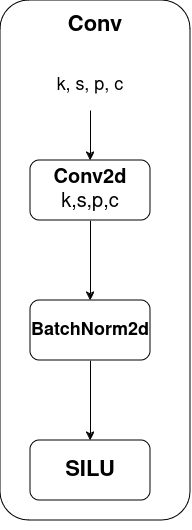
# YOLOv8 Architecture

YOLOv8 is made up of three main sections called Backbone , Neck and the Head.

The architecture contains different blocks which are described separately.

It consists of Convolution Block(Conv Block), C2f Block, BottleNeck Block, Spatial Pyramid Pooling Fast(SPPF) Block and finally Detect Block.

## Convolution Block (Conv Block)

It is the most basic block which consists of the Conv2d layer, BatchNorm2d layer and SiLU activation function.

### Conv2d Layer:

Convolution is a mathematical operation that involves sliding a small matrix (called a kernel or filter) over the input data, performing element-wise multiplication, and summing the results to produce a feature map. The convolution operation is especially useful for detecting patterns such as edges, textures, and other localized features in images.

The "2D" in Conv2D refers to the fact that the convolution is applied in two spatial dimensions, typically height and width. Conv2D layers are commonly used for processing 2D grid data, such as grayscale or color images. The parameters of a Conv2D layer include the number of filters (kernels), the size of the filters, the stride (step size of the filter movement), and padding (whether to pad the input to maintain spatial dimensions).

* **k:** Number of filters or kernels. It represents the depth of the output volume, and each filter is responsible for detecting different features in the input.
* **s:** Stride. It is the step size at which the filter/kernel slides over the input. A larger stride reduces the spatial dimensions of the output volume.
* **p:** Padding. Padding is the additional border of zeros added to the input on each side. It helps preserve spatial information and can be used to control the spatial dimensions of the output volume.
* **c:** Number of channels in the input. For example, in an RGB image, c would be 3 (one channel for each color: red, green, and blue).

### BacthNorm2d Layer:

Batch Normalization (BatchNorm) is a technique used in deep neural networks to improve the training stability and convergence speed. In the context of convolutional neural networks (CNNs), the BatchNorm2d layer specifically applies batch normalization to 2D inputs, which are typically the outputs of convolutional layers.

The BatchNorm2d layer operates on each channel independently, normalizing the activations within a mini-batch. It performs the following operations for each channel:

1. **Normalization:** Subtract the mean of the mini-batch and divide by the standard deviation. This helps to center and scale the activations.
2. **Scaling and Shifting:** Introduce learnable parameters (gamma and beta) to scale and shift the normalized values. This allows the model to adapt the normalized values to the specific needs of the layer.

It helps the neural networks by:

1. **Consistent Scaling:** It ensures that the numbers going through the network aren't too big or too small. This helps in preventing problems during training.
2. **Adjustment for Better Learning:** It fine-tunes the data as it passes through each layer, making it easier for the network to learn from the examples it's given.
3. **Quicker Training:** With BatchNorm2d, the neural network can learn the right patterns more quickly. This is like giving the network a set of tools to better understand and remember different features in the pictures.

### SiLU Activation Function:

SiLU, which stands for Sigmoid Linear Unit, is an activation function used in neural networks. It is also known as the Swish activation function.

The SiLU activation function is defined as follows:

SiLU(x)=x⋅σ(x)

where σ(x) is the sigmoid function, which is given by:

σ(x)=1/(1+e^-x)​

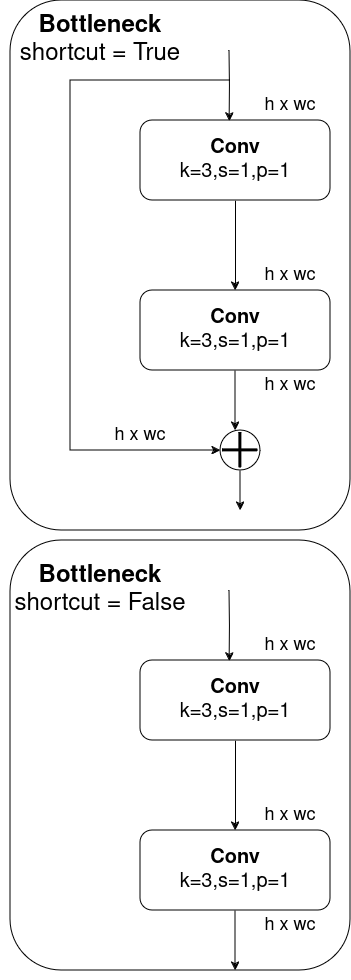
In simple terms, the SiLU activation function computes the element-wise product of the input x and the sigmoid function applied to x. It has been shown in the research that SiLU can provide better performance in certain deep learning tasks compared to other activation functions like ReLU (Rectified Linear Unit).

The key characteristic of SiLU is that it allows for smooth gradients, which can be beneficial during the training of neural networks. Smooth gradients can help avoid issues like vanishing gradients, which can impede the learning process in deep neural networks.

The normalized output from **BatchNorm2d** is then passed through the **SiLU** activation function. The SiLU function introduces non-linearity to the network. The function's output is the result of applying the SiLU function element-wise to the input.

The output of the SiLU activation function is the final output of the layer, and it serves as the input to the subsequent layer in the neural network. The SiLU activation function introduces non-linearity to the network by applying a smooth version of the rectified linear unit (ReLU: ReLU(x)=max(0,x)) with a sigmoid component. When combined with Batch Normalization, it can help improve the training stability and convergence of neural networks.

## Bottleneck Block



The bottleneck block consists of the Cov Block with a shortcut and Conv Block without a shortcut. If the shortcut=true then the shortcut is implemented in the bottleneck block else the input is passed through two Conv Blocks in a series.

### Shortcut Connection:

A bottleneck block is often used to improve the efficiency and performance of the network, especially in deep architectures. The bottleneck block consists of two convolutional layers and a shortcut connection.

The **shortcut connection**, also known as a skip connection or residual connection, is a direct connection that bypasses one or more layers in the network. It allows the gradient to flow more easily through the network during training, addressing the vanishing gradient problem and making it easier for the model to learn.

The main purpose of the shortcut connection is to facilitate the training of deep networks. As the network depth increases, it becomes more challenging for gradients to propagate through all the layers during backpropagation. This can lead to the vanishing gradient problem, where the gradients diminish as they are back propagated through the layers, making it difficult for the model to learn meaningful representations.

By introducing a shortcut connection, the model has the option to skip one or more layers, allowing the gradient to flow more directly from the input to the output during backpropagation. This helps in training deeper networks more effectively and can lead to better convergence and generalization performance.

