INF8225 - Project
YOLOv3 exploration

Ana Jade Boudreault (1893238) [1] , **Guillaume Thibault (1948612)** [1] ,
[1]Polytechnique Montréal
{g.thibault, ana-jade.boudreault}polymtl.ca, Code: https://github.com/guthi1/Yolo_project

Abstract

This work realized within the framework of the INF8225 course aims at exploring the YoloV3 model and its various applications. We have first done real time detection with the pre-trained model. Then, we modified the architecture to be able to detect a single class and train the model from the same weights as the article in order to detect a soccer ball in a match. Finally, we explored the accuracy of the model.

1 Introduction

YOLOv3 (You Only Look Once) is an object detection model introduced in 2018 known for its speed. This model quickly gained in popularity and remains popular to this day. Based on its name, YOLOv3 has two predecessors: YOLOv1, which introduced the model's general structure and YOLOv2, which added the use of predefined anchor boxes to improve bounding box predictions.

This report presents and compares the performance of YOLOv3 with pre-trained weights versus our own implementation. To begin, the performance of the pre-trained model was tested in real time using a webcam camera. The model was then implemented from scratch and trained on the MNIST for object detection dataset [Håkon Hukkelås, 2020]. The performance of both variations (pre-trained and MNIST) was then compared using the mean Average Precision (mAP) metric.

2 Previous works

Most object detectors fall within two classes: One-stage detectors and two-stage detectors.

2.1 Two-stage detectors

Two-stage detectors start by finding regions of interest with a region proposal network (RPN) and then use classification and regression to identify the object's class and find the bounding box coordinates. These detectors generally have a better performance than one-stage detectors, but are slower because two steps need to be completed before the result is output. Popular detectors include *R-CNN* [Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik,] and *Faster R-CNN* [Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun,].

R-CNN: This model uses a selective search to identify regions of interest and then uses a CNN to classify each region. This model's main disadvantage is that the selective search is not based on a neural network and acts as a bottleneck restricting R-CNN's speed. There is also no regression done to optimize the bounding box's coordinates.

Faster R-CNN: Faster R-CNN uses the same CNN for both object proposal and classification. This means performing a selective search is non longer necessary and the whole network can be trained at the same time. A region proposal network is used to define the bounding boxes, and a feature map is used for classification.

2.2 One-stage detectors

One-stage detectors treat both the classification and bounding box position tasks at the same time. These models are generally less accurate than two-stage methods, but are much faster. Popular detectors include HOG [Navneet Dalal, Bill Triggs,] and YOLOv1 [Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi,].

HOG: The HOG detector is based on the histogram of oriented gradients descriptor and SVM classifier. A detection box is used to scan the image in order to determine if an object of interest is present or not. Multi-scale detection is possible by keeping the size of the detection box constant and by changing the size of the image. This method is mainly used to detect humans.

YOLOv1: The bounding box location for each class is predicted simultaneously and is treated as a regression problem. The image is divided in a $S \times S$ grid and each cell produces B bounding boxes, each with their own confidence score regarding the presence of an object of interest as well as its class.

3 Preliminary concepts

3.1 Model architecture

The following figure [Ayoosh Kathuria, 2018] shows an overview of YOLOv3's architecture.

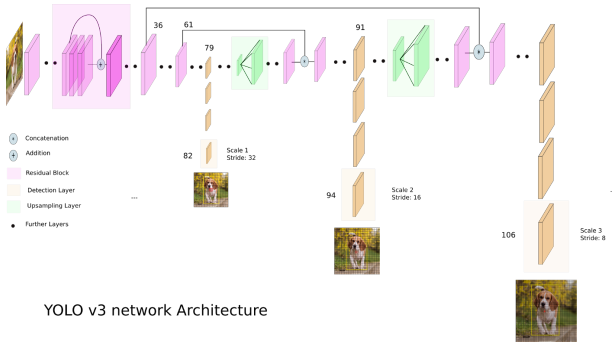

Figure 1: YOLOv3 architecture

YOLOv3 is composed 106 fully convolutional layers: 53 used for feature extraction and 53 used for detection and classification. Feature extraction is done by a backbone model called Darknet-53 trained on Imagenet. The 53 layers used for detection and classification are each combined with a batch normalization layer and use a Leaky ReLU activation function. These convolutional layers are used produce multiple feature maps by passing multiple filters over the images. Detections are made at three different places (layers 82, 94 and 106). Since the resolution of the image changes between each detection, this leads to a better performance when trying to find multiscale objects. Just like the previous versions of YOLO, the input image is separated in a $S \times S$ grid, and the cell at the center of the object is responsible for detecting it.

