COMPARISON OF DIFFERENT YOLO ARCHITECTURES IN MICROSCOPIC CELL COUNTING USING CONVOLUTIONAL NEURAL NETWORKS

A THESIS SUBMITTED TO

THE FACULTY OF ARCHITECTURE AND ENGINEERING

OF

EPOKA UNIVERSITY

BY

ALBA TUJANI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR

THE DEGREE OF MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

March 2023

Approval sheet of the Thesis

This is to certify that we have read this thesis entitled **“Comparison of different YOLO architectures in microscopic cell counting using convolutional neural networks”** and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Dr. Arban Uka
Head of Department
Date: March 10, 2023

Examining Committee Members:

Assoc. Prof. Dr. Carlo Ciulla	(Computer Engineering)	
Dr. Arban Uka	(Computer Engineering)	
Dr. Florenc Skuka	(Computer Engineering)	

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name Surname: Alba Tujani

Signature: ______________

ABSTRACT

COMPARISON OF DIFFERENT YOLO ARCHITECTURES IN MICROSCOPIC CELL COUNTING USING CONVOLUTIONAL NEURAL NETWORKS

Tujani, Alba

M.Sc., Department of Computer Engineering

Supervisor: Dr. Arban Uka

As automation continues to advance rapidly, cell detection and counting plays a significant, crucial role in medical image processing's automatic analysis. Accurately detecting and locating cells seems to be an extremely difficult task given their size, structure, and adherence to one another. In order to avoid time-consuming manual work, researchers are ever so eager in trying to implement and experiment with new technologies, to achieve a highly performative automatic cell segmentation.

Nevertheless, due to different image acquisition techniques, non-uniform backgrounds, variations in cell shapes or size, morphological properties and many other factors, obtaining perfect results is not always so easy. In this paper we study a convolutional neural network approach using YOLOv5 and YOLOv6 architectures for cell segmentation in microscopic images.

Key words: *medical image analysis, machine learning, yolo, classification, cell-counting, cnn*

ABSTRAKT

KRAHASIMI I ARKITEKTURAVE TË NDRYSHME YOLO NË NUMËRIMIN E QELIZAVE MIKROSKOPIKE ME PËRDORIMIN E RRJETEVE NEURALE KOVOLUCIONALE

Tujani, Alba

Master Shkencor, Departamenti i Inxhinierise Kompjuterike

Udhëheqësi: Dr. Arban Uka

Ndërsa automatizimi vazhdon të përparojë me shpejtësi, identifikimi dhe numërimi i qelizave luan një rol të rëndësishëm dhe vendimtar në analizën automatike të përpunimit të imazheve mjekësore. Zbulimi i saktë dhe lokalizimi i qelizave është një detyrë jashtëzakonisht e vështirë duke pasur parasysh madhësinë, strukturën dhe afërsinë e tyre me njëra-tjetrën. Për të shmangur punën manuale që kërkon shumë kohë, studiuesit janë gjithnjë në përpjekje për të zbuluar dhe eksperimentuar me teknologji të reja, për të arritur një segmentim automatik të qelizave sa më performues.

Sidoqoftë, për shkak të teknikave të ndryshme të marrjes së imazhit, sfondeve jo uniforme, variacioneve në forma ose madhësi të qelizave, vetive morfologjike dhe shumë faktorëve të tjerë, arritja e rezultateve të përsosura nuk është gjithmonë aq e lehtë. Në këtë punim ne studiojmë një qasje të rrjetit nervor konvolucional duke përdorur arkitekturat YOLOv5 dhe YOLOv6 për segmentimin e qelizave në imazhet mikroskopike.

Fjalët kyçe: *machine learning, mikroskopi, klasifikim, numerim qelizash, yolo, cnn*

ACKNOWLEDGEMENTS

I am deeply grateful to all those who have helped and supported me during my thesis journey. I would like to extend my heartfelt gratitude to my advisor Dr. Arban Uka, who has provided me with invaluable guidance, encouragement, and support throughout the entire process. Also, I would like to thank my family and friends, who have always been there for me with unwavering love and support. Your belief in me has been a constant source of strength.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

1. INTRODUCTION

1.1 Cell Counting

Cell counting plays a crucial role in the field of medicine, particularly in the diagnosis and treatment of diseases. The number of cells present in a microscopic image can provide important information about the health or growth of cells and tissues, and can be used to monitor the progression of various diseases, such as cancer. In the past, cell counting was typically performed manually by a human, which was time-consuming, prone to error, and lacked accuracy. However, with the advent of computer vision and artificial intelligence, automated cell counting has emerged as a more efficient and reliable solution for cell counting in medicine. [1] Automated cell counting algorithms use computer vision techniques to detect and count cells in images, reducing the workload of medical professionals and providing more accurate results. These algorithms have the potential to revolutionize the field of medicine, providing faster and more precise diagnoses, and improving the accuracy of clinical trials. The advancements in this field have the potential to positively impact patient care and overall health outcomes. In this paper, we will be analyzing two different YOLO architectures, YOLOv5 and YOLOv6. With their ability to process large amounts of data in a short amount of time, they are poised to revolutionize the field of cell counting.

1.2 Thesis Objective and scope of works

In this research our aim is to create some deep learning methods by making use of pre-existing convolutional neural network. The proposed systems will aim to accurately and efficiently count the number of cells in microscopic images. By integrating YOLO, we will be able to effectively detect and localize cells in the images, even in the presence of cluttered background and varying cell shapes and sizes. The performance of the two models will be evaluated in terms of accuracy and speed. The ultimate goal is to contribute to the advancement of cellular analysis by providing a reliable and efficient cell counting solution for researchers in the field of biology and medical science.

1.3 Organization of the thesis

This thesis is organized in six chapters as below:

I. In **Chapter 1** – we present an introduction to the objective. We talk about what our work consists of and the final goal of all the study.

II. **Chapter 2** consists of a literature review in medical imaging and the challenges faced during working on the thesis.

III. **Chapter 3** presents the neural networks, computer vision techniques and other methodologies we have used in this work.

IV. **Chapter 4** includes the necessary methodologies in supporting our thesis. We describe the dataset, the model architecture and also the implementation.

V. **Chapter 5** describes the results obtained from the implementation.

VI. **Chapter 6** concludes the interpretation of the results and explores the future work and research of this topic.

CHAPTER 2

2. LITERATURE REVIEW

2.1 Medical Imaging

In order to achieve an early detection, diagnosis but also treatment of different diseases, over many years there have been introduced different medical imaging techniques such as computed tomography, magnetic resonance imaging, positron emission tomography, ultrasounds, X-Ray etc. The results and interpretations are usually done by the experts of the fields. The reason we had to rely on the professionals was because of the knowledge they possess about the target domains. But lately, the researchers and medical professionals have started to gain from computer-assisted interventions due to the huge variances in pathology and the probable exhaustion of the human expertise. Given the introduction of many machine learning techniques, computational medical image processing is surely progressing.

2.2 Challenges

The main challenge of this research was the labelling of the images. We started with a raw dataset of cell images had to be processed and labelled manually along the contours of the nuclei. We experimented with many techniques and open-source applications to understand and see what would be the best fit for our type of images and models.

Because of the fact that the density of cells can be quite high, we often observed close proximity of them and this made it harder to differentiate from one another. Also, in some of the cases, it was not very easy to distinguish the nucleus properly. This was related to some of the cells appearing to have “burst”.

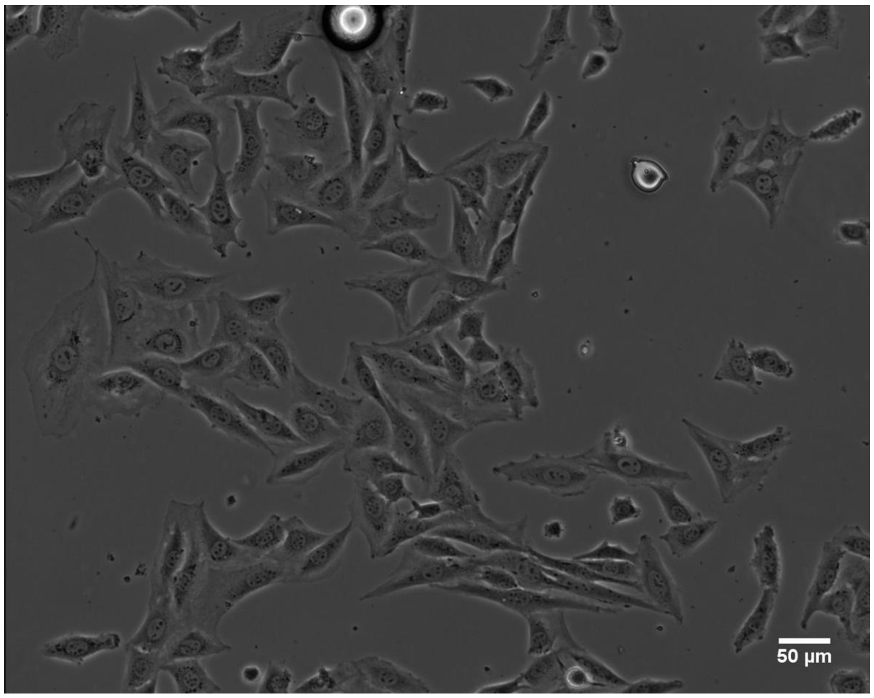

Figure 1- Cell Image example

Another challenge would be the resolution and contrast of the images that were used for our dataset. This can lead to a lower accuracy of the model, as it can produce false positive or false negative results when detecting the cells.

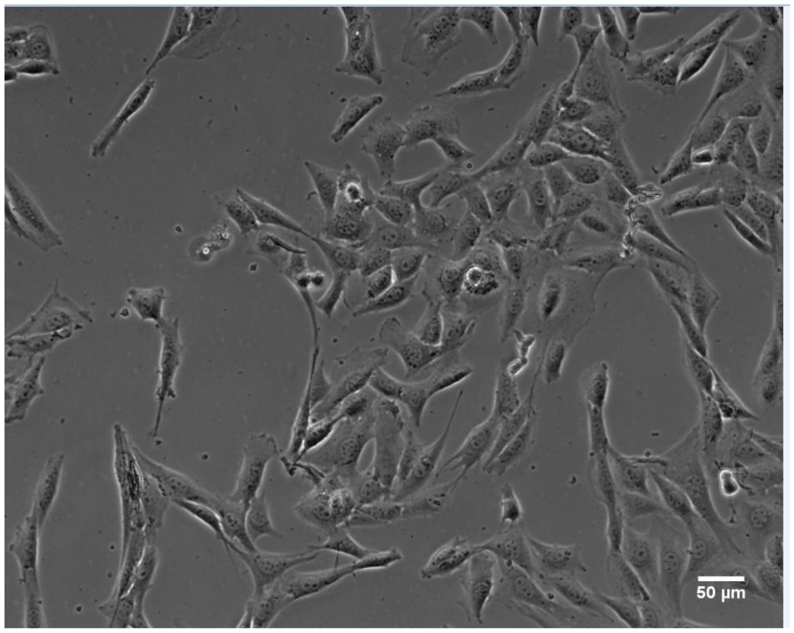

Figure 2- Cell Image with crowded cells

2.3 Image Analysis and Neural Networks

Convolutional Neural Networks (CNNs) have been used extensively for image processing in a variety of disciplines, including security, surveillance, deep fake identification, autonomous driving, space exploration, sign language translations and counting. Medical diagnostics has benefited greatly from the innovative designs of Convolutional Neural Networks. These implementations make different tasks like cancer or anomalies detection easier and rapider.

In a CNN, the input image is processed through multiple layers, each of which performs a specific function, such as feature detection or classification. The key to a CNN's ability to analyze images is its use of convolutional layers, which are designed to detect specific features in the image. These convolutional layers scan the image, looking for patterns and features, and then use this information to make decisions about the image. One of the key advantages of CNNs is their ability to reduce the dimensionality of the input data. This makes them more efficient and less prone to overfitting, which is a common problem in other types of neural networks. Another advantage of CNNs is their ability to handle data of varying scales, such as images with different resolutions or orientations. This makes them well-suited to a wide range of image analysis tasks, including object recognition, image classification, and segmentation.

In recent years, CNNs have achieved state-of-the-art results on many image - analysis tasks, outperforming traditional computer vision techniques. They have been applied to a wide range of domains, making them a crucial tool for the field of image analysis.

CHAPTER 3

3. NEURAL NETWORKS

3.1 Artificial Neuron

The first introduction of the Artificial Neurons was done in 1943 by McCulloch and Pitts in “A logical Calculus of Ideas Immanent in Nervous Activity”. Their presentation was very closely related to how the human nervous system (and neurons) works. The calculations are done by what we call propositional logic. It wasn’t until the 1990s when there was really a rise of interest in Artificial Neural Networks. [16]

The artificial neuron can have more than one inputs and only one output. The inputs can contain different values of information and when they are activated, the output information is activated as well. Many neurons can be organized together in the most basic, simple Artificial Neural Network (ANN), called a perceptron. Each input value is connected and multiplied to the weight. Weight starts as a randomly generated number. The result after the multiplication is passed to an activation function in the body. Then finally the product is outputted.

Many perceptrons can form a Multi-Layered Perceptron, the basic form of Deep Neural Networks. Different from what we saw above, here the middle layers that remain hidden are introduced.

First, the forward passing prediction is made by the backpropagation method. It reverses through each layer, measuring the contribution of each connection's error (reverse pass). The connection weights are slightly adjusted to lessen the inaccuracy. This is also called ‘tweaking’. [16]

The backpropagation technique can be combined with various activation functions. Some notable activation functions include:

1. *The Sigmoid (Logistic) Activation Function* $S(z) = 1/ (1 + e^{z)}$
   A S-shaped , differentiable, monotonic function that exists between 0 to 1. It is mostly used for those cases when there is a need to predict a probability of the output.

2. *The hyperbolic tangent function tanh (z) = 2σ(2z) – 1*

   It is S-shaped, very similar to the Sigmoid activation function. Any real value can be used as an input and it has an output value that ranges from –1 to 1. This range contributes to causing each layer's output to be almost normalized. The larger the input, the closer the output will be to 1. The hyperbolic tangent function is a continuous function.