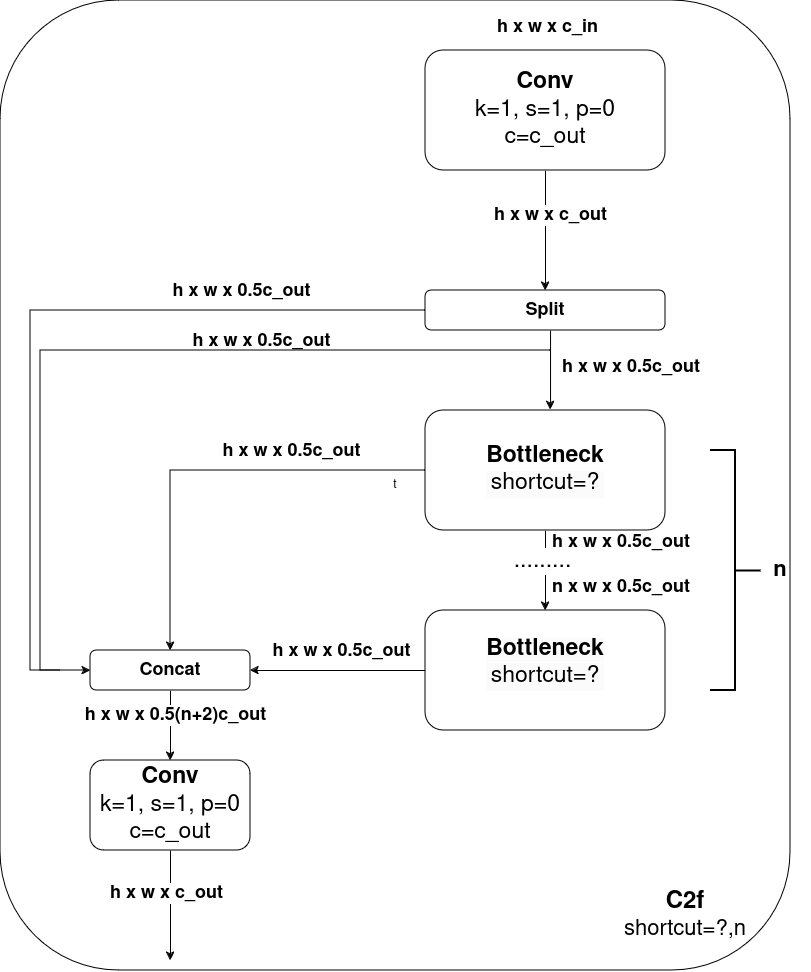
In the specific context of a bottleneck block in a CNN with two convolutional blocks and a shortcut, the shortcut connection allows the model to bypass the convolutional blocks if necessary. This way, the model can choose to use the identity mapping provided by the shortcut, making it easier to learn the identity function when needed. The inclusion of a shortcut connection enhances the ability of the model to learn complex representations and improves the training of deep CNNs.

**What is the vanishing gradient problem?**The vanishing gradient problem is a challenge that arises during the training of deep neural networks, particularly in architectures with many layers.It occurs when the gradients of the loss function with respect to the parameters (weights) of the network become extremely small as they are back propagated from the output layer to the input layer during the training process.

For Example: Imagine you're teaching a computer to recognize patterns, like identifying objects in images. In a deep learning model, there are many layers of computations. The vanishing gradient problem is like trying to teach the computer where it went wrong when it makes a mistake, but the feedback you give becomes so tiny that the computer doesn't learn much from it.

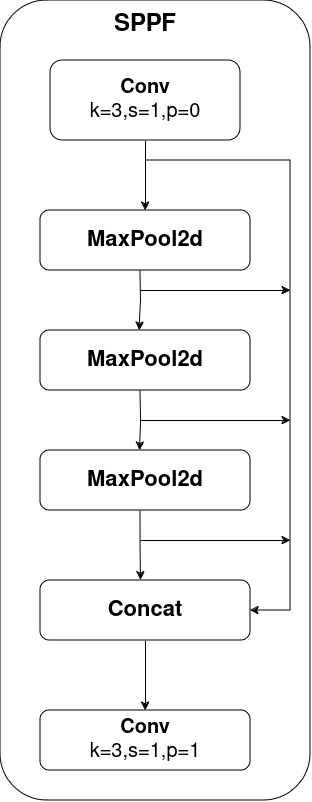
It's like telling someone a secret, and as the secret gets passed from person to person, each person whispers it softer and softer. Eventually, the last person hears only a faint trace of the original message, and the information is lost. In the same way, the guidance for learning in deep models becomes so faint that the early layers don't effectively learn the important features of the data.

## C2f Block



This C2f block consists of a convolutional block which then the resulting feature map will be split. One feature map goes to the Bottleneck block where other goes directly to the Concat block. In the C2f block we can have as many Bottleneck blocks as we like. At the end, the feature map from the bottleneck block and the split feature map is concatenated and inputted into a final convolutional block.

## Spatial Pyramid Pooling Fast (SPPF) Block:



The basic idea behind Spatial Pyramid Pooling is to divide the input image into a grid and pool features from each grid cell independently, allowing the network to handle images of different sizes effectively.

In essence, Spatial Pyramid Pooling enables neural networks to work with images of different resolutions by capturing multi-scale information through pooling operations at different levels of granularity. This can be particularly useful in tasks such as object recognition, where objects may appear at different scales within an image

While SPP offers advantages, it can be computationally expensive. SPP-Fast addresses this by using a simpler pooling scheme. Instead of using multiple pooling levels with different kernel sizes, SPP-Fast might use a single fixed-size kernel for pooling, reducing the number of computations needed. SPP-Fast offers a trade-off between accuracy and speed.

The **SPPF Block** consists of a convolutional block followed by three MaxPool2d layers. Every resulting feature map from the MaxPool2d layer is then concatenated at the end and fed to a convolutional block.

### MaxPool2d Layer:

A MaxPool2d layer is a type of pooling layer commonly used in convolutional neural networks (CNNs). The "2d" in MaxPool2d stands for two-dimensional, indicating that the pooling operation is applied independently to each individual 2D spatial slice of the input tensor.

Pooling layers are used to down sample the spatial dimensions of the input volume, reducing the computational complexity of the network and extracting dominant features. Max pooling is a specific type of pooling operation where, for each region in the input tensor, only the maximum value is retained, and the other values are discarded.

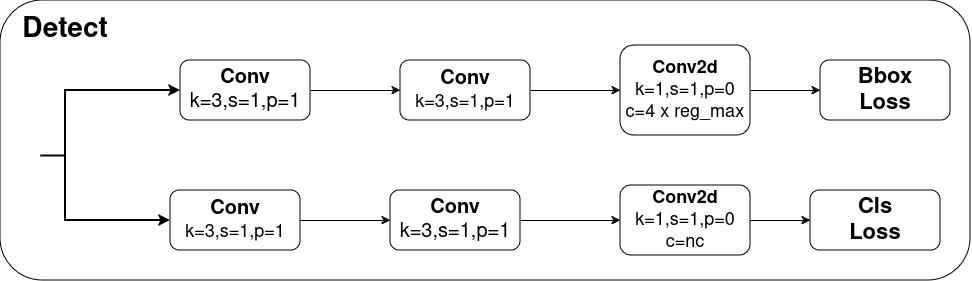
In the case of MaxPool2d, the pooling is applied in both the height and width dimensions of the input tensor. The layer is defined by specifying parameters such as the size of the pooling kernel (in our case k=3) and the stride (s=1). The kernel size determines the spatial extent of each pooling region, and the stride determines the step size between successive pooling regions.

