UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Adriana Carrillo Rios

COMPARAÇÃO DOS ALGORITMOS YOLOV3 E SSD PARA IDENTIFICAÇÃO DE OBJETOS EM IMAGENS, EM UM DATASET PARA NAVEGAÇÃO DE ROBÔS EM AMBIENTES INTERIORES.

Santa Maria, RS
2022

Adriana Carrillo Rios

COMPARAÇÃO DOS ALGORITMOS YOLOV3 E SSD PARA IDENTIFICAÇÃO DE OBJETOS EM IMAGENS, EM UM DATASET PARA NAVEGAÇÃO DE ROBÔS EM AMBIENTES INTERIORES.

Trabalho de conclusão de curso apresentado ao curso de Bacharelado em Engenharia de Controle e Automação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de **Bacharel em Engenheira de Controle e Automação**.

Orientador: Prof. Dr. Daniel Fernando Tello Gamarra

Santa Maria, RS
2022

Adriana Carrillo Rios

COMPARAÇÃO DOS ALGORITMOS YOLOV3 E SSD PARA IDENTIFICAÇÃO DE OBJETOS EM IMAGENS, EM UM DATASET PARA NAVEGAÇÃO DE ROBÔS EM AMBIENTES INTERIORES.

Trabalho de conclusão de curso apresentado ao curso de Bacharelado em Engenharia de Controle e Automação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de **Bacharel em Engenheira de Controle e Automação**.

Aprovado em 4 de fevereiro de 2022:

_____

Daniel Fernando Tello Gamarra, Prof. Dr. (UFSM)
(Presidente/Orientador)

_____

Daniel Welfer, Prof. Dr. (UFSM)

_____

Rodrigo da Silva Guerra, Prof. Dr. (UFSM)

Santa Maria, RS
2022

RESUMO

COMPARAÇÃO DOS ALGORITMOS YOLOV3 E SSD PARA IDENTIFICAÇÃO DE OBJETOS EM IMAGENS, EM UM DATASET PARA NAVEGAÇÃO DE ROBÔS EM AMBIENTES INTERIORES.

AUTOR: Adriana Carrillo Rios
ORIENTADOR: Daniel Fernando Tello Gamarra

Este trabalho apresenta um estudo relacionado à área de visão computacional. O objetivo principal é comparar os algoritmos YOLOv3 e SSD para identificação de objetos em imagens. Para viabilizar o objetivo pretendido, em um primeiro momento se realiza a comparação dos modelos sem executar o aumento de dados. Em seguida, é realizado adequadamente o aumento de dados de tal forma a melhorarmos o desempenho dos modelos.

O modelo YOLOv3 apresentou um desempenho um pouco melhor em relação ao SSD, sem realizar aumento de dados, pois o F1-score para YOLOv3 é de 75% e para SSD é de 73%. Por outro lado, ao realizar o aumento de dados, o modelo SSD é favorecido, o F1-score obtido foi de 81% e o YOLOv3 obteve 80%.

Palavras chaves: Yolov3, SSD, Aumento de Dados, Identificação de Objetos.

ABSTRACT

COMPARISON OF THE YOLOV3 AND SSD ALGORITHMS FOR OBJECT IDENTIFICATION IN IMAGES, IN AN INDOOR MOBILE ROBOT NAVIGATION DATASET.

AUTHOR: Adriana Carrillo Rios
ADVISOR: Daniel Fernando Tello Gamarra

This work presents a study related to the research area of computer vision. The main objective is to compare YOLOv3 and SSD algorithms for identifying objects in images. In order to achieve the intended objective, at first, the algorithms were trained and compared without data augmentation. After, the data augmentation was executed for improving the performance of the algorithms.

The YOLOv3 model presented a slightly better performance compared to the SSD, without performing data augmentation, because the F1-score for YOLOv3 is 75% and for SSD it is 73%. On the other hand, when performing data augmentation, the SSD model is favored, the F1-score obtained was 81% and, YOLOv3 obtained 80%.

Keywords: Yolov3, SSD, Data Augmentation, Object Identification.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Com o contínuo crescimento do número de dados, surge a necessidade de desenvolver técnicas cada vez mais eficientes que ajudem a extrair informações úteis desses dados, e assim, obter melhor aproveitamento dos dispositivos e tecnologias. Neste sentido, a Inteligência Artificial (IA) surge como uma ferramenta poderosa.

Uma aplicação importante é a do reconhecimento de objetos em imagens, que está se tornando cada vez mais útil, por exemplo, na área médica, de segurança e na navegação de veículos autônomos. Entre os modelos que ajudam a detectar objetos em imagens, tem-se o YOLO (*You Only Look Once*) e o SSD (*Single Shoot Multibox Detection*).

O trabalho de conclusão de curso explorará e comparará dois algoritmos para detecção de objetos em imagens, em primeiro momento sem aplicar a técnica de *data augmentation* e depois mediante *data augmentation*. Para isto, é necessário preparar os dados, treinar os algoritmos YOLO e SSD com e sem *data augmentation*, testar o treinamento para avaliar e comparar as estruturas usando as métricas mais conhecidas.

1.1 OBJETIVOS

1.1.1 Objetivo geral

O objetivo geral deste projeto é apresentar um estudo na área de visão computacional com o propósito de comparar os algoritmos YOLO e SSD para identificação de objetos em imagens. A comparação será executada em dois momentos, primeiro sem aumento de dados e depois realizando esse aumento.

1.1.2 Objetivos específicos

Os objetivos específicos deste trabalho são:

- Realizar o pré-processamento dos dados para os modelos YOLO e SSD.
- Executar aumento de dados para o *dataset* original, mas considerando apenas os dados para treinamento;

- Treinar os algoritmos YOLO e SSD em um *dataset* para navegação de robôs em ambientes interiores;
- Realizar predições com cada um dos modelos treinados;
- Avaliar e comparar os algoritmos YOLO e SSD para identificação de objetos em imagens.

1.2 DESCRIÇÃO DO PROBLEMA

Neste trabalho, o problema a ser estudado consiste na comparação de dois modelos que realizam a detecção de objetos em imagens, utilizando conhecimentos na área de visão computacional. Para isso é necessário estudar os modelos de detecção de objetos YOLO e SSD, e assim poder realizar uma adequada implementação deles. O maior desafio deste projeto é a execução de um apropriado aumento de dados de tal maneira a melhorar todas as métricas para os dois modelos.

2. REFERENCIAL TEÓRICO

Este capítulo apresenta a teoria necessária para melhor entender os conceitos utilizados no desenvolvimento deste projeto e assim lograr uma maior compreensão do mesmo.

2.1 DEEP LEARNING

O *deep learning* ou aprendizado profundo permite que modelos computacionais aprendam representações de dados com vários níveis de abstração, melhorando consideravelmente o reconhecimento visual de objetos [2].

As redes neurais são utilizadas em *deep learning*, entre outras coisas, para reconhecimento de objetos. Elas contêm camadas e, em cada camada, temos os neurônios. Os neurônios dessas redes são as unidades elementares que processam as informações, estabelecendo uma resposta para elas. Pense no funcionamento do cérebro humano para entender melhor, pois de maneira análoga funcionam as redes neurais artificiais, e por isso elas recebem esse nome.

2.2 REDES NEURAIS

Uma rede neural artificial é um dos métodos com os quais é possível implementar uma inteligência artificial [3]. Ela recebe dados de entrada (estímulos) que são detectados, por exemplo, por sensores. Esses dados são processados produzindo assim saídas que devem ser bem próximas das saídas desejadas, mas isso só acontece quando a rede é bem estruturada e treinada.

Esses dados de entrada sofrem alterações ao passarem pelos neurônios de camada em camada. Cada valor de entrada é ponderado, ou seja, multiplicado pelo seu peso correspondente. Em seguida, esses valores são somados formando uma combinação linear. A essa combinação linear é adicionada uma constante chamada *Bias*. O resultado dessa soma passa por uma função de ativação para assim obter a saída do neurônio.

Na saída dos neurônios é usada, geralmente, uma função de ativação. Essa função nos dá o processamento em cada neurônio para assim alterar a saída dele. A função de ativação é

importante, pois com ela é possível a aprendizagem da rede neural. Ela produz o entendimento do neurônio para que a informação seja descartada ou não. Na ausência dela, a rede neural vira só um modelo de regressão linear.

Na rede neural, o peso mostra a eficácia de uma entrada específica. Quanto maior o peso de entrada, mais ela influenciará na rede neural. Por outro lado, Bias é uma constante que ajuda o modelo de tal maneira que ele possa se adaptar melhor aos dados fornecidos [3].

As redes neurais precisam de uma boa aprendizagem ou treinamento. Neste caso, deve-se ter um banco de dados, que além de ter a quantidade de informações suficientes também é importante a qualidade delas. Durante o treinamento, é desejável que se faça a medição do erro para determinar quão válida é a nossa rede neural.

2.3 REDE NEURAL CONVOLUCIONAL

Uma Rede Neural Convolucional ou ConvNet é um tipo de rede neural que pode ser utilizada para classificar imagens, reconhecer objetos e para fazer agrupamento por similaridade. A característica que a diferencia de outros tipos de redes é o padrão de conexão entre os neurônios, que está baseado no córtex visual dos animais [4], ou seja, há uma sobreposição de neurônios de tal maneira que exista uma maior amplitude no campo de cobertura da rede. Em geral, os elementos que a compõem é a camada de entrada, as camadas de convolução, que vão alternando com camadas de *pooling*, e por último a camada completamente conectada.

A camada de convolução é a mais importante. Ela é composta por um conjunto de filtros ou *Kernels* capazes de aprender, eles se deslocam (realizando operações de convolução) até percorrerem toda a imagem. Esses filtros na verdade são matrizes, cujos valores em cada componente do arranjo matricial podem ser interpretados como sendo os pesos. Isto permite que sejam obtidos mapas de características (*feature maps*) de uma região e que, a partir desses mapas, características sejam extraídas. Vale salientar que o tamanho do filtro define o tamanho da vizinhança que cada neurônio da camada irá processar [4].

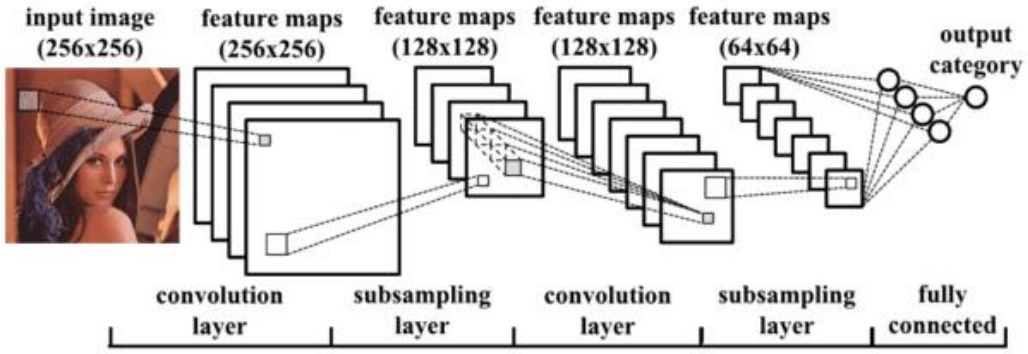
A camada convolucional possui um parâmetro importante para a sua execução, chamado de passo, *stride*. Ele determina o salto que é dado pelo filtro na convolução quanto maior o *stride*, menor é o tamanho da matriz resultante da operação convolução e menos sobreposição. Isto produz uma possível perda de informações importantes que possui a imagem, por esse motivo comumente o *stride* não é maior que 2.

