B_{nms} \leftarrow \emptyset$  Initialize empty set
3:   for  $b_i \in B$  do => Iterate over all the boxes
4:      $discard \leftarrow \text{False}$  Take boolean variable and set it as false. This variable indicates whether b(i) should be kept or discarded
5:     for  $b_j \in B$  do Start another loop to compare with b(j)
6:       if  $\text{same}(b_i, b_j) > \lambda_{nms}$  then If both boxes having same IOU
7:         if  $\text{score}(c, b_j) > \text{score}(c, b_i)$  then
8:            $discard \leftarrow \text{True}$  Compare the scores. If score of b(i) is less than that of b(j), b(i) should be discarded, so set the flag to True.
9:         if not  $discard$  then Once b(i) is compared with all other boxes and still the discarded flag is False, then b(i) should be considered. So add it to the final list.
10:           $B_{nms} \leftarrow B_{nms} \cup b_i$ 
11:   return  $B_{nms}$  Do the same procedure for remaining boxes and return the final list

```

---

Figure 8: Pseudo code for NMS algorithm

### 3.3) 2D convolution

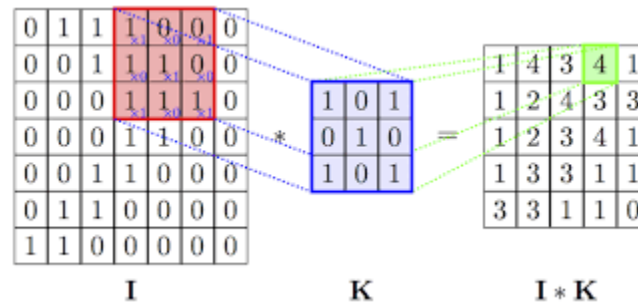
Two-dimensional convolution is a fundamental operation in image processing. It is commonly used for tasks such as image filtering, feature extraction, and pattern recognition. The mathematical representation of 2D convolution involves the convolution operation denoted by



the symbol "\*". Let's consider two functions  $f(x,y)$  and  $g(x,y)$ , where  $f$  is the input image and  $g$  is the filter (also known as kernel or mask).


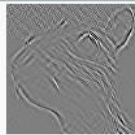
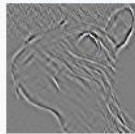


The 2D convolution of  $f$  and  $g$ , denoted as  $f * g$ , is defined as follows:

$$(f * g)(x,y) = \sum_m \sum_n f(m,n) \cdot g(x-m, y-n)$$



The convolution operation involves sliding the filter  $g$  over the input image  $f$  and computing the sum of element-wise products (also called dot product) at each position. For a specific position  $(x, y)$  in the result of the convolution, the convolution involves multiplying the corresponding elements of the filter and the input image at various positions and summing all these products.

Down bellow, you have a few examples of 3x3 kernels used for extracting different shapes and features from images or used to filter input images:

Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Ridge or edge detection	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

### 3.4) Batch normalization

Batch normalization is a technique commonly used in neural networks to improve training stability and speed. It normalizes the input of a layer by adjusting and scaling the result of activation functions. This is useful in dealing with problems like vanishing gradients (due to smaller values gradients tend towards 0 so in parameter update step during the backpropagation we basically do not update the parameters or the update is too small and no learning is performed which will make the training process slower) or exploding gradients (higher values might determine the gradients to take values that tend to infinity and the updated parameters have big changes in their values which might make the training unstable).

In the context of convolutional neural networks (CNNs) or 2D convolutional layers, batch normalization can be extended to work with mini-batches of data. This extension is known as Batch Normalization for 2D data or Batch 2D Normalization.

For a given mini-batch of data, the batch 2D normalization operation is applied independently to each channel along spatial dimensions. Let's denote the input to the batch 2D normalization as  $X \in \mathbb{R}^{N \times C \times H \times W}$ , where: N is the batch size, C is the number of channels, H is the height of the input and W is the width of the input.

We first have to compute the mean  $\mu$  and variance  $\sigma^2$  for each channel independently, over the mini batch:

$$\mu_c = \frac{1}{N \cdot H \cdot W} \sum_{n=1}^N \sum_{h=1}^H \sum_{w=1}^W X_{n,c,h,w}$$
$$\sigma_c^2 = \frac{1}{N \cdot H \cdot W} \sum_{n=1}^N \sum_{h=1}^H \sum_{w=1}^W (X_{n,c,h,w} - \mu_c)^2$$

Then we normalize the activations for each channel:

$$\hat{X}_{n,c,h,w} = \frac{X_{n,c,h,w} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$$

Where epsilon is added in the denominator for numerical stability. It can be choose arbitrarily and it's a small constant.

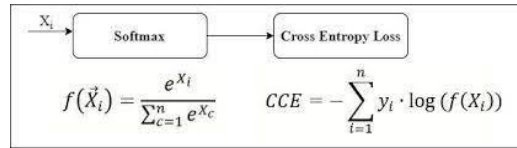
After this we have to scale and shift the normalized activations using learnable parameters  $\gamma, \beta$ :

$$Y_{n,c,h,w} = \gamma_c \hat{X}_{n,c,h,w} + \beta_c$$

The learnable parameters  $\gamma, \beta$  are introduced to allow the network to learn the optimal scaling and shifting for each channel.

### 3.5) Categorical cross entropy function and binary cross entropy function

Categorical cross entropy (CCE) function is a commonly used function for classification problems where multiple classed are involved. The useful scenario is when each example belongs to only one class. It is usually used alongside softmax activation function.



where  $y_i$  is the true probability distribution (the probability of an example to be in a class  $i$ -represented by a one),  $\hat{y}$  is the predicted probability distribution (the output from the softmax function  $f$  computed for a class  $i$ ).

This function is used to measure the dissimilarity between the true distribution of class labels ( $y$ ) and the predicted distribution  $f(X)$ .

In short terms, when the true class has a probability close to 1, the loss is close to 0 (because  $\log(1)=0$ ), but as the predicted probability  $f(X)$  tends to 0, the loss increases.

Binary Cross-Entropy (BCE) is a common loss function used in binary classification problems, where each example belongs to one of two classes (0 or 1). The mathematical expression for Binary Cross-Entropy loss is as follows:

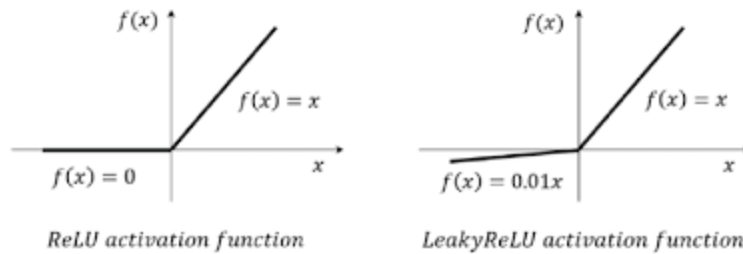
$$L(y, \hat{y}) = -[y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})]$$

Where  $y$  is the true label (either 0 or 1) and  $\hat{y}$  is the predicted probability of the positive class (class 1). The BCE loss function penalizes the model more when the predicted probability diverges from the true label for both positive (class 1) and negative (class 0) classes. The BCE loss function is widely used in binary classification problems and is often combined with the sigmoid activation function, which squashes model outputs to the range  $[0, 1]$ , representing probabilities. The combination of BCE and sigmoid ensures that the predicted values form a valid probability distribution over the two classes. In the context of YOLOv3 it is used to determine whether we have an object of interest in a specific grid cell during the prediction step.

### 3.6) Leaky ReLU

Leaky ReLU (Rectified Linear Unit) is an activation function used in artificial neural networks. It is a variation of the traditional ReLU activation function. The key difference is that Leaky ReLU allows a small, non-zero gradient for negative inputs. The mathematical expression for Leaky ReLU is given by:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$



You might be asking: Why using leaky ReLU instead of regular ReLU?

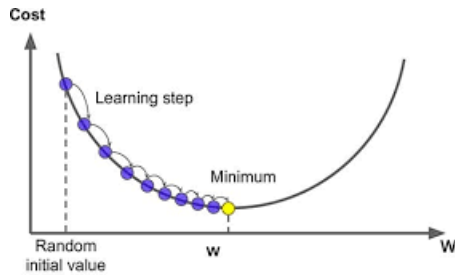
This variation is useful in dealing with the problem of the “dying neuron” which arises when neurons become inactive, meaning they consistently output zero for all inputs during training. This is a problem because:

- We lose the capacity to represent features: in a neural network, each neuron is responsible for learning specific patterns and features of the input data. If a neuron is inactive, it loses the ability to contribute to the learning process, thus, certain features may not be captured by the network.
- The learning capacity is reduced: inactivation of neurons reduces the overall learning capacity of the network. The model may struggle to adapt and learn complex relationships in the data, leading to suboptimal performance.
- Gradient saturation phenomena: In the case of ReLU activation, when the output is always zero (for negative inputs), the gradient during backpropagation becomes zero as well. This is known as gradient saturation, and it hinders the training process because weights are not updated, and the network fails to learn from certain examples.

### 3.7) Adam algorithm for optimization

We can remind a few basic optimization techniques: Gradient Descent, Stochastic Gradient Descent, Momentum Gradient Descent, Root Mean Square Propagation.

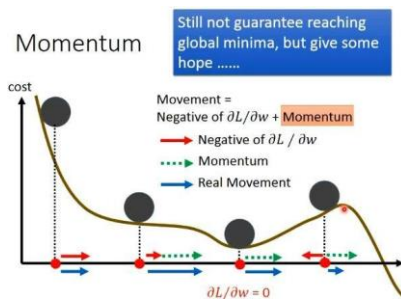
Gradient Descent is the simplest of all. It takes into account the gradient of the loss function with respect to the parameters and a predefined step size which combined, give us the direction and the step length that can lead us to a minimum. The drawbacks are that it converges to a local minimum and also uses the entire batch size, which can be very computationally inefficient.



$$\theta = \theta - \alpha \cdot \frac{\partial L}{\partial \theta}$$