**In simpler terms:**

Imagine you have an image, and you want to find the most important features in different parts of that image. The MaxPool2d layer is like a sliding window that moves across the image. At each step, it looks at a small part of the image (a 2x2 square, for example) and only keeps the biggest number in that square. This "biggest number" represents the most important information in that part of the image.

By doing this process across the entire image, the MaxPool2d layer helps reduce the size of the image while keeping the essential features. It's like zooming out and summarizing the important stuff while throwing away less important details. This downsizing makes the computations in the neural network more manageable and helps the network focus on the most critical aspects of the input data.

## Detect Block

****

Detect Block is responsible for the detection of the objects. Unlike in previous versions of YOLO , YOLOv8 is an anchor free model which means it predicts directly the center of an object instead of the offset from a known anchor box. Anchor free detection reduces the number of box predictions, which speeds up complicated post processing steps that sift through candidate detections after inference. In YOLOv8, the prediction is done in the grid cell.

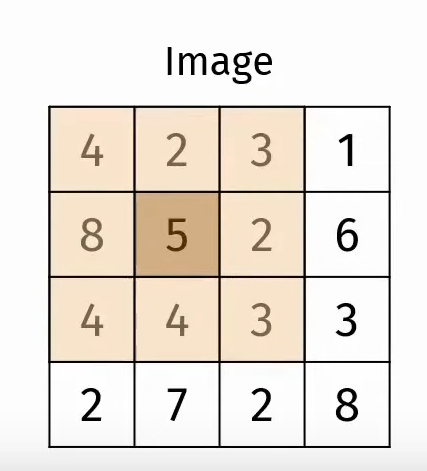
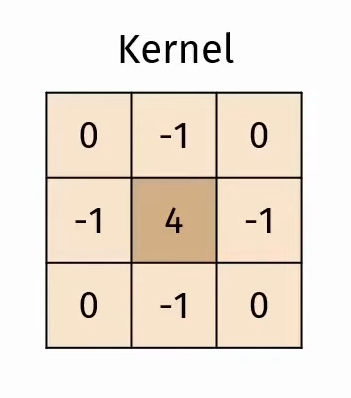
The Detect Block contains two tracks. The first track is for **bounding box** predictions and the second track is for **class** predictions.

Both tracks contain two convolutional blocks followed by a single Conv2d layer which give the Bounding Box loss and Class Loss respectively.

# Basic Terminology in CNN:

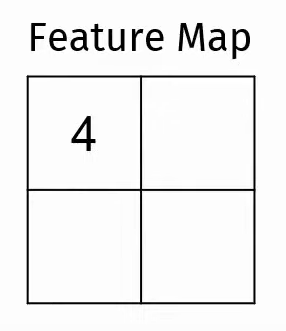
## Kernel:

Kernel is a two dimensional array or a small matrix used for convolutional operation. Kernels are also called feature detectors. The value in the kernel is weights that can be updated during the training of the neural networks. Kernel will move across the given image and perform a DOT operation between the input and the kernel matrix. The output from the dot operation produces the feature map matrix of the given image.



DOT operation performed:

(4\*0)+(2\*-1)+(3\*0)+(8\*-1)+(5\*4)+(2\*-1)+(0\*4)+(4\*-1)+(3\*0) = 4



Likewise the feature map is filled using the help of the kernel.

## Stride:

Stride is defined as the displacement distance during the convolution process. The smaller the resulting output, the larger the stride. It defines the step size or the number of pixels the convolutional filter (or kernel) moves across the input data at each step. The stride value determines how much the filter shifts or "slides" over the input data when performing the convolution operation.

For example, if we have a Image matrix of 8\*8 and kernel size of 3\*3 and stride=2 then our feature map matrix will be of 3\*3 matrix as the kernel moves two pixel at a time covering the 8\*8 matrix in 3\*3.

**Input size:**

1 2 3 4 5 6 7 8

9 10 11 12 13 14 15 16

17 18 19 20 21 22 23 24

25 26 27 28 29 30 31 32

33 34 35 36 37 38 39 40

41 42 43 44 45 46 47 48

49 50 51 52 53 54 55 56

57 58 59 60 61 62 63 64

**Output feature map:**

[1\*W 3\*W 5\*W]

[17\*W 19\*W 21\*W]

[33\*W 35\*W 37\*W]

Likewise for the same input and kernel size but if our stride=1 then the output feature map will be of 6\*6 matrix as the kernel needs to travel one step at a time in the 8\*8 matrix resulting in a 6\*6 output matrix.

**Input size:**

1 2 3 4 5 6 7 8

9 10 11 12 13 14 15 16

17 18 19 20 21 22 23 24

25 26 27 28 29 30 31 32

33 34 35 36 37 38 39 40

41 42 43 44 45 46 47 48

49 50 51 52 53 54 55 56

57 58 59 60 61 62 63 64

**Output feature map:**

[1\*W 2\*W 3\*W 4\*W 5\*W 6\*W]

[9\*W 10\*W 11\*W 12\*W 13\*W 14\*W]

[17\*W 18\*W 19\*W 20\*W 21\*W 22\*W]

[25\*W 26\*W 27\*W 28\*W 29\*W 30\*W]

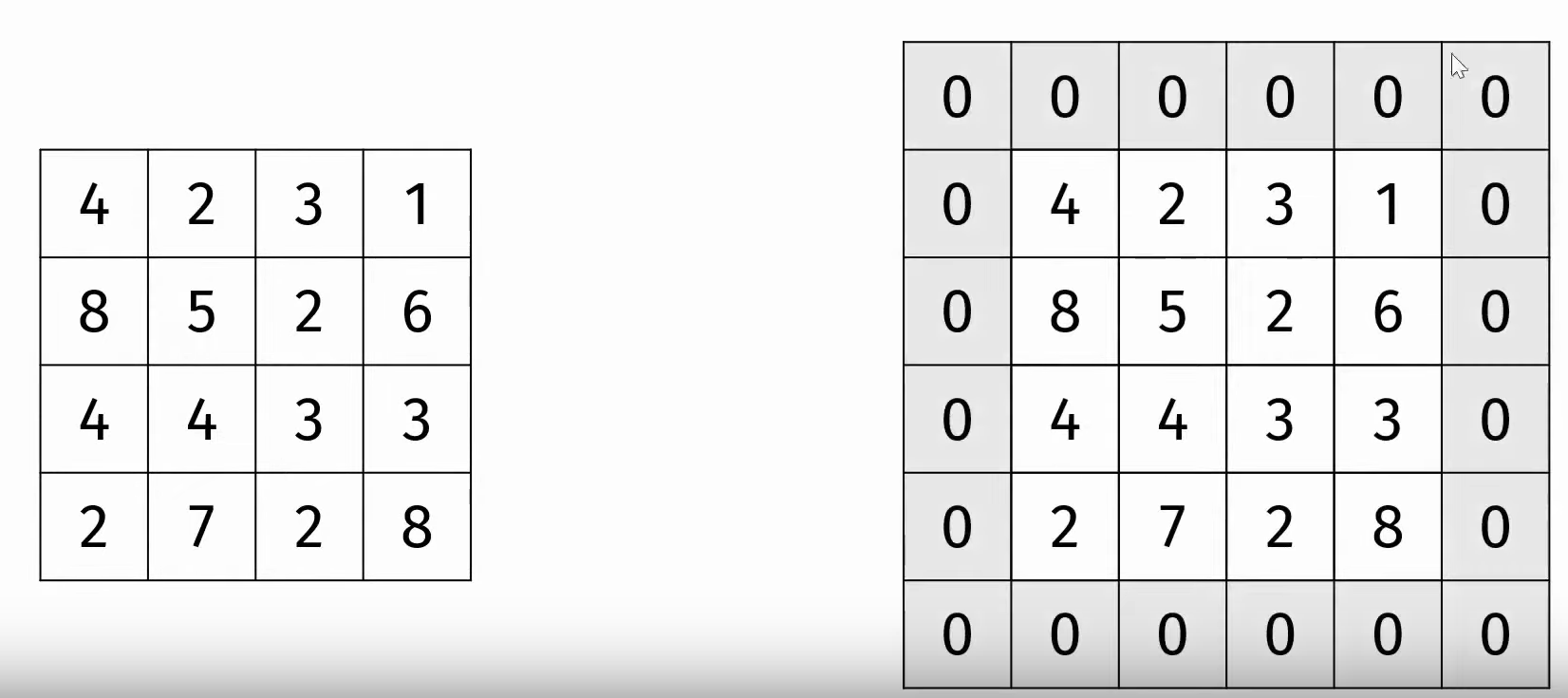
[33\*W 34\*W 35\*W 36\*W 37\*W 38\*W]