A camada de *pooling/subsampling* é onde são reduzidas as matrizes para uma de menor dimensão. Assim, representamos os dados de uma forma simplificada e por sua vez aceleramos o processamento que deve ser feito para as camadas seguintes. Outro benefício é o aumento da invariância espacial da rede, ou seja, ela fica mais robusta à possíveis variações na posição das características mapeadas. Este processo ajuda a evitar um possível *overfitting* (que se apresenta quando o modelo está muito bem ajustado aos dados de treinamento, mas não é possível prever novos resultados). Nesta camada é executada a operação de *pooling* que consiste em atribuir uma métrica para a região da matriz que é reduzida.

A camada totalmente conectada (*fully connected layer*) normalmente estabelece quais atributos de alto nível são os que mais se correlacionam com a classe do objeto, ou seja, é a camada responsável por classificar os dados uma vez que ela consegue separá-los para cada classe de resposta.

À medida que a rede apreende, os pesos são atualizados e isto é possível apresentando à rede os dados de entrada e as suas respectivas respostas (pares de treinamento). Quando começa o treinamento, o erro é propagado da saída para a entrada (*backpropagation*) ajustando os pesos. Esse ajuste é realizado até minimizar o erro observado ao comparar a saída da rede com o resultado esperado, chegando assim o mais próximo possível da saída desejada. Na saída do classificador é idealmente usada a função de ativação *Softmax*, que é considerada uma função para classificação multiclasse. Observe a arquitetura de uma rede neural convolucional comum na Figura 1.

Figura 1 – Modelo Arquitetural de uma rede neural convolucional comum.



Fonte: https://www.lambda3.com.br/2019/09/redes-neurais-artificiais-as-maquinas-pensam/

A importância do uso das ConvNets está no pré-processamento, pois é muito menor em comparação a outros algoritmos de classificação. Enquanto nos métodos anteriores a elas, os filtros são realizados à mão, as ConvNets têm a capacidade de aprender sozinhas esses filtros.

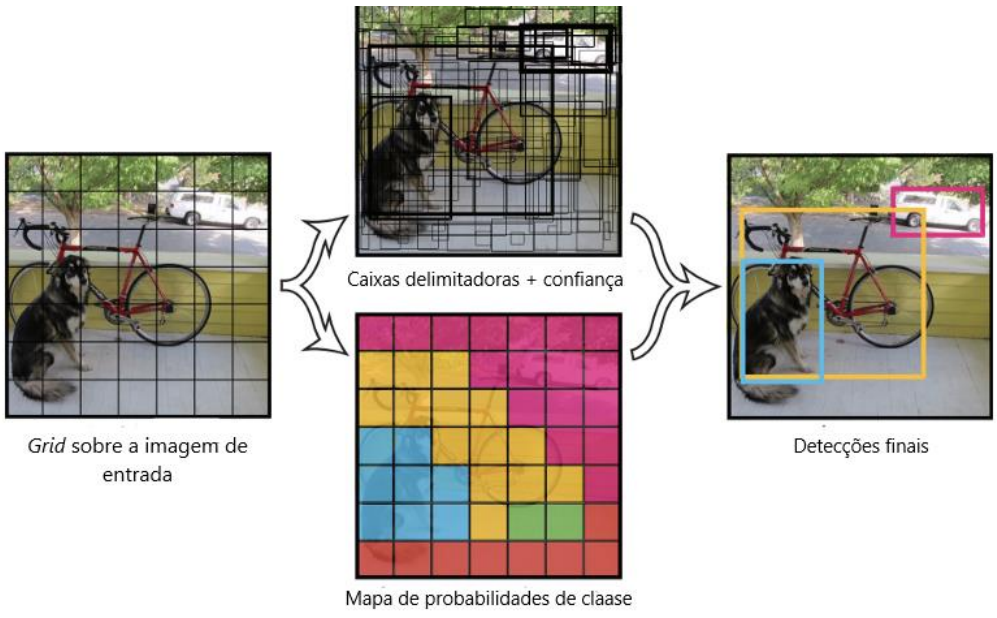
2.4 YOLO - YOU ONLY LOOK ONCE

YOLO é um algoritmo que usa redes neurais convolucionais para detecção de objetos em imagens, podendo ser ainda em tempo real. Essa arquitetura foi apresentada por Joseph Readmon em [5], atualmente existem cinco versões além das versões Fast ou Tiny YOLO.

A estratégia do modelo é a detecção de objetos numa única passada da rede, originando assim seu nome: You Only Look Once (você só olha uma vez), o que faz com que ele seja mais rápido do que outros algoritmos embora não seja o mais preciso. Comparando YOLO com as redes neurais convolucionais comuns a diferença é que essas redes não olham para o quadro completo, apenas para regiões individuais e isto as torna mais lentas.

Neste trabalho será utilizado o modelo YOLOv3, apresentado por Joseph Readmon em [6]. A Figura 2 apresenta um resumo do modelo divide-se a imagem de entrada em um grid de S x S células. Cada célula é responsável por fazer a predição de três caixas delimitadoras, para caso haja mais de um objeto naquela célula. Como YOLO pode detectar vários objetos em uma imagem, isto significa que ele divide a imagem em regiões e além de prever as caixas delimitadoras, também prevê as probabilidades para cada região, criando um mapa de probabilidade de classes. Por último, aplicam-se filtros para eliminar caixas delimitadoras com probabilidade abaixo de um limiar e aquelas que se sobrepõem, obtendo assim as detecções dos objetos na imagem.

Figura 2 – Resumo funcionamento YOLO



Fonte: http://www.lapix.ufsc.br/wp-content/uploads/2018/10/yolo2.png

Em [6] os autores mostram uma arquitetura mais profunda do extrator de características para YOLOv3, a Darknet-53, assim chamada por conter 53 camadas convolucionais, a Figura 3 apresenta a sua estrutura. As 53 camadas podem ser ajustadas, as 21 primeiras são para detecção, as 30 seguintes para extração de características e as duas últimas para classificação. Por último, a rede ainda utiliza algumas camadas residuais (estrutura semelhante a uma *Residual Network*) para ajudar a evitar o sobreajuste e para aprendizagem de recursos, à medida que a rede se aprofunda, mais difícil é aprender bem os recursos. Então, ao se adicionar um atalho, as camadas dentro dele aprenderão o quê adicionar ao recurso antigo para gerar um recurso melhor. Além disso, usa também uma estrutura semelhante a uma FPN (*Feature Pyramid Network*) que ajuda a melhorar bastante a precisão, pois permite que o mapa de recursos atual veja seus recursos em camadas "futuras" e utilize ambos para fazer previsões mais precisas. Com esta técnica, o modelo é mais capaz de capturar as informações do objeto, tanto de baixo nível quanto de alto nível.

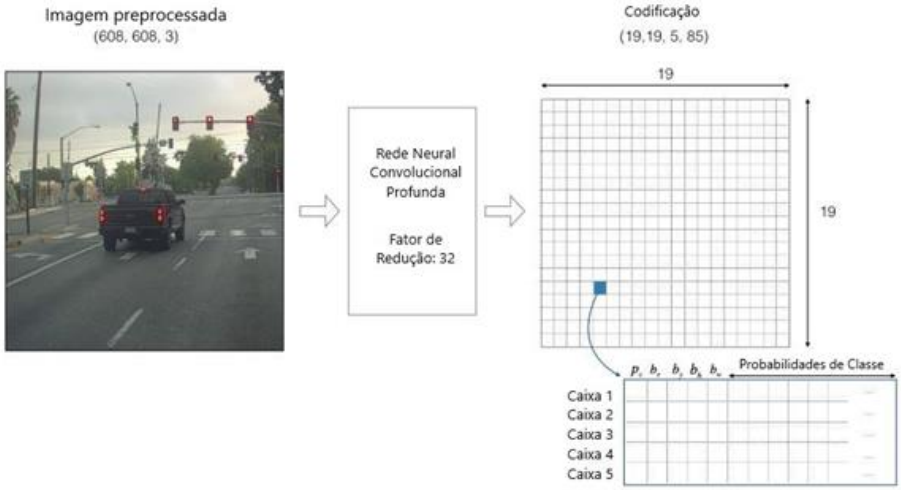
Figura 3 – Estrutura da Darknet-53

	Tipo	Filtros	Tamanho	Saída
	Convolucional	32	3 × 3	256 × 256
	Convolucional	64	3 × 3 / 2	128 × 128
1×	Convolucional	32	1 × 1	
	Convolucional	64	3 × 3	
	Residual			128 × 128
	Convolucional	128	3 × 3 / 2	64 × 64
2×	Convolucional	64	1 × 1	
	Convolucional	128	3 × 3	
	Residual			64 × 64
	Convolucional	256	3 × 3 / 2	32 × 32
8×	Convolucional	128	1 × 1	
	Convolucional	256	3 × 3	
	Residual			32 × 32
	Convolucional	512	3 × 3 / 2	16 × 16
8×	Convolucional	256	1 × 1	
	Convolucional	512	3 × 3	
	Residual			16 × 16
	Convolucional	1024	3 × 3 / 2	8 × 8
4×	Convolucional	512	1 × 1	
	Convolucional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Fonte:http://resources.dbgns.com/study/ObjectDetection/YOLOv3.pdf

Para o funcionamento de YOLOv3, o modelo reduz a amostragem da imagem por um fator chamado de passo da rede. Suponhamos que temos uma imagem de entrada 608×608 e passo da rede 32. Quando a imagem entra na rede neural profunda, sofre um fator de redução de 32. Portanto, o *grid* na imagem terá 19×19 células, pois 608/32 = 19. Observe a Figura 4.

Figura 4 – Imagem sendo processada por YOLOv3



Fonte: https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193

A seguir, a ideia é que o modelo cria o mapa de previsão e a maneira como este se interpreta é que cada célula pode prever um número fixo (3) de caixas delimitadoras. E, espera-se também que cada célula preveja um objeto através de uma dessas caixas delimitadoras, se o centro do objeto cair em uma célula do *grid*, essa célula será responsável por detectar esse objeto. Porém, há um problema com relação a isso, pode ser que tenhamos vários objetos na mesma célula então, é necessário usar retângulos predefinidos (chamados de caixas âncoras) para resolver o problema, segundo os autores em [6]. A ideia é que essas âncoras tenham diferentes formas para cada objeto e associar predições a cada uma delas.