3. *The Rectified Linear Activation Unit Function – ReLU (z) = max(0,z)*

   It is also continuous, but not differentiable at the point z = 0 (Gradient Descent bounces around since the slope changes abruptly). It returns 0 for any negative value as input, and the if the input is a positive value, it is returned as an output as well. This function is quick to be calculated. It does not have a maximum output value and this can help reduce some issues during Gradient Descent [32]. But because of the negative values input producing a 0 output, the ability of the model to train properly is decreased.

4. *Leaky ReLU – f(z) = max(0.01*x, x)*

   An attempt to solve the problem mentioned above. When the value is not 0.01 it is called Randomized ReLU. Range of this function is –infinity to infinity. It is also monotonic, together with its derivative.

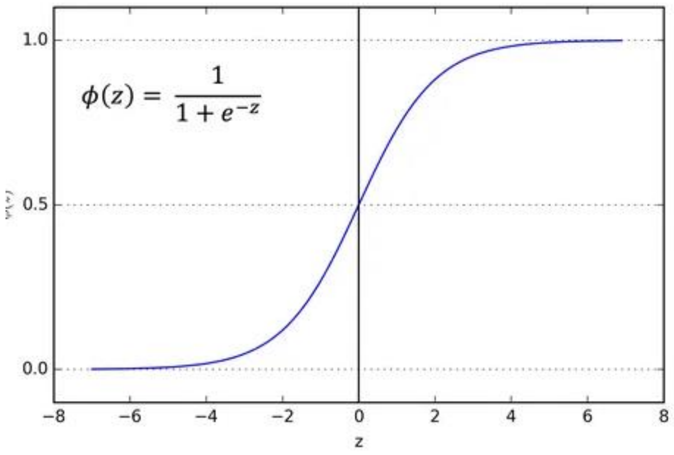

Figure 3- Sigmoid Function

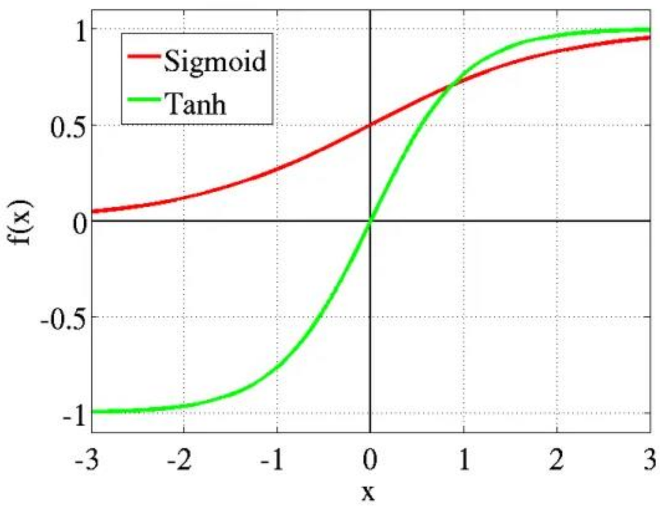

Figure 4 - Sigmoid vs Tanh Function

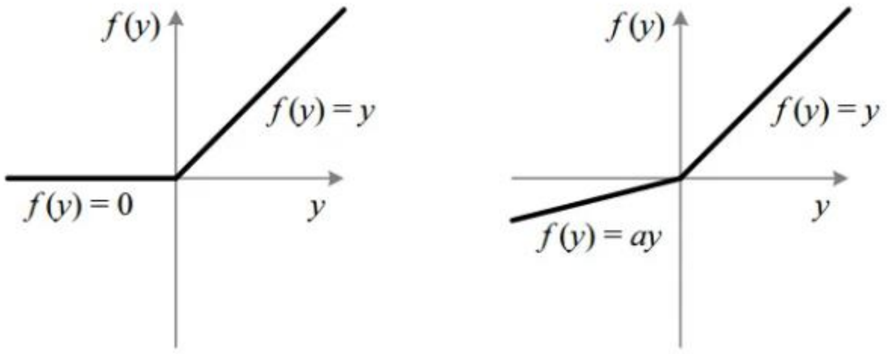

Figure 5 - ReLU vs Leaky ReLU

3.2 Convolutional Neural Network Layers

The input layer of CNNs serves the main goal of initializing the input image data, in order to make its dimensions zero-centered. The scale of the input data is then normalized in between [0,1] to accelerate the converging speed and decreases the redundancy by whitening the data.

The core of the CNNs is actually the *Convolutional Layer*. A CONV kernel filter slides on the original image. It then multiplies the value of each pixel of the data and adds them as the convolutional result. This rule is called a convolution [2]. Additionally, a technique known as shared weight is used to filter various portions of an image using the same CONV kernel, allowing neutral cells with the same feature to be identified and categorized as belonging to the same object category. The Kernel size, depth, stride, zero-padding, and filter number are a few examples of parameters. The following is the output size calculating algorithm:

$$H_{out} = 1 + \frac{H_{in} + (2*pad) - K_{height}}{S} \qquad W_{out} = 1 + \frac{W_{in} + (2*pad) - K_{width}}{S}$$

where H_{out} is the output height, H_{in} is the input height, K_{height} is the Kernel height (same applies for the width).

The next layer is the *Active Layer*, which makes the result of the Convolutional layer nonlinear. It solves the vanishing gradient problem of underfitting. Sigmoid, Tanh, ReLU, Leaky ReLU and more functions can be used for it, but Leaky ReLU is the most used one. [2]

The Pooling Layer decreases the dimension of results obtained from the Convolutional Layer. It is located between two convolutional layers. Three different pooling techniques exist: general pooling, overlapping pooling, and spatial pyramid pooling (SPP). The width of the general pooling typically coincides with stride. The two common methods are maximum pooling and average pooling. Typically, the stride is wider than the overlapping pooling. Any size image's convolutional features can be converted into the same number of dimensions using SPP. This SPP benefit enables CNN to handle a variety of image types while also preventing information loss due to cropping and warping.

Often, the last layers of CNNs are the *Fully Connected Layers*. They send the data processed to the output while simplifying and speeding up the calculation.

3.3 Computer Vision Techniques

Because it straddles a number of research and development domains, including computer science, physics, mathematics and biology, computer vision can be difficult to define. The fundamental function of computer vision is the automatic information extraction from digital images. [32] Making sense of images—that is, separating meaningful, semantic information from images —is a key objective of computer vision (objects present in images, their position and quantity). Several sub-domains can be created from this general issue:

1. *Object classification*

This is the simplest technique of computer vision and has a main goal to classify the image into one or more different categories. An object classifier takes an image as input and can differentiate between different objects in the picture. It would for example say that there are cars present in the image, or cells, or people. But it is limited and cannot elaborate in any more details about the data present, such as how many people are there, or the car color and its position.

2. *Object detection*

It makes use of image classification to detect the objects in visual data. It is used to identify the objects inside bounding boxes and also extract the type of the objects in an image.

3. *Semantic Segmentation*

It classifies every pixel of the image to specify what type of objects is in it. In other words, every pixel plays a role. It does this without differentiating between the object instances. Semantic segmentation is also defined as the task of clustering parts of an image together which belong to the same object class.

4. *Object and instance segmentation*

Similar to semantic segmentation but on a more complex level, instance segmentation can categorize the objects in an image at the pixel level. It implies that Instance Segmentation can group comparable object types into various groups. For instance, if the image consists of different vehicles, semantic segmentation will allow us to recognize that there are different cars, whereas instance segmentation will allow us to identify the cars according to their color, shape, etc.

We define a *feature* in computer vision as a piece of information that is taken from data that is related to the task at hand and is frequently formally expressed as a one- or two-dimensional vector. Features include some prominent visual elements, distinct edges, patches, and more. The features should be easy to be distinguished and extracted from new, other images. And they also must contain all the necessary information that is needed for more recognition.

Convolutional Neural Network can handle multidimensional data. It uses three-dimensional data (height, width, and depth) as input for images and arranges its own neurons in a similar volume.

This leads to a unique feature of CNNs: each neuron in a CNN only has access to select components in the nearby region of the previous layer, as opposed to fully connected networks, where neurons are connected to all elements from the previous layer. This area is referred to as the neurons' **receptive field**, and it is typically square and encompasses all channels (or the filter size) [32]

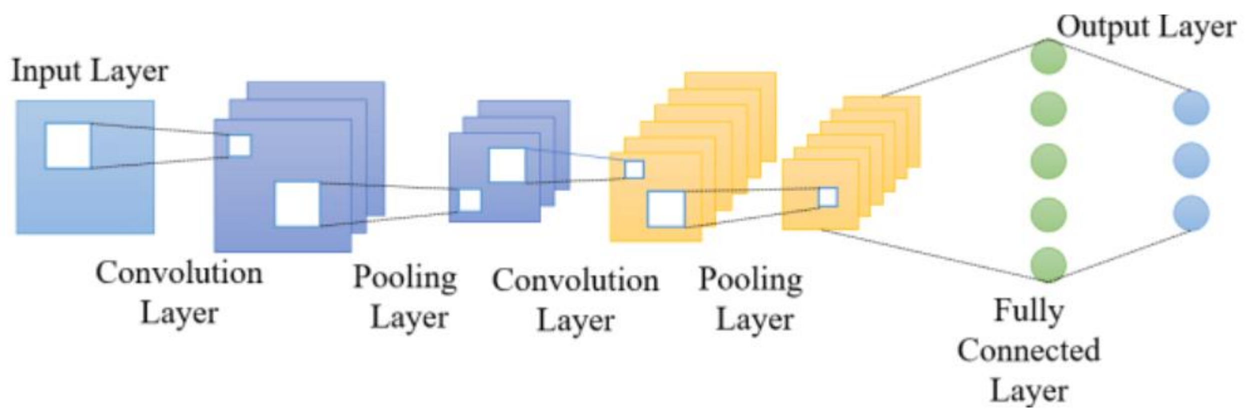

Figure 6 - Receptive field and layers of CNN

3.4 Evaluating Model's performance

When working with neural networks, something we encounter in every research is the *Mean Average Precision* (mAP) metric. We use mAP to analyze the performance of object detection and segmentation systems. The formula of mAP is based on: the Confusion Matrix, Intersection over Union (IoU), Recall and Precision.

In order to understand the Confusion Matrix, we need to be introduced to the basic four elements of it:

- *True Positives* (TP) – The model has predicted a label which matches correctly as the ground truth
- *True Negatives* (TN) – The model did not predict a label which is not a ground truth
- *False Positives* (FP) – The model predicted a label, which is not part of the ground truth (Type I error)
- *False Negatives* – The model did not predict a label, which is a ground truth (Type II error)

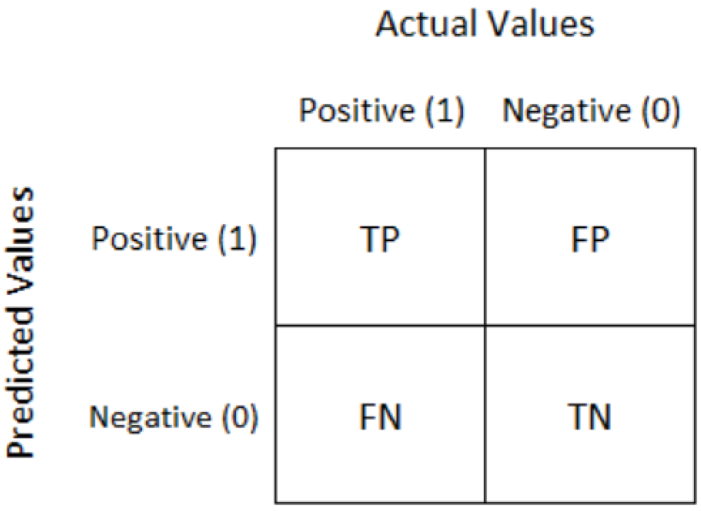

Figure 7- Confusion Matrix

True and false positives are defined by the number of predictions matching or not matching the ground truth boxes. Intersection over Union (IoU), also known as the Jaccard index, displays the overlap of the predicted bounding box coordinates to the ground truth box. Higher IoU implies a closer match between the predicted and actual bounding box coordinates.

$$IoU(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A + B| - |A \cap B|}$$

A and B are the number of elements that each set contains. $A \cap B$ is the intersection of the two sets, otherwise meaning the number of elements they have in common. $A \cup B$ is the union of the sets and therefore $|A \cup B|$ represents the total number of elements the two sets cover together.

The ratio of IoU varies from 0 (the sets/boxes do not overlap at all) to 1 (the sets/boxes overlap entirely).

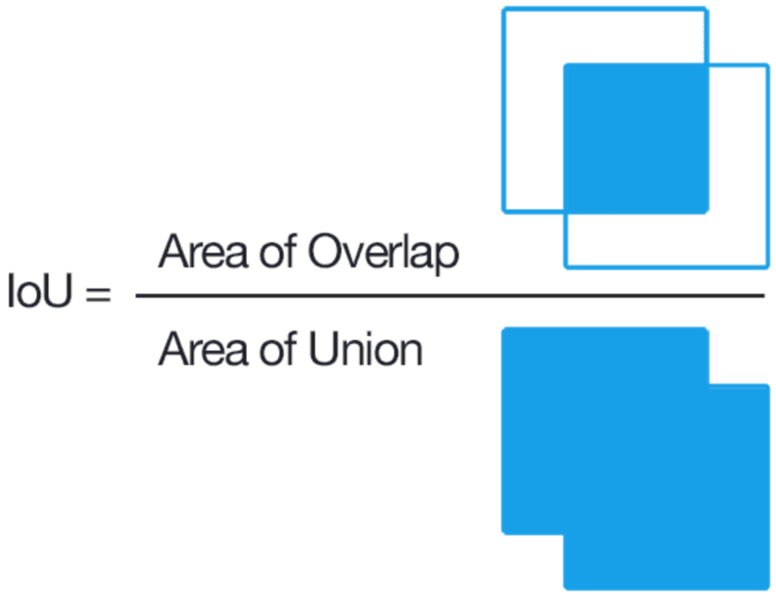

Figure 8 - Intersection over Union

Recall is calculated as the ratio between the TP to the total number of samples. What Recall is actually measuring is the model's ability to detect the positive samples. The higher this value is, the better the model is considered to be.

$$Recall = \frac{TP}{TP + FN}$$

Precision value measures how well the model can find the true positives out of all positive predictions. In other words:

$$Precision = \frac{TP}{TP + FP}$$

Since sometimes it can be difficult to have a clear comparison of models with low precision and high recall (or vice versa), F-score is used to help us measure it in a more efficient way. It uses Harmonic Mean in place of Arithmetic Mean by punishing extreme values more. F1 is designed to work well on unbalanced data.

$$F1 = \frac{2\ x\ precision\ x\ recall}{precision + recall}$$

Finally, coming back to the mAP, it is calculated as the weighted mean of precisions at each threshold; the weight is the increase in recall from the prior threshold. Depending on different contexts, its interpretation can also vary.