[41\*W 42\*W 43\*W 44\*W 45\*W 46\*W]

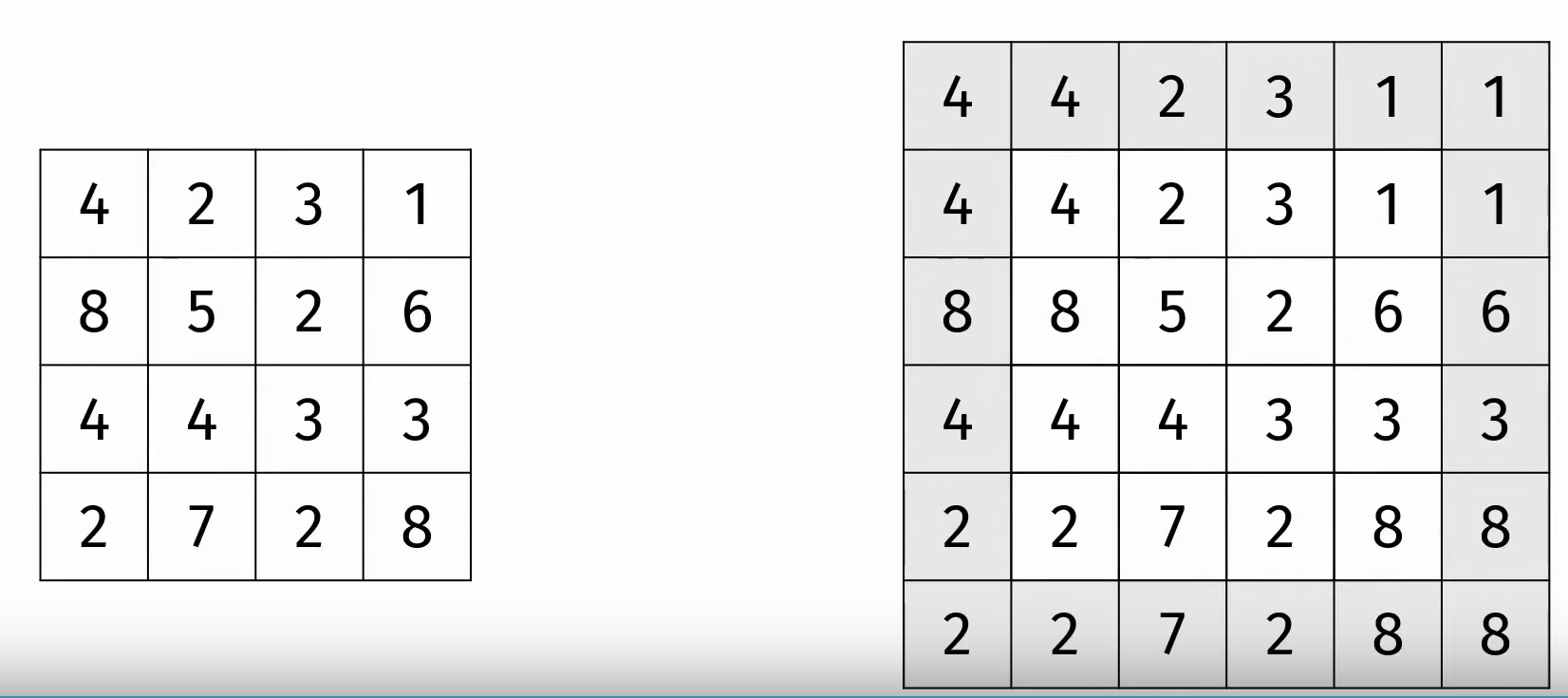
## Padding:

Padding is a technique used in the context of convolutional neural networks (CNNs) to add extra pixels around the input data before applying the convolution operation. These additional pixels are usually set to zero (zero-padding), although other padding strategies exist. Another padding strategy is replication padding where the outermost element of the image is replicated in the padding region The primary purpose of padding is to preserve the spatial dimensions of the input and prevent the reduction in size that can occur as a result of convolutional and pooling operations.

Zeros padding:



Replication Padding:



# YOLOv8 Architecture Explained:

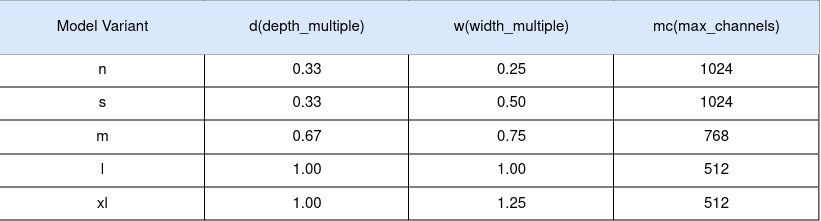
It consists of three components: Backbone, Neck, and Head.

Backbone is the deep learning architecture that acts as a feature extractor of the inputted image.

Neck combines the features acquired from the various layers of the Backbone module.

Head predicts the classes and the bounding box of the objects which is the final output produced by the object detection model.

The YOLO models are classified in terms of three parameters which are depth\_multiple, width\_multiple and max\_channels.



From the above table we can see that there are 5 types of Yolov8 model. In our project , we are using **yolov8n.pt** which is the **“n”** variant of the YOLO model consisting of d=0.33, w= 0.25 and mc= 1024.



