Instituto Tecnológico de Aeronáutica

CT-213

Laboratório 9 - Detecção de Objetos

Aluno: Pedro Elardenberg Sousa e Souza
Professor: Marcos Ricardo Omena de Albuquerque Maximo

14 de abril de 2024

Conteúdo

1 Resumo

Neste Laboratório, foi utilizado o algoritmo YOLO baseado no uso de uma Rede Neural Convolucional (CNN) para detecção de objetos, a saber, imagens de bola e traves da competição de futebol de robôs do time da ITAndroids.

A atividade consistiu em implementar a rede neural YOLO em Keras tal como ela foi projetada para o robô humanoide da ITAndroids, mas foi utilizada uma rede já treinada para realizar a tarefa.

Além da criação da arquitetura da rede, foram implementados os métodos de detecção do objeto, processamento da imagem e tratamento do *output* da rede, a qual retornava as imagens testadas com os objetos indicados nela.

Os resultados foram então mostrados e discutidos neste relatório.

2 Introdução

2.1 YOLO

You Only Look Once (YOLO) é uma rede neural que faz predições em tempo real da posição e dimensão de objetos, isto é, *boundin boxes* e da probabilidade de esse objeto estar na imagem, com base na premissa de passar por cada sub-elemento da imagem apenas uma vez, fazendo-o passar por suas camadas convolucionais e de *pooling* para, no fim, ser classificado com ajuda de uma camada totalmente conectada [1].

O algoritmo YOLO recebe uma imagem como *input* e usa uma rede neural convolucional profunda para detectar os objetos nessa imagem. A arquitetura do modelo da CNN que forma o escopo da YOLO é mostrada na figura 1

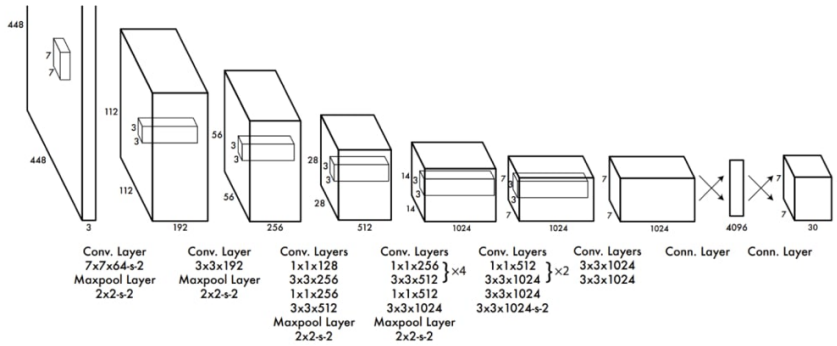


Figura 1: Arquitetura da Rede Neural Convolucional (CNN) YOLO

Como a rede neural teria que ser executada no computador de bordo de um robô humanoide que não possuía GPU, foi utilizada uma versão simplificada da Fast YOLO, mantendo, assim, a capacidade da rede em executar detecções em tempo real dentro das limitações do processador disponível.

As primeiras camadas convolucionais do modelo são treinadas previamente usando ImageNet. Esse modelo pré-treinado é feito para aumentar a performance, já que pesquisas anteriores mostraram que adicionar camadas convolucionais e conectadas melhoram a performance do modelo. A última camada totalmente conectada da rede prediz as probabilidades da classe buscada estar na imagem e a posição e tamanho da *bounding box* desse objeto, caso haja.

YOLO divide a imagem inicial em um *grid* de $S \times S$ células. Se o centro do objeto estiver dentro da célula, essa célula é a responsável por detectar o objeto. Cada célula do *grid* prediz B *bounding boxes* e as suas respectivas Confianças. Confiança é uma medida que reflete quanto o modelo acredita

que que aquela célula contém o centro da imagem e quão preciso ele acredita que sua predição é.