While average precision gives information about the performance of a model for a single class, we use mAP to get a global score. This corresponds to the mean of the average precision for each class. mAP summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight:

$$AP = \sum_{n} (R_n - R_{n-1})P_n$$

P_n and R_n are the precision and recall respectively at the nth threshold.

CHAPTER 4

4. METHODOLOGY

4.1 YOLO

YOLO is a real time object recognition algorithm that is based on regression. This means that instead of selecting interesting parts of an image, it predicts classes and bounding boxes for the whole image in one run of the algorithm.

Each bounding box contains four descriptors that are: the center of the bounding box, the width and height, the value that corresponds to the class of the object. The network divides the image into regions and predicts the probabilities for each of them. Each grid cell in YOLO will have an associated vector in the output that tells if the object exists in that grid, the class of the object (which in our case is the cell nucleus) and the predicted bounding box for that object.

YOLO makes use of only convolutional layers and is invariant to the size of the input image. Candidate boxes from images are extracted directly by the network and then object detection is performed through the entire image features.

YOLO surpasses CNN's maximum speed cap and achieves an outstanding balance between speed and accuracy. It is extremely fast while outperforming state-of-the-art techniques like Faster R-CNN. Secondly, YOLO uses a global reasoning approach and encodes contextual data about the image. False positives on background are therefore less likely to be predicted.

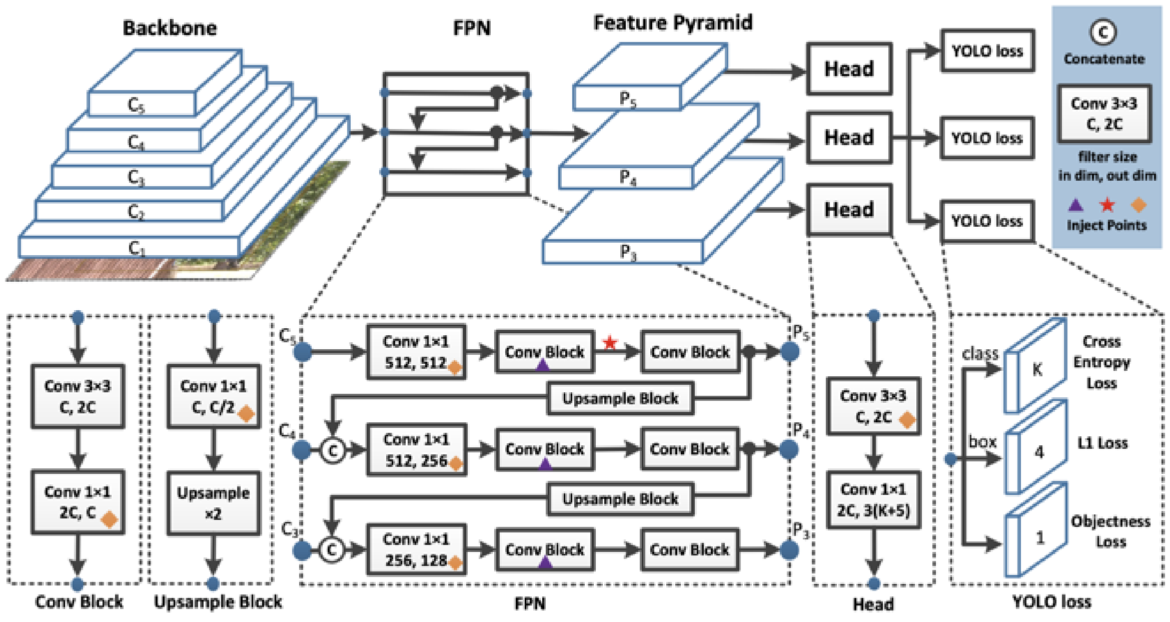

Figure 9- Basic YOLO Architecture (Adapted from [26])

A Fully Convolutional One-stage Object Detection uses pixel-wise prediction for object detection.

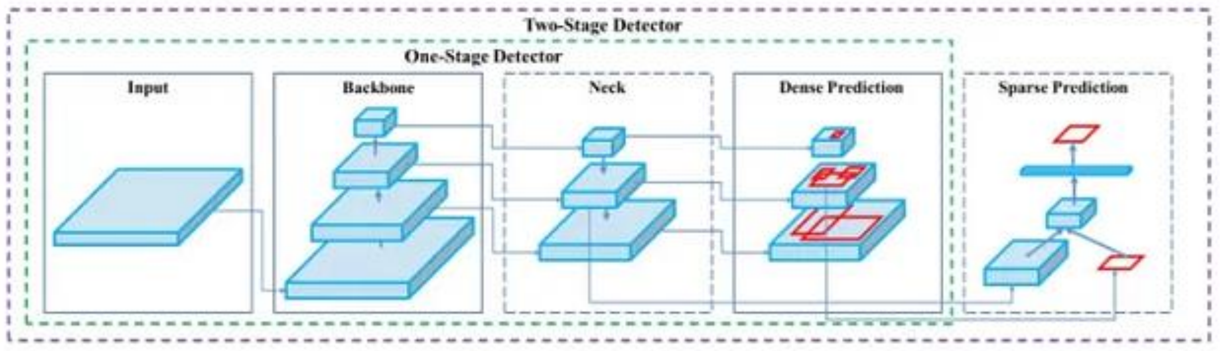

Figure 10 – One Stage Detector

The deep learning architecture that serves as a feature extractor is called the **Backbone**. In essence, each backbone model is a classification model. There are a few models like *VGG16*, *SqeueezeNet, MobileNet and ShuffleNet* that can be utilized in the backbone. They are all intended only for CPU training. Its design has a critical influence on the inference efficiency because it carries a high portion of computation cost [21].

The feature aggregator, also known as the **Neck**, is a subset of the bag of specials. It can collect feature maps from different stages of the backbone. It builds up the pyramid feature maps.

Head is made up of different convolutional layers. The final detection result according to the features passed by the neck, is predicted in the end. The head can be categorized in two ways: anchor based and anchor free. What this really means is a structural perspective, a parameter-coupled head or parameter-decoupled head [21].

4.2 YOLOv5

YOLOv5 was released in June 2020 and is available in five models: n, s,m,l,x. Each letter corresponds to: *nano* – extra small model, *small* model, *medium* size model, *large* model and *extra-large* model.

The detection accuracy and performance of them differs and can be better observed in the graph below:

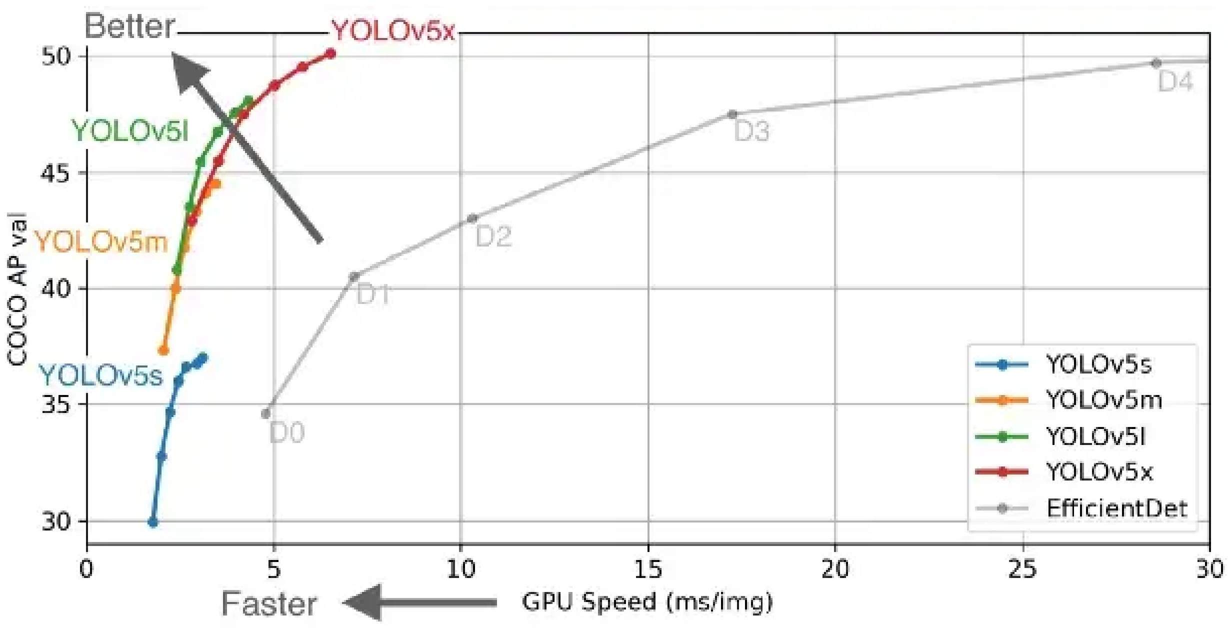

Figure 11 - Comparison of YOLOv5 models (Adapted with permission from [23])

The difference of the five models is the number of layers and the parameters. But no difference in terms of operations was observed.

YOLOv5 supports also instance segmentation. This is achieved by introducing a fully connected neural network, the *ProtoNet*, in addition to the object detection head. Prototype masks are produced by *ProtoNet* for the segmentation model, in a similar way to a Fully Connected Network.

All YOLOv5 models contain a *CSP-Darknet53* as a backbone, *Spatial Pyramid Pooling* (*SPP*) and *Path Aggregation Network (PANet)* as a neck and the head used is the same as the one in YOLOv4. The activation functions are the Sigmoid Linear Unit and the Sigmoid Activation Function.

It has been observed that YOLOv5 is faster to train than its ancestor YOLOv4. The model is also considerably lighter. It is however worthy to be noted that both YOLOv4 and YOLOv5 have the same mAP, at least in the benchmark and studies made by Roboflow.

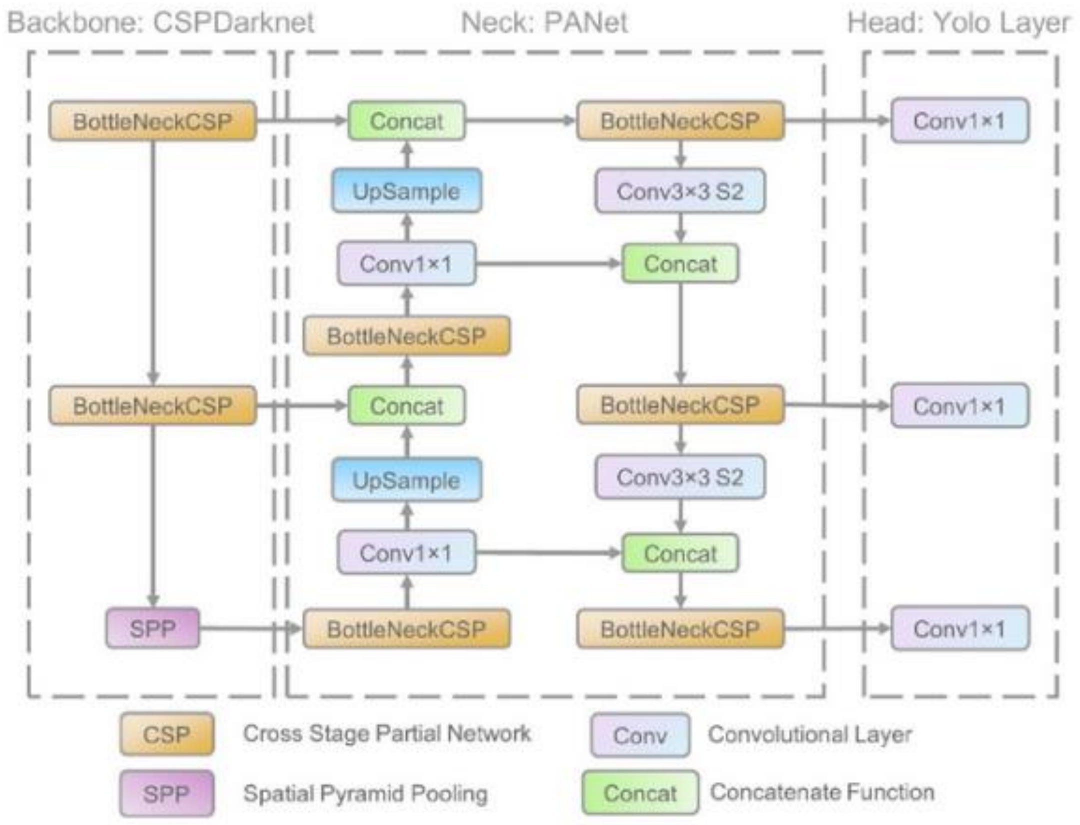

Figure 12 - Network architecture of YOLOv5 (Adapted from [23])

4.3 YOLOv6

The network design, label assignment, loss function, data augmentation, business-friendly changes, quantization, and deployment are the components that make up the renovated YOLOv6 design. [21] It uses a *Varifocal loss* (VFL) for classification and *Distribution Focal loss* (DFL) for detection.
Focal loss gives rise to VFL. This indicates that it already handles and weighs difficult and simple cases differently during training. The importance of the positive and negative examples is also handled differently by VFL. The learning signals from the two samples are balanced as a result.

DFL is used for box regression loss in the YOLOv6 Medium and Large models. The continuous box location distribution is viewed by DFL as a discretized probability distribution.

When the borders of the ground truth are hazy, it is particularly useful for detection. It was also tried using DFLv2, which added a lightweight sub-network. However, this required additional computations, and no improvements above DFL were noted. Therefore, DFL is still used as the localization loss function.