As âncoras de uma maneira mais formal são nuances do que seriam as dimensões das caixas delimitadoras que temos de fato no *dataset* de treinamento e elas mostram para a rede qual é o detector que deve ser responsável por cada uma das caixas. As âncoras são calculadas sobre todas as caixas do conjunto de treinamento, por sua vez, o conjunto é clusterizado de acordo com o número de detectores que será aplicado no método, por padrão, cinco Clusters, mas para YOLOv3 são 3 para cada escala. Os centroides de cada Cluster são utilizados como âncoras para cada detector. O modelo aplica pequenas transformações às caixas âncora para assim prever as caixas delimitadoras.

As fórmulas a continuação descrevem a maneira como a saída da rede é transformada, isto é, os cálculos realizados pelo modelo para dar a previsão das caixas delimitadoras.

$$b_x = \sigma(t_x) + c_x \tag{1}$$

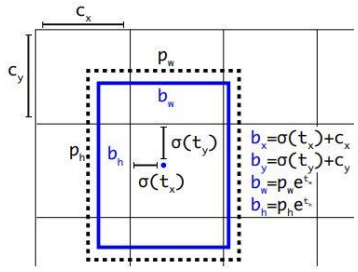
$$b_y = \sigma(t_y) + c_y \quad (2)$$

$$b_w = p_w \, . e^{t_w} \quad (3)$$

$$b_h = p_h \, . e^{t_h} \quad (4)$$

Os valores b_x e b_y são as coordenadas do centro; b_w e b_h a largura e a altura da caixa delimitadora prevista. Além disso, t_x, t_y, t_w, t_h é o que a rede gera. As coordenadas do canto superior esquerdo da célula são c_x e c_y. E, as dimensões das âncoras para a caixa são p_w e p_h (as caixas âncoras são definidas apenas por largura e altura). Note que as coordenadas centrais da caixa delimitadora são previstas usando uma função Sigmoide (σ), isto é para manter o centro na célula que está realizando a previsão. Na Figura 5, pode-se observar a caixa âncora e a previsão de caixa delimitadora (a caixa azul).

Figura 5 – Caixas delimitadoras com previsão de localização

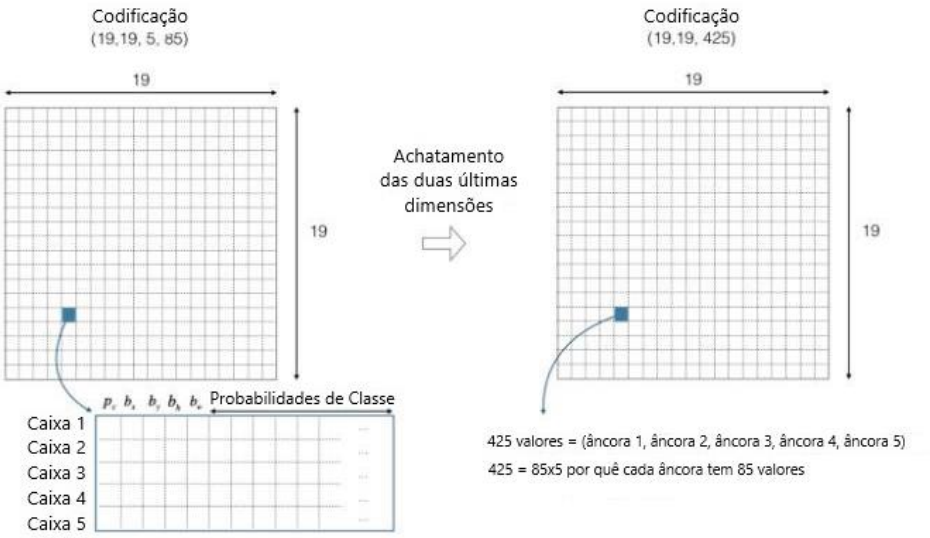


Fonte: https://pjreddie.com/media/files/papers/YOLOv3.pdf

Supondo que temos ainda uma imagem de entrada 608×608, passo da rede 32 e que agora temos 5 caixas âncoras e 80 classes de objetos. Com isto, continuamos a descrever os próximos passos desenvolvidos pelo modelo YOLOv3.

Note que para cada célula do *grid* o modelo constrói um arranjo (tensor). Esse arranjo possui atributos para cada caixa âncora, que descrevem a probabilidade P_c de existir ou não um objeto na caixa delimitadora (*objectivity score*), as coordenadas do centro, as dimensões da caixa e as probabilidades de classe. Ou seja, o modelo faz a predição de um tensor de tamanho $m_1 \times m_2 \times k \times (n+5)$, onde $m_1 \times m_2$ é o tamanho do *grid,* k é o número de caixas âncoras e n o número de classes. Para o exemplo, temos 19×19×5×(80+5). Para continuar com o raciocínio sobre o funcionamento do YOLOv3, achata-se as duas últimas dimensões da codificação (19,19,5,85). Portanto, a saída é dada por a codificação (19,19,425), isto é, cada célula do *grid* 19×19 retorna 425 valores, observe a Figura 6.

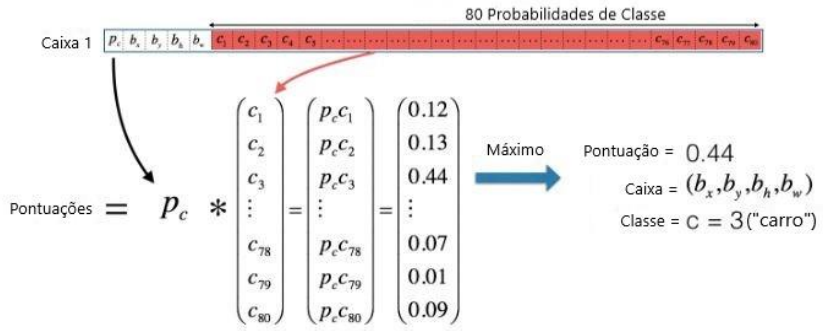
Figura 6 – Flatten das duas últimas dimensões



Fonte: https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193

Agora, para cada caixa (de cada célula), se calcula o vetor dado pelo produto escalar de P_c e o vetor que contém as 80 probabilidades de classe. Assim, se extrai uma probabilidade de que a caixa contenha uma determinada classe. Logo, escolhe-se a maior componente do vetor obtido, observe a Figura 7.

Figura 7 – Detecção de um carro pela caixa (b_x, b_y, b_h, b_w)



Fonte: https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193

Para entender melhor o que YOLO está prevendo em uma imagem note que, para cada uma das células da grade 19x19, ele encontra o máximo dos scores de probabilidade, mas isto se faz calculando o máximo entre as cinco caixas âncoras e as diferentes classes.

São muitas as caixas para as quais o modelo atribuiu uma alta confiança. Logo, deve-se filtrar a saída do algoritmo para um número bem menor de caixas. Então, se aplicam dois filtros, um serve para eliminar as caixas com pontuação baixa, neste caso escolhe-se um

limiar. O outro filtro seleciona apenas uma caixa quando várias caixas se sobrepuserem sempre e quando detectem o mesmo objeto, ou seja, aplica-se a supressão não máxima (NMS). Este último executa um loop verificando se há sobreposições entre as caixas delimitadoras, para cada classe. Para isto utiliza o índice IoU (*Intersection over Union*), o qual é apresentado na seção 2.5. Para duas caixas da mesma classe que estiverem acima de um certo limiar de IoU, o algoritmo NMS descarta a caixa com a pontuação de confiança mais baixa**,** eliminando assim os retângulos que mais se superpõem. Assim, YOLOv3 obtém o resultado, observe a Figura 8, só o retângulo que interessa para a detecção.

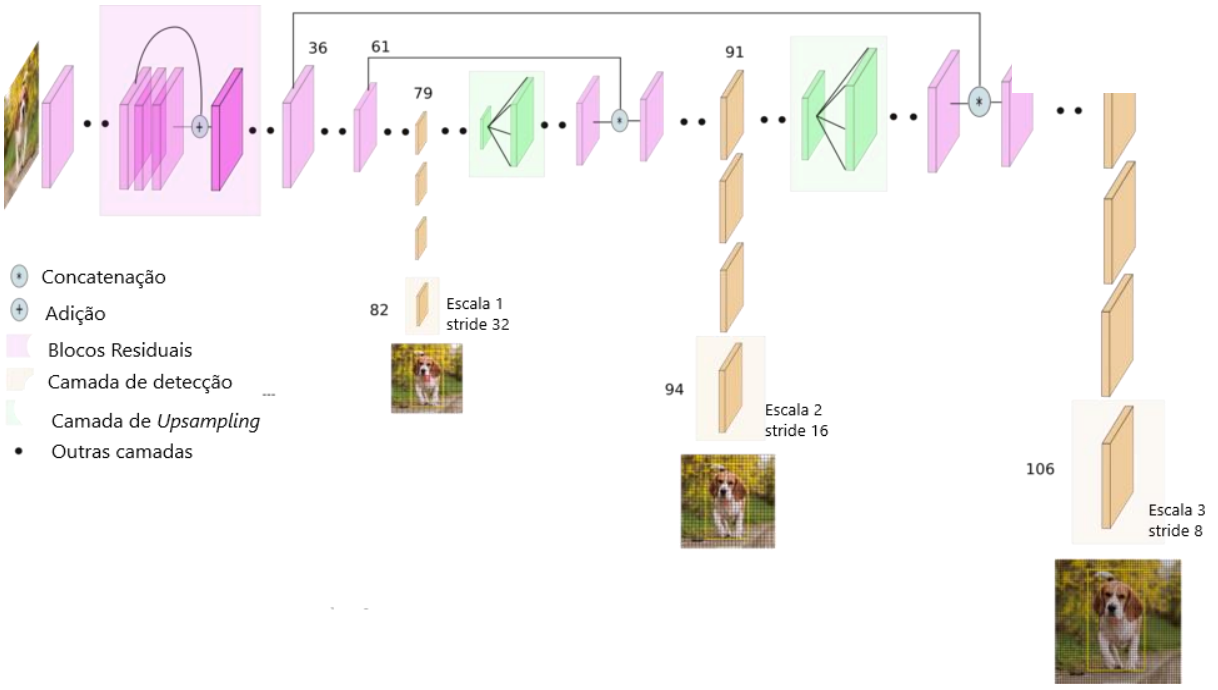
Para classificação, a camada *Softmax* foi substituída por uma camada convolucional 1x1 com função logística. Pois, ao usar *Softmax*, assume-se que cada saída pertence a uma única classe, mas em alguns casos temos rótulos semanticamente semelhantes (por exemplo, Mulher e Pessoa), então *Softmax* não permite que a rede generalize bem a distribuição de dados. Portanto, uma das melhorias para YOLOv3 é usar uma função logística para poder realizar classificação multi-rótulo.

Figura 8 – Resultado final do YOLOv3

Fonte: https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193

YOLOv3 faz previsões em três escalas diferentes, segundo os autores em [6] é para que ele possa fazer a previsão de objetos pequenos pois as versões anteriores do YOLO tinham problemas ao tentar detectar eles, observe a Figura 9. Os autores também realizam um procedimento de *Upsampling*, que pode ajudar o modelo a aprender recursos refinados, que são instrumentos para detectar objetos pequenos.