Stochastic Gradient Descent has the same principle as GD but with a small improvement. It doesn't take the whole batch of data into account at one iteration, but instead, it takes a specified number of random samples. Now we overcome the computational limitation of memory, but we are still stuck with the local minima problem.

The first solution to this problem could be using Momentum Gradient Descent. It speeds up training by accelerating gradients in the right directions by adding a fraction of the previous gradient to the current one. This principle is pretty simple, let's take an example: we assume that the gradient descent is like a ball rolling down a hill. Normally, it will take fixed steps as the learning rate is the same throughout. That means you calculate the gradient at each step and take a step in that direction of value  $\alpha$ . The Momentum technique helps you to realize that, since the last few steps have been pretty much in the same direction, you could apply a boost to the current step to move in that direction so you need to take fewer steps.



$$\begin{cases} v_t = \gamma \cdot v_{t-1} + \eta \frac{\partial L}{\partial \theta} \\ \theta = \theta - v_t \end{cases}$$

Though it proved to be a good method, it still has a high chance to get stuck pretty easily into a local minima.

As you noticed, the learning rate  $\eta$  is the same for all the values in  $\theta$ . We basically do not account for different changes in the variation of particular dimensions of  $\theta$ . The main idea behind RMSprop is to adapt the learning rates for each parameter individually. It does this by maintaining a moving average of the squared gradients of the parameters (which is basically an estimation of the variance of gradients). The algorithm adjusts the learning rates based on the magnitudes of these moving averages.

At each iteration of the training process, calculate the squared gradients for each parameter by squaring the gradient of the loss function with respect to that parameter.

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta) \left( \frac{\partial L}{\partial \theta} \right)^2$$

Where  $E[g^2]_t$  is the moving average of the squared gradients at time  $t$ ,  $\beta$  is a decay rate (typically between (0.85-0.9)) and  $\frac{\partial L}{\partial \theta}$  is the gradient of the loss function with respect to the parameters at time  $t$ .

Then, we adjust the parameters based on the computed learning rates:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2] + \varepsilon}} \frac{\partial L}{\partial \theta} |_{\theta_t}$$

Where  $\frac{\partial L}{\partial \theta} |_{\theta_t}$  is the gradient of the loss function with respect to the parameters at time  $t$  and  $\varepsilon$  is a small constant added to the denominator to prevent the division by zero. When the variance is high, it suggests that the parameters are changing rapidly, and using a larger learning rate may lead to overshooting or instability. Conversely, when the variance is low, it indicates a more stable situation, and a larger learning rate may be appropriate for faster convergence.

Adam optimization combines both the Momentum gradient descent and RMSprop techniques in the following order:

We compute the exponential moving average of the gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L}{\partial \theta} |_{\theta_t}$$

where  $m_t$  is the exponentially decaying average of past gradients and  $\beta_1$  controls the exponential decay ( approx. 0.9).

Then we take the exponential moving average of squared gradients, like in RMSprop:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left( \frac{\partial L}{\partial \theta} |_{\theta_t} \right)^2$$

We perform a bias correction because the moving averages are initialized with zero and can be biased towards zero, especially in the beginning of the training :

$$\begin{cases} \hat{m}_t = \frac{m_t}{1 - \beta_1} \\ \hat{v}_t = \frac{v_t}{1 - \beta_2} \end{cases}$$

And finally the correction step:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

## 3.8) Upsampling using transposed convolutions and nearest neighbor approach

### 3.8.1) Transposed convolutions

Before talking about transposed convolutions it should be clear that in spite of the naming Deconvolutional layer, transposed convolutions is different from the actual deconvolution (which is the reverse operation of convolution). They are similar in the sense that the spatial dimension generated by both are the same. Transposed convolution doesn't reverse the standard convolution by values, rather by dimensions only.

A transposed convolution looks like a standard convolutional layer but it is applied on a different feature map. While the standard convolution operation gives a feature map with less or an equal dimension than the input, the transposed convolution is used to generate an output feature map that has spatial dimensions greater than the feature map. Also, this operation is defined by padding and stride  $p$  and  $s$ .

We need to introduce a few notations:

- $i$  is the size of the input
- $k$  which is the kernel's size
- $o$  output feature map size

We know that after a convolution, the output size  $o$  is:

$$o = \frac{i + 2p - k}{s} + 1$$

Firstly, we have to determine the desired dimensions of the upsampled matrix. We can take a look at the above formula and consider the following change:  $o$  is now the dimension of the input matrix (so we denote it by  $i$ ) and  $i$  is the desired output matrix (which we want to be higher, so we denote it by  $o$ ).

$$i = \frac{o + 2p - k}{s} + 1 \rightarrow o = (i - 1)s + k - 2p$$

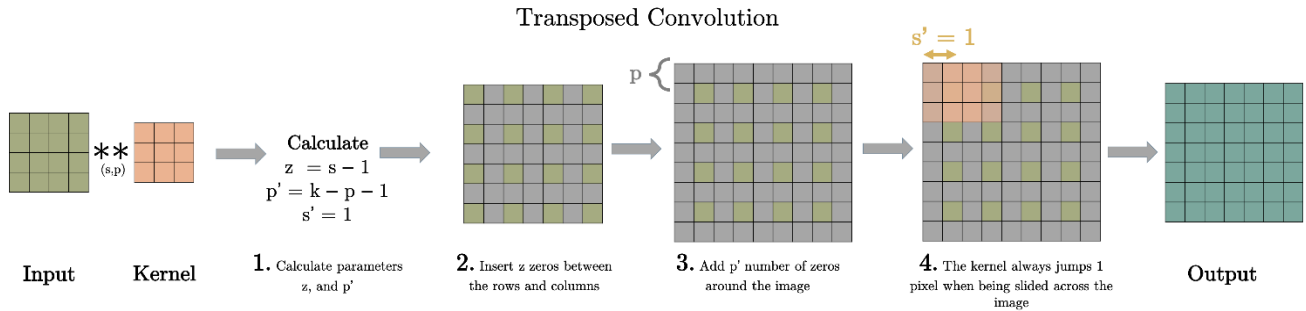
We actually described the dimensions of the new upsampled matrix, considering the dimensions of the input, kernel size, stride and padding.

Like we mentioned before, this operation has two parameters: padding and stride.

Algorithmically it looks like:

- 1) we compute  $z = s - 1$  and  $p' = k - p - 1$
- 2) between each row and columns of the input, we insert  $z$  number of zeros. This will increase the image size of the input to  $(2i - 1) \times (2i - 1)$ .
- 3) We pad the modified input image with  $p'$  zeros

4) Now we can carry out a standard convolution on the image generated from 2 and 3 with a stride of 1.



Above we have an example for upsampling an input image of 4x4, with a kernel of 3x3, stride 2 and padding 1, in order to obtain a 7x7 image.

This operation is interesting because we can learn how to upsample a smaller image and preserve or even enhance the key features on a bigger scale.

Another approach to explain transpose convolution begins from the normal convolution itself.

Let's consider we have the following kernel:

	0	1	2
0	1	2	4
1	1	1	3
2	1	2	4

Kernel (3, 3)

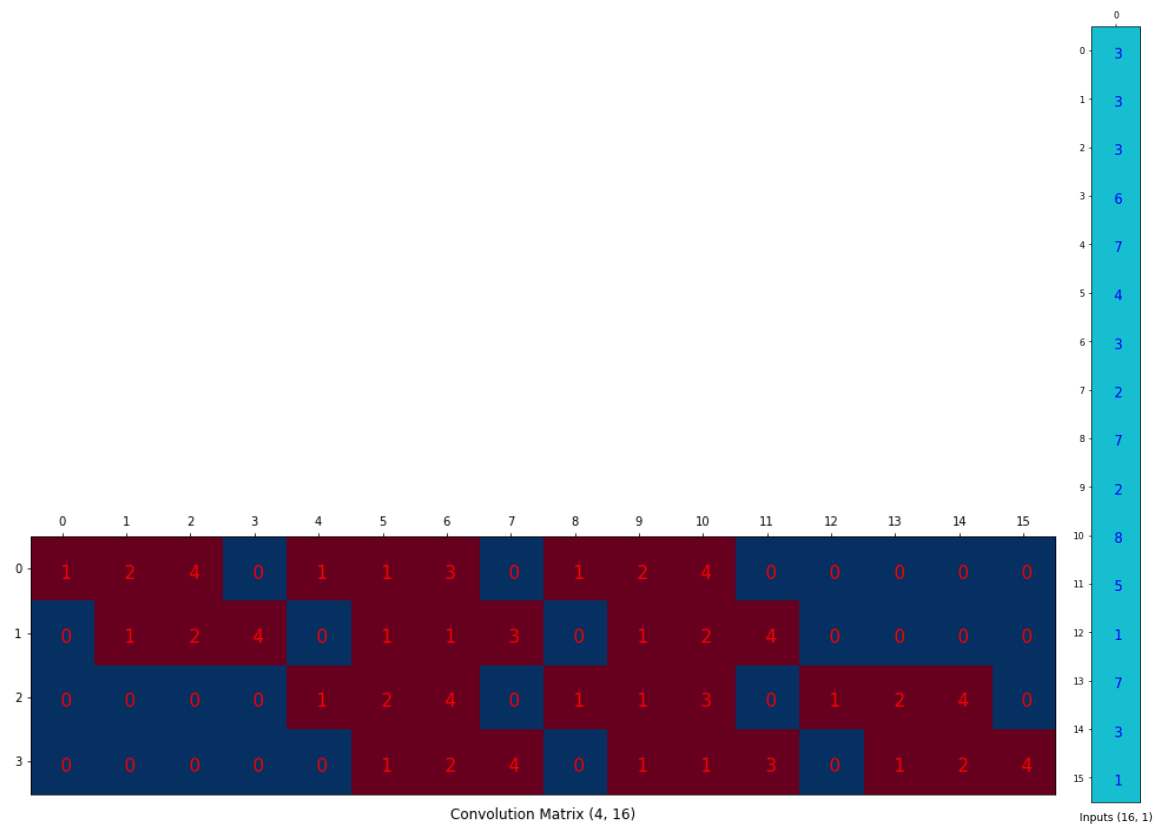
And our input matrix is:

	0	1	2	3
0	3	3	3	6
1	7	4	3	2
2	7	2	8	5
3	1	7	3	1

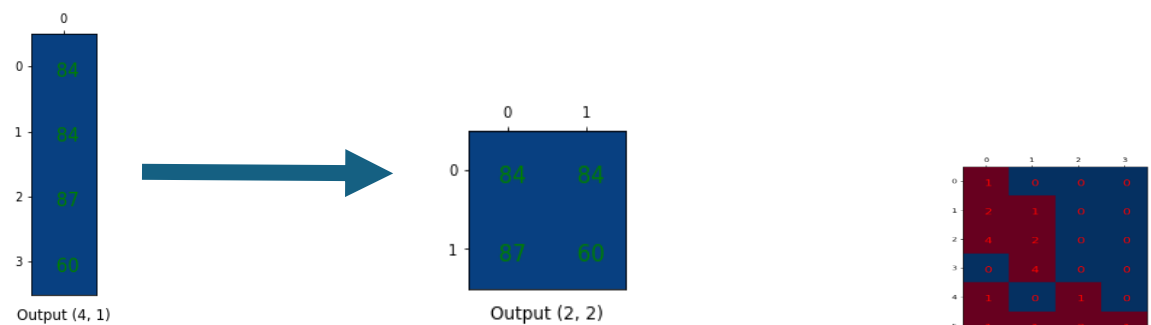
Inputs (4, 4)

Performing a convolution is equivalent to transforming the kernel and the input matrix into:





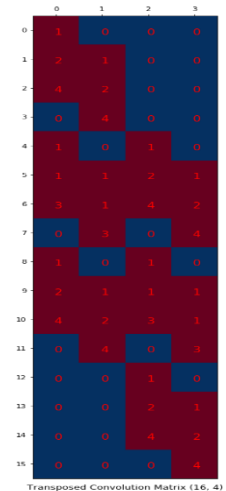
We can now perform the matrix multiplication between the matrix and the kernel and rearrange the terms in order to obtain the result:



As the name of this operator says, transposed convolution requires the transposed of the above convolution matrix:

In the context of neural networks, during the forward pass, a convolutional layer applies a set of filters to the input data to produce feature maps. During the backward pass (back-propagation), the gradients are calculated and used to update the weights of the convolutional layer.

It has been shown that the transposed convolution operation is equivalent to a backward pass of the convolutional layer with the same weights as in the forward pass. This



means that using transposed convolutions with the same weights as in the forward pass will produce the same result as the backward pass of the original convolutional layer.

### 3.8.2) Nearest neighbor upsampling

This was the procedure used to upsample the matrix in my implementation and it is the most basic one.

Suppose we have a  $n \times n$  matrix and we want to get a  $(m \cdot n) \times (m \cdot n)$  matrix, with  $m \in \mathbb{N}$ .

The basic approach is to generate for each element a block of  $m \times m$  with that element in the newly created matrix as shown bellow.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 & 2 \\ 3 & 3 & 3 & 4 & 4 & 4 \\ 3 & 3 & 3 & 4 & 4 & 4 \\ 3 & 3 & 3 & 4 & 4 & 4 \end{bmatrix}$$

The main downside to this procedure is that the network does not have any additional learnable parameters in this upsampling process, but on the other hand, it is simpler and quicker than the transposed convolution.