In YOLOv6 there have been also *introduced longer training epochs*, *quantization* and *knowledge distillation*. YOLOv6 uses knowledge distillation to increase the models' accuracy. This is also possible without having a significant computation expense. A teacher model is used in knowledge distillation to train a student model. Along with the ground truth, the instructor model's predictions serve as soft labels to train the student model. In order to duplicate the performance of the teacher model, we essentially train a smaller and simpler model (relative to the teacher). [21]

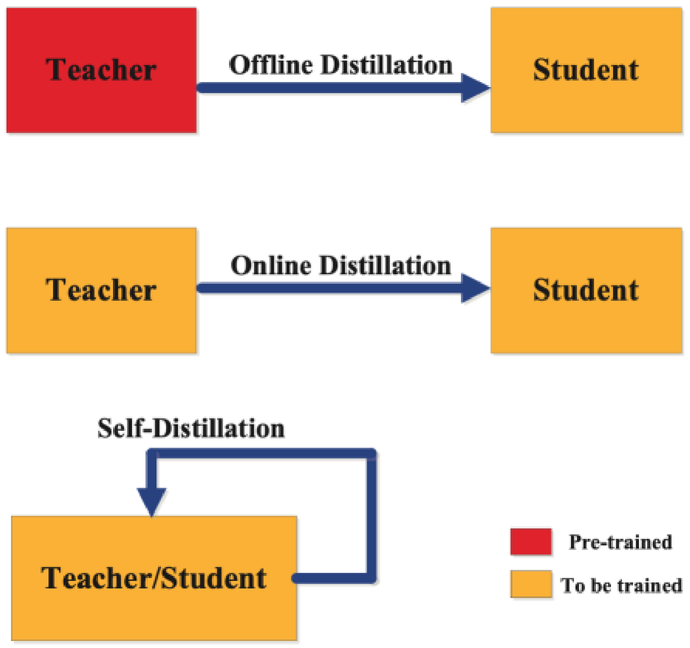

Figure 13 Teacher Student in Self-Distillation

The type of knowledge distillation will determine whether the teacher model is pre-trained or not. Additionally, the teacher model may be a bigger model or perhaps the same model. Since YOLOv6 trains users through self-distillation, the student model serves as the teacher model. But in this instance, the teacher model has already been taught. The optimization procedure reduces the KL-divergence between the student's and teacher's predictions for YOLOv6 training.

ResNets can perform better in classification performance, but they lack speed during inference. VGGs on the other hand are much faster as they contain effective 3*3 convolutions. Their downside is as expected, the accuracy. YOLOv6 models use *reparametrized backbones*. The network structure changes during training and inference. In the backbone, RepBlocks with skip connections (reparametrized VGG) are the foundation of the network for nano, tiny and small architectures. During inference, RepConv blocks are used instead.

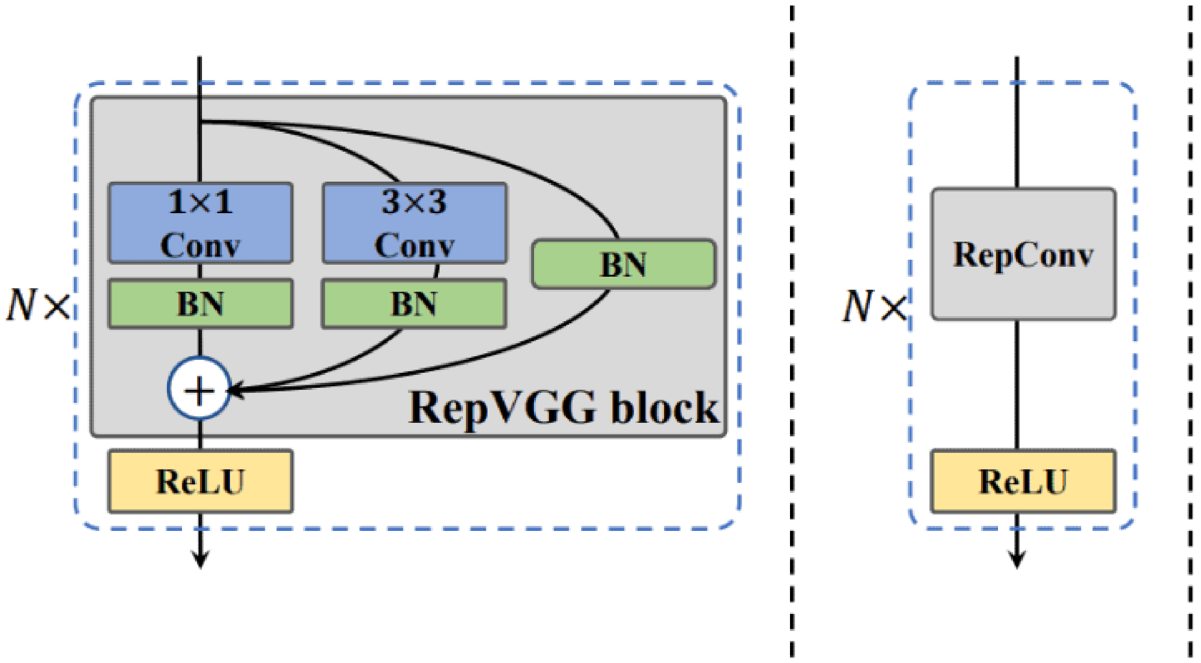

Figure 14 - RepVGG during training and RepConv during inference

In Medium and Large models of the network, the architecture actually uses reparametrized versions of the CSP backbone, also known as the **CSPStackRep**.

As for the Neck, following YOLOv4 and YOLOv5, the YOLOv6 aggregates the multi-scale feature maps using Path Aggregation Network (PAN) topology [25]. PAN is intended to boost information flow in a proposal-based instance segmentation framework [25]. In particular, bottom-up route augmentation shortens the information flow between lower layers and the topmost feature, enhancing the feature hierarchy with precise localization signals in lower layers. The use of adaptive feature pooling, which connects feature grid and all feature levels, allows for the direct propagation of important information from each feature level to the subsequent proposal subnetworks. To further enhance mask prediction, a supplemental branch is constructed that captures several perspectives for each proposal.
In order to have RepPAN, the neck is enhanced using RepBlocks or CSPStackRep Blocks.

We can also observe some changes in the head structure. To increase its efficiency, decoupled head is simplified and referred to as the **Efficient Decoupled Head**. As a result, the classification and detection branches split off from the backbone independently and do not exchange any parameters. This further cuts down on computations and also offers greater precision.

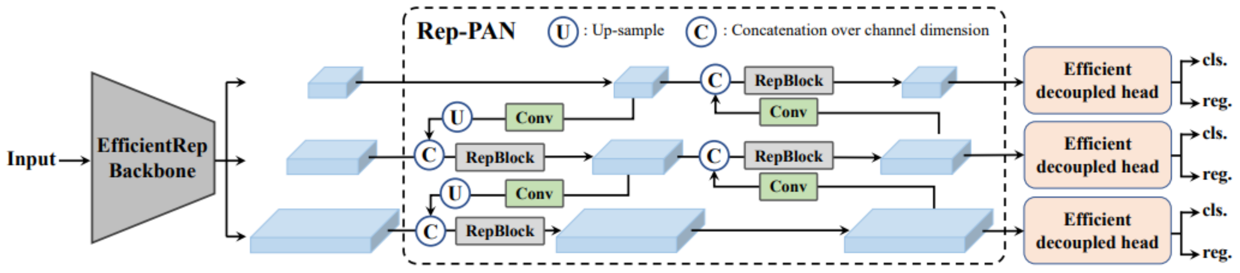

Figure 15 - YOLOv6 (N and S) Framework

We can see how the efficient decoupled head is written in our implementation code [cit yolov5]. It is optimized with hybrid channels methods.

```
  def __init__(self, num_classes=80, num_layers=3, inplace=True, head_layers=None, use_dfl=True, reg_max=16):  # detection
layer
        super().__init__()
        assert head_layers is not None
        self.nc = num_classes  # number of classes
        self.no = num_classes + 5  # number of outputs per anchor
        self.nl = num_layers  # number of detection layers
        self.grid = [torch.zeros(1)] * num_layers
        self.prior_prob = 1e-2
        self.inplace = inplace
        stride = [8, 16, 32] if num_layers == 3 else [8, 16, 32, 64] # strides computed during build
        self.stride = torch.tensor(stride)
        self.use_dfl = use_dfl
        self.reg_max = reg_max
        self.proj_conv = nn.Conv2d(self.reg_max + 1, 1, 1, bias=False)
        self.grid_cell_offset = 0.5
        self.grid_cell_size = 5.0

        # Init decouple head
        self.stems = nn.ModuleList()
        self.cls_convs = nn.ModuleList()
        self.reg_convs = nn.ModuleList()
        self.cls_preds = nn.ModuleList()
        self.reg_preds = nn.ModuleList()

        # Efficient decoupled head layers
        for i in range(num_layers):
            idx = i*5
            self.stems.append(head_layers[idx])
            self.cls_convs.append(head_layers[idx+1])
            self.reg_convs.append(head_layers[idx+2])
            self.cls_preds.append(head_layers[idx+3])
            self.reg_preds.append(head_layers[idx+4])
```

YOLOv6 model is then built with its respective head, neck and backbone:

```
def build_network(config, channels, num_classes, num_layers, fuse_ab=False, distill_ns=False):
    depth_mul = config.model.depth_multiple
    width_mul = config.model.width_multiple
    num_repeat_backbone = config.model.backbone.num_repeats
```

```
channels_list_backbone = config.model.backbone.out_channels
fuse_P2 = config.model.backbone.get('fuse_P2')
cspsppf = config.model.backbone.get('cspsppf')
num_repeat_neck = config.model.neck.num_repeats
channels_list_neck = config.model.neck.out_channels
use_dfl = config.model.head.use_dfl
reg_max = config.model.head.reg_max
num_repeat = [(max(round(i * depth_mul), 1) if i > 1 else i) for i in (num_repeat_backbone + num_repeat_neck)]
channels_list = [make_divisible(i * width_mul, 8) for i in (channels_list_backbone + channels_list_neck)]

block = get_block(config.training_mode)
BACKBONE = eval(config.model.backbone.type)
NECK = eval(config.model.neck.type)

if 'CSP' in config.model.backbone.type:
    backbone = BACKBONE(
        in_channels=channels,
        channels_list=channels_list,
        num_repeats=num_repeat,
        block=block,
        csp_e=config.model.backbone.csp_e,
        fuse_P2=fuse_P2,
        cspsppf=cspsppf
    )
    neck = NECK(
        channels_list=channels_list,
        num_repeats=num_repeat,
        block=block,
        csp_e=config.model.neck.csp_e
    )
else:
    backbone = BACKBONE(
        in_channels=channels,
        channels_list=channels_list,
        num_repeats=num_repeat,
        block=block,
        fuse_P2=fuse_P2,
        cspsppf=cspsppf
    )
    neck = NECK(
        channels_list=channels_list,
        num_repeats=num_repeat,
        block=block
    )
```

YOLOv6 is an *anchor-free detector*. This makes it stand out among other models, as a result of its optimized generalization ability and the simplification in decoding prediction results. A reduction in cost of post-processing is observed. In YOLOv6 we use anchor point-based paradigm. Its box regression branch actually predicts the distance from the anchor point to the four sides of the bounding boxes [21].

4.4 Dataset

We have accumulated many microscopic images of cells and have labelled manually each of them using the open-source annotation-tool Roboflow [22]. The main focus was the nucleus of the cell. The user can select a specific area in the image, right where the cell nucleus would be. These are our bounding boxes and we have labelled them with a C – meaning Cell.

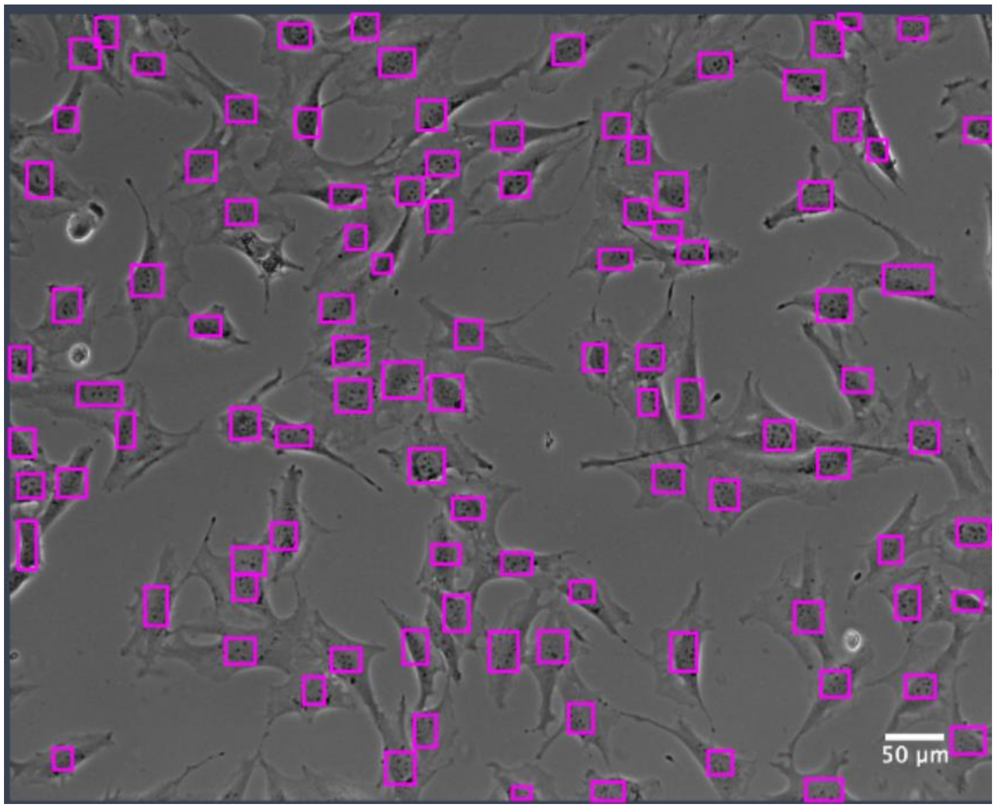

Figure 16 - Labelling of the dataset

Roboflow can divide the dataset randomly into three separate folders: Train, Valid and Test, with a default of 70%, 20% and 10% of the images respectively.