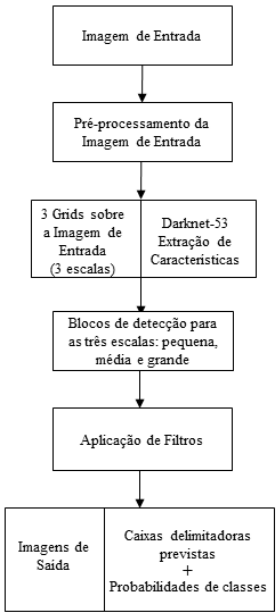
Figura 9 – Resumo Arquitetura YOLOv3



Fonte: https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b

Por último, apresentamos um diagrama que permite ver resumidamente o funcionamento do modelo YOLOv3, observe a Figura 10.

Figura 10 – Diagrama do funcionamento do modelo YOLOv3



Fonte: O Autor.

2.5 SSD - SINGLE SHOT MULTIBOX DETECTOR

O modelo de detecção de objetos SSD: *Single Shot Multibox Detector* foi apresentado por Wei Liu *et al* em [7]. Essa técnica é chamada assim, pois as tarefas de localização e classificação de objetos são realizadas em uma única passagem da rede, ainda em tempo real. *Multibox*, pois é o nome de uma técnica para regressão de caixa delimitadora a qual foi integrada ao SSD. Essa técnica é inspirada no trabalho de Szegedy em [8] e consiste em uma proposta rápida de coordenadas de caixas delimitadoras independentes de classe.

A detecção de objetos no modelo SSD é composta por duas partes: extração de mapas de recursos e aplicação de filtros convolucionais para detecção. Resumidamente, o SSD usa uma única rede neural profunda. Ele tem como base uma rede convolucional *feed-forward* que produz uma coleção de tamanho fixo de caixas delimitadoras e scores para a presença de instâncias de classe de objeto nessas caixas, seguida por uma etapa de supressão não máxima (análogo a YOLOv3) para as detecções finais. As primeiras camadas da rede têm uma arquitetura padrão usada para classificação de imagem de alta qualidade, e foi chamada de rede base. Logo, adiciona-se uma estrutura auxiliar a rede para produzir detecções, ela permite extrair recursos em várias escalas e diminuir progressivamente o tamanho da entrada para cada camada subsequente. A continuação apresentamos os principais recursos dessa rede segundo os autores em [7].

Mapas de recursos em várias escalas para detecção. Adiciona camadas de recursos convolucionais ao final da rede base. Essas camadas vão diminuindo de tamanho e permitem previsões de detecções em várias escalas.

Preditores convolucionais para detecção. Depois de serem extraídos os mapas de recursos, o SSD aplica filtros convolucionais 3×3 para que cada localização (que é o equivalente a uma célula no YOLO) possa fazer previsões. Cada filtro gera uma quantidade de canais que são as pontuações para cada classe mais uma caixa delimitadora.

Caixas padrão e proporções de aspecto. Associa-se um conjunto de caixas delimitadoras padrão a cada localização do mapa de recursos. A posição de cada caixa em relação à localização correspondente é fixa. Em cada localização do mapa de recursos se faz previsão dos deslocamentos relativos às formas de caixa padrão, além das pontuações por classe que indicam a presença de uma instância de classe em cada uma dessas caixas. Neste caso, esses valores correspondem a $\Delta(c_x, c_y, w, h)$, que são as compensações para a caixa

padrão em cada localização. Observe que aqui, essas caixas padrão são aplicadas a vários mapas de recursos de diferentes resoluções.

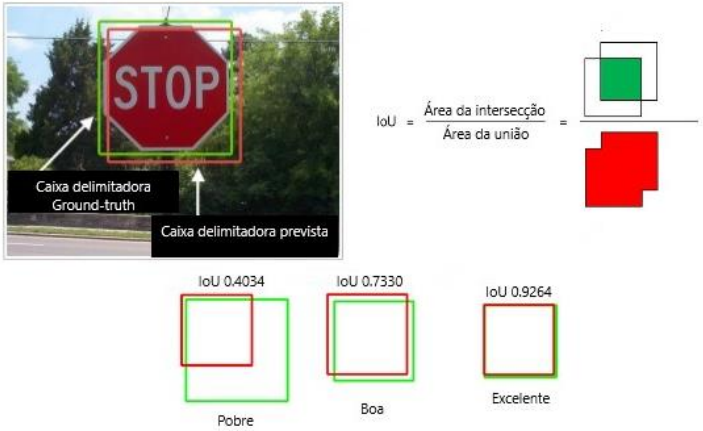
Então, de fato está sendo possível ter diferentes formas de caixa padrão e isso permite discretizar com eficiência o espaço de possíveis formas de caixa de saída. Ou seja, o modelo discretiza esse espaço num conjunto de caixas padrão em diferentes proporções ou taxas de aspecto (*aspect ratios*) e os escala de acordo com a localização do mapa de recurso identificado.

As caixas padrão são pré-selecionadas manualmente e com cuidado para cobrir uma grande variedade de objetos da vida real. Elas são caixas delimitadoras predefinidas e de tamanho fixo que se aproximam da distribuição das caixas *Ground-truth*. Como todas as localizações do mapa de recursos são associadas a um conjunto de caixas padrão de diferentes dimensões e proporções, em teoria isso deve permitir que o SSD generalize qualquer tipo de entrada, sem a necessidade de uma fase de pré-treinamento. Observe que para cada uma das camadas do mapa de recursos se tem o mesmo conjunto de caixas padrão e ainda elas estão centralizadas na localização correspondente. Porém, camadas diferentes usam conjuntos diferentes de caixas padrão para personalizar as detecções de objetos em diferentes resoluções.

A continuação apresenta-se um exemplo para entender melhor alguns dos conceitos mostrados anteriormente, mas antes disto vejamos o que significa mineração negativa.

Ao fazermos o treinamento podemos ter que, a maioria das caixas delimitadoras tenham baixo IoU (*Intersection over Union*). O IoU pode ser entendido como um índice no qual as caixas delimitadoras previstas que se sobrepõem fortemente às caixas *Ground-truth* têm pontuações mais altas do que aquelas com menos sobreposição, observe a Figura 11.

Figura 11 – Entendendo o IoU



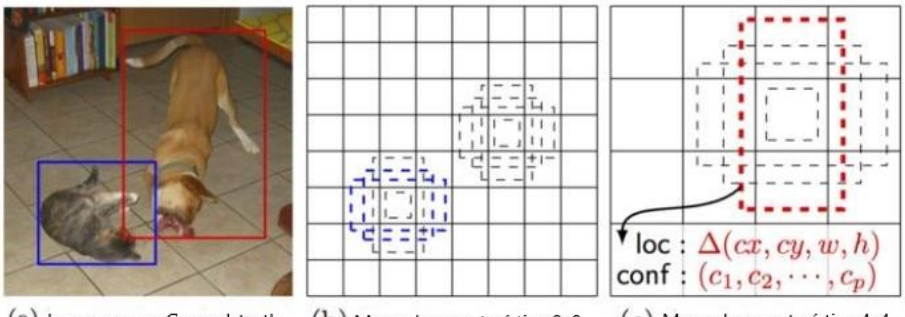
Fonte: Adaptado de https://image.slidesharecdn.com/ml-180824083037/95/deep-learning-and-object-detection-6-638.jpg?cb =1535099507

Assim, quando a maioria das caixas delimitadoras apresentam baixo IoU, o SSD interpreta como exemplos negativos de treinamento. De fato, pode-se acabar com uma quantidade desproporcional de exemplos negativos no conjunto de treinamento. Logo, em vez de usar todas as previsões negativas, é recomendável manter uma proporção de exemplos negativos para positivos de aproximadamente, 3 para 1. A razão pela qual é necessário manter amostras negativas é que o modelo precisa ser informado explicitamente sobre o que constitui uma detecção incorreta e assim poder apreender e saber identificá-la.

A seguir apresenta-se um exemplo que mostra uma ideia da proposta para detecção do SSD, envolvendo alguns dos conceitos vistos anteriormente, para melhor compreensão. No item (a) da Figura 12 tem-se a imagem de entrada e as caixas *Ground-truth* para cada objeto durante o treinamento. De uma maneira convolucional, avalia-se um pequeno conjunto de caixas padrão (neste exemplo são 4, mas podem ser 6) de diferentes proporções, em cada localização, em vários mapas de recursos, com escalas diferentes, por exemplo, 8×8 em (b) e 4×4 em (c).

Para cada caixa padrão, se preveem os offsets da forma e as confianças para todas as categorias de objetos, (c_1, c_2, · · ·, c_p). No treinamento, primeiro combinam-se essas caixas padrão com as caixas *Ground-truth*. Por exemplo, combinam-se duas caixas padrão com o gato e uma com o cachorro, que são tratadas como positivas e o resto são tratadas como negativas. Ainda no exemplo, o cão corresponde a uma caixa padrão (a que tem cor vermelha) na camada de mapa de recursos 4×4, mas não corresponde a nenhuma caixa padrão no mapa de recursos 8×8 (de resolução mais alta). Por outro lado, o gato é detectado apenas pela camada do mapa de recursos 8×8 em 2 caixas padrão (as de cor azul). Outra coisa interessante que pode ser visualizada na Figura 12, é que no mapa de recursos temos um *grid* com células como tínhamos no YOLO, mas no SSD não é usado o termo célula e sim, localização. Além disso, o *grid* é criado sobre o mapa de recursos e não sobre a imagem de entrada.

Figura 12 – Caixas padrão no SSD nos mapas de recursos 8x8 e 4x4

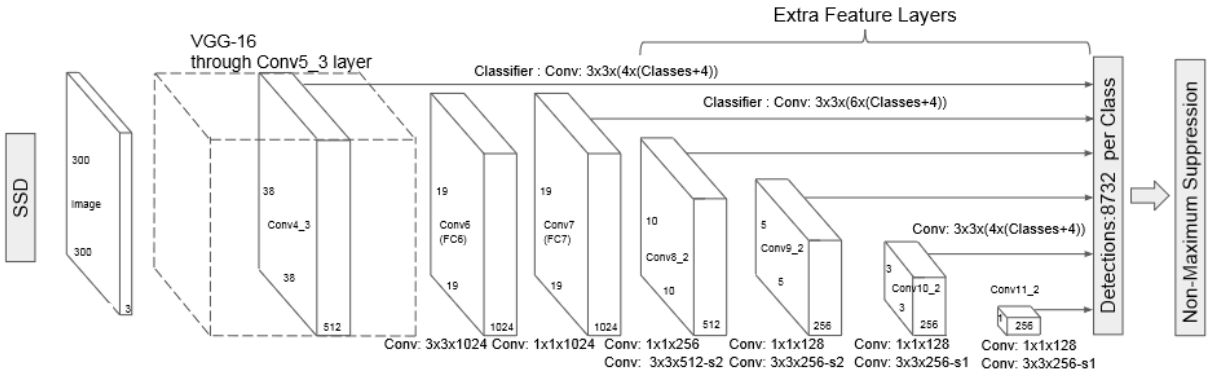


Fonte: https://arxiv.org/pdf/1512.02325.pdf

Durante a previsão, o modelo gera scores para cada categoria de objeto presente em cada caixa padrão. Além disso, produz ajustes para a caixa padrão que melhor se adequar ao formato do objeto. Para isso, a rede combina predições de vários mapas de recursos com diferentes resoluções, o que ajuda a lidar com objetos de diversos tamanhos. É importante dizer que os mapas de recursos em várias resoluções melhoram significativamente a precisão.