3.2 Convolutional layers with up and down sampling

A convolutional layer passes a fixed-size, learnable filter over an image to produce feature maps. YOLOv3 uses a series of 3×3 and 1×1 convolutional layers, each having a specific purpose.

3×3 conv: Used to produce feature maps at different image resolutions based on up/down sampling at that specific layer.

1×1 conv: Used to change the number of channels for that specific layer or used to perform object detection.

Strides are used instead of pooling to prevent the loss of low-level features. Strides of 2 per layer are used to down sample the image until it reaches a total stride of 32 at layer 84. Layer 79 is then passed through a couple other convolutional layers, before being up-sampled by two. The resulting feature map will then be concatenated with the feature map at layer 61. The same process is applied to the layer 91, which is concatenated with layer 36.

3.3 Darknet-53

Darknet-53 is a 53 layer convolutional network trained on Imagenet that uses 3×3 and 1×1 filters with skip connections. Other models such as ResNet also use skip connections, but Darknet is much more efficient without compromising performance. The following image [Joseph Redmon, Ali Farhadi, 2018] presents the Darknet architecture in greater detail.

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

Figure 2: Darknet-53 architecture

3.4 Detection

The detection is done by applying a 1×1 detection kernel on the feature map. Each cell on the feature map can predict B bounding boxes. For each of the generated boxes, 4 attributes describing its size, one value describing the probability of an object being present in the box (objectness score), and C, the number of classes, values describing the probability of the object belonging to a specific class (class confidences). When trained on the COCO dataset, each bounding box is described by 85 numbers (4x bounding box position, 1x objectness score, 80x class confidences) compared to 25 when trained on the VOC 2009 (20 classes) dataset.

The following table presents an overview of the three detections done in the model.

Table 1: Overview of the detection layers in YOLOv3

Layer	Strides	Object size
82	32	Big
94	16	Medium
106	8	Small

YOLOv3 uses the binary cross-entropy loss function for the classification task and uses logistic regression for object confidence and class predictions.

3.5 Hyperparameters

The following list presents the model's hyperparameters.

1. Maximum number of boxes
    - Default value: 200
    - Description: Maximum number of bounding boxes that can be kept for one image
2. Objectness score
    - Default value: 0.5
    - Description: Probability of an object being inside a given bounding box. Used to eliminate boxes that don't have an object in them.
3. Class threshold
    - Default value: 0.25
    - Description: Minimum class confidence to detect an object
4. Non-max suppression threshold
    - Default value: 0.6
    - Description: Minimum probability of the object being in a bounding box. Helps to overcome the problem of detecting the same object multiple times

3.6 Differences with YOLOv1 and YOLOv2

Here is an overview of the changes made with YOLOv3 [Ayoosh Kathuria, 2018]:

- Detection at different layers to help with the detection of small objects
- More bounding boxes are predicted per cell, going from 5 in YOLOv2 to 9 (3 per scale) in YOLOv3
- Use of cross-entropy loss instead of the sum-squared error for object confidence and class predictions. In YOLOv3, those values are predicted using logistic regression
- Multi-label classification is now possible with the use of logistic regression and a threshold instead of softmax.

4 Performance with pre-trained weights

The first step of the project was to explore the performance of the project. For this we used the pre-trained weights that we loaded into the model. We then connected the images from a webcam as input and ran the model on the graphics card of a computer (GTX1070).

The results obtained are nothing short of impressive. The model can process images at a rate of 12 frames per second, which allows to obtain an output video. The detection quality of the model is extremely accurate as can be seen in the following figure.

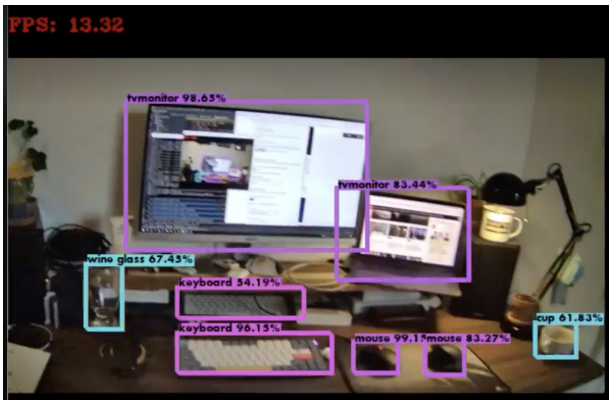

Figure 3: YOLOv3 Webcam result

5 Transfer learning

The next step of this work is to use the YoloV3 model, change it to adapt it to our needs and train it from the same weights used in the article. Our use case is football detection to allow a camera to follow the ball or just to help a camera operator to follow the action.