YOLO prediz várias *bounding boxes* por célula. No treinamento da rede, queremos que apenas um *bounding box predictor* seja responsável por cada objeto. YOLO atribui um preditor para ser responsável por cada objeto previsto baseado na predição que possui a maior precisão no momento. Isso leva a "especialização"entre *bounding box predictors*. Cada *predictor* fica melhor em prever certos tamanhos, *aspect ratios* ou classes de objetos, de modo a obter uma melhor performance geral.

3 Análise dos Resultados

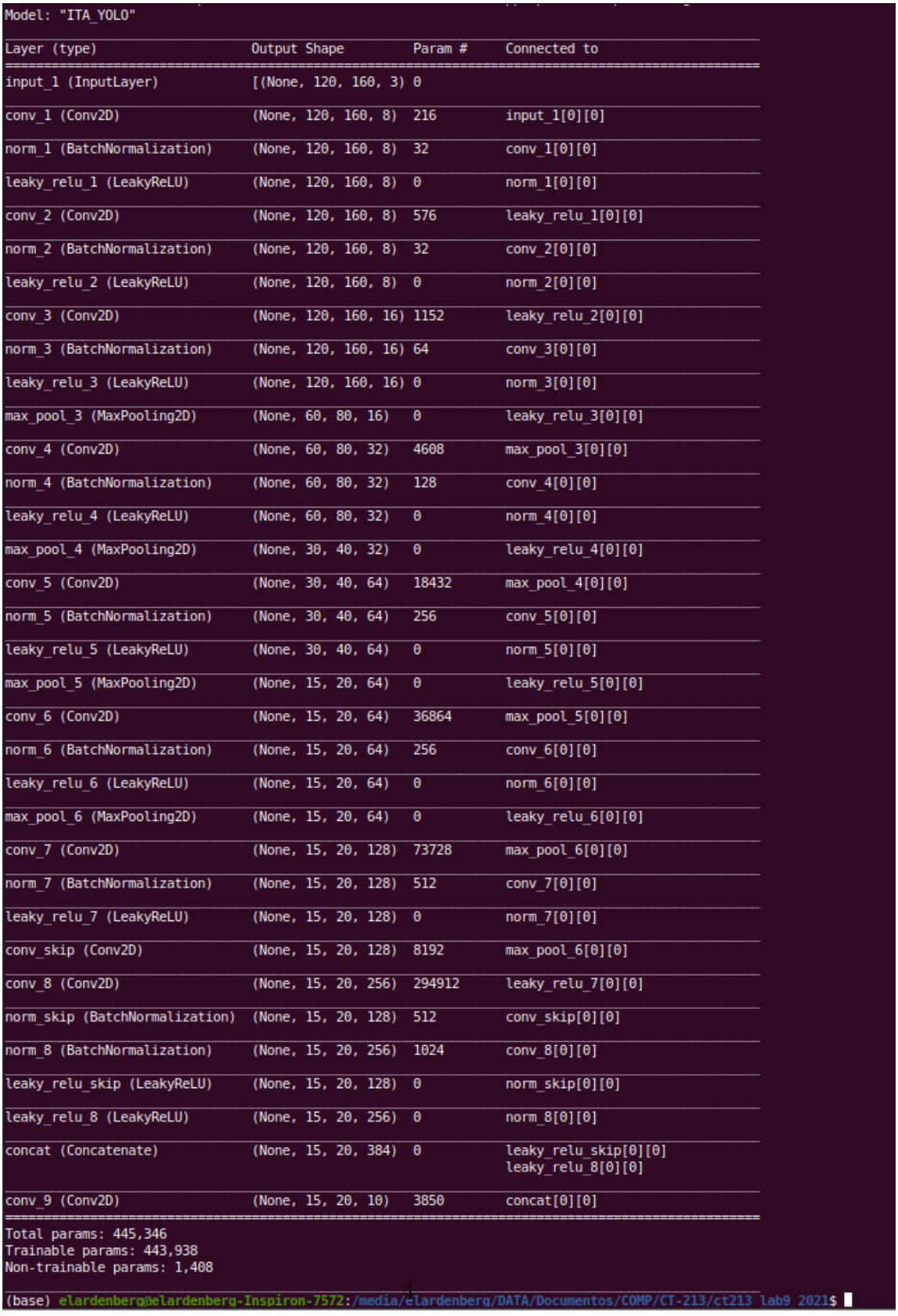
3.1 Implementação do modelo da Rede Neural