Then we can select the format of the annotations to be used for our network. We have used 90 images of size 1280x1024 for our dataset with around 10 hours of labelling. Because of the very small size of the objects we want to detect, a resizing was not done on the images. This is to ensure that the quality of the photos would not be lost.

Roboflow can then export the images and the respective annotations in our desired format (YOLOv5 or v6) with a simple API key that is added to the code, or in the traditional download way.

4.5 Implementation

The first implementation we are going to observe is the YOLOv5. We have defined only one class – C, as mentioned above. It bounds the nucleus of the cells. The other part of the image will be considered a background. We have used an existing repository from Github [23]

The network used a batch size of 64 and trained for 300 epochs. The best results were reached at epoch 194. We did the pre-processing steps and training on Google Colab and lasted for about 18 minutes.

The generic COCO pretrained checkpoint is used for the weights. Also, to enable a faster training, cache is set on.

For the second implementation with YOLOv6, we have similarly cloned the repository from Github [24]. The preparation of the dataset included the separation of the images in the train, valid and test folders. This separation was also done on the label files.

In the **data.yaml** file we have specified the path for each of the folders, number of classes (nc = 1) and the name of the class (nucleus).
For the configuration of the network structure, training settings, optimizers and data augmentation hyperparameters, we have downloaded the pre-trained YOLOv6s finetuned model. In this file we can see the different layers and types used for the backbone, neck and head. It is to be noted that this configuration also downloads the pre-trained weights from YOLOv5. Batch number in this second implementation was set to 64 as well and we trained for 400 epochs for around 51 minutes.

CHAPTER 5

5. RESULTS AND DISCUSSION

5.1 YOLOv5

This network was able to achieve a mean average precision of 73% in terms of accuracy (0.738). We can see the increase of the mAP over the epochs in the graph below. It is apparent that there are a few fluctuations in the curve. It is important to mention that even though accuracy was 73%, every cell nucleus in the cell is counted. A lower than 100% accuracy in this case indicates that the predicted bounding boxes surface is not entirely correct. But the element within it is detected.

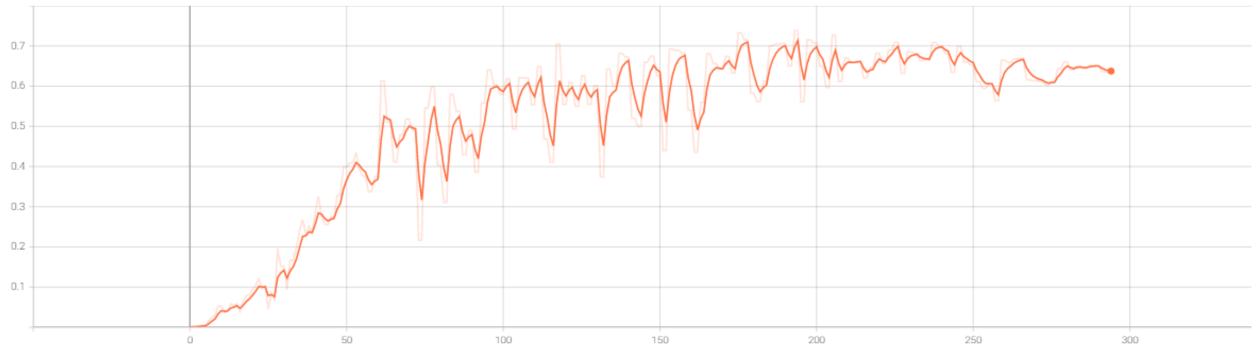

Figure 17 - YOLOv5 mAP

A similar behavior is seen in the precision curve. The final precision value for our training reached 0.763.

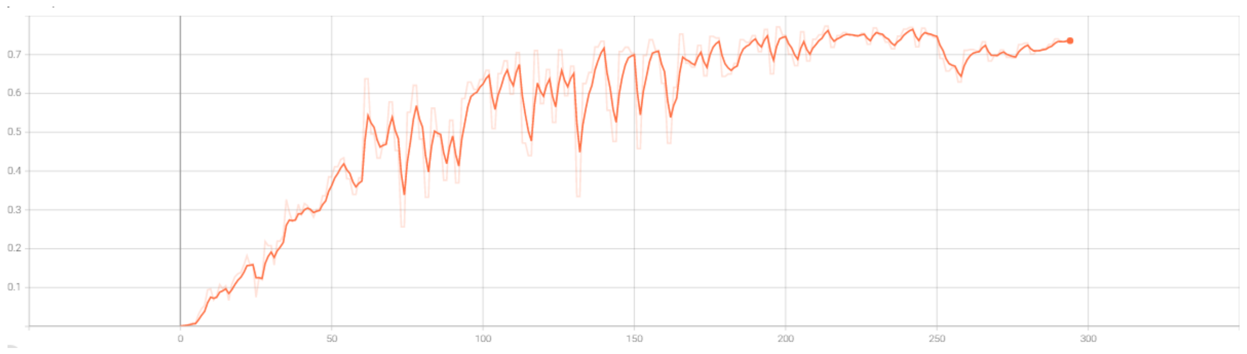

Figure 18 - YOLOv5 Precision

The fluctuations in Recall are not as big as in the previous curves. Value looks stable in the last 100 epochs or so. Recall reached an average value of 0.734.

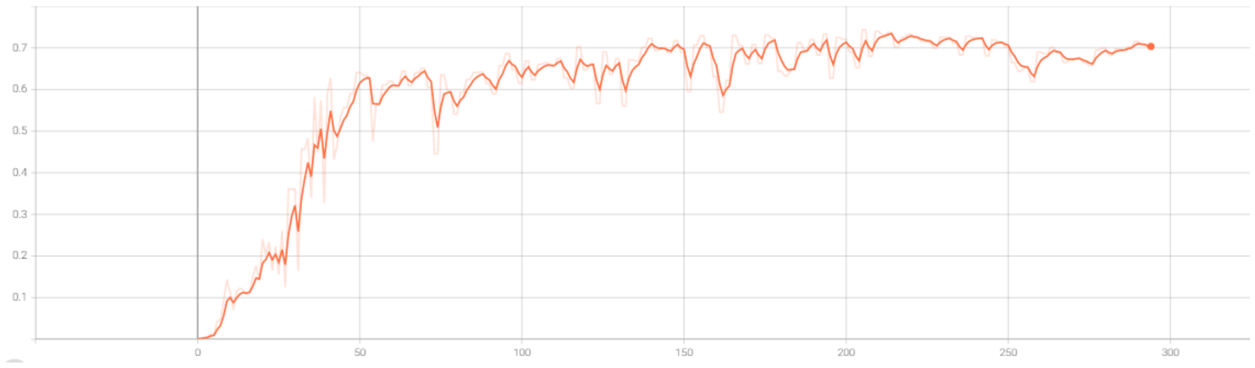

Figure 19 - YOLOv5 Recall

We can also observe the box loss for our latest experiment. This basically measures how "tight" the predicted bounding boxes are to the ground truth object. The model shows an improvement in locating the center of the object, with a final value of 0.05239 in the latest epoch.

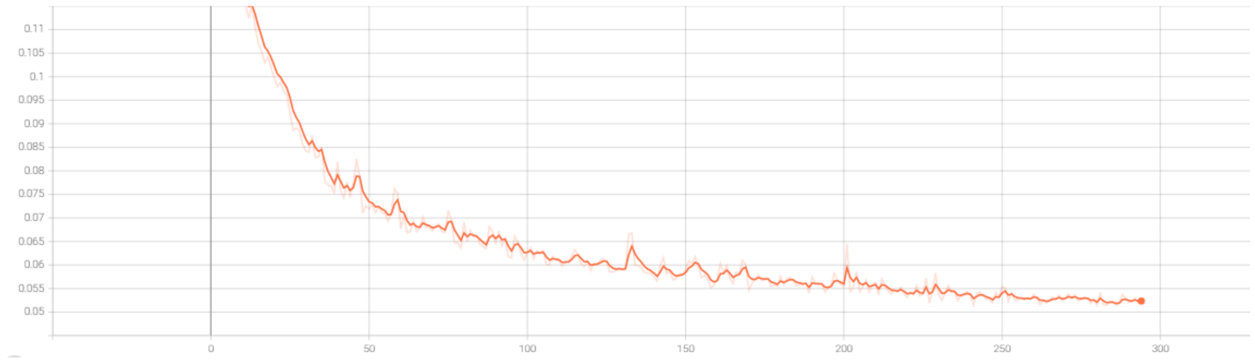

Figure 20 - YOLOv5 Box loss

Having the precision and recall values, we can now calculate the F1 score, resulting in 0.748.

mAP	*PRECISION*	*RECALL*	**BOX LOSS**	**F1 - SCORE**
0.738	0. 763	0.734	0.05239	0.748

Table 1 – *YOLOv5 Results*

We have also plotted the F1-Confidence curve that shows the best F1 score of 0.75 at a threshold of confidence of 0.416.

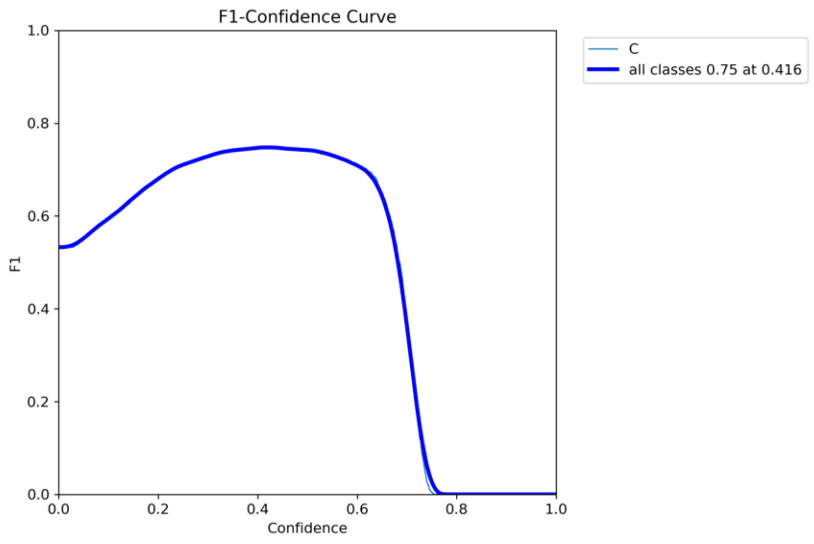

Figure 21 - YOLOv5 F1 Score

In a similar way we have also plotted the precision-confidence curve, with the highest value 1 reached at the confidence threshold of 0.770, and the precision-recall curve. This shows the tradeoff between precision and recall for different thresholds. The higher the Area under the Curve (AUC) is, the better the performance of the model is considered to be.

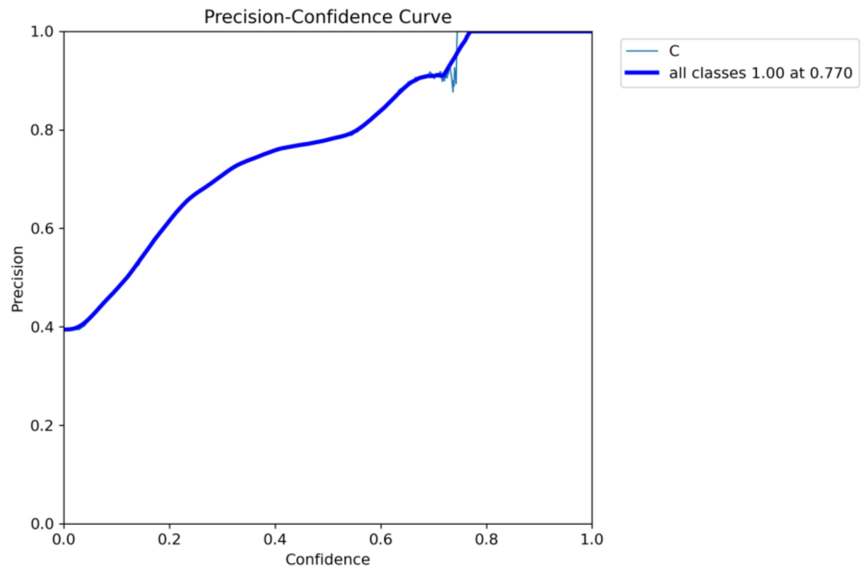

Figure 22 - YOLOv5 Precision Confidence Curve

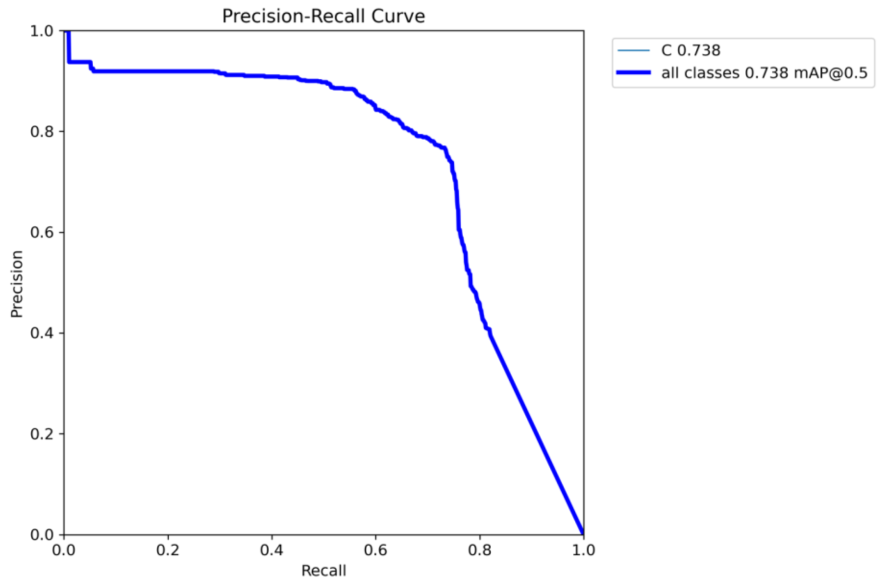

Figure 23 - YOLOv5 Precision Recall Curve

After the training, we have run the detection on some of our test images, to see the output of the model. You can see in the figures to come, the detected cells in our medical images.