5.1 When is transfer learning useful?

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task. The use of transfer learning is widely used as a starting point in model training in order to reduce the time and resources needed to train a model. In ours case, we will use the Darknet framework model with pre-train weights to train the YoloV3 model with custom a custom class from scratch.

5.2 Implementation with YOLOv3

To explore transfer learning, we based ourselves on the YoloV3 architecture implemented with the Darknet framework by Joeph Redmon [Joseph Redmon, a]. Darknet is an open source neural network framework written in C and CUDA. The goal of the transfer learning explored in this work is to detect a football ball in a image or a video. One use of this model would be to create a camera system to automatically follow the action during a football game.

The first step for this part of the project is the installation of CUDA and CudNN, these allow to use a graphics card in order to parallelize the calculations in order to have much better results in the efficiency of the real time processing (20 frames per second instead of 0.1 frame per second with the use of a GTX1070 graphics card) Then, the OpenCV library was installed from the source with Cmake. When these elements are installed and functional, Darknet is install in order to train the model with it.

Before the model can be trained, it is first necessary to get labeled data. For this, the Google image library [Google,] is used to import 1000 images containing football balls. These picture are processed with the OIDv4 ToolKit [Vittorio Mazzia,] to get the labels in the right format for the training.

Once all the preparations are completed, we need to change the YoloV3 architecture to adapt it to our class, since it is de-

signed to work with 80 classes instead of one. The Darknet framework already contain the YoloV3 architecture, where we have to adjust the maximum number of batches (4000), the steps (3200,4600) number of classes (1) and the convolution filters (18). Then, we train the model.

For more information on the transfer learning, see the Github repos readme.md https://github.com/guthi1/Yolo_project/tree/main/Transfer%20learning

5.3 Trainning

To train the model, we used the same weights that the authors of YoloV3 used to train their model, that is to say the darknet53.conv.74 pre-train weights for the convolutions trained on imagenet [Joseph Redmon, b]. The dataset used was limited to 1000 training data so the training of the model was not very long, about 2h for a total of 900 epoch.

5.4 Results

Prediction made on an image that the network has never seen before. The prediction is done in 0.0227 seconds on average.