## 4) YOLOv3 architecture

### 4.1) Backbone architecture

In the context of Convolutional Neural Networks, the term backbone refers to the core architecture that is responsible for extracting features from images. It typically consists of a series of of convolutional and pooling layers that learn hierarchical representations of the input terms. To put it simple, the backbone architecture is responsible for learning which are the key features (horizontal, vertical, diagonal lines, as well as certain patterns or textures that can be useful in the classification problem) that need to be extracted in order to get an accurate classification of the input.

Compared to the previous YOLO architecture, YOLOv3 has a better feature extractor called Darknet-53 that has shorted connection, residual blocks as well as Leaky ReLU activation function. The number 53 indicates the number of layers in the classification network.

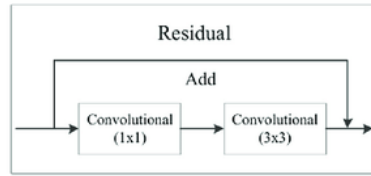


Figure 9: Residual block [3]

	Type	Filters	Size	Output
	Convolutional	32	$3 \times 3$	$256 \times 256$
	Convolutional	64	$3 \times 3 / 2$	$128 \times 128$
1x	Convolutional	32	$1 \times 1$	
	Convolutional	64	$3 \times 3$	
	Residual			$128 \times 128$
	Convolutional	128	$3 \times 3 / 2$	$64 \times 64$
2x	Convolutional	64	$1 \times 1$	
	Convolutional	128	$3 \times 3$	
	Residual			$64 \times 64$
	Convolutional	256	$3 \times 3 / 2$	$32 \times 32$
8x	Convolutional	128	$1 \times 1$	
	Convolutional	256	$3 \times 3$	
	Residual			$32 \times 32$
	Convolutional	512	$3 \times 3 / 2$	$16 \times 16$
8x	Convolutional	256	$1 \times 1$	
	Convolutional	512	$3 \times 3$	
	Residual			$16 \times 16$
	Convolutional	1024	$3 \times 3 / 2$	$8 \times 8$
4x	Convolutional	512	$1 \times 1$	
	Convolutional	1024	$3 \times 3$	
	Residual			$8 \times 8$
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 10: YOLOv3 backbone (Darknet-53) [2]

As we can see from Fig. 9, Darknet-53 has residual blocks that look like short connections in the network. In the next sections we will detail their importance and use in the entire architecture.

Each convolutional layer is a combination of a 2D convolutional layer alongside batch normalization layer and leaky ReLU activation function, as shown in Fig.11.

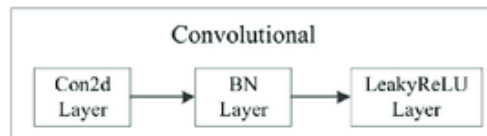


Figure 11: Convolutional layer structure [3]

Now that we have presented the overall structure of the backbone, it is worth mentioning that in order to do the object detection we need additional layers.

## 4.2) YOLOv3 architecture

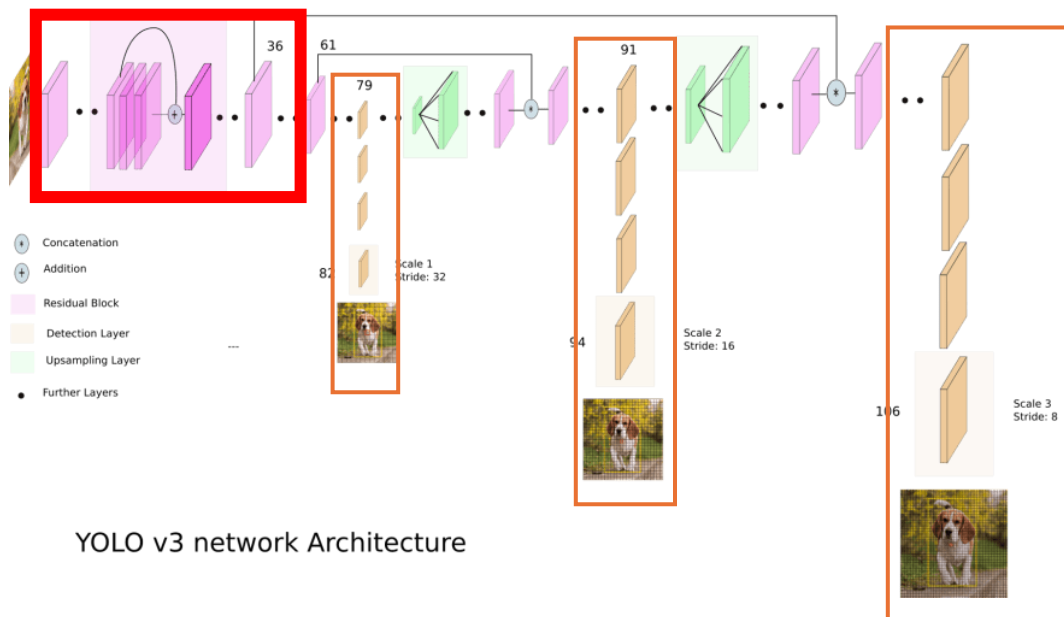


Figure 12: Full architecture of YOLOv3

The first 53 (red square) layers can be considered backbone layers. Then we have convolutional layers and upscaling layers (represented in Fig.12 with green). As you can see, there are no fully connected layers, everything being only convolution layer, even the three different scale predictions (highlighted with orange boxes).

YOLOv3 structure doesn't contain any pooling layers, so the question is "how can we get smaller scale predictions without any pooling layers?". Usually pooling layers are used to downsample the image by keeping the "most dominant" features as shown in Fig.13.

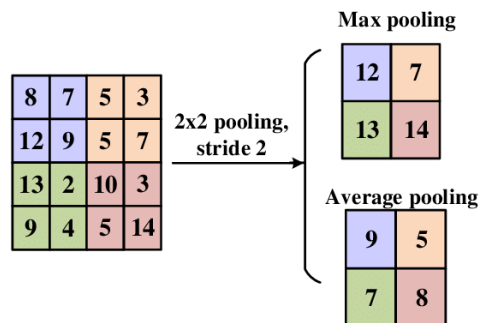


Figure 13: Types of pooling layers [4]

So, instead of using the pooling layers, they use convolutional layers with stride 2, this preventing the loss of information, unlike the pooling layer.

Taking the example of max pooling, we take the biggest element inside a grid, because we can consider that it preserves the key features inside that grid. In fact, this procedure has a loss of information. If we use a convolution layer with stride 2, as shown in Fig.14, we can still learn

features and also perform the downsampling without loss of information. Shortly, by doing this, we can make our network learn how to extract key features without any loss of information.

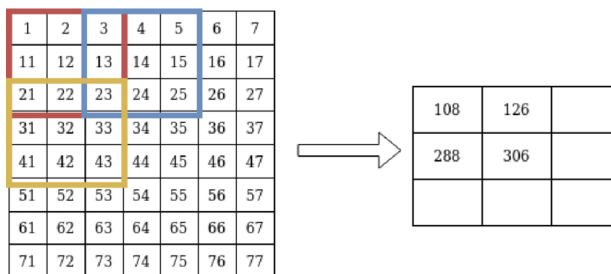


Figure 14: Convolution of with 3x3 kernel and stride 2 [5]

Because of the lack of extra info in the YOLOv3 paper or any related work I presumed that YOLOv3 uses transposed convolutions (also known as deconvolutions or fractionally strided convolutions) or nearest neighbor approach to upsample the concatenated feature maps. This operation increases spatial resolutions of the feature maps.

### 4.3) Grid cells and Anchor boxes

Similar to the previous versions of YOLO, v3 follows the same grid cell approach which divides the image into smaller grids, each grid being responsible for detecting a particular object.

Anchor boxes, Fig.15 purpose is to predict and refine bounding box coordinates and classify objects within those boxes.

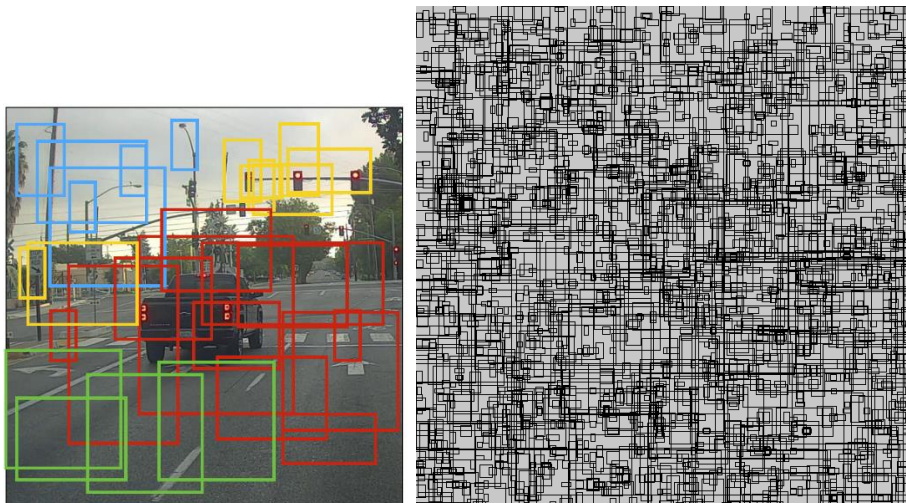
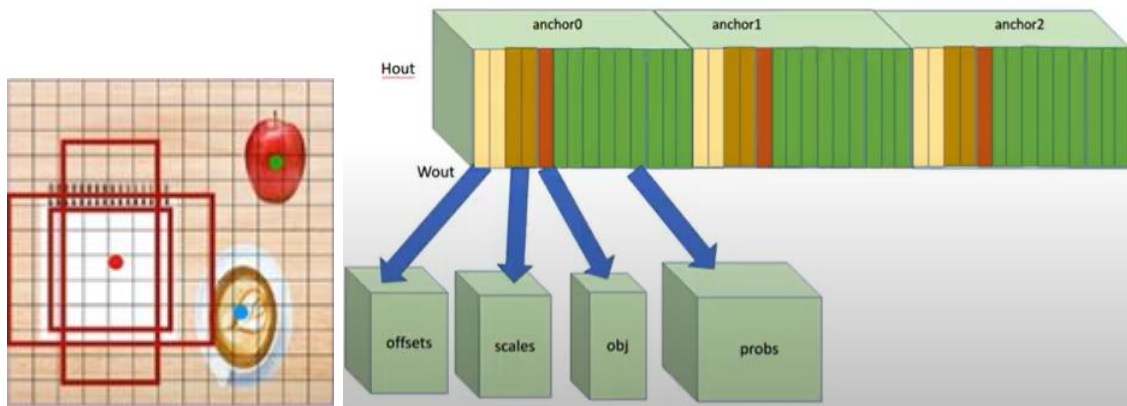


Figure 15: Illustrative example of anchor boxes placed on image

For bounding box prediction, the goal is to identify and locate objects within an image and draw bounding boxes around them. YOLO divides an input image into multiple grid cells and predicts bounding boxes and class probabilities for each grid cells. Each cell in the grid is associated with multiple anchor boxes (in this case, three anchors). The anchor box whose dimensions are most similar to the ground-truth bounding box for an object is responsible for predicting that object.

Let's take a look at the example shown in the left picture of Fig.16. Each grid cell has three sets of predictions corresponding to each anchor box. Each anchor box predicts: the offsets which are xy offsets with respect to the real bounding box, scales which are width and height of the anchor box, the objectness score (which is a measure of how likely it is that the anchor box contains an object of interest-will be detailed in the section about YOLOv3's loss function) and class probabilities (which tell us how likely it is that the object inside that anchor box belongs to a class-in the case of my project, I trained the network on Pascal VOC which has a total of 20 classes so we have 20 class probabilities). This means that (considering training the network for 20 classes) each anchor will return 25 values. Considering that we have three anchors we get a total of  $25 \times 3 = 75$  predictions for each grid cell.



*Figure 16: Anchor box prediction*

Now that we have presented how the predictions are made, we can go further and present how is the prediction made at three different scales.

#### 4.4) Multi-scale prediction and Skip connections

As shown in Fig.12, there are made three predictions at three different levels. By the time of the first prediction (highlighted with the first orange rectangle) the image had been downsampled by 32. This means that for a  $416 \times 416$  initial image, the first prediction is made for a  $13 \times 13$  feature vector size. After the first prediction, the image is upsampled till we obtain a  $26 \times 26$  feature vector and finally, the last prediction is made for a  $52 \times 52$  scale vector. In other words, to detect different objects at different scales, we predict the objects at different scales. The first prediction done on a  $13 \times 13$  feature vector will help detecting bigger objects, while the last prediction which is done on a  $52 \times 52$  feature vector will help finding smaller objects. The intermediate  $26 \times 26$  feature vector is useful for predicting medium sized objects.

It is important to mention that for each particular scale, we still have three anchor boxes predicting offsets, scales, objectness score and class probabilities. To make an idea of how many boxes YOLOv3 predicts we can make the following calculations: we have  $13 \times 13 = 169$  plus  $26 \times 26 = 676$  plus  $52 \times 52 = 2704$  equals 3549 grid cells. For each grid cell we have 3 anchor boxes

responsible for predictions, so we have a total of 10647 predicted boxes. So YOLOv3 predicts more than ten times the number of boxes predicted by YOLOv2 (only 845).

In YOLOv3 (You Only Look Once, version 3), skip connections are employed to connect earlier layers with later layers in the network. These skip connections, also known as shortcut connections, allow the information from earlier layers to bypass some of the subsequent layers and be directly fed into later layers. This helps in preserving fine-grained details and gradients during the training process.

## 4.5) Residual blocks

Residual blocks were introduced to facilitate the training of very deep networks. The main idea behind residual blocks is the use of skip connections, also known as shortcut connections, to allow the network to learn the identity function. This helps in preserving important information during the training process.

In the context of YOLOv3, the architecture consists of many residual blocks stacked on top of each other. Each residual block typically has the following components:

- Convolutional Layers:

YOLOv3 residual blocks usually begin with a series of convolutional layers that perform feature extraction. These layers help in capturing complex patterns and high-level features from the input.

- Skip Connection:

The key element of a residual block is the skip connection, which directly connects the input of the block to its output. This skip connection allows the gradient to flow easily through the network during backpropagation, mitigating the vanishing gradient problem.