Neste Laboratório, foi implementado no código **make_detector_network.py** o modelo em Keras da rede neural YOLO, com a seguinte arquiterura:

Tabela 1: Arquitetura da Lenet-5 para este Laboratório

#Layer	**Tipo**	**# Filt.**	**Saída**	**Kernel**	**Stride**	**Ativação**
Entrada	Imagem	3	120×160	-	-	-
1	Conv2D	8	120×160	3×3	1	leaky_relu
2	Conv2D	8	120×160	3×3	1	leaky_relu
3	Conv2D	16	120×160	3×3	1	leaky_relu
3	MaxPooling2D	16	60×80	2×2	2	-
4	Conv2D	32	60×80	3×3	1	leaky_relu
4	MaxPooling2D	32	30×40	2×2	2	-
5	Conv2D	64	30×40	3×3	1	leaky_relu
5	MaxPooling2D	64	15×20	2×2	2	-
6	Conv2D	64	15×20	3×3	1	leaky_relu
6	MaxPooling2D	64	15×20	2×2	2	-
7	Conv2D	128	15×20	3×3	1	leaky_relu
skip	Conv2D	128	15×20	3×3	1	leaky_relu
8	Conv2D	256	15×20	3×3	1	leaky_relu
9	Conv2D	10	15×20	3×3	1	leaky_relu