Figure 4: Prediction after 100 epoch

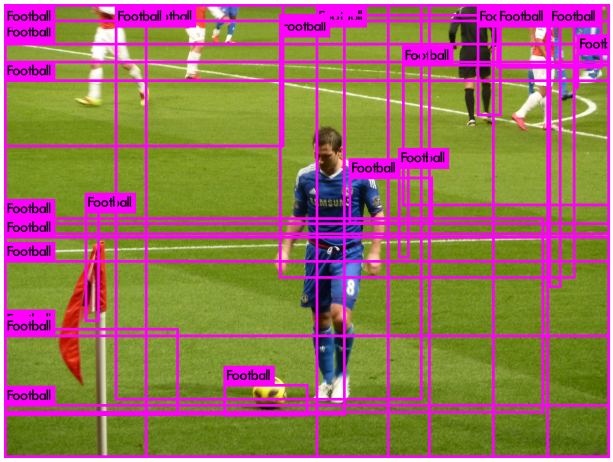

Figure 5: Prediction after 200 epoch

Figure 6: Prediction for epoch 300 to 700

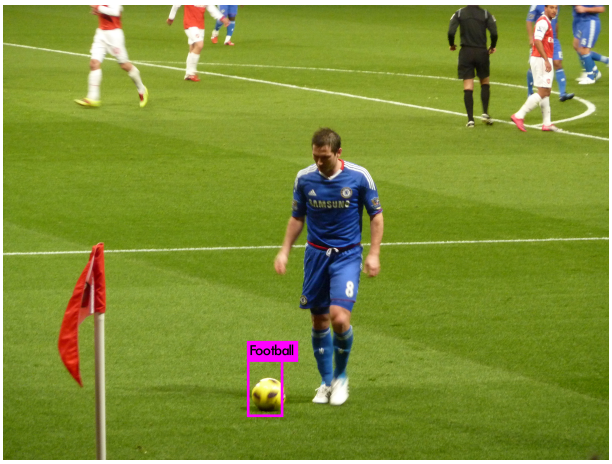

Figure 7: Prediction after 800 epoch

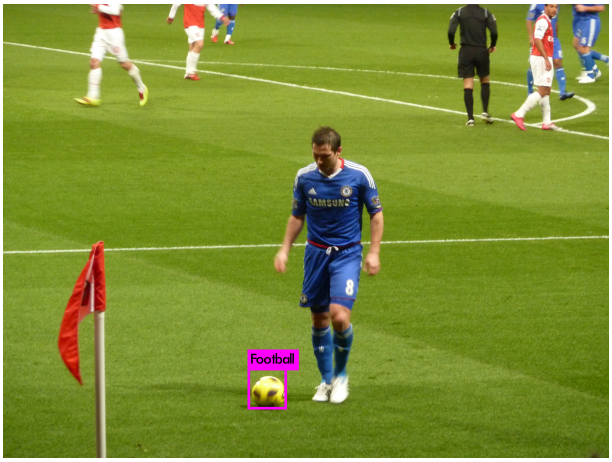

Figure 8: Prediction after 900 epoch

Table 2: Results with default hyperparameters

Epoch	Number of prediction	average confidence
100	200	75
200	35	62
300	0	-
400	0	-
500	0	-
600	0	-
700	0	-
800	1	52
900	1	57

5.5 Discussion

The results obtained for this small number of images, compared to the more commonly used dataset of nearly a million images like the COCO dataset, are quite accurate after 2 hours of training. As we can see on the following figures, the model starts by predicting a lot of balls everywhere, and as the training goes on, the model understands that a ball is rarer. After 800 iterations, the model manages to find the balloon in the image, and after 100 more iterations, the model manages to surround the balloon well. Furthermore, we can see that the confidence of the detection increases between the last two results and it would be interesting to continue the training to see if the confidence continues to increase.

Due to the fact that the vast majority of the images used are images of soccer games, the model has difficulty generalizing to images where the action is not on a field. For example, me holding a ball in my hands in my kitchen. The developed model is then specialized to images of soccer game, that was desired at the beginning, but the model may have difficulty at certain times for camera angles that it has never seen before.

6 Model precision

After exploring how to use YoloV3 with the framework developed by the author, we want to explore the accuracy of the model developed by the author.

6.1 Implementation

Several Github repositories were consulted and tested in order to find one that fit closely enough the scope of our project. The following list presents most of the tested Github repositories in order.

1. YOLOv3 Implemented in TensorFlow 2.0
    - GitHub
    - Google Colab (ours)
2. YOLOv3 in PyTorch
    - GitHub
    - Google Colab (ours)
3. TensorFlow2.x - YOLOv3
    - GitHub
    - Google Colab (ours)

6.2 Choosing the training dataset

COCO dataset

The first obvious choice for the training dataset was the COCO dataset. On top of being the one with the most classes, it is also the dataset used to train the original YOLO weights. In order to have the best comparison possible, training the model from scratch on this dataset would have been ideal. However, the huge amount of images in the COCO dataset forced us to find an alternative. Since we were running everything on Google Colab to have access to their GPUs, we had limited storage space and our runtime would often get disconnected because we had exceeded the maximum disk space allowed.

Pascal VOC dataset

Our second choice was the Pascal VOC dataset. Since its number of training samples was greatly inferior to the number in the COCO dataset, we hoped using Pascal's VOC dataset would solve our problems. For a while it did, but the format of the annotations (XML file) made it difficult to calculate the mean Average Precision, used to calculate the model's performance.

MNIST for object detection dataset

Finally, we chose to train YOLOv3 on the MNIST for object detection dataset [Håkon Hukkelås, 2020]. Not only is it possible to choose the number of training samples that are to be generated, but the annotation are also provided in a simple .txt file. This means going over the disk space limit will not be a problem, and it will be a lot easier to parse the ground truth information needed to calculate the mAP. For those reasons, we decided to choose to train YOLOv3 on this dataset.

6.3 Modifications made to the GitHub code

1. YOLOv3 implemented in TensorFlow 2.0

The first minor modification made to this implementation was to change the version of cuDNN running in Google Colab. The second modification needed was to change the training dataset from Pascal VOC to COCO. This was a lot more time-consuming than anticipated, mostly because of the change in the annotation file format and because we had no prior experience with TensorFlow. After trying for a couple of days, we decided to try and find another implementation, this time in PyTorch.