**depth\_multiple(d):**

depth\_multiple parameter determines how many Bottleneck Blocks are used in C2f block.

This scales the number of layers in the network. A value less than 1 reduces the depth (fewer layers), making the model smaller and faster but potentially less accurate. Conversely, a value greater than 1 increases the depth (more layers), leading to a larger and potentially more accurate model but slower to run.

**width\_multiple (w):**

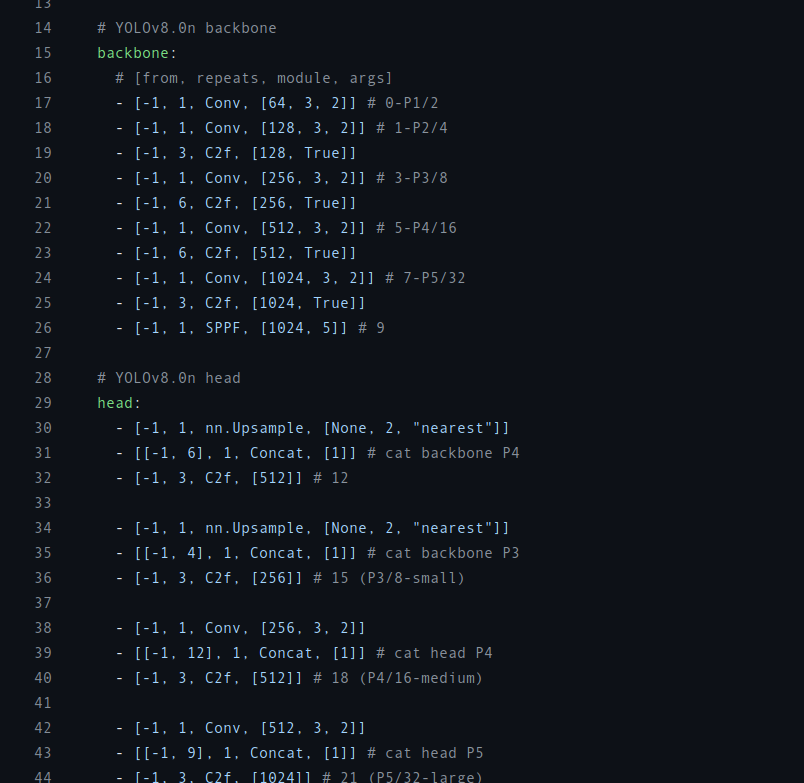
This scales the number of channels in the convolutional layers. A value less than 1 thins the network (fewer channels), resulting in a smaller and faster model but potentially sacrificing some accuracy. On the other hand, a value greater than 1 widens the network (more channels), creating a larger and potentially more accurate model but requiring more processing power.

**max\_channels (mc):**

This parameter sets an upper limit on the number of channels allowed in the network. It acts as a safety measure to prevent the model from becoming too wide (too many channels) especially when width\_multiple is set high. This can help control the model size and prevent overfitting.

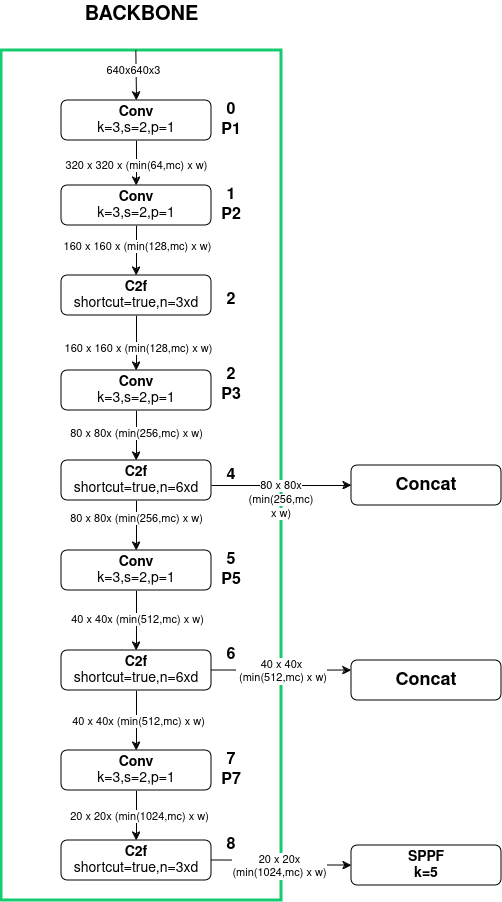
**Numbering in the blocks:**

The numbering present in the architecture blocks are from the yolo.yaml file in the official document where numbering is given to blocks used in it.



Here 0,1,2… etc number denotes the modules used in the architecture and P1,P2, P3,P4,P5 are the different resolutions from which the features are extracted and then later on concatenated in the Neck block using up-sampling or down-sampling.

## Backbone Block:



### BLOCK 0 and 1:

The backbone begins with two convolutional blocks with kernel size=3, stride=2 and padding=1.

The spatial resolution is reduced when stride=2 is used.

For example:

The input image in the above model is of 640\*640 resolution. By using stride=2, we reduce the resolution of the image by half i.e 320\*320 because the kernel moves in 2 pixel increments so the output feature map becomes 320\*320 resolution.

To obtain the output channel of the convolution block we use the following formula :

**min(64,mc)\*w**

Here,

64 is the base output channel

mc is the max\_channel (since we are using “n” type model so our mc= 1024)

w is the width\_multiple (since we are using “n” type model so our w= 0.25)

So our final output channel becomes = min(64,1024)\*0.25

= 64\*0.25

= 16

The total output of the first Convolutional Block is 320\*320\*16.

Likewise the same process is applied in the second Convolutional Block as the output of the second convolutional block is : 160\*160\*32

### BLOCK 2:

Block 2 is C2f block which contains two parameters i.e. **shortcut** and **n**.

Here **shortcut** is the boolean parameter that denotes if the Bottleneck block utilizes the shortcut or not.

If the value of the shortcut= true then the bottleneck block inside the C2f block utilizes the shortcut else it doesn't.

Here **n** determines how many bottleneck blocks are used inside the C2f block. In the case of Block 2 n is given by:

**n= 3\*d**

where d= depth\_multiple

In our case the depth\_multiple of “n” type Yolo model is 0.33 so,

the number of bottleneck block used inside the C2f becomes (n) = 3\* 0.33

=0.99 i.e. 1 bottleneck block is used.

In C2f block since it doesnot require kernels or strides, the resolution and the output channel is unchanged.

### BLOCK 3:

Block 3 is another Convolutional Block which takes the input from the C2f block and reduces its resolution by half utilizing stride=2.

The output of the BLOCK 3 becomes 80\*80\*64

### BLOCK 4:

Block 4 is another C2f Block with shortcut=true and n= 6\*d = 6\*0.33 = 1.98 i.e 2 bottleneck blocks used.

Block 4 is concatenated in the Neck Module using the same image resolution of 80\*80\*64 at BLOCK 14.

### BLOCK 5:

Block 5 is another Convolutional Block which takes the input from the C2f block and reduces its resolution by half utilizing stride=2.