Em relação à arquitetura do SSD, observe a Figura 13, ela pode ter como base a rede VGG-16, mas podem ser usadas outras redes e elas também produziriam bons resultados segundo os autores em [7]. Os motivos principais pelos quais a VGG-16 foi utilizada como rede base é seu desempenho robusto em tarefas de classificação de imagem de alta qualidade e ela é conveniente quando há transferência de aprendizado, pois ajuda a melhorar os resultados. As camadas VGG totalmente conectadas originais foram trocadas por um conjunto de camadas de convolução auxiliares, que permitem extrair recursos em várias escalas e diminuir progressivamente o tamanho da entrada para cada camada subsequente [7]. Essas camadas adicionadas são seis, das quais cinco são para detecção de objetos sendo que em três dessas camadas se faz seis previsões no lugar de quatro.

Figura 13 – Arquitetura do SSD para uma entrada 300×300×3



Fonte: https://arxiv.org/abs/1512.02325

O SSD basicamente utiliza a VGG-16 para extrair mapas de recursos e, depois faz detecção de objetos, iniciando pela camada Conv4_3. Além disso, a Conv4_3 detecta objetos na menor escala (0,2 ou em alguns casos 0,1) e essa escala, vai aumentando linearmente para a camada mais a direita até chegar em 0,9. Logo, pode-se dizer que s_{min}=0,2 e s_{max}=0,9. Assim, é possível calcular as escalas das caixas padrão usando a equação 5.

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1), \quad k \in [1, m] \tag{5}$$

Na equação 5, m é o número de mapas de recursos. Agora, para cada escala, s_k temos cinco proporções não quadradas dadas pela equação 6.

$$a_r \in \left\{1, 2, 3, \frac{1}{2}, \frac{1}{3}\right\} \quad \left(w_k^a = s_k \sqrt{a_r}\right) \quad \left(h_k^a = {s_k}/{\sqrt{a_r}}\right) \tag{6}$$

Onde os valores referentes aos w's representam largura, e os h's as alturas. Esses valores de largura e altura nos permitem determinar cinco caixas delimitadoras não quadradas. Além disso, para a proporção 1:1, a escala é dada pela equação 7.

$$s_k^{'} = \sqrt{s_k s_{k+1}} \tag{7}$$

Neste caso, temos uma caixa delimitadora quadrada. Como visto, podemos obter no total, seis caixas delimitadoras com diferentes proporções, mas também podemos ter camadas com quatro caixas delimitadoras [11], basta eliminar os valores 3 e 1/3.

Para conhecer o número de previsões por localização, utilizaremos um exemplo. Suponha que a Conv4_3 é de tamanho 38×38 e sabendo que para cada localização ela prevê 4 objetos, ou seja, há quatro caixas delimitadoras e que cada previsão é composta por uma caixa delimitadora e 21 scores para 21 classes (supondo 20 classes, e ainda se considera a classe que tem 0 objetos, ou seja a classe 0). Então, escolhe-se o maior score como a classe do objeto delimitado e assim, tem-se 38×38×4 previsões por localização, independentemente da profundidade do mapa de recursos. Na linguagem utilizada pelo SSD entende-se que fazer várias previsões contendo caixas delimitadoras e índices de confiança é a mesma coisa que ter uma caixa múltipla como mencionado em [10] observe a Figura 14.

Figura 14 – SSD fazendo previsões

Fonte: Adaptado de https://medium.com/@jonathan_hui/ssd-object-detection-single-shot-multibox-detector-for-real-time-pro cessing-9bd8deac0e06

Para o SSD fazer uma previsão, diferentes camadas de mapas de recursos também estão passando por filtros (convoluções 3×3) em cada localização. Cada filtro produz 25 canais dos quais 21 são os scores para cada classe e os outros quatro definem a caixa delimitadora.

A função perda do SSD consiste em dois termos L_{conf} e L_{loc} (perda de confiança e perda de localização). A perda de localização mede a "distância" entre as caixas delimitadoras previstas e as caixas *Ground-truth* do conjunto de treinamento. O SSD só penaliza previsões de correspondências positivas. Aqui as correspondências negativas podem ser ignoradas.

A perda de confiança é a perda em fazer uma previsão de classe (mede o grau de confiança quanto à objetividade da caixa delimitadora calculada). Para cada previsão de correspondência positiva, penaliza-se a perda de acordo com a pontuação de confiança da classe correspondente. Para previsões de correspondência negativa, penaliza-se a perda de acordo com a pontuação de confiança da classe 0.

Na equação 8 temos a perda dada pela MultiBox, que mede o quão longe a previsão ficou com relação à realidade. O valor do parâmetro α ajuda a equilibrar a contribuição da perda de localização.

$$\text{Multibox\_loss} = \text{perda de confiança} + (\alpha * \text{perda de localização}) \quad (8)$$

SSD também usa o filtro NMS para remover previsões. A partir do maior score de confiança, o modelo avalia se alguma caixa delimitadora previamente prevista tem IoU maior que 0.45 com relação a previsão atual para a mesma classe. E se encontrada, a previsão atual é ignorada [11]. E, quando há muitas caixas delimitadoras semelhantes, tanto em forma, dimensão e objeto, é mantida só a que possui uma maior acurácia. Além disso, como muitas previsões não contêm objetos, o SSD elimina qualquer previsão com pontuação de confiança de classe menor que 0,01.

A diferença do SSD para as redes neurais convolucionais é que nelas ao classificar temos a mínima confiança de existir algum objeto na região escolhida e no SSD as caixas delimitadoras são encontradas ao realizar classificações em todas as posições da imagem, utilizando diferentes formas e escalas.

Por último, é importante mencionar que para a execução deste projeto será utilizado o modelo SSD MobileNet v2, ou seja, a rede base no lugar de ser a VGG-16 será a MobileNet versão 2. Isto é, MobileNet v2 compõe a base para extração de características. E, essa escolha foi feita porque o modelo apresentado dessa forma pode ser suportado pelo Google Colaboratory. Além disso, em [26] se mostrou que a MobileNet se equipara ao VGG-16 com 1/30 de custo computacional e tamanho do modelo.

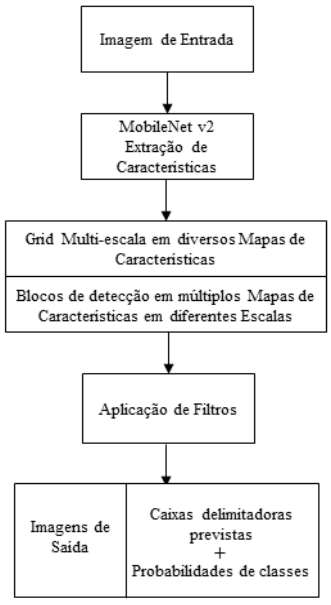
MobileNet v2 é uma rede neural totalmente convolucional e sua estrutura é baseada em convoluções separáveis em profundidade, camadas gargalo e blocos residuais invertidos [54].

Uma convolução separável em profundidade trabalha em dois estágios: primeiro há uma camada de convolução em profundidade que filtra a entrada, seguida por uma camada de convolução (pontual) 1×1 que combina esses valores filtrados para criar novos recursos. Essa camada de convolução pontual diminui o número de canais e por isso é chamada de camada de projeção ou de gargalo, ou seja, ela reduz o número de dados que flui pela rede. Por outro lado, antes disso, para poder aplicar os filtros sobre esses dados e extrair a maior quantidade de informações, precisamos "descompactar" esses dados com a camada de expansão.

Em resumo, a camada de expansão restaura os dados em sua forma completa, depois a camada de profundidade realiza a filtragem de informações importantes e finalmente, a camada de projeção "compacta" os dados para torná-los pequenos novamente. O que faz tudo funcionar é que as expansões e projeções são realizadas utilizando camadas convolucionais com *learnable parameters*, ou seja, parâmetros que têm a capacidade de apreender, e assim o modelo é capaz de conhecer a melhor "compactação" e "descompactação" dos dados em cada estágio.

Por último, apresentamos um diagrama que nos permite ver resumidamente o funcionamento do modelo SSD, observe a Figura 15.

Figura 15 – Diagrama do funcionamento do modelo SSD



Fonte: O Autor.

2.6 AUMENTO DE DADOS

O *data augmentation* é uma técnica que consiste, como seu nome indica, criar artificialmente dados a partir dos existentes e assim aumentar o conjunto de dados. Esses dados podem ser imagens, mas é recomendável realizar esse aumento só para o conjunto de treinamento.

Um dataset com uma quantidade de dados insuficiente torna-se um problema, pois classes abaixo da amostra sofrem baixa precisão específica da classe, neste sentido, o *data augmentation* permite obter o número de exemplos necessários para cada uma das classes. Outro objetivo da técnica é se prevenir ainda mais do problema de um possível *overfitting*, tornando o modelo mais generalizado e robusto. Outro objetivo não menos importante, é que para problemas de detecção é interessante ter um número maior de dados e as vezes é impossível ter certo número de amostras e neste sentido, o *data augmentation* torna-se uma ferramenta bem útil.

O aumento de dados pode ser realizado automaticamente ou manualmente por assim dizer. No primeiro caso, teríamos que aplicar GANs (*Generative Adversarial Networks*). Elas são redes neurais que ao serem treinadas criam imagens que tentam imitar as originais, e para isto realizam uma série de operações sobre as imagens de entrada. É importante salientar que apesar de que as GANs têm mostrado resultados bastante satisfatórios, elas não são comumente utilizadas, pois elas apresentam alguns desafios, por exemplo, dificuldade para obter imagens com alta resolução. instabilidade no treinamento e não convergência, e a necessidade de grandes conjuntos de dados para treinamento. Porém, elas proporcionam aumento de dados bem mais sofisticados e complexos se comparados aos realizados pelas bibliotecas comumente conhecidas para este objetivo.

Uma segunda forma, bem mais utilizada, é realizar o aumento de dados digamos que manualmente, isto é, usando as diversas bibliotecas disponíveis. Esta segunda abordagem é um pouco trabalhosa, mas em compensação permite ter um controle sobre o tipo de aumento que está sendo realizado sobre cada uma das imagens. Neste último caso, podem ser utilizadas as diversas bibliotecas que nos ajudam com o aumento de dados, por exemplo, CLoDSA disponível em [50] ou Keras usando a classe ImageDataGenerator disponível em [53]. Vale salientar que também existe a plataforma na Internet chamada Roboflow, onde é possível realizar aumento de dados só executando algumas configurações, mas o serviço não é gratuito e nem todas as técnicas de aumento estão disponíveis.