- Batch Normalization and Activation:

Batch normalization and activation functions (e.g., Leaky ReLU) are applied to the output of convolutional layers to improve training stability and introduce non-linearity.

- More Convolutional Layers:

Additional convolutional layers follow the skip connection. These layers refine the features and further contribute to the representation learning process.

- Final Output:

The final output of the residual block is the sum of the input and the output of the additional convolutional layers. Mathematically, this can be represented as:  $\text{output} = \text{input} + F(\text{input})$ , where  $F(\text{input})$  is the output of the convolutional layers.

The use of residual blocks in YOLOv3 enables the training of a deep neural network, allowing it to learn intricate features and patterns in images. The skip connections ensure that the gradient information can easily propagate through the network, making it easier to train deep architectures effectively.

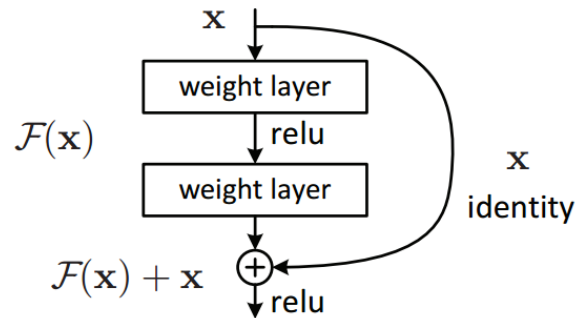


Figure 17: Schematic representation of a Residual block

## 4.6) Bounding box prediction

As we mentioned before, apart from the class probabilities, there are predicted five more values which are:  $t_x, t_y, t_w, t_h$  and  $t_o$  that correspond to bounding box offset coordinates  $x$  and  $y$ , width and height of the predicted bounding box and objectness score which tells us the probability that a bounding box contains an object of interest.

One problem with  $t_x, t_y, t_w, t_h$  is that they are unbounded.

Let's talk about the first pair which is  $(t_x, t_y)$ . We need to calculate  $(B_x, B_y)$ , both of them being normalized values (this implies that their value should belong to  $(0;1)$ ). Shifting from  $(-\infty; \infty)$  to  $(0; 1)$  can be done by using the sigmoid function  $\sigma(x) = \frac{1}{1+e^{-x}}$ .

We can write:  $(B_x, B_y) = (\sigma(t_x) + c_x; \sigma(t_y) + c_y)$ , where  $c_x$  and  $c_y$  are the offsets from the top left of the image, respectively from the top of the image, as shown in Fig.18.



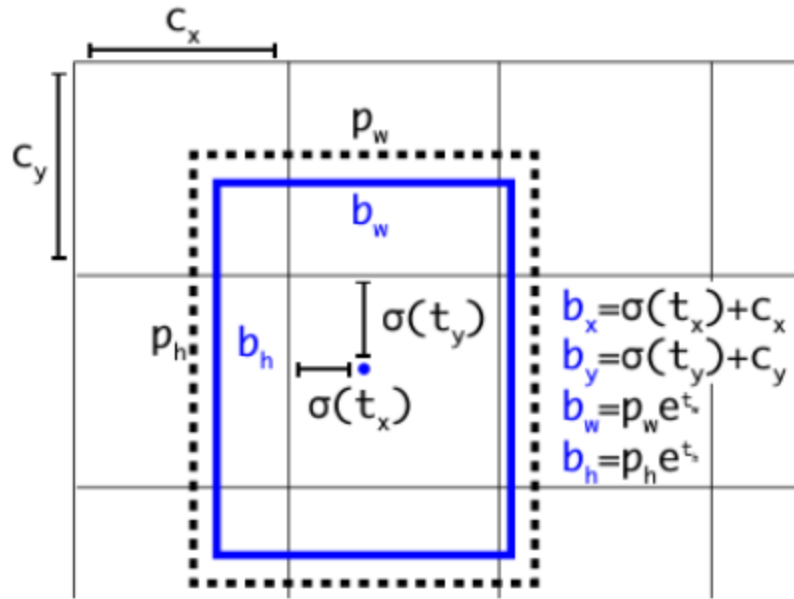


Figure 18: Bounding box prediction [2]

The xy predictions are made directly with respect to the image and the grid cell. They do not consider the anchor box. This is why they call it direct location prediction.

On the other hand, width and height depend on the anchor boxes. Now, how do we compute the actual width and height?

First of all,  $t_w$  and  $t_h$  have values between  $(-\infty; \infty)$  and we should somehow restrict those values between  $(0; \infty)$  (not  $(0; 1)$ , because those values can go beyond the grid level). We can do this by applying the exponential  $e^x$  to those two predicted values. Let's not forget that width and height of the bounding boxes depend on the width and height of the anchor boxes (let's call them  $A_w$  and  $A_h$ ).

The width and height of the bounding boxes are:  $e^{t_w} * A_w$  and  $e^{t_h} * A_h$ .

We have left only one parameter and that is objectness score  $t_o$  which tells how confident is a box that it contains an object of interest. It is computed by applying the sigmoid function to a linear transformation (similar to a classification problem where we compute the probability of the predicted value of being or not in a class):  $\sigma(w \cdot x + bias)$

## 4.7) Class prediction

For class prediction, in YOLOV3, is used multi-label classification.

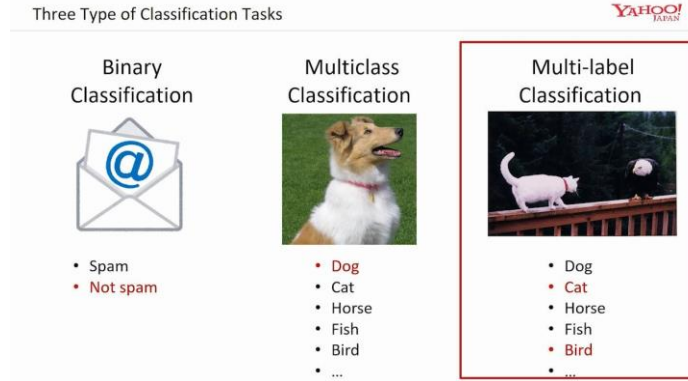


Figure 19: Multi-labeled classification compared to other classifications

Multi-labeled classification implies predicting multiple classes from one single image. In the context of YOLOv3, this means that for every single bounding box we are predicting multiple labels.

If we want to perform a multiclass classification, we use the softmax function which will tell us how likely a certain object is to be in a certain class. If we want to make a multi-labeled classification, we will use the sigmoid function which will tell us how likely that each object of interest in the image is in a certain class. So, for each prediction node, we use the sigmoid function to perform this multi-labeled classification.

To anticipate the next section's topic about YOLOv3 loss function, it's worth mentioning that in case of using the sigmoid function we need to use binary cross entropy function for loss calculation, while for the softmax function we need to use categorical cross entropy function.

## 4.7) YOLOv3 loss function

YOLOv3 loss function has the following form:

$$\begin{aligned}
 & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
 & + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[ (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\
 & + \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{\text{obj}} \left[ \hat{C}_i \log(C_i) + (1 - \hat{C}_i) \log(1 - C_i) \right] + \\
 & \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{\text{noobj}} \left[ \hat{C}_i \log(C_i) + (1 - \hat{C}_i) \log(1 - C_i) \right] + \\
 & \sum_{i=0}^{S^2} I_{ij}^{\text{obj}} \sum_{c \in \text{classes}} \left[ \hat{p}_i(c) \log(p_i(c)) + (1 - \hat{p}_i(c)) \log(1 - p_i(c)) \right]
 \end{aligned}$$

Figure 20: YOLOv3 loss function

Let's now break it in small sub-components and analyze each one of them.

#### 4.7.1) Regression loss

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\ + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]$$

Figure 21: Localization loss

This loss is also called Localization loss and measures the error in predicting the bounding box coordinates  $(x, y)$ , alongside their width and height  $(w, h)$ . YOLOv3 uses the mean squared error (which in fact computes the Euclidean distance between those points) to calculate the localization error. The localization error is only applied to the bounding box responsible for detecting an object (for example, the object with the highest intersection over union with the ground truth box).

The notations in Fig. 21 have the following meanings:

- $S^2$  is the number of grid cells;
- $B$  is the number of predicted bounding boxes in each grid cell;
- $\lambda_{coord}$  is an importance coefficient that adjusts the importance of localization loss function;
- $1_{ij}^{obj}$  is an indicator function that equals 1 if object  $j$  is present in a grid cell  $i$ , otherwise, it's 0;
- $x_i, y_i, w_i, h_i$  are the ground truth values for the bounding box;
- $\hat{x}_i, \hat{y}_i, \hat{w}_i, \hat{h}_i$  are the predicted bounding box coordinates and dimensions;

#### 4.7.2) Confidence loss

$$\sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{obj} \left[ \hat{C}_i \log(C_i) + (1 - \hat{C}_i) \log(1 - C_i) \right] + \\ \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{noobj} \left[ \hat{C}_i \log(C_i) + (1 - \hat{C}_i) \log(1 - C_i) \right] +$$

Figure 22: Confidence loss

This loss function (because of convergence reasons), does not use MSE loss function, but instead, it used binary cross entropy loss function where:

- $\hat{C}_i$  is the predicted confidence score for box  $j$  in grid cell  $i$  (it denotes the confidence that an object is present or not in a grid cell);

- $C_i$  is the true confidence score;
- $\lambda_{coord}$  is a coefficient that down-weights the confidence loss for grid cells without objects. (because we want to give more importance for the grid cells that contain objects of interest)

This loss is a loss function which takes into account how confident is each box that it contains or not an object of interest, without taking into account the class probabilities of the possible object of interest in that box.

#### 4.7.3) Class loss

$$\sum_{i=0}^{S^z} I_{ij}^{obj} \sum_{c \in classes} \left[ \hat{p}_i(c) \log(p_i(c)) + (1 - \hat{p}_i(c)) \log(1 - p_i(c)) \right]$$

This loss function uses cross entropy loss (more specific, categorical cross entropy loss) and it measures the error in predicting the class probabilities for each bounding box.

- $\hat{p}_i$  are the predicted class probabilities for each box in grid cell  $i$
- $p_i$  are the true class probabilities for each box in grid cell  $i$

## 5) Implementation

### 5.1) Used packages

- torch = framework mainly used for tensor computation (similar to numpy) and supports GPU acceleration;
- torch.nn = framework that supports building neural network models, alongside training, evaluating and testing neural networks;
- torch.optim = package that offers support for different optimization algorithms used in training deep neural networks;
- PIL = image processing library, mainly used in this project for loading, storing, or reshaping images;
- albumentations = package used to support working with deep convolutional neural networks and offers capabilities in augmenting images (from the initial data we generate a new set of transformed images in order to increase the diversity)

### 5.2) Main implemented blocks

- CNN block