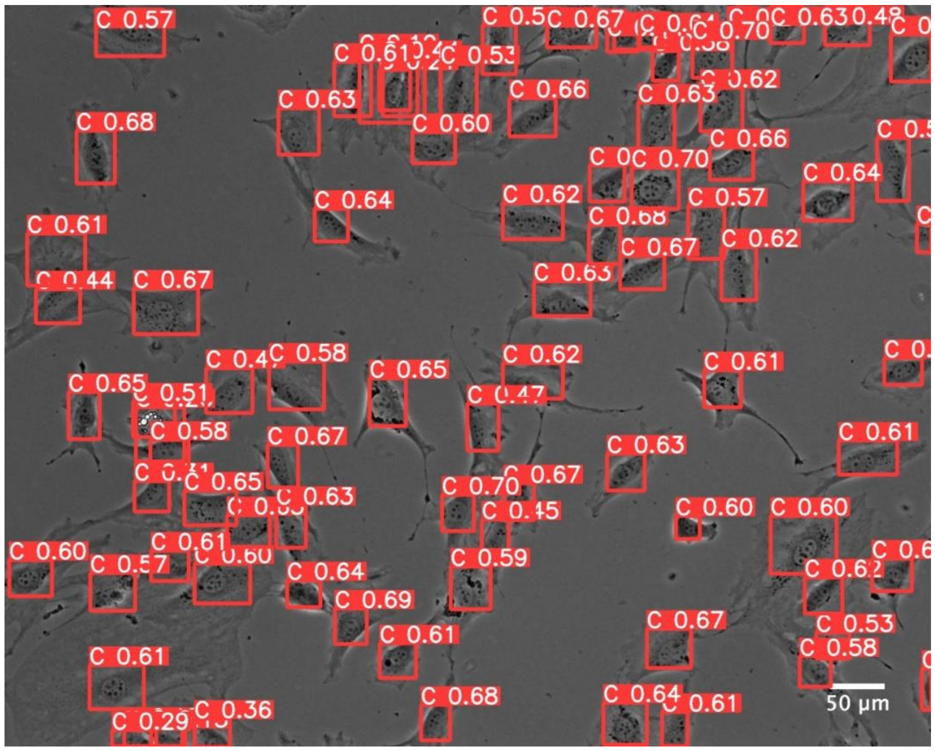
Figure 24 - Detection results in YOLOv5

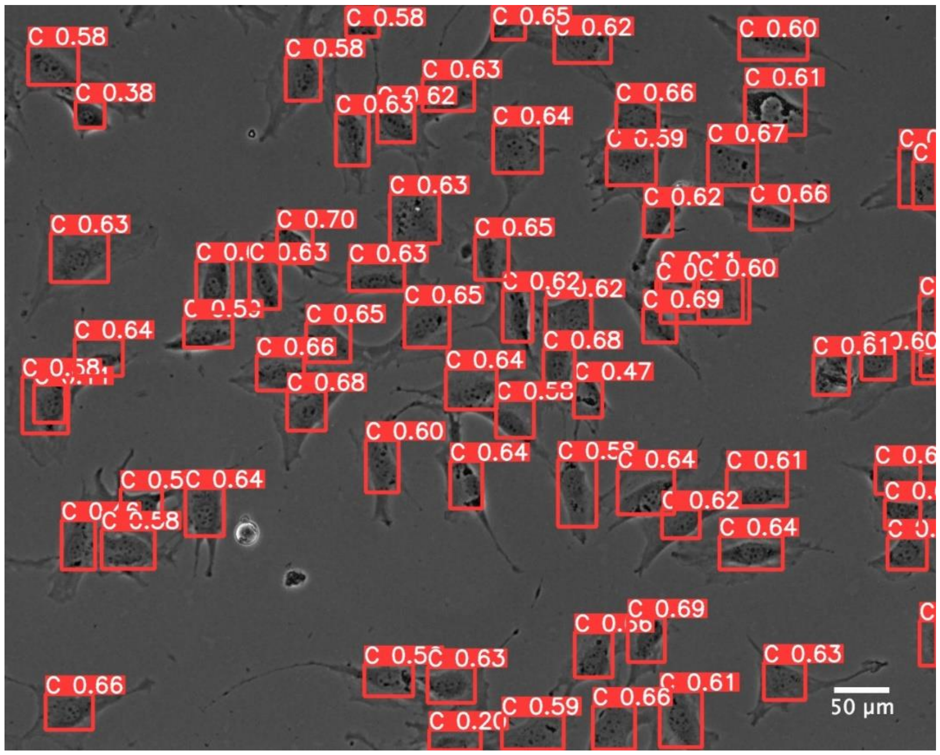
Figure 25 - Detection results in YOLOv5

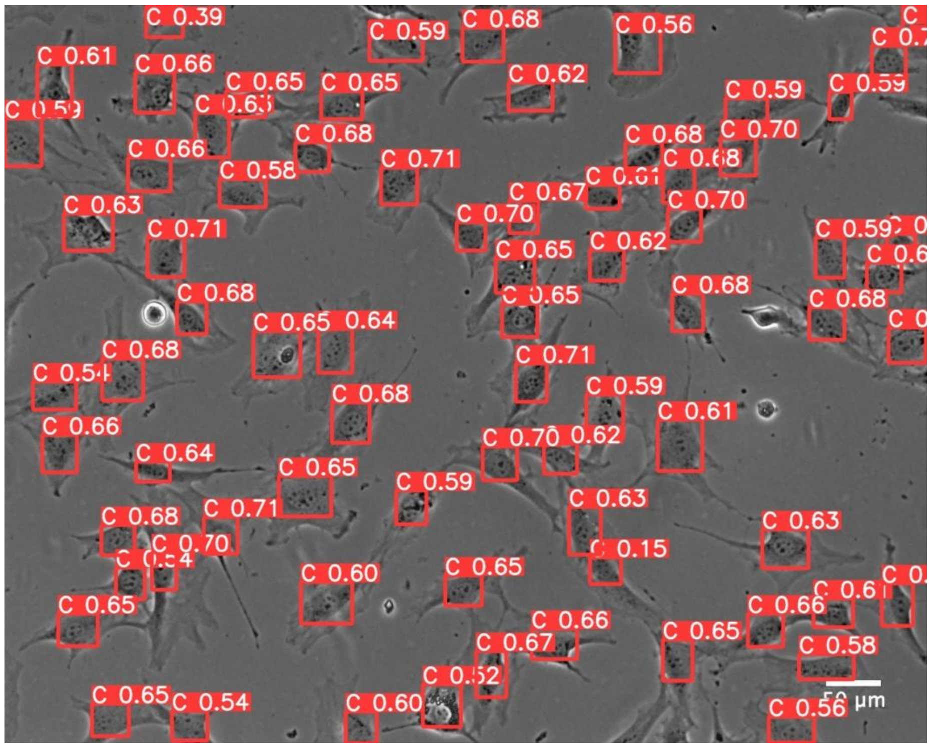

Figure 26- Detection results in YOLOv5

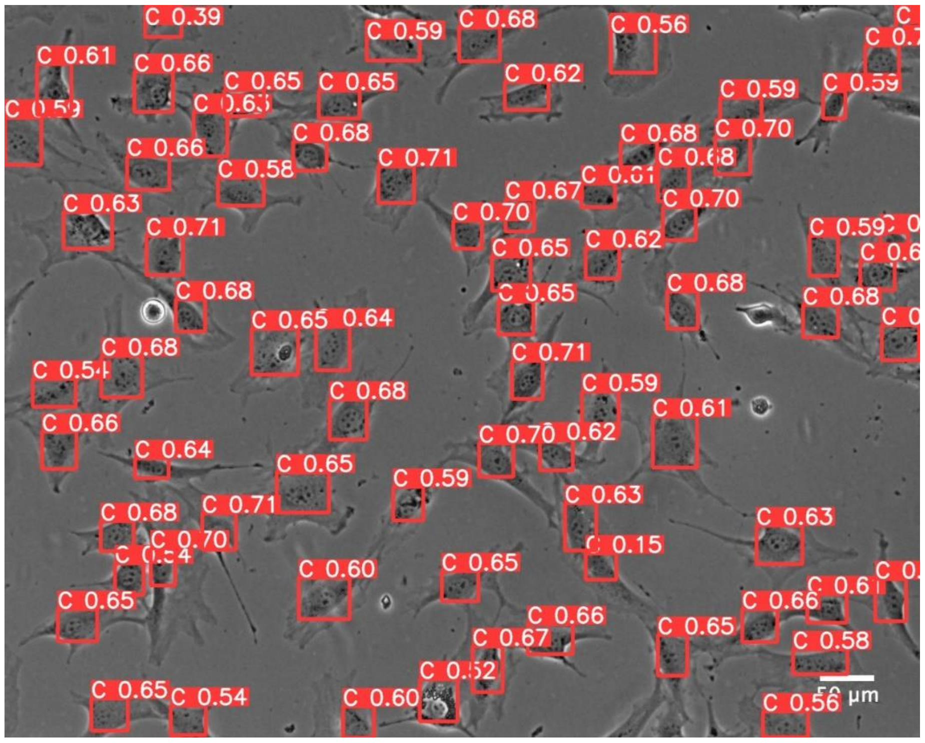

Figure 27 - Detection results in YOLOv5

We can see that the model was able to detect almost every nucleus present in the images.

5.2 YOLOv6

Class loss started off a bit high in the beginning of the training. It then experienced a decrease and appeared stable in the last iterations.

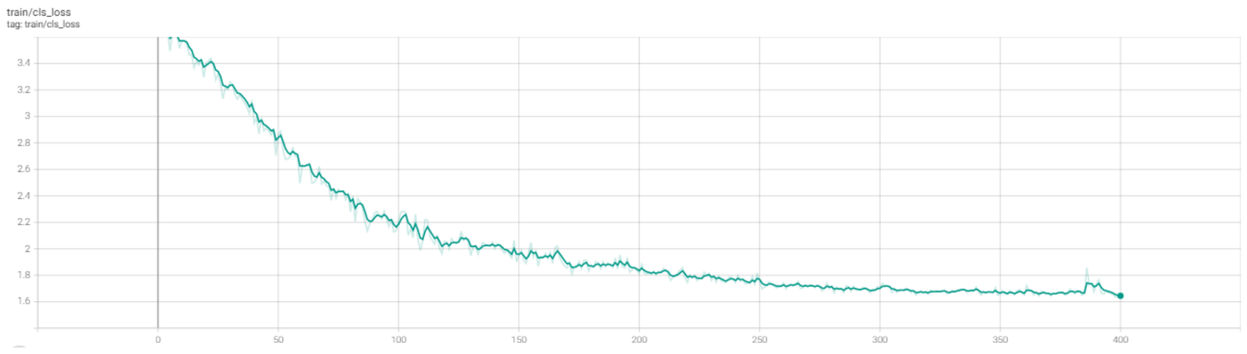

Figure 28 - Class loss YOLOv6

Here we can observe a comparison between different runs we have done of the training process (latest is in green):

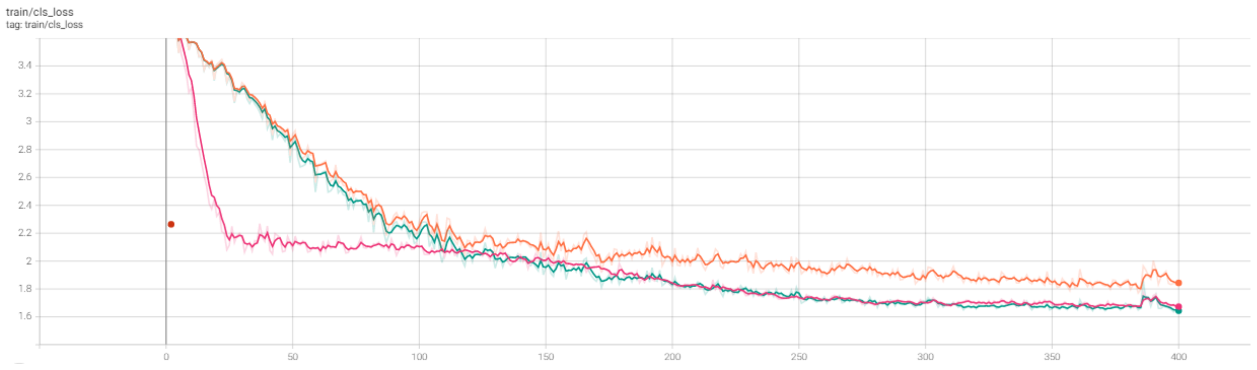

Figure 29 - Comparison in class loss

For Intersection over Union loss, we see that our latest experiment showed a bigger decrease than the others:

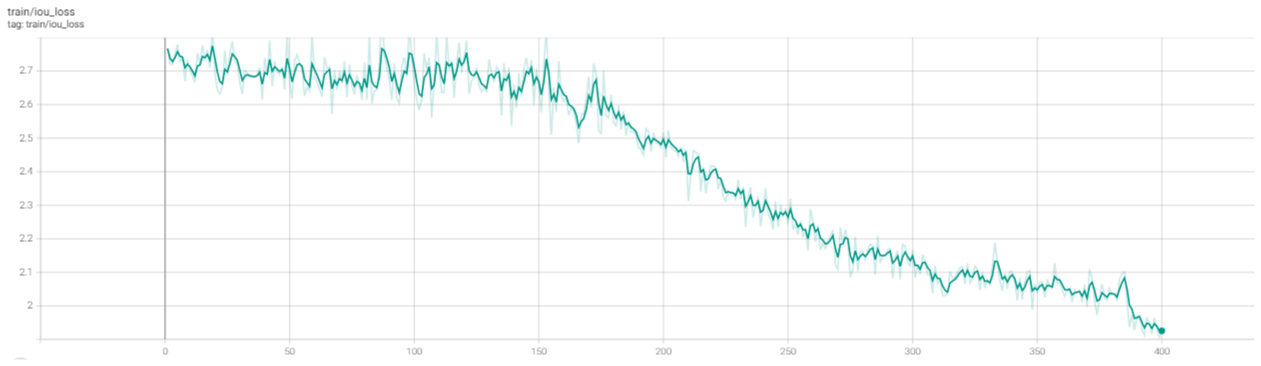

Figure 30 - IoU loss

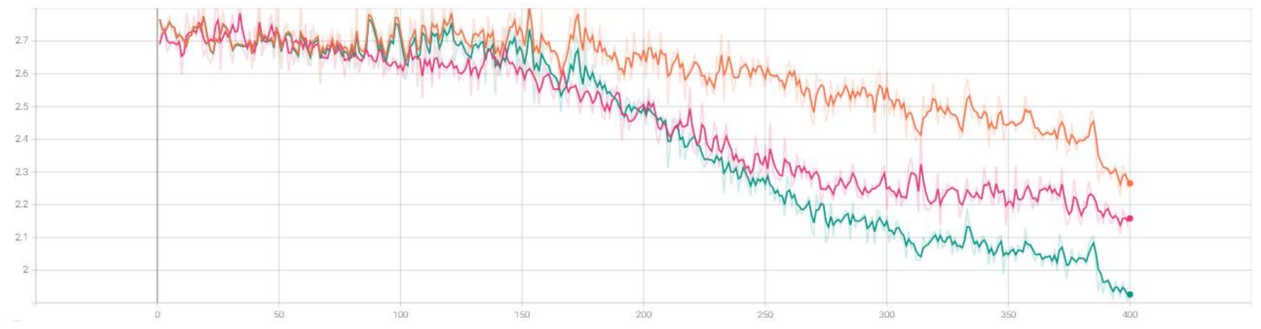

Figure 31 - IoU loss comparison

The mAP did not appear promising in the beginning. However, we observe here as well an improvement in its value, compared to the other experiments.