Depois de executado o código, o resultado obtido foi o sumário mostrado na figura 2.



```
Model: "ITA_YOLO"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #   Connected to
=================================================================================================
input_1 (InputLayer)         [(None, 120, 160, 3)] 0
_________________________________________________________________________________________________
conv_1 (Conv2D)              (None, 120, 160, 8)  216         input_1[0][0]
_________________________________________________________________________________________________
norm_1 (BatchNormalization)  (None, 120, 160, 8)  32          conv_1[0][0]
_________________________________________________________________________________________________
leaky_relu_1 (LeakyReLU)     (None, 120, 160, 8)  0           norm_1[0][0]
_________________________________________________________________________________________________
conv_2 (Conv2D)              (None, 120, 160, 8)  576         leaky_relu_1[0][0]
_________________________________________________________________________________________________
norm_2 (BatchNormalization)  (None, 120, 160, 8)  32          conv_2[0][0]
_________________________________________________________________________________________________
leaky_relu_2 (LeakyReLU)     (None, 120, 160, 8)  0           norm_2[0][0]
_________________________________________________________________________________________________
conv_3 (Conv2D)              (None, 120, 160, 16) 1152        leaky_relu_2[0][0]
_________________________________________________________________________________________________
norm_3 (BatchNormalization)  (None, 120, 160, 16) 64          conv_3[0][0]
_________________________________________________________________________________________________
leaky_relu_3 (LeakyReLU)     (None, 120, 160, 16) 0           norm_3[0][0]
_________________________________________________________________________________________________
max_pool_3 (MaxPooling2D)    (None, 60, 80, 16)   0           leaky_relu_3[0][0]
_________________________________________________________________________________________________
conv_4 (Conv2D)              (None, 60, 80, 32)   4608        max_pool_3[0][0]
_________________________________________________________________________________________________
norm_4 (BatchNormalization)  (None, 60, 80, 32)   128         conv_4[0][0]
_________________________________________________________________________________________________
leaky_relu_4 (LeakyReLU)     (None, 60, 80, 32)   0           norm_4[0][0]
_________________________________________________________________________________________________
max_pool_4 (MaxPooling2D)    (None, 30, 40, 32)   0           leaky_relu_4[0][0]
_________________________________________________________________________________________________
conv_5 (Conv2D)              (None, 30, 40, 64)   18432       max_pool_4[0][0]
_________________________________________________________________________________________________
norm_5 (BatchNormalization)  (None, 30, 40, 64)   256         conv_5[0][0]
_________________________________________________________________________________________________
leaky_relu_5 (LeakyReLU)     (None, 30, 40, 64)   0           norm_5[0][0]
_________________________________________________________________________________________________
max_pool_5 (MaxPooling2D)    (None, 15, 20, 64)   0           leaky_relu_5[0][0]
_________________________________________________________________________________________________
conv_6 (Conv2D)              (None, 15, 20, 64)   36864       max_pool_5[0][0]
_________________________________________________________________________________________________
norm_6 (BatchNormalization)  (None, 15, 20, 64)   256         conv_6[0][0]
_________________________________________________________________________________________________
leaky_relu_6 (LeakyReLU)     (None, 15, 20, 64)   0           norm_6[0][0]
_________________________________________________________________________________________________
max_pool_6 (MaxPooling2D)    (None, 15, 20, 64)   0           leaky_relu_6[0][0]
_________________________________________________________________________________________________
conv_7 (Conv2D)              (None, 15, 20, 128)  73728       max_pool_6[0][0]
_________________________________________________________________________________________________
norm_7 (BatchNormalization)  (None, 15, 20, 128)  512         conv_7[0][0]
_________________________________________________________________________________________________
leaky_relu_7 (LeakyReLU)     (None, 15, 20, 128)  0           norm_7[0][0]
_________________________________________________________________________________________________
conv_skip (Conv2D)           (None, 15, 20, 128)  8192        max_pool_6[0][0]
_________________________________________________________________________________________________
conv_8 (Conv2D)              (None, 15, 20, 256)  294912      leaky_relu_7[0][0]
_________________________________________________________________________________________________
norm_skip (BatchNormalization) (None, 15, 20, 128)  512         conv_skip[0][0]
_________________________________________________________________________________________________
norm_8 (BatchNormalization)  (None, 15, 20, 256)  1024        conv_8[0][0]
_________________________________________________________________________________________________
leaky_relu_skip (LeakyReLU)  (None, 15, 20, 128)  0           norm_skip[0][0]
_________________________________________________________________________________________________
leaky_relu_8 (LeakyReLU)     (None, 15, 20, 256)  0           norm_8[0][0]
_________________________________________________________________________________________________
concat (Concatenate)         (None, 15, 20, 384)  0           leaky_relu_skip[0][0]
                                                              leaky_relu_8[0][0]
_________________________________________________________________________________________________
conv_9 (Conv2D)              (None, 15, 20, 10)   3850        concat[0][0]
=================================================================================================
Total params: 445,346
Trainable params: 443,938
Non-trainable params: 1,408
_________________________________________________________________________________________________
(base) elardenberg@elardenberg-Inspiron-7572:/media/elardenberg/DATA/Documentos/COMP/CT-213/ct213_lab9_2021$
```

Figura 2: Sumário da versão do modelo da YOLO criado para este laboratório

3.2 Implementação da detecção de objetos com YOLO

No código *yolo_detector.py*, foram implementados os métodos *detect(), preprocess_image()* e *process_yolo_output()*. Esse código consiste na criação da classe *YoloDetector* cujo objeto é chamado pelo código *test_vision_detector.py*, por meio da função *detect()*.

O método *detect()* recebe o *output* do método *preprocess_image()* e o utiliza para fazer a previsão da rede com o método *predict()* do modelo YOLO. Ele então passa esse *output*, depois de processado por *process_yolo_output()*, para os objetos que se deseja encontrar na imagem, que são a bola e as traves de futebol.

O método *preprocess_image()* recebe uma imagem, que originalmente é 680×480 px e a transforma em um *array* que representa os 3 canais de cor, em escala de 0 a 1, da mesma imagem em dimensões 160×120.