Essas bibliotecas dispõem por sua vez de diversas operações para aumento de dados, por exemplo, translações horizontais e verticais, rotações randômicas, vários tipos de filtros, aumento e redução de brilho, aumento e redução de resolução, entre outras. Uma consideração importante que deve ser levada em conta no momento de realizar o aumento de dados, é que as operações aplicadas devem condizer com as posições dos objetos nas imagens de acordo à aplicação. Por exemplo, se queremos detectar números de placas de veículos, não faz sentido realizar uma operação de rotação de 180º para a uma imagem que contém o objeto "carro". Porém, se quisermos estudar imagens de acidentes de trânsito, com certeza esse tipo de operação faria sentido.

2.7 TRABALHOS RELACIONADOS

Existem inúmeras aplicações nas quais YOLOv3 e SSD são de relevante ajuda e grande importância para o desenvolvimento de diversos trabalhos. A seguir mencionamos alguns exemplos.

Yang *et al* em [42] utiliza YOLOv3 para detectar e monitorar componentes cruciais do sistema ferroviário. Esses componentes são aplicados em trilhos operacionais e com a detecção é possível evitar falhas, auxiliando na manutenção preventiva. Anhu *at el* em [43] propõe um método de detecção de veículos com diferentes tipos de interseções de tráfego em tempo real para o fluxo de tráfego. Eles também usam o modelo YOLOv3 como base do estudo. Alguns estudos também apresentam o modelo YOLOv3 como ferramenta base para a melhoria do desempenho de robôs, conforme apresentado por Li *et al* em [44], um método de detecção de resíduos baseado no YOLOv3, que permite a detecção de objetos em tempo real e de alta precisão em ambientes aquáticos dinâmicos.

Por outro lado, Kelly *et al* em [45] usam o modelo SSD para detectar janelas ou intervalos de um processo contínuo operando em um estado de estabilidade, esta aplicação é útil especialmente quando modelos de estado estacionário estão sendo utilizados para otimizar o processo ou planta em tempo real. Li *et al* em [46] visam alcançar a detecção precisa e em tempo real de defeitos de superfície usando o modelo SSD e como base de rede a estrutura MobileNet. Um método de detecção de defeitos de superfície foi proposto com base no SSD MobileNet.

Vários estudos comparando os diferentes modelos de detecção de objetos em imagens são encontrados na literatura, mas não comparando as arquiteturas YOLOv3 e SSD MobileNet v2.

Uma análise comparativa bem completa dos diferentes modelos pode ser encontrada em [47]. Os autores resumem os resultados de artigos individuais para que possam ser vistos juntos. O objetivo é apresentar vários pontos de vista em um contexto, com a ideia de melhor compreender o cenário de atuação de cada modelo. Além disso, os autores ressaltam que os modelos muitas vezes são implementados nos mais diversos ambientes e uma comparação direta entre eles não pode ser feita.

Bussyreddy em [48] também analisa e compara várias arquiteturas derivadas dos modelos YOLO e SSD, mas o desempenho do modelo SSD MobileNet v2 não é apresentado.

Kurdthongmee em [49] analisa e compara o desempenho dos modelos YOLOv3 e SSD MobileNet v1. Os resultados apresentados pelo autor mostram que o modelo SSD MobileNet v1 supera o YOLOv3 em termos de taxa de detecção e erro médio de localização. No entanto, YOLOv3 fornece melhores resultados quando são considerados: desvio padrão e erros de localização máximo e mínimo.

Rios A.C. *et al* em [52] analisam e comparam o desempenho dos modelos YOLOv3 e SSD MobileNet v2. Os resultados apresentados pelos autores mostram que o modelo YOLOv3 apresentou um desempenho melhor para um *dataset* personalizado usado em robótica.

A principal diferença do artigo que foi publicado em [52] para este trabalho é que o *dataset* utilizado no artigo não era equilibrado, e o conjunto de dados era exatamente igual ao criado pelo autor em [12]. Além disso, no artigo não se realizou aumento de dados para melhorar os desempenhos dos modelos. Por último, com relação ao artigo mais um pequeno diferencial foi implementado no presente projeto, e é que aqui se seguiu fielmente a rotulação realizada pelo autor em [12], utilizando a plataforma Roboflow.

Este trabalho é importante porque até o momento no qual foi enviado o artigo para avaliação, não tinha sido publicado nenhum outro artigo que comparasse os dois modelos objetos de estudo em um dataset multi-rótulos personalizado e nem em conjuntos de dados conhecidos. Vale salientar que esse artigo foi utilizado como base para a realização deste projeto.

3. MATERIAS E MÉTODOS

Neste capítulo são apresentados os recursos computacionais necessários para a execução do presente projeto e a metodologia utilizada no desenvolvimento do mesmo.

3.1 RECURSOS PARA YOLOv3

Os principais recursos de programação utilizados para o desenvolvimento do projeto quanto ao modelo YOLOv3 são apresentados a continuação.

a) Google Colaboratory

O Google Colaboratory também é conhecido como Google Colab ou simplesmente Colab, é basicamente um serviço de nuvem gratuito hospedado pelo Google para incentivar pesquisas de Aprendizado de Máquina e Inteligência Artificial. No Colab é possível importar conjuntos de imagens, treinar um classificador e avaliar o modelo, entre outras coisas. O Colab na verdade é uma implementação do Jupyter Notebook, se executam células com códigos utilizando os servidores em nuvem do Google. Isto é, utiliza-se o hardware que disponibiliza o Google, incluindo GPUs e TPUs, independentemente do poder da máquina que esteja sendo utilizada. Na verdade, tudo que é necessário para poder executar o Colab seria um navegador [13].

b) CUDA

CUDA é uma plataforma de computação paralela e modelo de programação desenvolvido pela NVIDIA para computação em unidades de processamento gráfico, GPUs. Usando a CUDA, é possível acelerar a velocidade de processamento, aproveitando o poder da GPU. Em aplicativos que são acelerados por GPU, a parte sequencial da carga de trabalho é executada na CPU (que é otimizada para desempenho de encadeamento único) enquanto a parte de computação intensiva do aplicativo é executada em milhares de núcleos de GPU em paralelo [14]. Neste projeto foi utilizada a CUDA 11.1 (v11.1.105).

c) cuDNN

A biblioteca de rede neural profunda NVIDIA CUDA® [cuDNN] é uma biblioteca para redes neurais profundas acelerada por GPU. O cuDNN fornece implementações altamente ajustadas para rotinas padrão. Ela também é usada para aceleração de GPU de alto desempenho. Isso permite que desenvolvedores não gastem esforços com o ajuste de desempenho da GPU de baixo nível. O cuDNN acelera estruturas como Caffe, Keras, MATLAB, TensorFlow, PyTorch entre outras [15]. Neste projeto foi utilizada a versão cuDNN v8.1.1 para CUDA v11.1.

d) Darknet

Darknet é uma rede neural de código aberto escrita em linguagem C e CUDA. Joseph Readmon é o seu criador e está disponível em [19]. O YOLO é sua subdivisão otimizada para detecção de objetos em tempo real. A Darknet tem suporte para uso de CPUs e GPUs da Nvidia. O YOLO que é um dos métodos utilizados neste projeto é uma das ramificações da Darknet, mas para podermos fazer uso do YOLO devemos ter instalada a Darknet.

e) OpenCV

A OpenCV é uma biblioteca de visão computacional de código aberto e multiplataforma, escrita nas linguagens C e C ++. A sua documentação encontrasse disponível em [18]. A OpenCV foi criada com o objetivo de ter eficiência computacional em aplicações que são executadas em tempo real. Ela fornece uma infraestrutura simples de usar que ajuda a construir aplicações razoavelmente sofisticadas de maneira rápida. A OpenCV pode executar o reconhecimento de padrões estatísticos e agrupamento e é altamente útil para diversas tarefas no Aprendizado de máquina.

f) LabelImg

LabelImg é uma ferramenta de código aberto escrita em Python, que serve para rotular imagens graficamente, ou seja, podemos criar caixas delimitadoras e etiquetas nos objetos da

imagem. LabelImg suporta etiquetagem no formato PascalVOC e YOLO. As anotações são salvas como arquivos XML e TXT, respectivamente.

Também é possível etiquetar um maior número de imagens se comparado a outros aplicativos gratuitos.

g) Anaconda

Anaconda é uma distribuição gratuita e de código aberto das linguagens de programação Python e R para computação científica, que visa simplificar o gerenciamento e implantação de pacotes. A distribuição inclui pacotes de ciência de dados adequados para Windows, Linux e macOS. É desenvolvido e mantido pela Anaconda, Inc [28].

3.2 RECURSOS PARA SSD

Para o desenvolvimento do modelo SSD para detecção de objetos em imagens, foi necessário utilizar novamente alguns dos recursos mencionados anteriormente para YOLOv3. Eles são: Google Colaboratory, NumPy, Matplotlib, OpenCV, LabelImg e Anaconda. A continuação, são apresentados outros recursos computacionais importantes que foram utilizados para o modelo SSD MobileNet v2, e que ainda não foram apresentados.

a) TensorFlow

TensorFlow é uma biblioteca de código aberto compatível com Python para computação numérica que torna o aprendizado de máquina mais rápido e fácil. *TensorFlow* é a segunda geração de um sistema (chamado *DistBelief*) projetado pelo Google Brain (segmento do Google voltado para pesquisa na área de inteligência artificial). *Tensorflow* pode também ser executado em múltiplas CPUs e GPUs. Além disso, está disponível para diversos sistemas operacionais e plataformas de computação móveis, incluindo Android e iOS [29].

b) TF-Slim

TF-Slim é uma biblioteca leve para definir, treinar e avaliar modelos complexos no *TensorFlow*. Os componentes do TF-Slim podem ser misturados livremente com o *TensorFlow* nativo, bem como com outros frameworks [30]. Slim torna mais fácil estender modelos de visão computacional e ajuda bastante na inicialização de algoritmos de treinamento, usando partes de pontos de verificação de modelos pré-existentes.

3.3 FERRAMENTAS PARA AUMENTO DE DADOS

Neste trabalho realizou-se aumento de dados utilizando algumas ferramentas. Principalmente considerando a "CLoDSA", que se encontra disponível em [50]. CLoDSA é uma biblioteca de aumento de imagem de código aberto e tem a vantagem de apresentar imagens de boa qualidade, após operações de aumento. Ela suporta uma ampla variedade de técnicas de aumento e permite ao usuário combiná-las facilmente. Além disso, também pode ser aplicada em listas de imagens, por exemplo, vídeos.