The output of the BLOCK 5 becomes 40\*40\*128

### BLOCK 6:

Block 6 is another C2f Block with shortcut=true and n= 6\*d = 6\*0.33 = 1.98 i.e 2 bottleneck blocks used.

Block 6 is concatenated in the Neck Module using the same image resolution of 40\*40\*128 at BLOCK 11.

### BLOCK 7:

Block 7 is another Convolutional Block which takes the input from the C2f block and reduces its resolution by half utilizing stride=2.

The output of the BLOCK 5 becomes 20\*20\*256

### BLOCK 8:

Block 8 is another C2f Block with shortcut=true and n= 3\*d = 3\*0.33 = 0.99 i.e 1 bottleneck block used.

Block 6 is then passed to the SPPF block with the same image resolution of 20\*20\*256 at BLOCK 9.

### BLOCK 9 (SPPF) :

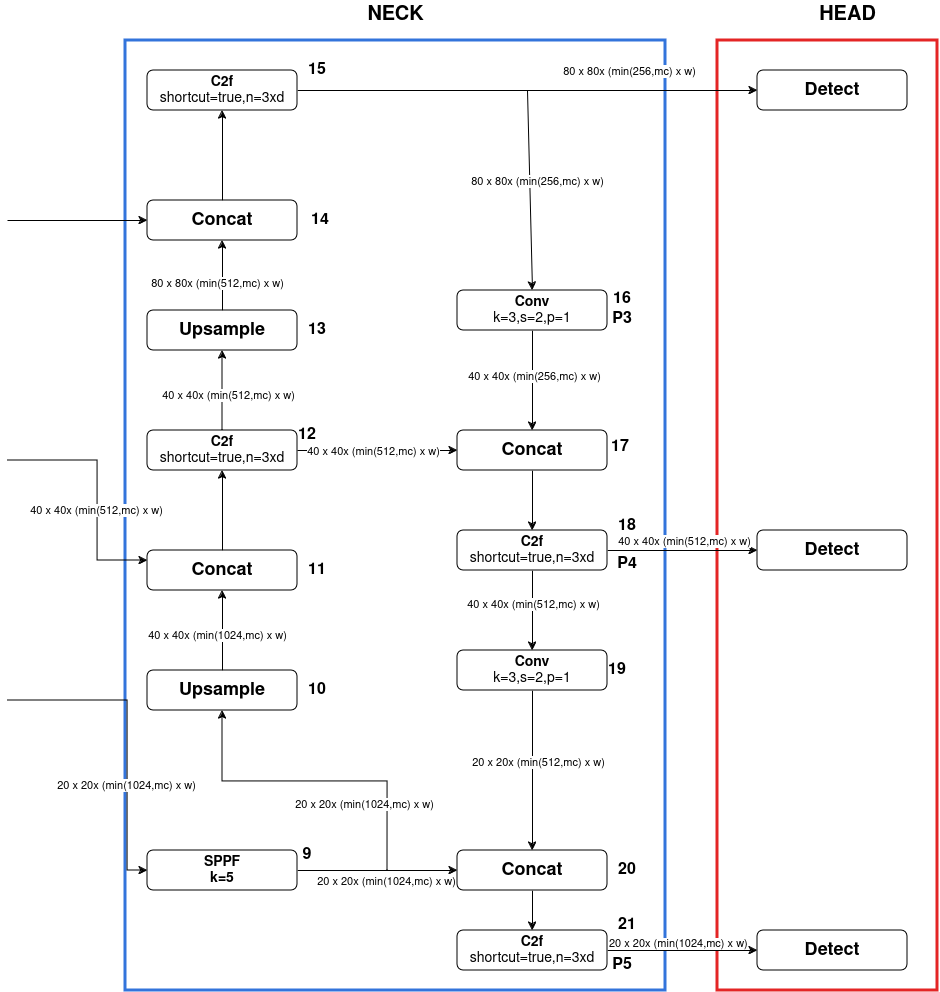
SPPF Block is used after the last convolution layer of the C2f block in the Backbone.

The main function of the SPPF block is to generate the fixed feature representation of the object in various sizes in an image without resizing the image or introducing spatial information loss.

The SPPF Block is also concatenated in BLOCK 20 with 20\*20\*256 resolution.

## 

## Neck Block and Head Block:



### BLOCK 10:

Block 10 contains the Upsample layer. This Upsample layer increases the feature map resolution of the SPPF (20\*20) to match with the feature map resolution of the Block 6 (40\*40).

The Upsample layer simply increases the feature map by double in this case to match the feature map of the Block 6 without making any changes in the output channel (256).

### BLOCK 11:

Block 11 is a Concat Block that sums the output channel of the blocks that are concatenated without any change in resolution.

Here Block 6 and Block 10 are concatenated which means the output channel of Block 6 (128) and Block 10 (256) is added = 128+256 = 384. And the resolution is unchanged (40\*40).

So, the final output of the Concat Block is 40\*40\*384.

### BLOCK 12:

Block 12 is a C2f Block with shortcut=false which means it does not use shortcut in bottleneck block and n= 3\*d = 3\*0.33 = 0.99 i.e 1 bottleneck block used.

Block 12 is concatenated in BLOCK 17 with the resolution of 40\*40\*384.

### BLOCK 13:

Block 13 contains the Upsample layer. This Upsample layer increases the feature map resolution of the C2f Block (40\*40) to match with the feature map resolution of the Block 4 (80\*80).

The Upsample layer simply increases the feature map by double in this case to match the feature map of the Block 4 without making any changes in the output channel (384).

### BLOCK 14:

Block 14 is a Concat Block that sums the output channel of the blocks that are concatenated without any change in resolution.

Here Block 4 and Block 13 are concatenated which means the output channel of Block 4 (64) and Block 13 (384) is added = 64+384 = 448. And the resolution is unchanged (80\*80).

So, the final output of the Concat Block is 80\*80\*448.

### BLOCK 15:

Block 14 is another C2f Block with shortcut=false and n= 3\*d = 3\*0.33 = 0.99 i.e 1 bottleneck block used.

This block will reduce the channel size of the feature map. The feature map of this block (80\*80\*448) is used as an input for the Detect Block in the Head.

**This Detect block specializes in detecting small objects.** (First Detect Block in the Head)

The output of Block 15 is also used as input in Convolutional Block (BLOCK 16) with input of 80\*80\*448.

### BLOCK 16:

The convolutional block uses kernel size=3, stride=2 and padding=1.

Block 16 is another Convolutional Block which takes the input from the C2f block and reduces its resolution by half utilizing stride=2.

The output of the BLOCK 16 becomes 40\*40\*64

### BLOCK 17:

Block 17 is a Concat Block that sums off the output channel of the blocks that is being concatenated without any change in resolution.

Here Block 12 and Block 16 are concatenated which means the output channel of Block 12 (384) and Block 16 (64) is added = 384+64 = 448. And the resolution is unchanged (40\*40).

So, the final output of the Concat Block is 40\*40\*448.

### BLOCK 18:

Block 18 is another C2f Block with shortcut=false and n= 3\*d = 3\*0.33 = 0.99 i.e 1 bottleneck block used.