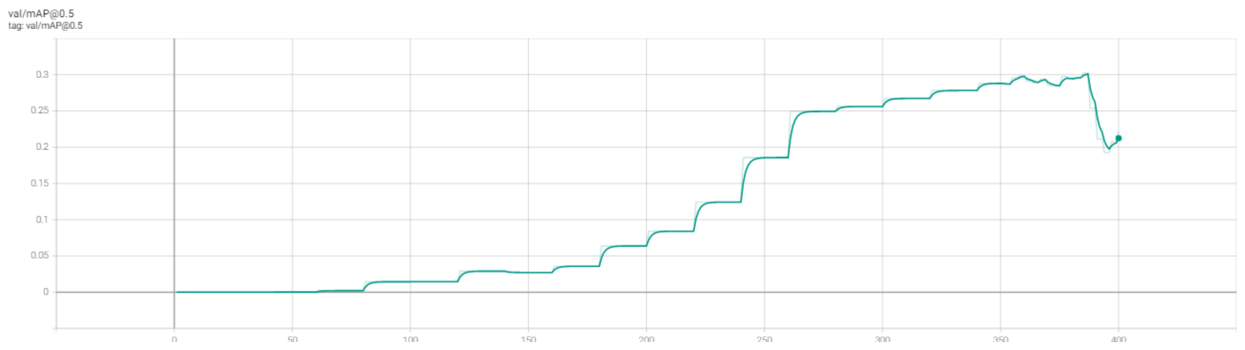

Figure 32 - mAP in YOLOv6

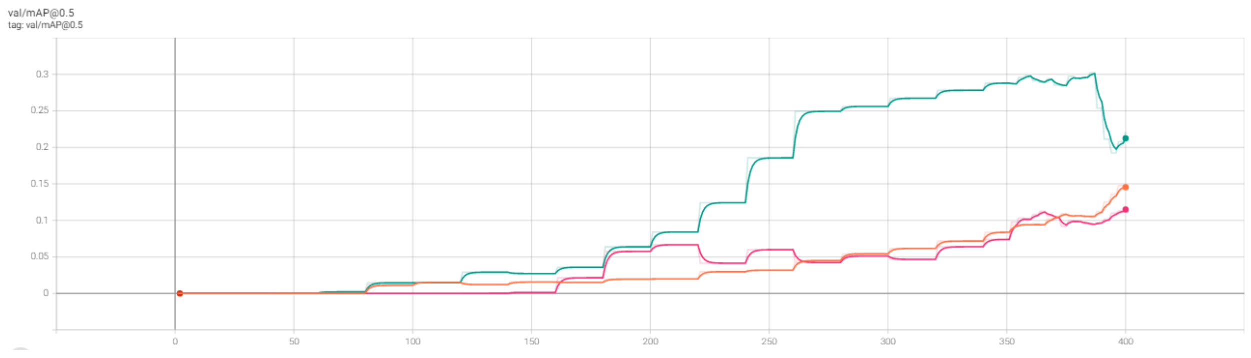

Figure 33 - mAP Comparison

After 350 epochs, mAP showed a decrease from 0.3 to 0.2.

The mAP@[0.5:0.95], showing the average mAP over different IoU thresholds (from 0.5 to 0.95), had the highest value in 0.073.

The model achieved a precision value of 0.118 and a Recall value of 0.098.

We have calculated the F1-score with the above values, resulting in 0.2141.

mAP	*PRECISION*	*RECALL*	*F1 - SCORE*
0.3	0.118	0.098	0.2141

Table 2 – *YOLOv6 Results*

Lastly in the table and chart below, we can have a clearer view in the comparison of the two models.

	mAP	*PRECISION*	*RECALL*	*F1-SCORE*
YOLOv5	0.73	0.76	0.75	0.74
YOLOv6	0.3	0.118	0.098	0.2141

Table 3 – *Comparison of YOLOv5 and YOLOv6*

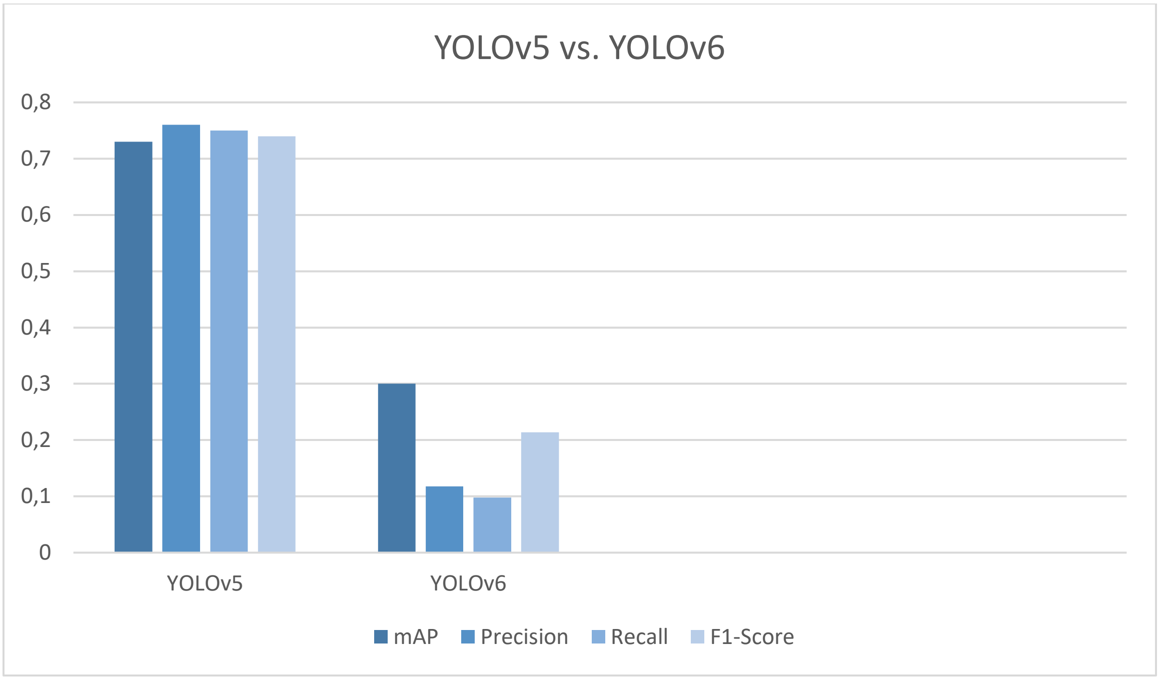

Figure 34 - Comparison between YOLOv5 and YOLOv6

CHAPTER 6

6.1 Conclusions

Through these experiments we have observed the behavior of two different YOLO models when trying to achieve a detection of cells in microscopic images.

In YOLOv5 the results show a good model for this task. The F1-score is high, meaning that in the prediction the number of false positives and false negatives were low. YOLOv5 showed a high accuracy as well, while not obtaining a lot of time to train.

The same, unfortunately could not be determined for YOLOv6. For this kind of detection, its ancestor showed more promising results, so we propose using the version 5 for cell segmentation in images. The trainings, however, do suggest that with larger amount of data, or by using a bigger YOLOv6 model, for example the large one with more parameters, the results could reach better levels.

We can also conclude that high accuracy can be achieved even without pre-processing steps or techniques done to the images. In a much larger dataset, with more training epochs, the model can be trained to better identify the cells' nuclei.

6.2 Future Work

YOLO is a very powerful and fast detection model. However, there is the challenge of high number of operations and computational demand. Future research can be concentrated on implementing deep learning frameworks more quickly and on developing more effective pruning techniques for smaller, more precise detection networks. The networks can be then incorporated in mobile device CPUs, due to their small weight.

Furthermore, pruning strategies can be used for additional detection tasks where hardware storage and computing resources are scarce, and focus not only on cell counting.

With CNN's architectural improvements, the extraction of features can be made more robust by using cutting-edge convolutional techniques including tiled, transposed, and dilated convolution.

These convolutions may be used to further enhance the procedure, depending on the applications and underlying datasets. Models that have been trained for one job in particular may not perform well on other tasks that are comparable, leading to the model's inability to detect new data. For more accurate generalized models, various regularization techniques such as LP-Normalization, dropouts could be tested.

CHAPTER 7

7. REFERENCES

[1] C. Liu, Y. Tao, J, Liang, K.Li, Y.Chen "Object Detection Based on YOLO Network", 2015, IEEE 4th Information Technology and Mechatronics Engineering Conference.

[2] Juan Du, "Understanding of Object Detection Based on CNN Family and YOLO" 2018 J. Phys.: Conf. Ser. 1004 012029

[3] A. Bochkovskiy, Ch. Wang, H Liao "YOLOv4: Optimal Speed and Accuracy of Object Detection" *April 2020,* arXiv.

[5] A. Uka, X. Polisi, A. Halili, C. Dollinger, and N. E. Vrana, "Analysis of cell behavior on micropatterned surfaces by image processing algorithms," in *IEEE EUROCON 2017 -17th International Conference on Smart Technologies*, 2017, pp. 75–78, doi: 10.1109/EUROCON.2017.8011080.

[6] C. X. Hernández, M. M. Sultan, and V. S. Pande, "Using Deep Learning for Segmentation and Counting within Microscopy Data," Feb. 2018, [Online]. Available: http://arxiv.org/abs/1802.10548.

[7] Godfrey N. Hounsfield., "Computed Medical Imaging" Feb. 2015, [Online]. Available: https://www.science.org/doi/10.1126/science.6997993

[8] M. Chen, X. Shi, Y. Zhang, D. Wu and M. Guizani, "Deep Feature Learning for Medical Image Analysis with Convolutional Autoencoder Neural Network," in IEEE Transactions on Big Data, vol. 7, no. 4, pp. 750-758, 1 Oct. 2021, doi: 10.1109/TBDATA.2017.2717439.

[9] D. Shen, G. Wu, H. Suk, "Deep Learning in Medical Image Analysis"*2017, [Online] Available:* https://www.annualreviews.org/doi/10.1146/annurev-bioeng-071516-044442

[10] S. Anwar, M. Majid, A. Qayyum, M. Awais, M. Alnowami, M. Khan, "Medical Image Analysis using Convolutional Neural Networks,"*2018,* doi: https://doi.org/10.1007/s10916-018-1088-1

[11] C. N. Vasconcelos and B. N. Vasconcelos, “Convolutional Neural Network Committees for Melanoma Classification with Classical And Expert Knowledge Based Image Transforms Data Augmentation,” Feb. 2017, [Online]. Available: http://arxiv.org/abs/1702.07025.

[12] D. Wang *et al.*, “AFP-Net: Realtime Anchor-Free Polyp Detection in Colonoscopy,” Sep. 2019, [Online]. Available: http://arxiv.org/abs/1909.02477.

[13] X. Dong *et al.*, “Air, bone and soft-tissue Segmentation on 3D brain MRI Using Semantic Classification Random Forest with Auto-Context Model,” Nov. 2019, [Online]. Available:

[14] J. H. Tan, U. R. Acharya, S. V. Bhandary, K. C. Chua, and S. Sivaprasad, “Segmentation of optic disc, fovea and retinal vasculature using a single convolutional neural network,” *J. Comput. Sci.*, vol. 20, pp. 70–79, May 2017, doi: 10.1016/j.jocs.2017.02.006.

[15] X. Feng, K. Qing, N. J. Tustison, C. H. Meyer, and Q. Chen, “Deep convolutional neural network for segmentation of thoracic organs-at-risk using cropped 3D images,” *Med. Phys.*, vol. 46, no. 5,

[16] B. Planche and E. Andres, *Hands-On Computer Vision with TensorFlow 2*. 2019.

[17] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” Sep. 2014, [Online]. Available: http://arxiv.org/abs/1409.1556.