As técnicas que foram aplicadas utilizando CLoDSA são listadas a continuação:

- *Average blurring*: Suaviza a imagem usando um filtro médio.
- *Change to HSV*: Altera o espaço de cores de RGB para HSV.
- *Change to LAB*: Altera o espaço de cores de RGB para tipo laboratório.
- *Dropout*: Define alguns pixels da imagem para ser zero.
- *Gaussian blurring*: Desfoca a imagem usando um filtro gaussiano.
- *Invert*: Inverte todos os valores na imagem, isto é, um pixel de valor "v" passa a valer "250-v".
- *Raise saturation*: Aumenta a saturação.
- *Resize*: Redimensiona a imagem.
- *Rotate*: Gira a imagem em 45 ou 30 graus.
- *Salt and Pepper*: Adiciona ruído de sal e pimenta à imagem.
- *Shearing*: Corta a imagem.
- *Translation*: Translada a imagem horizontalmente.

Além disso, utilizaram-se as técnicas: reflexão sobre o eixo "y", translação vertical e brilho randômico. Elas não estão disponíveis na biblioteca CLoDSA, mas podem ser encontradas em [53], utilizando a classe ImagemDataGenerator da biblioteca Keras.

- *brightness_range*: Aumenta ou diminui o brilho.

- *horizontal_flip*: Realiza reflexões sobre o eixo y.
- *height_shift_range:* Realiza translações verticais.

A seguir apresentamos alguns exemplos de imagens obtidas mediante aumento de dados. Na Figura 16 mostramos a imagem original. Na Figura 17 essa mesma imagem, mas depois de ter sido aplicada a operação denominada '*Inverter*' e na Figura 18 novamente a mesma imagem, mas depois de termos aplicado a operação denominada '*Dropout*'.

Figura 16 – Imagem do *dataset* original

Fonte: Douglas H. dos Reis, disponível em [12].

Figura 17 – Imagem após operação de aumento '*Inverter'*

Fonte: O Autor.

Figura 18 – Imagem após operação de aumento ‘*Dropout*’

Fonte: O Autor.

3.4 DATASET

Neste projeto foi tomado como base o *dataset* criado por Douglas H. dos Reis em [12], para navegação de robôs em ambientes interiores. Esse conjunto de dados foi projetado para ser utilizado com o modelo de detecção de imagens, YOLOv2. Ele contém 1120 imagens de tamanhos iguais 800×600 pixels, com suas respectivas anotações contidas em arquivos do tipo TXT em um formato tal que YOLO consegue fazer a leitura das informações (quais os objetos e onde se encontram dentro da imagem), no total 1120 arquivos TXT.

Esse conjunto de dados não está distribuído de maneira proporcional. Além disso, eles se encontram agrupados em quatro pastas de acordo ao tipo de objeto etiquetado (Caixa, Mesa, Cadeira com rodas, Cadeira Comum), por exemplo, na primeira pasta temos todas as imagens nas quais foram etiquetadas as “Caixas” e seus respectivos arquivos TXT.

Outra característica importante do *dataset* é com relação aos fundos das imagens. Eles são de dois tipos, fundo uniforme e fundo estruturado. Isto para melhor treinar o conjunto de dados.

A seguir apresentamos alguns exemplos de imagens do *dataset* original.

Figura 19 – Imagem do *dataset* original contendo o objeto ‘Mesa’

Fonte: Douglas H. dos Reis, disponível em [12].

Figura 20 – Imagem do *dataset* original contendo o objeto ‘Cadeira com rodas’

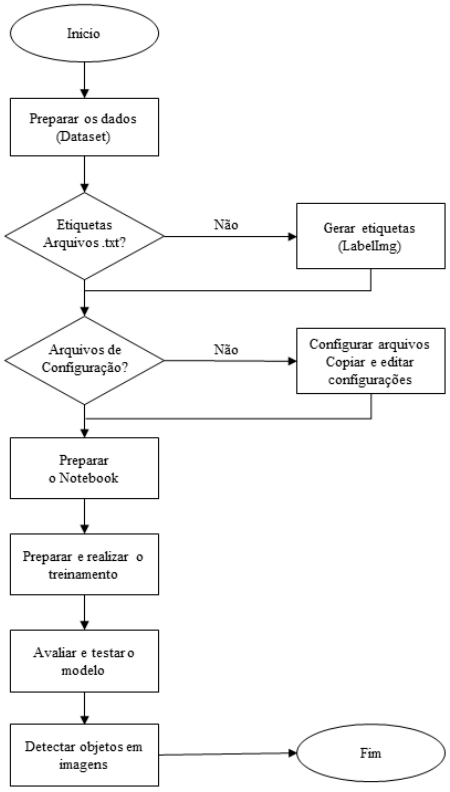
Fonte: Douglas H. dos Reis, disponível em [12].

3.5 METODOLOGIA PARA YOLOv3

Nesta sessão apresenta-se o desenvolvimento da metodologia utilizada para a preparação de dados, treinamento e avaliação do modelo YOLOv3. Com o objetivo de resumir e entender melhor a metodologia usada para o modelo YOLOv3 para detecção de

objetos em imagens, foi construído um fluxograma que mostra os passos mais importantes, observe a Figura 21.

Figura 21 – Fluxograma metodologia YOLOv3



Fonte: O Autor.

3.5.1 Preparação dos dados

Antes de realizar o treinamento de um modelo de detecção de objetos em imagens, é necessário preparar um conjunto de dados. Esses dados são importantes porque nos ajudam a fazer um bom treinamento dos modelos e assim obter melhores resultados quanto ao desempenho. Para cada um dos modelos, o *dataset* deve ser preparado adequadamente de tal forma que cada um deles possa compreender as informações, para tanto devemos ter imagens condizentes com os objetos que desejamos detectar e etiquetas adequadas.

Para a execução da primeira parte deste trabalho, foram tomadas 963 imagens do *dataset* base, apresentado na seção 3.4. Algumas imagens apresentam um único objeto enquanto outras apresentam múltiplos objetos, isso para que o modelo apreenda a realizar a detecção dos objetos em diversos contextos.

O motivo de escolher esse *dataset* para *benchmark* é que apesar de que existem diversos conjuntos de dados famosos, eles geralmente contêm um número considerável de classes e assim o trabalho tornar-se-ia bastante árduo quanto ao número de imagens por classe para realizar o aumento de dados e as suas etiquetagens. Além disso, seria necessário maior poder computacional para a realização dos treinamentos.

O conjunto de dados completou-se de maneira a obter 1000 imagens, visando ter um *dataset* equilibrado, 250 imagens para cada classe. Para as classes ‘Mesa’ e ‘Cadeira com rodas’ se equilibraram os conjuntos de imagens de teste por *resampling*, amostrando mais frequentemente essas duas classes subamostradas. Por outro lado, para as classes ‘Caixa’ e ‘Cadeira comum’ foram removidas exposições dessas duas classes sobreamostradas Isto foi realizado para não deixar em vantagem nenhuma das quatro classes. Além disso, também se consideraram os arquivos TXT dessas imagens.

Assim, para cada classe temos um total de 250 imagens das quais 200 (80%) são utilizadas para treinamento e 50 (20%) para teste e avaliação do modelo. Além disso, são utilizadas 10 imagens não conhecidas pelo modelo, para ver como ele se comporta quando está realizando as detecções.

Para execução da segunda parte deste trabalho com relação ao modelo YOLOv3, foram utilizadas 2000 imagens. Essas imagens correspondem as 1000 imagens descritas anteriormente e, as outras 1000 imagens correspondem a 800 imagens resultado do aumento de dados realizado sobre as imagens de treinamento e 200 novas imagens baixadas da internet que são adicionadas ao conjunto de imagens para teste e avaliação. Vale salientar que neste caso, foi necessário executar o processo de etiquetagem dessas 1000 imagens e para isto se utilizou a ferramenta LabelImg disponível em [32], mas também pode se utilizar a ferramenta Yolo_mark disponível em [20].

Ainda com relação à execução da segunda parte mencionada no parágrafo anterior, para cada classe tem-se um total de 500 imagens das quais 400 (80%) são utilizadas para treinamento e 100 (20%) para teste e avaliação. Além disso, serão utilizadas as mesmas 10 imagens para realizar detecções para avaliar de maneira quantitativa o comportamento do modelo como realizado sem o aumento de dados.

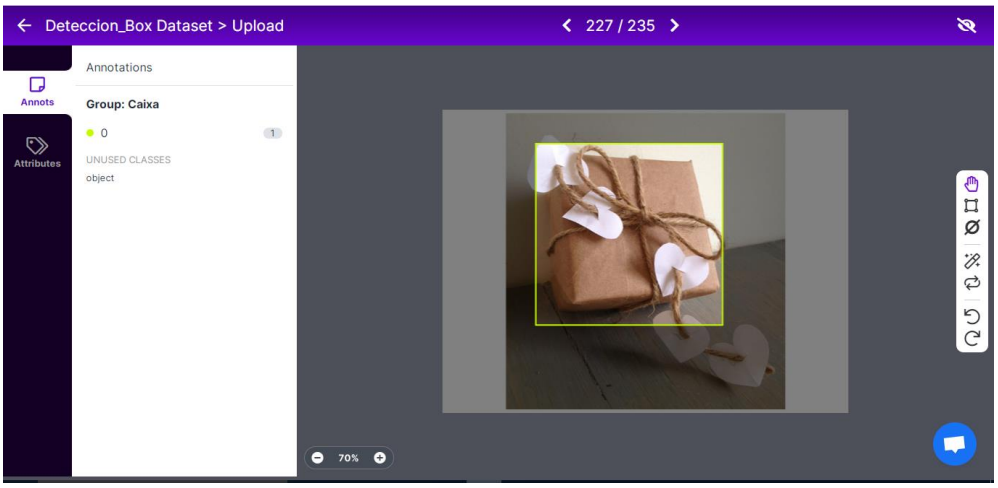
Vale salientar que na preparação dos dados, tomou-se cuidado com relação aos tamanhos das imagens de treinamento, deixando todas elas com o mesmo tamanho. Apesar de que o modelo YOLOv3 aceita imagens de entrada de tamanhos diferentes, existem alguns problemas que poderiam aparecer no momento do treinamento, por exemplo, ao trabalhar

com GPUs em paralelo é necessário concatenar lotes de imagens, que quando têm diferentes tamanhos seria impossível realizar essa operação.

3.5.2 Rotulação de imagens

Após aumento de dados, é necessário rotular ou etiquetar os objetos nas imagens obtidas. Para poder realizar adequadamente este processo, utilizou-se a ferramenta "Roboflow", disponível em [51]. Roboflow basicamente é uma plataforma online que permite desde a visualização das imagens (com as caixas delimitadoras dos objetos) até treinamentos de conjuntos de dados. Alguns recursos oferecidos são gratuitos e nesse caso basta fazer um cadastro. Na execução deste trabalho, foi necessário utilizar Roboflow para visualização das imagens etiquetadas. Na Figura 22 observa-se uma imagem do *dataset* original com o objeto e sua respectiva caixa delimitadora.