This block will reduce the channel size of the feature map. The feature map of this block (40\*40\*448) is used as an input for the Detect Block in the Head.

**This Detect block specializes in detecting medium size objects.** (Second Detect Block in the Head)

### BLOCK 19:

The convolutional block uses kernel size=3, stride=2 and padding=1.

Block 19 is another Convolutional Block which takes the input from the C2f block and reduces its resolution by half utilizing stride=2.

The output of the BLOCK 19 becomes 20\*20\*112

### BLOCK 20:

Block 20 is a Concat Block that sums off the output channel of the blocks that are concatenated without any change in resolution.

Here Block 9 (SPPF) and Block 19 are concatenated which means the output channel of Block 9 (256) and Block 19 (112) is added = 256+ 112 = 368. And the resolution is unchanged (20\*20).

So, the final output of the Concat Block is 20\*20\*368.

### BLOCK 21:

Block 21 is another C2f Block with shortcut=false and n= 3\*d = 3\*0.33 = 0.99 i.e 1 bottleneck block used.

This block will reduce the channel size of the feature map. The feature map of this block (20\*20\*368) is used as an input for the Detect Block in the Head.

**This Detect block specializes in detecting large objects.** (Third Detect Block in the Head)

### Total Amount of blocks used in the Yolov8 architecture:

**Convolution Block:**

Conv2d layer: 1

BatchNorm2d: 1

SiLU Activation Function : 1

**Bottleneck Block**:

Convolutional Block: 2

**C2f Block:**

Convolutional Block: 2

Bottleneck Block : as required

**SPPF Block:**

Convolutional Block: 2

MaxPool2d Layer: 3

**Detect Block:**

Convolutional Block: 4

Conv2d Layer: 2

Total Individual Convolutional Block used in the Architecture: 7

Total C2f Block used in the architecture: 8

Total SPPF Block used in the architecture: 1

Total Detect Block used in the architecture: 3

Total Bottleneck Block with shortcut true used for “n” type of YOLO model in different C2f Block: 10

#### Total no. of Conv2d layer used in Yolov8 architecture:

7\*1+(8\*2)+(10\*2)+1\*2+3\*(4+2)= 63 with Bottleneck block

7\*1+(8\*2)+1\*2+3\*(4+2)= 43 without Bottleneck block

#### Total no. of BatchNorm2d Layer used in Yolov8 architecture:

7\*1+(8\*2)+(10\*2)+1\*2+3\*4= 57 with Bottleneck block

7\*1+(8\*2)+1\*2+3\*4= 37 without Bottleneck block

#### Total no. of SiLU Activation Function used in Yolov8 architecture:

7\*1+(8\*2)+(10\*2)+1\*2+3\*4= 57 with Bottleneck block

7\*1+(8\*2)+1\*2+3\*4= 37 without Bottleneck block

#### Total no. of MaxPool2d layer used in Yolov8 architecture: 3

**Note:**

In Yolov8, the value of Stride is 2 because the kernel can navigate to every pixel in the image without focusing on the middle pixel much often. Here if the stride=1 then the pixel evaluated by the kernel can again be calculated in the next evaluation which causes the model to extract features from the same set of pixels multiple times. This leads to prioritizing the middle of the image for feature extraction rather than equally from the image. Taking stride=1 less prioritizes the feature of the corner of the images for feature extraction which leads to poor performance of the model. So stride=2 is used to prioritize the whole image rather than focusing in the middle of the image for feature extraction.

# Facts of Yolov8 Model:

## Input image size:

Yolov8 model is trained in 640\*640 resolution by default.

We typically input the resolution of 2160\*3840 pixel and 1920\*1080 pixels input video.

Our model takes each frame from the given video so our input size is also 2160\*3840. Then the Yolov8 model resize the image to 640\*360 to maintain the aspect ratio of the image and apply padding to the image to make its resolution to 640\*640. When the resizing is done, it runs detection, and finally scales and clips the bounding boxes to the original image size.

Simply, Image size taken by our model is 640\*640 pixels.

## Batch Size:

Default batch size of Yolov8= 16

Batch size for training, indicating how many images are processed before the model's internal parameters are updated.

Decrease in batch size leads to increase in training time but produces more accurate result whereas increasing it can lead to slightly worse results

In our training, we used the default batch size = 16

## Worker Thread:

Default worker thread of Yolov8= 8

These worker threads asynchronously load and preprocess data while the main training process focuses on computations such as forward and backward passes through the neural network.

Decrease in worker thread leads to increase in training time but produces more accurate results whereas increasing it can lead to slightly worse results.

In our training, we used the default batch size = 16

Default Learning Rate of Yolov8=0.01

Optimizer used : AdamW

Time to train the model: 10 epochs completed in 23.071 hours.

**Q. Can you explain the concept of anchor boxes and how they are utilized in YOLOv8?**

Anchor boxes are predefined bounding boxes of different aspect ratios and scales used by YOLOv8 to predict object locations and sizes within an image. They help improve detection accuracy by handling objects of various shapes and sizes.

**Q. How does YOLOv8 handle multi-object detection in an image?**

YOLOv8 employs a single neural network to simultaneously predict multiple bounding boxes and class probabilities for objects within an image, enabling efficient multi-object detection.

**Q. What is non-maximum suppression (NMS), and why is it important in object detection tasks?**

Non-maximum suppression (NMS) is a post-processing technique used to suppress multiple overlapping bounding box predictions for the same object, retaining only the most confident detection.

**Q. What is mAP?**

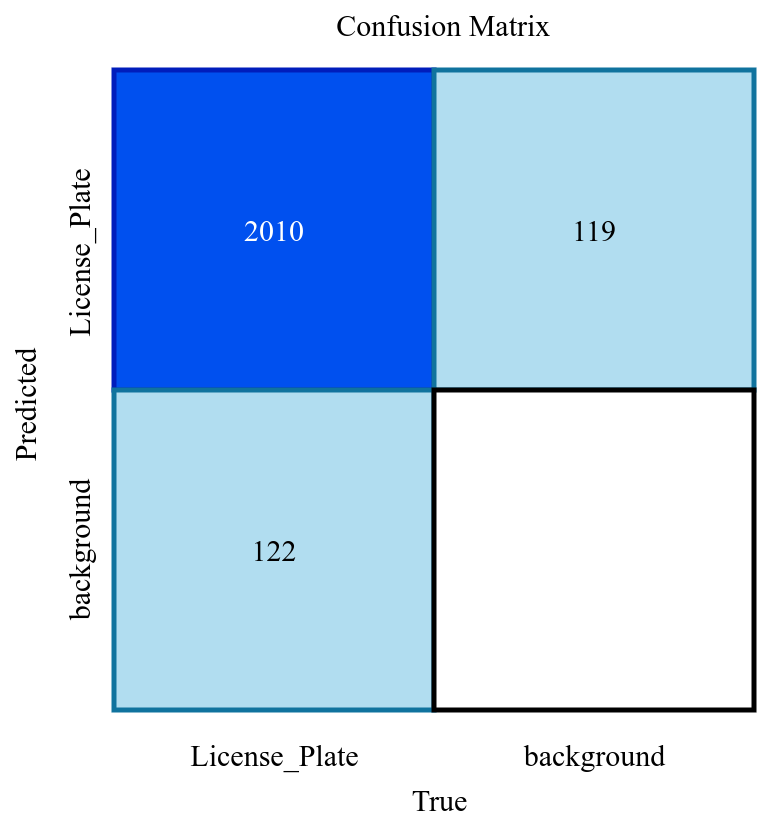
It stands for mean Average Precision. It is a common metric used to evaluate the performance of object detection models.