```
1  # (7)
2  from torch import nn
3
4
5  # Defining CNN Block
17 usages
6  class CNNBlock(nn.Module):
7      def __init__(self, in_channels, out_channels, use_batch_norm=True, **kwargs):
8          super().__init__()
9          self.conv = nn.Conv2d(in_channels, out_channels, bias=not use_batch_norm, **kwargs)
10         self.bn = nn.BatchNorm2d(out_channels)
11         self.activation = nn.LeakyReLU(0.1)
12         self.use_batch_norm = use_batch_norm
13
14     def forward(self, x):
15         # Applying convolution
16         x = self.conv(x)
17         # Applying BatchNorm and activation if needed
18         if self.use_batch_norm:
19             x = self.bn(x)
20             return self.activation(x)
21         else:
22             return x
```

- Residual Block

```

7  class ResidualBlock(nn.Module):
8      def __init__(self, channels, use_residual=True, num_repeats=1):
9          super().__init__()
10
11         # Defining all the layers in a list and adding them based on number of
12         # repeats mentioned in the design
13         res_layers = []
14         for _ in range(num_repeats):
15             res_layers += [
16                 nn.Sequential(
17                     nn.Conv2d(channels, channels // 2, kernel_size=1),
18                     nn.BatchNorm2d(channels // 2),
19                     nn.LeakyReLU(0.1),
20                     nn.Conv2d(channels // 2, channels, kernel_size=3, padding=1),
21                     nn.BatchNorm2d(channels),
22                     nn.LeakyReLU(0.1)
23                 )
24             ]
25         self.layers = nn.ModuleList(res_layers)
26         self.use_residual = use_residual
27         self.num_repeats = num_repeats
28
29         # Defining forward pass
30         def forward(self, x):
31             for layer in self.layers:
32                 residual = x
33                 x = layer(x)
34                 if self.use_residual:
35                     x = x + residual
36             return x

```

- Scale prediction block

```

7  class ScalePrediction(nn.Module):
8      def __init__(self, in_channels, num_classes):
9          super().__init__()
10         # Defining the layers in the network
11         self.pred = nn.Sequential(
12             nn.Conv2d(in_channels, 2 * in_channels, kernel_size=3, padding=1),
13             nn.BatchNorm2d(2 * in_channels),
14             nn.LeakyReLU(0.1),
15             nn.Conv2d(2 * in_channels, (num_classes + 5) * 3, kernel_size=1),
16         )
17         self.num_classes = num_classes
18
19         # Defining the forward pass and reshaping the output to the desired output
20         # format: (batch_size, 3, grid_size, grid_size, num_classes + 5)
21         def forward(self, x):
22             output = self.pred(x)
23             output = output.view(x.size(0), 3, self.num_classes + 5, x.size(2), x.size(3))
24             output = output.permute(0, 1, 3, 4, 2)
25             return output

```

- YOLOv3

```
11 class YOLOv3(nn.Module):
12     def __init__(self, in_channels=3, num_classes=20):
13         super().__init__()
14         self.num_classes = num_classes
15         self.in_channels = in_channels
16
17         # Layers list for YOLOv3
18         self.layers = nn.ModuleList([
19             CNNBlock(in_channels, out_channels=32, kernel_size=3, stride=1, padding=1),
20             CNNBlock(in_channels=32, out_channels=64, kernel_size=3, stride=2, padding=1),
21             ResidualBlock(channels=64, num_repeats=1),
22             CNNBlock(in_channels=64, out_channels=128, kernel_size=3, stride=2, padding=1),
23             ResidualBlock(channels=128, num_repeats=2),
24             CNNBlock(in_channels=128, out_channels=256, kernel_size=3, stride=2, padding=1),
25             ResidualBlock(channels=256, num_repeats=8),
26             CNNBlock(in_channels=256, out_channels=512, kernel_size=3, stride=2, padding=1),
27             ResidualBlock(channels=512, num_repeats=8),
28             CNNBlock(in_channels=512, out_channels=1024, kernel_size=3, stride=2, padding=1),
29             ResidualBlock(channels=1024, num_repeats=4),
30             CNNBlock(in_channels=1024, out_channels=512, kernel_size=1, stride=1, padding=0),
31             CNNBlock(in_channels=512, out_channels=1024, kernel_size=3, stride=1, padding=1),
32             ResidualBlock(channels=1024, use_residual=False, num_repeats=1),
33             CNNBlock(in_channels=1024, out_channels=512, kernel_size=1, stride=1, padding=0),
34             ScalePrediction(in_channels=512, num_classes=num_classes),
35             CNNBlock(in_channels=512, out_channels=256, kernel_size=1, stride=1, padding=0),
36             nn.Upsample(scale_factor=2),
37             CNNBlock(in_channels=768, out_channels=256, kernel_size=1, stride=1, padding=0),
38             CNNBlock(in_channels=256, out_channels=512, kernel_size=3, stride=1, padding=1),
39             ResidualBlock(channels=512, use_residual=False, num_repeats=1),
```

```

38         CNNBlock( in_channels: 256, out_channels: 512, kernel_size=3, stride=1, padding=1),
39         ResidualBlock( channels: 512, use_residual=False, num_repeats=1),
40         CNNBlock( in_channels: 512, out_channels: 256, kernel_size=1, stride=1, padding=0),
41         ScalePrediction( in_channels: 256, num_classes=num_classes),
42         CNNBlock( in_channels: 256, out_channels: 128, kernel_size=1, stride=1, padding=0),
43         nn.Upsample(scale_factor=2),
44         CNNBlock( in_channels: 384, out_channels: 128, kernel_size=1, stride=1, padding=0),
45         CNNBlock( in_channels: 128, out_channels: 256, kernel_size=3, stride=1, padding=1),
46         ResidualBlock( channels: 256, use_residual=False, num_repeats=1),
47         CNNBlock( in_channels: 256, out_channels: 128, kernel_size=1, stride=1, padding=0),
48         ScalePrediction( in_channels: 128, num_classes=num_classes)
49     ])
50
51     # Forward pass for YOLOv3 with route connections and scale predictions
52     def forward(self, x):
53         outputs = []
54         route_connections = []
55
56         for layer in self.layers:
57             if isinstance(layer, ScalePrediction):
58                 outputs.append(layer(x))
59                 continue
60             x = layer(x)
61
62             if isinstance(layer, ResidualBlock) and layer.num_repeats == 8:
63                 route_connections.append(x)
64
65             elif isinstance(layer, nn.Upsample):
66                 x = torch.cat( tensors: [x, route_connections[-1]], dim=1)
67                 route_connections.pop()
68         return outputs

```



- Defining YOLO loss