O método *process_yolo_output()* recebe o *array* previsto pela rede, que possui dimensões $15 \times 20 \times 10$. Os valores 15×20 são o *grid* criado pela rede neural e 10 é o vetor de *features* que a rede obtém. Esse vetor corresponde à tupla [probabilidade, posição x, posição y, largura, altura] da bola ligada a uma tupla semelhante para a trave. Assim, para cada uma das células do *grid*, foi realizada uma busca para saber qual era aquele com maior probabilidade para a bola e quais as duas células de maior probabilidade para a trave. Esses valores foram guardados nos respectivos vetores de *features* e, antes de retornar o valor, foi realizada uma normalização dos valores com o auxílio de mais uma função implementada, denominada de *object_normalization()*.

O método *object_normalization()* recebe um desses objetos, sua posição (x, y) e o tamanho relativo de sua *bounding box*, denominado de *anchor*, com as transformações dadas a seguir:

$$\begin{aligned} obj[0] &= \sigma(obj[0]) \\ obj[1] &= (x + \sigma(obj[1])) * 32 \\ obj[2] &= (y + \sigma(obj[2])) * 32 \\ obj[3] &= anchor\_x * exp(obj[3]) * 640 \\ obj[4] &= anchor\_y * exp(obj[4]) * 640 \end{aligned} \tag{1}$$

Após isso, foram geradas imagens com os objetos detectados para cada uma das 10 imagens recebidas inicialmente. As figuras 3 a 7 mostram alguns dos resultados obtidos.