2. YOLOv3 in PyTorch

The first challenge with this implementation was having to build and install the *pycocotools* library. Next we also had to load the entire COCO dataset. It was at this point we realized the dataset was simply too big for Google Colab's disk space and decided to simply use the VOC dataset instead. Finally, the main reason this implementation didn't end up being the one we used was an issue with the code found on Github.

3. TensorFlow2.x - YOLOv3

This implementation is the one we decided to use since it had built-in functions to calculate the mAP. However, because of long training time, we weren't able to train the model long enough to get interesting results.

7 Methodology

7.1 Using pre-trained weights

Adapting a pre-trained model to make it work in real time with a webcam was a good place to start. It allowed us to familiarize ourselves with the model and visualize its expected performance. Modifying some code found on Github [theAIGuysCode, 2020] so it could work locally made it possible to establish a performance baseline and give us a general understanding of the model's structure.

7.2 Transfer learning

For this section, we had a lot of trouble using the Darknet framework on Windows because it was developed to work on Linux. After doing some research, we found a Git repository with the necessary source code modifications to make it work. After installing Cuda on our computer and OpenCV from source with Cmake, we compiled Darknet as an executable, we encountered library problems when it came time to train the model.

To make things easier, we decided to change the operating system and go directly to Linux to train the model and the use of Colab pro allowed us to run the model on Linux with only a few adjustments to the Makefile in order to use the Cuda library compatible with the colab graphics card.

We just have to upload the elements of the Transfer learning folder in colab in order to use it. All the instructions are in the readme of this folder on the Github of the project.

7.3 Model precision

As stated in section 6, our implementation was once again based on some code found on Github [YunYang1994, 2020]. Starting from some code that already had the model structure implemented definitely had its pros and cons.

Pros:

- Since the model was implemented with Tensorflow, we got the opportunity to work with a framework we weren't familiar with.
- The time saved by not having to implement the model structure allowed us to familiarize ourselves with transfer learning.

Cons:

- We didn't manually code the model layer by layer, resulting in a less thorough understanding of YOLOv3's architecture.

In our opinion, the pros outweigh the cons for a couple of reasons. First, implementing models from scratch is something that was practised intensively during other assignments for this class. Therefore, it was much more interesting for us to learn about something we hadn't done before (transfer learning). The same argument could also be used by choosing to work with TensorFlow versus PyTorch. Finally, given the short time-span within which the project needs to be completed, along with the fact that the submission date is in the middle of our final exams, we felt it was best to optimize our time and focus on trying to get presentable results.

8 Conclusion

Through this project, we managed to get a better understanding of YOLOv3, learn how to setup a realtime detection camera, explore multiple (sometimes disappointing) GitHub repos implementing the model in Python and learn a lot about transfer learning. In the end, it would have been better to directly roll up your sleeves and use the framework written in C for everything instead of being lazy and trying to use python at all costs. After seeing how much faster the model could be trained in C, we now have a better understanding as to why many Python libraries are also written in that language. While the work presented in this report isn't exactly what was planned in the original scope of the project, we feel like it is just as, if not more, interesting. And in the spirit of Joseph Redmond, don't email us (Because we're finally done with this semester).

References

[Ayoosh Kathuria, 2018] Ayoosh Kathuria. What's new in YOLO v3?, 2018.

[Google,] Google. Open Images Dataset. https://storage.googleapis.com/openimages/web/index.html.

[Håkon Hukkelås, 2020] Håkon Hukkelås. MNIST Object Detection dataset, 2020.

[Joseph Redmon, a] Joseph Redmon. Darknet. https://pjreddie.com/darknet/.

[Joseph Redmon, b] Joseph Redmon. YOLO: Real-Time Object Detection. https://pjreddie.com/darknet/yolo/.

[Joseph Redmon, Ali Farhadi, 2018] Joseph Redmon, Ali Farhadi. YOLOv3: An Incremental Improvement, 2018.

[Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi,] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection.

[Navneet Dalal, Bill Triggs,] Navneet Dalal, Bill Triggs. Histograms of Oriented Gradients for Human Detection.

[Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik,] Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation.

[Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun,] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.

[theAIGuysCode, 2020] theAIGuysCode. Object-Detection-API, 2020.

[Vittorio Mazzia,] Vittorio Mazzia. OIDv4$_{ToolKit}$. $https://github.com/EscVM/OIDv4_{ToolKit}$.

[YunYang1994, 2020] YunYang1994. TensorFlow2.x-YOLOv3, 2020.