```

11 class YOLOLoss(nn.Module):
12     def __init__(self):
13         super().__init__()
14         self.mse = nn.MSELoss()
15         self.bce = nn.BCEWithLogitsLoss()
16         self.cross_entropy = nn.CrossEntropyLoss()
17         self.sigmoid = nn.Sigmoid()
18
19     def forward(self, pred, target, anchors):
20         # Identifying which cells in target have objects
21         # and which have no objects
22         obj = target[..., 0] == 1
23         no_obj = target[..., 0] == 0
24
25         # Calculating No object loss
26         no_object_loss = self.bce(
27             (pred[..., 0:1][no_obj]), (target[..., 0:1][no_obj]),
28         )
29
30         # Reshaping anchors to match predictions
31         anchors = anchors.reshape(1, 3, 1, 1, 2)
32         # Box prediction confidence
33         box_preds = torch.cat(
34             tensors=[self.sigmoid(pred[..., 1:3]),
35                     torch.exp(pred[..., 3:5]) * anchors
36                     ], dim=-1)
37         # Calculating intersection over union for prediction and target
38         ious = iou(box_preds[obj], target[..., 1:5][obj]).detach()
39         # Calculating Object loss
40         object_loss = self.mse(self.sigmoid(pred[..., 0:1][obj]),
41                                ious * target[..., 0:1][obj])
42
43         # Predicted box coordinates
44         pred[..., 1:3] = self.sigmoid(pred[..., 1:3])
45         # Target box coordinates
46         target[..., 3:5] = torch.log(1e-6 + target[..., 3:5] / anchors)
47         # Calculating box coordinate loss
48         box_loss = self.mse(pred[..., 1:5][obj],
49                             target[..., 1:5][obj])
50
51         # Calculating class loss
52         class_loss = self.cross_entropy((pred[..., 5:][obj]),
53                                         target[..., 5:][obj].long())
54
55         # Total loss
56         return (box_loss + object_loss + no_object_loss + class_loss)

```

## 6) Train and infrastructure

For training the network, Pascal VOC was used. It consists of 16550 training samples and 5000 testing data images, from 20 different classes.

For training I tried multiple platforms including Google Colab (not a good idea if you don't pay subscription), Kaggle Kernel (with a modified version of torch, 2.1.2) and a remote pc from utcn (found that training was not working due to the used python version – this is my theory).

After tens of hours of total training, I managed to create a batch session on Kaggle with the right hyperparameters for the network. The batch ran for 9 hours (maximum time admitted for a batch training on Kaggle GPU). The network was trained for 122 epochs (I intended tot train it for 200, but it took too much time), for a batch size of 32 images.

In Fig. 23 is shown the evolution of loss function during the training process. It is worth noting that after a certain number of epochs, the loss function reaches a steady state thus, the network does not improve as much as in the beginning of the training process.

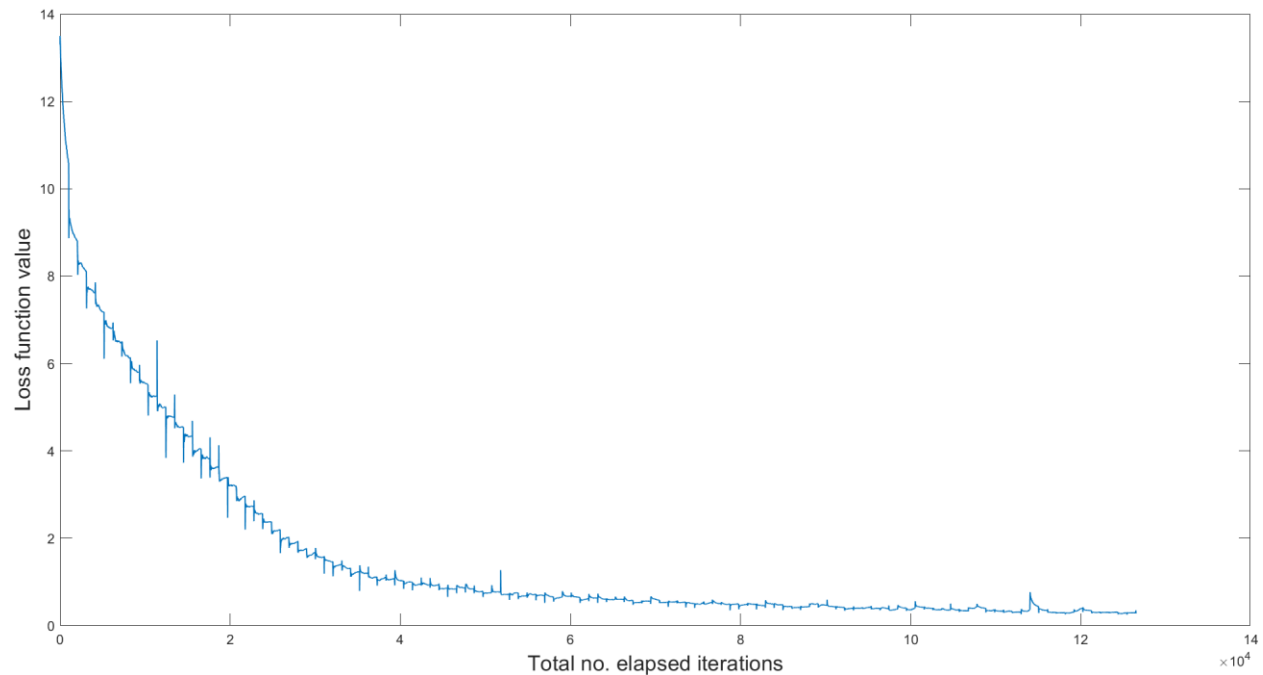
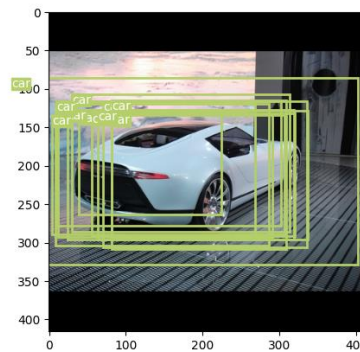
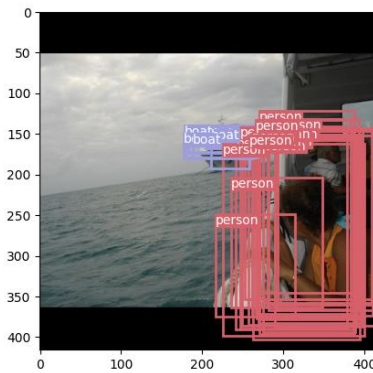
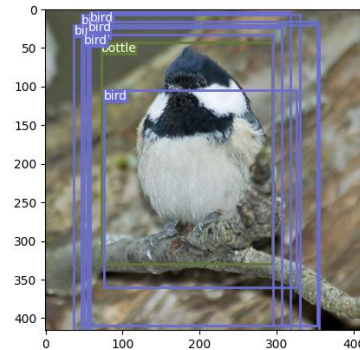
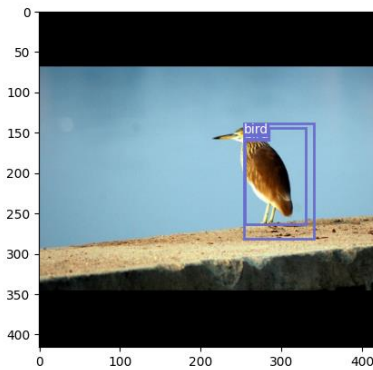
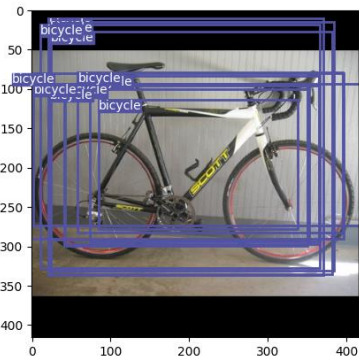
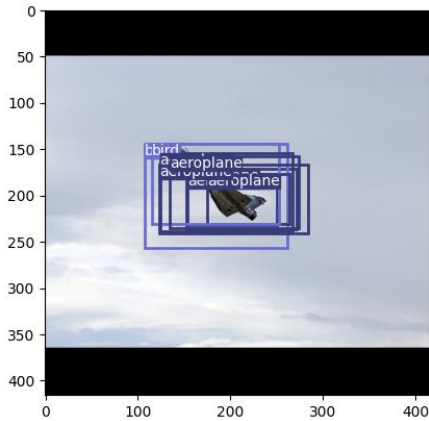
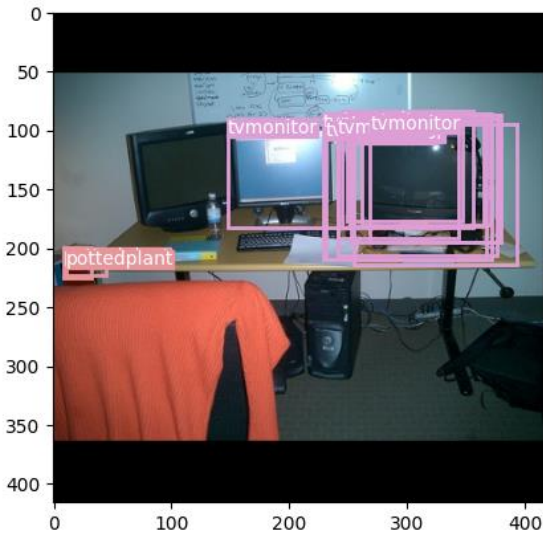
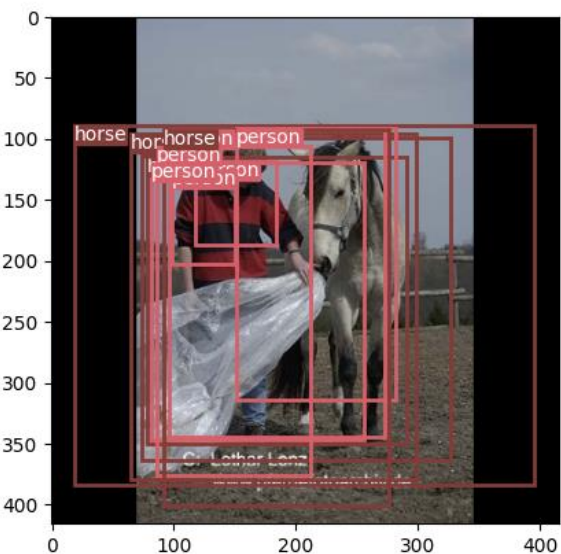
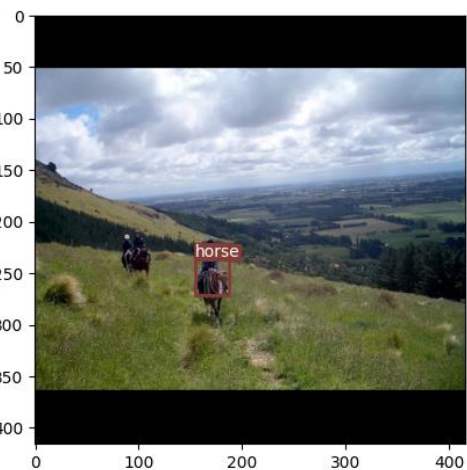
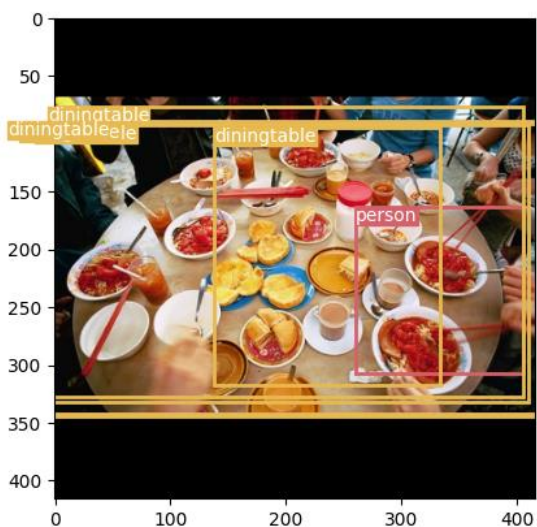
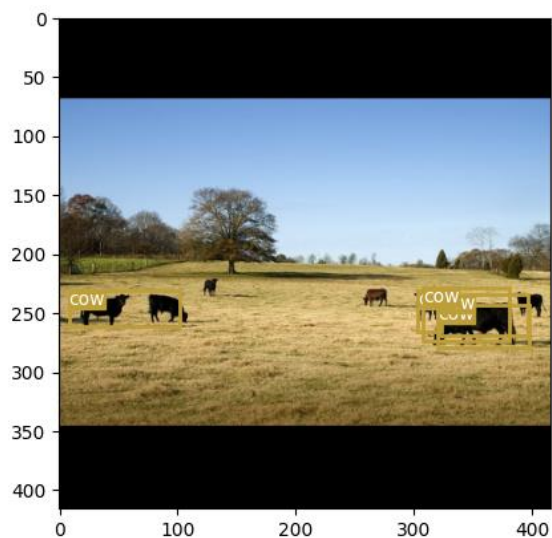
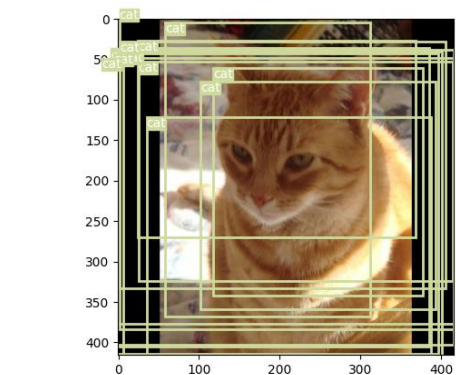


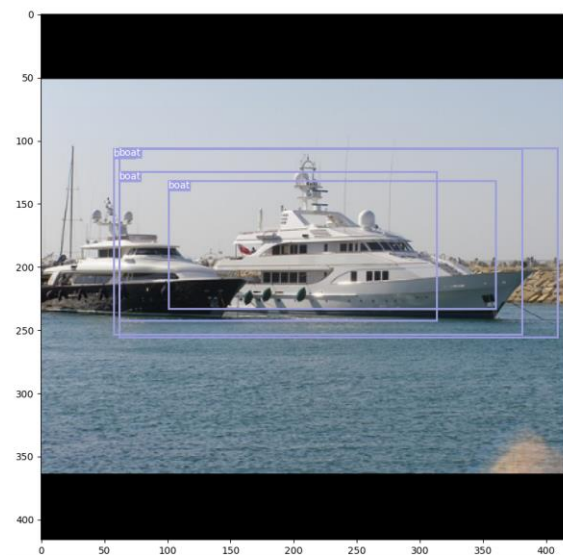
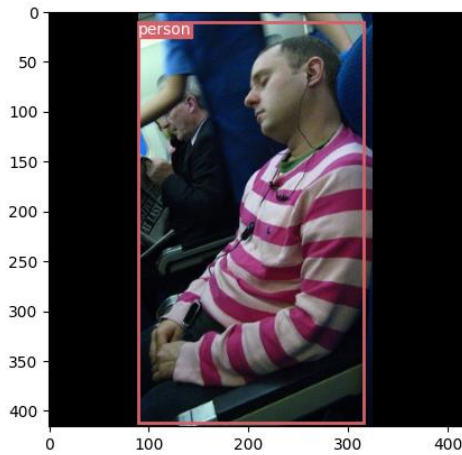
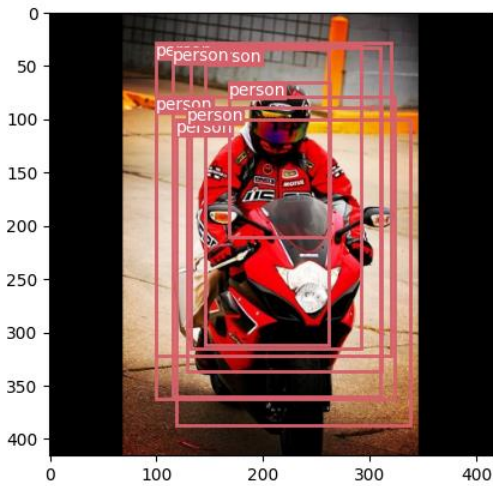
Figure 23: Loss function progress

## 7) Results

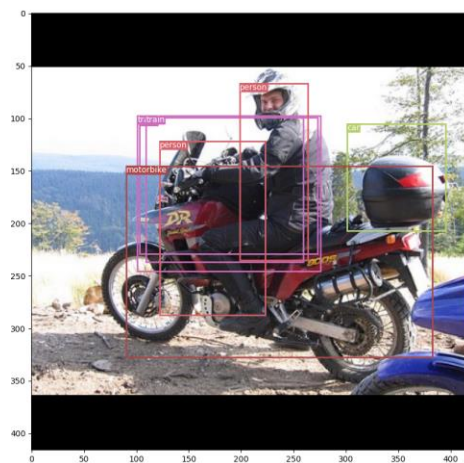
Bellow we have a few results from the testing process (sometimes, only one object was classified because, in certain cases, I set the confidence threshold to the object with the highest confidence from all the predictions).







Of course, there were some failed/weird predictions like (but it can happen to any person to mistake a sheep for a dog):





## 8) Bonus: Lagrange multipliers, SVD visualization for images and formulation for optimization of learning rate

### 8.1) SVD visualization for RGB and grayscale image

Out of curiosity I tried applying SVD (singular value decomposition) on a RGB and a grayscale image in order to observe how the image behaves when trying to approximate a matrix of rank N with it's best approximation of rank K, with  $K < N$ .

Bellow is the code and a few results.