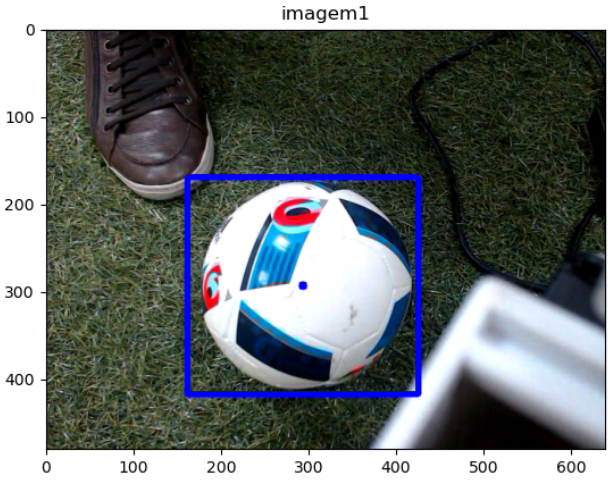


Figura 3: Objeto detectado na imagem pelo algoritmo YOLO

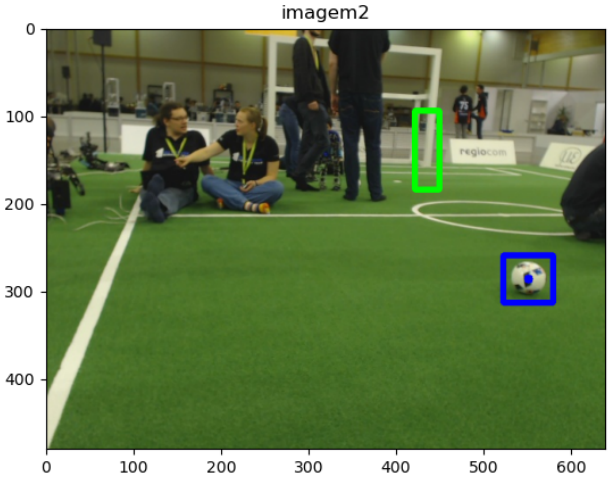


Figura 4: Objeto detectado na imagem pelo algoritmo YOLO

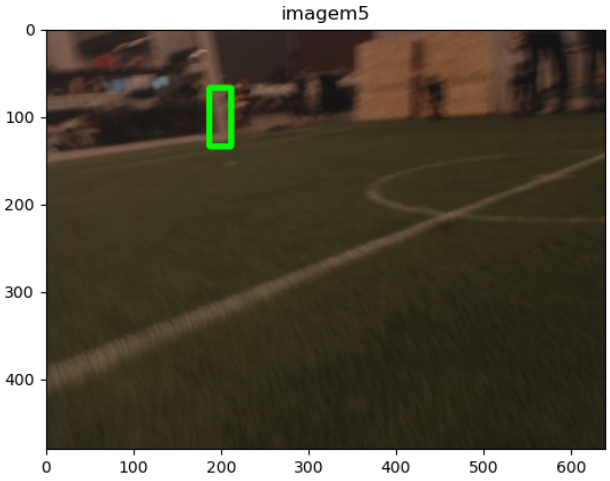


Figura 5: Objeto detectado na imagem pelo algoritmo YOLO

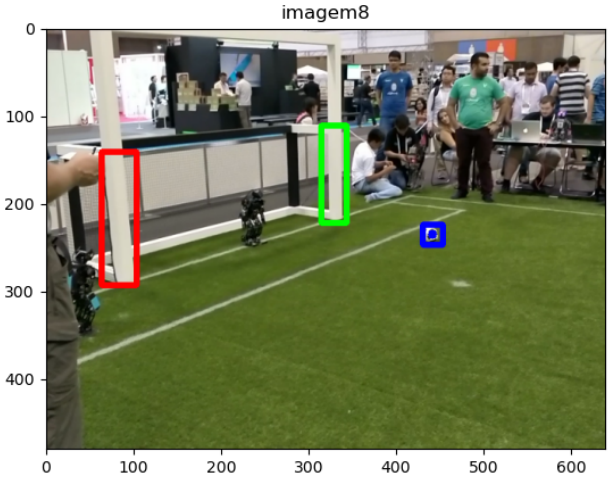


Figura 6: Objeto detectado na imagem pelo algoritmo YOLO

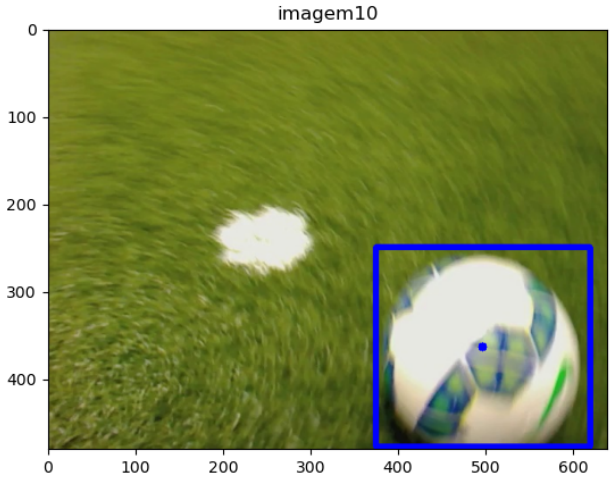


Figura 7: Objeto detectado na imagem pelo algoritmo YOLO

As imagens mostram uma detecção relativamente precisa dos objetos, sendo o algoritmo capaz de detectá-los de vários tamanhos, com imagens com variada qualidade de foco, e perceber se há de fato um objeto na imagem ou não. As demais imagens que não entraram neste relatório apontam para a mesma conclusão.

4 Conclusão

Dos exemplos mostrados nas imagens deste relatório, bem como da literatura a respeito desse método, é possível perceber que o algoritmo YOLO é capaz de detectar a presença ou não de objetos predefinidos de uma forma bem ampla e com alta precisão, com diferentes tamanhos e qualidades de foco da imagem.

Embora o treinamento da rede não tenha sido feito, já utilizando-se de uma rede com os hiperparâmetros devidamente otimizados, a execução dessa rede revelou-se bastante rápida, haja vista que o YOLO é um algoritmo feito para detecção em tempo real, e o processador da máquina utilizada para este laboratório foi o Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, com mais de 6 anos de uso.

Referências

[1] Yolo: Algorithm for object detection explained. Acessado em 14 de Abril de 2024. URL: https://www.v7labs.com/blog/yolo-object-detection.