[18] Y. Hu, S. Sun, J. Li, X. Wang, Q. Gu, “A novel channel pruning method for deep neural network compression” Sep. 2014, Research Center of Precision Sensing and Control, Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China

[19] R. Shi, T. Li, Y. Yamaguchi, “An attribution-based pruning method for real-time mango detection with YOLO network, [Online] Available: https://www.sciencedirect.com/science/article/abs/pii/S0168169919313717?via%3Dihub

[20] Y. He, X. Zhang, J. Sun, “Channel Pruning for Accelerating Very Deep Neural Networks", July 2017, arXiv:1707.06168

[21] C. Li, H. Jiang, L. Li, K. Weng, Y. Geng, L. Li, Z. Ke, Q. Li, M. Cheng, W. Nie, Y. Li, B. Zhang, Y. Liang, L. Zhou, X. Xu, X. Chu, X. Wei, X. Wei “YOLOv6: A single Stage

Object Detection Framework for Industrial Applications", Sept 22, arXiv:2209.02976

[22] “Roboflow.” https://roboflow.com/

[23] “YOLOv5.” https://github.com/ultralytics/yolov5/ Accessed on February 2023

[24] “YOLOv6.” https://github.com/meituan/YOLOv6 Accessed on February 2023

[25] S. Liu, L. Qi, H. Qin, J. Shi, J. Jia, "Path Aggregation Network for Instance Segmentation, March 2018, arXiv:1803.01534

[26] J. Solawetz, J. Nelson, "PP-YOLO surpasses YOLOv4 - State of the Art Object Detection Techniques", August 2020, Online, Available https://blog.roboflow.com/pp-yolo-beats-yolov4-object-detection/

[27] Uka, A., Tare, A., Polisi, X., & Panci, I. (2020, December). FASTER R-CNN for cell counting in low contrast microscopic images. In *2020 International Conference on Computing, Networking, Telecommunications & Engineering Sciences Applications (CoNTESA)* (pp. 64-69). IEEE

[28] Uka, A., Polisi, X., Barthes, J., Halili, A. N., Skuka, F., & Vrana, N. E. (2020, August). Effect of Preprocessing on Performance of Neural Networks for Microscopy Image Classification. In *2020 International Conference on Computing, Electronics & Communications Engineering (iCCECE)* (pp. 162-165). IEEE.

CHAPTER 8

8. Appendix

YOLOv5 train.py

```
# Model
  check_suffix(weights, '.pt')  # check weights
  pretrained = weights.endswith('.pt')
  if pretrained:
      with torch_distributed_zero_first(LOCAL_RANK):
          weights = attempt_download(weights)  # download if not found locally
      ckpt = torch.load(weights, map_location='cpu')  # load checkpoint to CPU to avoid CUDA memory leak
      model = Model(cfg or ckpt['model'].yaml, ch=3, nc=nc, anchors=hyp.get('anchors')).to(device)  # create
      exclude = ['anchor'] if (cfg or hyp.get('anchors')) and not resume else []  # exclude keys
      csd = ckpt['model'].float().state_dict()  # checkpoint state_dict as FP32
      csd = intersect_dicts(csd, model.state_dict(), exclude=exclude)  # intersect
      model.load_state_dict(csd, strict=False)  # load
      LOGGER.info(f'Transferred {len(csd)}/{len(model.state_dict())} items from {weights}')  # report
  else:
      model = Model(cfg, ch=3, nc=nc, anchors=hyp.get('anchors')).to(device)  # create
  amp = check_amp(model)  # check AMP

  for epoch in range(start_epoch, epochs):  # epoch ------------------------------------------------------------------
      callbacks.run('on_train_epoch_start')
      model.train()

      # Update image weights (optional, single-GPU only)
      if opt.image_weights:
          cw = model.class_weights.cpu().numpy() * (1 - maps) ** 2 / nc  # class weights
          iw = labels_to_image_weights(dataset.labels, nc=nc, class_weights=cw)  # image weights
          dataset.indices = random.choices(range(dataset.n), weights=iw, k=dataset.n)  # rand weighted idx

      # Update mosaic border (optional)
      # b = int(random.uniform(0.25 * imgsz, 0.75 * imgsz + gs) // gs * gs)
      # dataset.mosaic_border = [b - imgsz, -b]  # height, width borders

      mloss = torch.zeros(3, device=device)  # mean losses
      if RANK != -1:
          train_loader.sampler.set_epoch(epoch)
      pbar = enumerate(train_loader)
      LOGGER.info(('\n' + '%11s' * 7) % ('Epoch', 'GPU_mem', 'box_loss', 'obj_loss', 'cls_loss', 'Instances', 'Size'))
      if RANK in {-1, 0}:
          pbar = tqdm(pbar, total=nb, bar_format=TQDM_BAR_FORMAT)  # progress bar
      optimizer.zero_grad()
      for i, (imgs, targets, paths, _) in pbar:  # batch -------------------------------------------------------------
          callbacks.run('on_train_batch_start')
          ni = i + nb * epoch  # number integrated batches (since train start)
          imgs = imgs.to(device, non_blocking=True).float() / 255  # uint8 to float32, 0-255 to 0.0-1.0

          # Warmup
          if ni <= nw:
              xi = [0, nw]  # x interp
              # compute_loss.gr = np.interp(ni, xi, [0.0, 1.0])  # iou loss ratio (obj_loss = 1.0 or iou)
              accumulate = max(1, np.interp(ni, xi, [1, nbs / batch_size]).round())
              for j, x in enumerate(optimizer.param_groups):
                  # bias lr falls from 0.1 to lr0, all other lrs rise from 0.0 to lr0
                  x['lr'] = np.interp(ni, xi, [hyp['warmup_bias_lr'] if j == 0 else 0.0, x['initial_lr'] * lf(epoch)])
                  if 'momentum' in x:
                      x['momentum'] = np.interp(ni, xi, [hyp['warmup_momentum'], hyp['momentum']])

          # Multi-scale
          if opt.multi_scale:
              sz = random.randrange(int(imgsz * 0.5), int(imgsz * 1.5) + gs) // gs * gs  # size
              sf = sz / max(imgs.shape[2:])  # scale factor
              if sf != 1:
                  ns = [math.ceil(x * sf / gs) * gs for x in imgs.shape[2:]]  # new shape (stretched to gs-multiple)
                  imgs = nn.functional.interpolate(imgs, size=ns, mode='bilinear', align_corners=False)

          # Forward
          with torch.cuda.amp.autocast(amp):
              pred = model(imgs)  # forward
              loss, loss_items = compute_loss(pred, targets.to(device))  # loss scaled by batch_size
              if RANK != -1:
                  loss *= WORLD_SIZE  # gradient averaged between devices in DDP mode
              if opt.quad:
                  loss *= 4.

          # Backward
          scaler.scale(loss).backward()

          # Optimize - https://pytorch.org/docs/master/notes/amp_examples.html
          if ni - last_opt_step >= accumulate:
              scaler.unscale_(optimizer)  # unscale gradients
              torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=10.0)  # clip gradients
              scaler.step(optimizer)  # optimizer.step
              scaler.update()
              optimizer.zero_grad()
              if ema:
                  ema.update(model)
              last_opt_step = ni

          # Log
          if RANK in {-1, 0}:
              mloss = (mloss * i + loss_items) / (i + 1)  # update mean losses
              mem = f'{torch.cuda.memory_reserved() / 1E9 if torch.cuda.is_available() else 0:.3g}G'  # (GB)
              pbar.set_description(('%11s' * 2 + '%11.4g' * 5) %
                                   (f'{epoch}/{epochs - 1}', mem, *mloss, targets.shape[0], imgs.shape[-1]))
              callbacks.run('on_train_batch_end', model, ni, imgs, targets, paths, list(mloss))
              if callbacks.stop_training:
                  return
          # end batch ------------------------------------------------------------------------------------------------
      if RANK in {-1, 0}:
          # mAP
          callbacks.run('on_train_epoch_end', epoch=epoch)
          ema.update_attr(model, include=['yaml', 'nc', 'hyp', 'names', 'stride', 'class_weights'])
          final_epoch = (epoch + 1 == epochs) or stopper.possible_stop
          if not noval or final_epoch:  # Calculate mAP
              results, maps, _ = validate.run(data_dict,
                                              batch_size=batch_size // WORLD_SIZE * 2,
                                              imgsz=imgsz,
                                              half=amp,
                                              model=ema.ema,
                                              single_cls=single_cls,
                                              dataloader=val_loader,
                                              save_dir=save_dir,
                                              plots=False,
                                              callbacks=callbacks,
                                              compute_loss=compute_loss)

          # Update best mAP
          fi = fitness(np.array(results).reshape(1, -1))  # weighted combination of [P, R, mAP@.5, mAP@.5-.95]
          stop = stopper(epoch=epoch, fitness=fi)  # early stop check
          if fi > best_fitness:
              best_fitness = fi
          log_vals = list(mloss) + list(results) + lr
          callbacks.run('on_fit_epoch_end', log_vals, epoch, best_fitness, fi)

          # Save model
          if (not nosave) or (final_epoch and not evolve):  # if save
              ckpt = {
                  'epoch': epoch,
                  'best_fitness': best_fitness,
                  'model': deepcopy(de_parallel(model)).half(),
                  'ema': deepcopy(ema.ema).half(),
                  'updates': ema.updates,
                  'optimizer': optimizer.state_dict(),
                  'opt': vars(opt),
                  'git': GIT_INFO,  # {remote, branch, commit} if a git repo
                  'date': datetime.now().isoformat()}

              # Save last, best and delete
              torch.save(ckpt, last)
              if best_fitness == fi:
                  torch.save(ckpt, best)
              if opt.save_period > 0 and epoch % opt.save_period == 0:
                  torch.save(ckpt, w / f'epoch{epoch}.pt')
              del ckpt
              callbacks.run('on_model_save', last, epoch, final_epoch, best_fitness, fi)
```

YOLOv5 detect.py

```
# Process predictions
        for i, det in enumerate(pred):  # per image
            seen += 1
            if webcam:  # batch_size >= 1
                p, im0, frame = path[i], im0s[i].copy(), dataset.count
                s += f'{i}: '
            else:
                p, im0, frame = path, im0s.copy(), getattr(dataset, 'frame', 0)

            p = Path(p)  # to Path
            save_path = str(save_dir / p.name)  # im.jpg
            txt_path = str(save_dir / 'labels' / p.stem) + ('' if dataset.mode == 'image' else f'_{frame}')  # im.txt
            s += '%gx%g ' % im.shape[2:]  # print string
            gn = torch.tensor(im0.shape)[[1, 0, 1, 0]]  # normalization gain whwh
            imc = im0.copy() if save_crop else im0  # for save_crop
            annotator = Annotator(im0, line_width=line_thickness, example=str(names))
            if len(det):
                # Rescale boxes from img_size to im0 size
                det[:, :4] = scale_boxes(im.shape[2:], det[:, :4], im0.shape).round()

                # Print results
                for c in det[:, 5].unique():
                    n = (det[:, 5] == c).sum()  # detections per class
                    s += f"{n} {names[int(c)]}{'s' * (n > 1)}, "  # add to string

                # Write results
                for *xyxy, conf, cls in reversed(det):
                    if save_txt:  # Write to file
                        xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / gn).view(-1).tolist()  # normalized xywh
                        line = (cls, *xywh, conf) if save_conf else (cls, *xywh)  # label format
                        with open(f'{txt_path}.txt', 'a') as f:
                            f.write(('%g ' * len(line)).rstrip() % line + '\n')

                    if save_img or save_crop or view_img:  # Add bbox to image
                        c = int(cls)  # integer class
                        label = None if hide_labels else (names[c] if hide_conf else f'{names[c]} {conf:.2f}')
                        annotator.box_label(xyxy, label, color=colors(c, True))
                    if save_crop:
                        save_one_box(xyxy, imc, file=save_dir / 'crops' / names[c] / f'{p.stem}.jpg', BGR=True)
```

YOLOv6 train.py (Main function of training)

```
def main(args):
    '''main function of training'''
    # Setup
    args.local_rank, args.rank, args.world_size = get_envs()
    cfg, device, args = check_and_init(args)
    # reload envs because args was chagned in check_and_init(args)
    args.local_rank, args.rank, args.world_size = get_envs()
    LOGGER.info(f'training args are: {args}\n')
    if args.local_rank != -1: # if DDP mode
        torch.cuda.set_device(args.local_rank)
        device = torch.device('cuda', args.local_rank)
        LOGGER.info('Initializing process group... ')
        dist.init_process_group(backend="nccl" if dist.is_nccl_available() else "gloo", \
                init_method=args.dist_url, rank=args.local_rank, world_size=args.world_size)

    # Start
    trainer = Trainer(args, cfg, device)
    # PTQ
    if args.quant and args.calib:
        trainer.calibrate(cfg)
        return
    trainer.train()

    # End
    if args.world_size > 1 and args.rank == 0:
        LOGGER.info('Destroying process group... ')
        dist.destroy_process_group()


if __name__ == '__main__':
    args = get_args_parser().parse_args()
    main(args)

def check_and_init(args):
    '''check config files and device.'''
    # check files
    master_process = args.rank == 0 if args.world_size > 1 else args.rank == -1
    if args.resume:
        # args.resume can be a checkpoint file path or a boolean value.
        checkpoint_path = args.resume if isinstance(args.resume, str) else find_latest_checkpoint()
        assert os.path.isfile(checkpoint_path), f'the checkpoint path is not exist: {checkpoint_path}'
        LOGGER.info(f'Resume training from the checkpoint file :{checkpoint_path}')
        resume_opt_file_path = Path(checkpoint_path).parent.parent / 'args.yaml'
        if osp.exists(resume_opt_file_path):
            with open(resume_opt_file_path) as f:
                args = argparse.Namespace(**yaml.safe_load(f))  # load args value from args.yaml
        else:
            LOGGER.warning(f'We can not find the path of {Path(checkpoint_path).parent.parent / "args.yaml"},'\
                           f' we will save exp log to {Path(checkpoint_path).parent.parent}')
            LOGGER.warning(f'In this case, make sure to provide configuration, such as data, batch size.')
            args.save_dir = str(Path(checkpoint_path).parent.parent)
        args.resume = checkpoint_path  # set the args.resume to checkpoint path.
    else:
        args.save_dir = str(increment_name(osp.join(args.output_dir, args.name)))
        if master_process:
            os.makedirs(args.save_dir)

    cfg = Config.fromfile(args.conf_file)
    if not hasattr(cfg, 'training_mode'):
        setattr(cfg, 'training_mode', 'repvgg')
    # check device
    device = select_device(args.device)
    # set random seed
    set_random_seed(1+args.rank, deterministic=(args.rank == -1))
    # save args
    if master_process:
        save_yaml(vars(args), osp.join(args.save_dir, 'args.yaml'))

    return cfg, device, args
```