```
1  #-----computing SVD for simple matrix-----
2  from numpy import *
3  from scipy.linalg import svd
4  ##define matrix A mxn, U mxm, S mxn, VT nxn
5  A = array([[1, 2], [3, 4], [5, 6]])
6  #print(A)
7  ##SVD
8  U,S,VT=svd(A)
9  #print(U)
10 #print(S)
11 #print(VT)
12
13 ##Reconstructing A
14 ##for this we need to make S a mxn "diagonal matrix": sigma has a diagonal matrix of nxn elements
15 sigma=zeros((A.shape[0],A.shape[1]))
16 sigma[:A.shape[1],:A.shape[1]]=diag(S)
17 A_reconstruct=U.dot(sigma.dot(VT))
18
19 #print(A_reconstruct)
20
21 #-----showing matrix as image-----
22 from matplotlib import pyplot as plt
23 """
24 plt.figure(1)
25 plt.subplot(121)
26 plt.imshow(A)
27 plt.subplot(122)
28 plt.imshow(A_reconstruct)
29 plt.show()
30 """
31
```

```

29 plt.show()
30 ""
31
32 ✓ #coloanele lui U sunt left-singular vectors of A
33 #coloanele lui V (nu VT= V transpose) sunt right singular vectors of A
34
35 #-----svd for grayscale image-----
36 from PIL import Image
37 img=Image.open("C:\Master\An1\sem1\Machine learning\RGB image.jfif")
38
39 img_gray=img.convert('L')
40
41 #convert PIL image to matrix
42 img=asarray(img)
43 img_gray=asarray(img_gray)
44 #print(img_gray)
45
46 plt.figure(1)
47 plt.subplot(411)
48 plt.imshow(img)
49 plt.subplot(412)
50 plt.imshow(img_gray,cmap=plt.cm.gray)
51
52
53 U,s,VT=svd(img_gray)
54 ✓ #print(img_gray.shape)
55
56 #keep in mind  $A=U*\Sigma*V^T \sim s_1*U_1*V_1^T + \dots + s_k*U_k*V_k^T$ 
57 #where  $s_i$ = the i-st greatest element of all the singular values and
58 #and  $U_i$  and  $V_i$  the corespondin left and right singular vectors
59

```

```

59
60 k=20#number of first singular values to approximate A with a k rank matrix
61 #getting indexes for sorted singular values
62 s_sort_ind=argsort(s)[::-1]
63 s_sorted=s[s_sort_ind]
64 #getting coresponding vectors
65 U_sort=U[:,s_sort_ind]
66 V=transpose(VT)
67 V_sort=V[:,s_sort_ind]
68
69 sk=s_sorted[0:k]
70 Uk=array(U_sort[:,0:k])
71 Vk=V_sort[:,0:k]
72 VkT=transpose(Vk)
73
74 ✓ #print(array(vstack(Uk)).size)
75 #print(array(vstack(VkT)).size)
76 img_svd_k=zeros(img_gray.size)
77 img_svd_k=hstack(img_svd_k)
78
79 #check how to do this without iterations
80 ✓ for i in range(k):
81 ✓     #print(Uk[:,i].size)
82     #print(VkT[i,:].size)
83     Ui=array(Uk[:,i])
84     VkTi=array(VkT[i,:])
85     #print(img_svd_k.size)
86     img_svd_k=img_svd_k+hstack(array(sk[i]*outer(Ui,VkTi)))
87
88 img_svd_k=img_svd_k.reshape(img_gray.shape)
89 plt.subplot(*args: 4,1,3)

```



```

87
88 img_svdk=img_svdk.reshape(img_gray.shape)
89 plt.subplot(*args: 4,1,3)
90 plt.imshow(img_svdk,cmap=plt.cm.gray)
91
92
93 #-----svd for rgb image-----
94 from Apply_svd import *
95
96 k=int(floor(len(img[:, :, 0])/10))
97 R=img[:, :, 0]
98 G=img[:, :, 1]
99 B=img[:, :, 2]
100
101 R_svdk=svd_k_components(R,k)
102 G_svdk=svd_k_components(G,k)
103 B_svdk=svd_k_components(B,k)
104
105
106 img_rgb_svdk=zeros(img.shape)
107 img_rgb_svdk[:, :, 0]=R_svdk
108 img_rgb_svdk[:, :, 1]=G_svdk
109 img_rgb_svdk[:, :, 2]=B_svdk
110 print(img_rgb_svdk)
111 print(img)
112
113 plt.subplot(*args: 4,1,4)
114 plt.imshow(img_rgb_svdk.astype('uint8'))#!!!!!!!!!!!!!!
115 plt.show()

```