In our model it is typically measured in 50% and 50-95%.

An mAP@50 value of 0.961 indicates that when evaluating the model's performance on a test dataset, the average precision across all classes (or on average) is 96.1% when considering a detection as correct if the intersection over union (IoU) between the predicted bounding box and the ground truth bounding box is greater than or equal to 50%.

In simpler terms, it means that when the model makes predictions, roughly 96.1% of the objects it identifies are correctly localized and classified with a moderate degree of overlap (at least 50%) with the ground truth bounding boxes. A higher mAP@50 value generally indicates better performance in object detection tasks.

Likewise, mAP@50-95 calculates the average precision by averaging the precision values obtained at IoU thresholds ranging from 50% to 95%. An mAP@50-95 value of 0.735 indicates that, on average, the model achieves a precision of 73.5% across this range of IoU thresholds.



**Recall** = TP / (TP + FN)

=2010/(2010+122)

=0. 942

**Precision** = TP / (TP + FP)

= 2010/(2010+119)

=0.944

**F1 Score** = 2 \* (Precision \* Recall) / (Precision + Recall)

= 2\*(0.942\*0.944)/(0.942+0.944)

= 0.94299894

**Accuracy** = (TP + TN) / (TP + TN + FP + FN)

= (2010+0)/(2010+119+122+0)

=0.89293647267

Note:

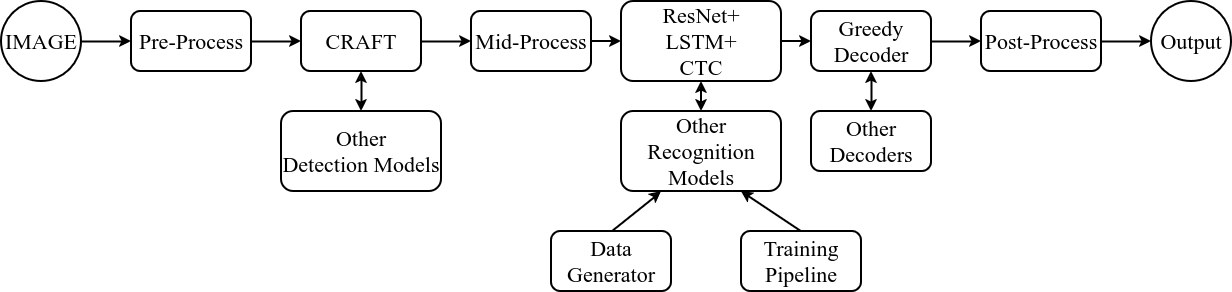
TP = True Positives

TN = True Negatives

FP = False Positives

FN = False Negatives

# EasyOCR Framework Overview



EasyOCR is a powerful Optical Character Recognition (OCR) framework designed to make text extraction from images simple and efficient. It supports over 80 languages and various writing scripts. Here's how it works:

**Core Components**

* **Pre-Process:**
  + **Image Preparation:** Images might be resized, have noise reduction applied, or have their contrast adjusted to enhance the text before recognition.
  + **Other Detection Models:** EasyOCR is flexible; it can incorporate other text detection models if CRAFT doesn't suit your needs.
* **CRAFT:**
  + **Text Detection:** This is a core component within EasyOCR. CRAFT (Character Region Awareness For Text Detection) is a deep learning-based model that locates text regions within an image, drawing bounding boxes around them.
* **Mid-Process:**
  + **Feature Extraction:** A convolutional neural network (CNN), typically ResNet, extracts essential visual features from the detected text regions.
  + **LSTM:** A Long Short-Term Memory (LSTM) network analyzes the sequential relationship between features, crucial for understanding the order and context of characters.
  + **CTC:** Connectionist Temporal Classification (CTC) is a decoding algorithm that translates the sequence of features into the final text output.
* **Greedy Decoder:**
  + **Text Formation:** The most probable text sequence from the CTC output is determined, often using a basic "greedy" algorithm that picks the most likely character at each step.
  + **Other Decoders:** Alternative decoders with more sophisticated language modeling could be used.
* **Post-Process:**
  + **Text Refinement:** Corrections for spelling, basic formatting, or adjustments based on domain knowledge (e.g., recognizing specific patterns in license plates) can be applied.

We use the Reader object from the EasyOCR library for our character recognition to detect the “en” language which is available in the EasyOCR library.

We are setting the parameter gpu= true inorder to use us resources from the GPU if it is available during processing.

The EasyOCR library automatically loads the detector model and the recognition model in the memory during processing. the default value of detector and recognizer = true.

readtext method:

The readtext method is the main method of Reader Object which takes the arguments of the image for recognition.

In our usages , just the license plate crop image is passed as an argument to the reader.readtext() method for character recognition.  
It uses greedy decoder as the default decoder for decoding output from the CTC.

EasyOCR uses the Batch size of 1 , worker threads =0.

It takes the image size up to 2560. Images bigger than this value will be resized down.

# OpenCv

## Grayscale:

In OpenCV, the standard method to convert an RGB image to grayscale uses a weighted sum of the red, green, and blue channels. This method typically results in a single-channel grayscale image. So, effectively, OpenCV converts RGB to a single layer of grayscale.

**Grayscale Images:** These have only one channel, representing intensity values. A single value determines the shade of gray for a particular pixel, ranging from 0 (black) to 255 (white).

**Conversion Process:** OpenCV's cv2.cvtColor() function with the flag cv2.COLOR\_BGR2GRAY is used for conversion. This function essentially combines the information from the three RGB channels into a single grayscale channel.

1. The function takes the weighted average of the Red, Green, and Blue values. Common weighting schemes include averaging all channels equally or using specific factors based on human perception of brightness.
2. The calculated intensity value is then assigned to the single channel of the grayscale image.

## Thresholding:

In OpenCV, converting a grayscale image to binary typically involves **one layer of thresholding**.

In our project we use binary thresholding to convert grayscale images in binary images.

We use 64 as the threshold value. Pixels in the grayscale image with intensity values **greater than** 64 will be classified as foreground (white) in the output binary image.

Since we use cv.THRESH\_BINARY\_INV, the flag inverts the usual thresholding behavior. So, pixels brighter than the threshold become black(background), and pixels darker than the threshold become white(license plate number)

**Pixel Classification:**

Pixels with intensity **greater than threshold value(64)** are classified as foreground and assigned the maximum intensity value (255) in the binary image, making them appear white.

Pixels with intensity **less than or equal to 64** are classified as background and assigned a value of 0, making them appear black.