```

4  def svd_k_components(matrix,k=10):
5      U,s,VT=svd(matrix)
6
7      s_sort_ind=argsort(s)[::-1]
8      s_sorted=s[s_sort_ind]
9      #getting corresponding vectors
10     U_sort=U[:,s_sort_ind]
11     V=transpose(VT)
12     V_sort=V[:,s_sort_ind]
13
14     sk=s_sorted[0:k]
15     Uk=array(U_sort[:,0:k])
16     Vk=V_sort[:,0:k]
17     VkT=transpose(Vk)
18
19     #print(array(vstack(Uk)).size)
20     #print(array(vstack(VkT)).size)
21     img_svd=zeros(matrix.size)
22     img_svd=hstack(img_svd)
23
24     #check how to do this without using for
25     for i in range(k):
26         #print(Uk[:,i].size)
27         #print(VkT[i,:].size)
28         Ui=array(Uk[:,i])
29         VkTi=array(VkT[i,:])
30         #print(img_svd.size)
31         img_svd=img_svd+hstack(array(sk[i]*outer(Ui,VkTi)))
32
33     img_svd=img_svd.reshape(matrix.shape)
34     return img_svd

```

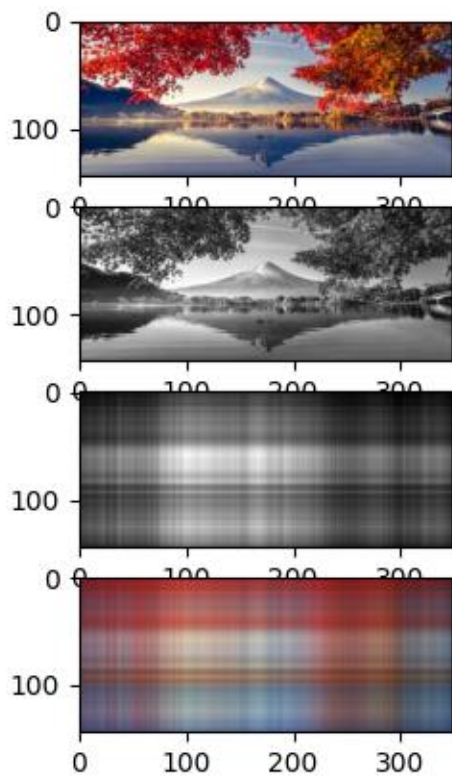


Figure 24: K=1

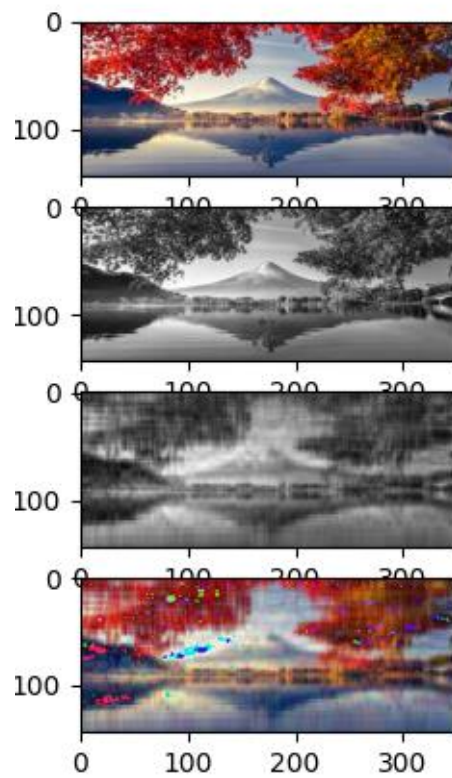


Figure 25: K=10

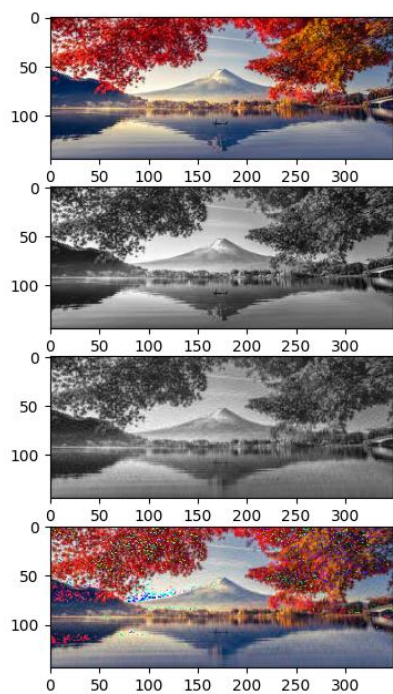


Figure 26: K=50

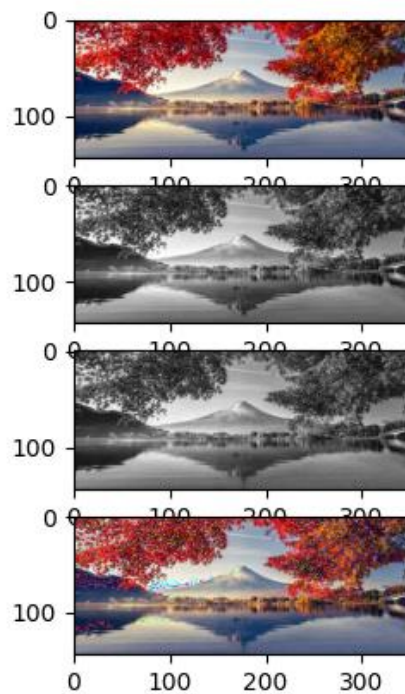


Figure 27: K=90

## 8.2) Solving optimization problem with constraints using Lagrange multipliers and Karush-Kuhn-Tucker conditions (MATLAB code)

Here is the script with a few examples of functions that need to be optimized under certain inequality/equality constraints (g is the constraint and is  $\leq 0$ ):

```
13     syms x1 x2
14     f=x1^2+x1*x2+2*x2^2+2*x2+3;
15     restrict=x1+x2-1;
16     func_var=[x1;x2];
17     [points,type,lambdas]=met_Lagrange(f,func_var,restrict);
18
19
20     syms x1 x2 real
21     f=x1^4+2*x1^2*x2+x1^2+4*x2^2+6*x2;
22     restr=x1^2+x2^2-1;
23     func_var=[x1;x2];
24     [points,type,lambdas]=met_Lagrange(f,func_var,restr);
25
26     syms x1 x2 real
27     f=x1^2*x2^2+113*x1*x2-1;
28     g=4*x1+3*x2-1;
29     func_var=[x1;x2];
30     [points,type,lambdas]=met_Lagrange(f,func_var,g);
31
```

```

32 function [points,type,lambdas]=met_Lagrange(func,func_var,restrict)
33 %func is a symbolic function
34
35 %func_var is a column vector with the necessary variables
36
37 %restrict is a column vector with restriction functions (by default equal
38 %to 0)
39
40     type=[];%gives the type of extremum points
41     n_var=length(func_var);
42
43     %declaring enough lambda params for restrictions
44     n_res=length(restrict);
45     lam=[];
46     for i=1:n_res
47         lam=[lam;sym(strcat('l',num2str(i)),'real')];
48     end
49
50     %constructing Lagrange L function
51     L=func+lam'*restrict;
52     varL=[func_var;lam];
53
54     %computing L jacobian
55     dL=jacobian(L,varL);
56
57     %computing the extremum points
58     sol=solve(dL==0,varL);
59     fn=fieldnames(sol);
60     sol_num=[];
61     for k=1:numel(fn)
62         sol_num=[sol_num,eval(sol.(fn{k}))];
63     end
64     dim=size(sol_num);
65     n_sol=dim(1);%number of solutions
66     sol_num=sol_num';%each column contains a solution tuple
67

```

```

65     n_sol=dim(1);%number of solutions
66     sol_num=sol_num';%each column contains a solution tuple
67
68     %computing Hessian for L
69     H_L=jacobian(dL,varL);
70
71     %making [z*I 0;0 0] matrix
72     syms z real
73     Z=[z*eye(n_var),zeros(n_var,n_res);zeros(n_res,n_var),zeros(n_res)];
74
75     % %preparing variable array to make the subs in H_L
76     % var=[];
77     % for i=1:length(varL)
78     %     var=[var;{varL(i)}];
79     % end
80
81     %for each solution, compute its type
82     for i=1:n_sol
83         current_sol=sol_num(:,i);
84         %computing value of hessian for the current solution
85         H_Ln=eval(subs(H_L,varL,current_sol));
86
87         %computing solutions for det(H_L-[z*I 0;0 0])=0
88         z_sol=eval(solve(det(H_Ln-Z),z));
89
90         if sum(z_sol>0)==length(z_sol)
91             type=[type;'Local minimum'];
92         elseif sum(z_sol<0)==length(z_sol);
93             type=[type;'Local maximum'];
94         elseif sum(z_sol==0)~=0
95             type=[type;'Undefined'];
96         else
97             type=[type;'Saddle'];
98         end
99     end
100     points=sol_num(1:n_var,:);
101     lambdas=sol_num(n_var+1:end,:);
102 end

```

### 8.3) Learning rate optimization

I've been thinking (starting from the first course of Machine Learning about Gradient Descent), how can the learning rate be optimized using an optimization algorithm (rather than only decreasing it at a certain rate after each iteration).

Let's consider a loss function  $f(x)$ , where  $x = (x_1, x_2, \dots, x_n)$ .

We want to find  $x_0$  so that  $\underset{x}{\operatorname{argmin}} f(x) = x_0$ . Using Gradient descent, the update step can be written as:

$$x_{k+1} = x_k - \alpha \frac{\partial f}{\partial x}$$

Let's suppose that the value of  $\alpha$  is not known a priori, so we can consider that  $x_{k+1}$  depends on  $\alpha$ :

$$x_{k+1} = x_{k+1}(\alpha) = x_k - \alpha \frac{\partial f}{\partial x}$$

Now, at each step, we can say that we want to minimize  $f(x_{k+1}(\alpha)) \leq f(\alpha) = f\left(x_k - \alpha \frac{\partial f}{\partial x}\right)$ . At this point we have to minimize the univariate function  $f(\alpha)$ . At this point, we can use a simpler minimization algorithm for univariate functions, such as Golden Section Search or Brent's Method to find a near-optimal  $\alpha_{nopt}$  and thus, perform the update.

$$x_{k+1} = x_k - \alpha_{nopt} \frac{\partial f}{\partial x}$$

## 9)References

- [1] SHENODA, Michael. Real-time Object Detection: YOLOv1 Re-Implementation in PyTorch. *arXiv preprint arXiv:2305.17786*, 2023.
- [2] Redmon, Joseph, and Ali Farhadi. "Yolov3: An incremental improvement." *arXiv preprint arXiv:1804.02767* (2018).
- [3] Tomato Diseases and Pests Detection Based on Improved Yolo V3 Convolutional Neural Network - Scientific Figure on ResearchGate. Available from: [https://www.researchgate.net/figure/Darknet-53-network-structure\\_fig1\\_342217605](https://www.researchgate.net/figure/Darknet-53-network-structure_fig1_342217605)
- [4] Yingge, Huo & Ali, Imran & Lee, Kang-Yoon. (2020). Deep Neural Networks on Chip - A Survey. 589-592. 10.1109/BigComp48618.2020.00016.
- [5] Analysis of Convolutional Neural Network based Image Classification Techniques - Scientific Figure on ResearchGate. Available from: [https://www.researchgate.net/figure/Convolution-Operation-with-filter-33-with-stride-2\\_fig3\\_352726526](https://www.researchgate.net/figure/Convolution-Operation-with-filter-33-with-stride-2_fig3_352726526)