Figura 22 – Imagem mostrada por Roboflow



Fonte: O Autor.

A ideia era saber como tinha sido realizada a etiquetagem pelo autor do *dataset* base em [12]. Isto porque no conjunto de dados, havia imagens nas quais aparecia o mesmo objeto várias vezes (por exemplo, uma imagem com uma mesa e seis cadeiras), e nesses casos nem todos os objetos foram etiquetados por ele.

Em seguida, usando a ferramenta LabelImg [32] foram rotuladas as 800 imagens obtidas no aumento de dados, no formato YOLO. Além disso, também foram etiquetadas as 200 novas imagens de teste. Mais detalhes sobre o processo de rotulação de imagens são apresentados na metodologia para SSD MobileNet v2, pois esse modelo precisou de que todas

as imagens fossem etiquetadas. Um detalhe importante, é que a ferramenta Roboflow com assinatura paga tem a possibilidade de conversão das etiquetas do formato YOLO para PascalVOC, que é o utilizado pelo modelo SSD.

3.5.3 Configuração de arquivos

É necessário criar alguns arquivos que YOLOv3 utiliza para execução do treinamento e depois para avaliação e teste. Alguns também foram criados por Douglas H. dos Reis em [12], mas precisaram ser adaptados para YOLOv3.

O arquivo “obj.names” não precisa ser modificado pois, as classes são as mesmas. Nele estão contidos os nomes dos objetos a serem detectados podendo ter só um nome por linha, observe a Figura 23. Vale salientar que ao etiquetarmos todas as imagens utilizando labelImg, esse arquivo é gerado automaticamente.

Figura 23 – Estrutura do arquivo “obj.names”

```
1 Caixa
2 Mesa
3 Cadeira com rodas
4 Cadeira comum
```

Fonte: Douglas H. dos Reis, disponível em [12].

O arquivo "obj.data" contém o número de classes, o diretório das imagens com os rótulos para treinamento, o diretório para as imagens de teste e o diretório do arquivo de backup para continuar o treinamento, caso seja necessário retomá-lo a partir da iteração na qual ele parou. Na Figura 24 pode-se observar a estrutura desse arquivo.

Figura 24 – Estrutura do arquivo “obj.data”

```
classes = 4
train =/content/gdrive/MyDrive/YOLOV3/train.txt
valid =/content/gdrive/MyDrive/YOLOV3/test.txt
names =/content/gdrive/MyDrive/YOLOV3/obj.names
backup =/content/gdrive/MyDrive/YOLOV3/backup/
```

Fonte: O Autor.

Os arquivos “train.txt” e “test.txt” que foram criados em [12], tiveram que ser reformulados adequadamente com relação à quantidade de imagens de treinamento e de teste, e com relação aos diretórios de cada uma das imagens. O arquivo “train.txt” deve conter os diretórios das imagens de treinamento e “test.txt” os diretórios das imagens para teste.

Para realizar essa tarefa de “separar” o conjunto de imagens sendo que uma parte é para treinamento e o restante é para teste, podem ser usados códigos que fazem a seleção aleatória das imagens baseados na porcentagem especificada para cada conjunto.

Em [12] também foi utilizado um arquivo chamado “IntegradorTrainingVOC.cfg”, que é para configurar o modelo. Nesse arquivo encontramos a estrutura do YOLOv2 e nele devem ser modificados parâmetros, de acordo à aplicação. Como nosso objetivo é utilizar o modelo YOLOv3, é necessário usar o arquivo “yolov3.cfg” e realizar as alterações necessárias usando como base a documentação do YOLOv3.

- *batch*: Este parâmetro se utiliza para que no treinamento, o dataset de imagens seja agrupado em conjuntos (*batches*) e não uma imagem por vez, pois isto implicaria realizar muitas mudanças nos pesos da rede. Além disso, não pode se realizar todo ao mesmo tempo devido às limitações de memória. Então, para executar o treinamento, tomamos 64 imagens para cada etapa do treinamento. Por default esse parâmetro é 1.
- *subdivision*: Este parâmetro corresponde ao número de grupos no qual dividiremos os *batches* para acelerar o processo de treinamento e para melhorar a generalização. Neste caso também é necessário modificar este valor que por default também é 1. No nosso caso, será 32. Com isto, também diminuímos os requisitos da GPU.
- *max_batches*: Este parâmetro está diretamente relacionado com o número de iterações que serão feitas no treinamento. O número mínimo aconselhado é (2000*número de classes). Neste trabalho, escolheu-se 12000. Alguns desenvolvedores aconselham que esse número seja 50200 (ou seja, 10 vezes menor que o valor default) para uma maior exatidão, mas o processo de treinamento torna-se muito demorado.
- *steps*: Este parâmetro está diretamente relacionado ao *max_batches*. Na verdade, podem ser dados vários valores, mas se aconselham ter dois, um será 80% do *max_batches* e o outro será 90% do *max_batches*. Eles são separados por vírgulas dentro da mesma linha de código no arquivo .cfg.
- *classes*: Refere-se ao número de classes que queremos classificar, ou seja, 4. Esse valor deve ser trocado nas linhas 610, 696 e 783 do arquivo.
- *filters*: Este parâmetro será modificado somente nas linhas 603, 689 e 776 do arquivo. O número de filtros depende diretamente do número de classes e satisfaz a seguinte relação, *filters* = (*classes* + 5) * 3. Para nosso caso, 27.

Existem também duas formas de parar o treinamento, configurando como fizemos no arquivo “yolov3.cfg” ou também durante o processo de treinamento aparecem indicadores de

erro “AVG IOU” e se recomenda parar quando o valor deste não continue a diminuir, em várias iterações.

No caso do YOLO existem duas opções para executar o treinamento. YOLO pode ser treinada do zero ou podem ser utilizados pesos previamente treinados para as camadas convolucionais. Esses pesos são provenientes do modelo darknet53. É melhor usar a segunda opção pois, não custará demasiado para o modelo encontrar a configuração ótima dos pesos, reduzindo consideravelmente o tempo de treinamento. Para a execução deste trabalho optou-se pela segunda opção e para isso foi baixado o arquivo “darknet53.conv.74” de pesos pré-treinados no conjunto de dados ImageNet. Observe que darknet53.conv.74 são pesos YOLOv3 iniciais para dados personalizados de treinamento, enquanto yolov3.weights são pesos totalmente treinados que podem ser usados para detecção.

3.5.4 Configuração do Colab

O próximo passo a ser executado é a configuração do Google Colaboratory e a integração deste com o Google Drive.

Primeiro, é preciso ter uma conta Google. Depois, acessamos o Colab e configuramos o *Runtime* para poder trabalhar com a GPU.

Como queremos trabalhar diretamente nos arquivos do computador local, criamos uma pasta no Google Drive chamada “YOLOV3”. Esta pasta é onde teremos todos os arquivos necessários para a implementação do modelo YOLOv3. É interessante ter um bom desempenho ao baixar dados do Drive para a Colab. Logo, é necessário ter o mínimo de arquivos na pasta raiz do Drive, isto ajudará a agilizar tudo que for executado.

A seguir, realizamos a conexão do Colab ao Google Drive. No Colab importamos a biblioteca “drive” e montamos o Google Drive como um *drive* local da máquina virtual. Assim, podemos acessar os arquivos do Drive pelo Colab, acessando o caminho: “/content/gdrive/MyDrive/”.

Ao executarmos o código mencionado anteriormente, deve aparecer um link. Ele deve ser acessado para poder obter o código de autorização. Esse código é copiado e depois colado na caixa que solicita o mesmo. Em seguida, aparece a mensagem “`Mounted at /content/gdrive`” indicando que o processo foi realizado com sucesso.

3.5.5 Clonar e compilar Darknet

O objetivo é termos a rede neural que ajuda YOLOv3 no processo de detecção de objetos em imagens. E para isto, devemos clonar uma versão melhorada da Darknet, criada por AlexeyAB em [22]. O autor em [22] aplicou algumas modificações para possibilitar o carregamento de arquivos do Drive e assim poder reduzir o número de logs no console, para acelerar o notebook.

Depois de clonar e compilar a Darknet, pode-se copiar a versão compilada para o Google Drive. Assim, não será necessário compilá-la novamente em cada execução do Notebook, ajudando a melhorar o desempenho.

3.5.6 Treinamento e Avaliação

Para iniciar o treinamento do YOLOv3 é necessário criar algumas pastas e depois fazer o upload dessas pastas e dos arquivos configurados na seção 3.5.3.

Todas as pastas e arquivos são colocados no Google Drive dentro da pasta "YOLOV3".

O subconjunto de imagens tomado do *dataset* construído por Douglas H. dos Santos em [12], segue a metodologia aplicada pelo autor. Foram criadas quatro pastas, uma para cada classe. Para a primeira classe (que aparece na primeira linha do arquivo "obj.names") construímos uma pasta nomeada "001" na qual se encontram todas as imagens que contêm a etiqueta "Caixa". O mesmo processo é realizado para as outras classes. Todas as pastas serão colocadas dentro de uma chamada "img" que denota imagens. Vale salientar que cada uma dessas quatro pastas não só possui imagens, elas contêm também os respectivos arquivos TXT de cada imagem.

No Google Drive também são criadas outras pastas: "weights", "backup", e "check", que descrevemos a seguir. Baixamos o arquivo "darknet53.conv.74" de pesos pré-treinados, esse arquivo pode ser encontrado em [23] e na pasta "weights" será feito o upload desse arquivo. Na pasta backup são salvos os pesos que estão sendo obtidos durante o treinamento. Por último, a pasta "check" será utilizada quando seja preciso fazer uma verificação.

Finalmente, podemos treinar o nosso modelo YOLOv3, e para isso utilizamos o comando "detector train". Durante o treinamento, depois de uma certa quantidade de iterações serão salvos os arquivos ".weights" e eles ficarão na pasta "backup", mas para não perder

informações copiamos o arquivo que foi gerado pelo modelo, chamado “yolov3_last.weights” na pasta de nome “weights” e assim, se precisarmos retomar o treinamento não teremos informações sobrepostas.

Também é bem possível que o treinamento por alguma razão pare e tenha que ser retomado. Então, para começar o treinamento no ponto em que o último tempo de execução terminou, é necessário implementar o mesmo código utilizado para iniciar o treinamento, mas desta vez o arquivo de pesos será “yolov3_last.weights”.

Após conseguirmos os novos pesos resultado do treinamento, pode-se detectar objetos em imagens e para poder realizar as detecções é necessário criar mais uma pasta que chamamos “DetecImg”, que também se encontra na pasta “YOLOV3”.

Terminado o treinamento, podemos analisar os resultados e avaliar o modelo. Escolhe-se entre os diversos arquivos “.weights” qual deles apresenta melhores resultados (IoU e mAP mais altas). O último arquivo “.weights” não necessariamente entrega os melhores resultados porque pode se produzir um *overfitting* o que significa que o modelo consegue detectar os objetos nas imagens de treinamento, mas em outras imagens, não consegue detectar nada.

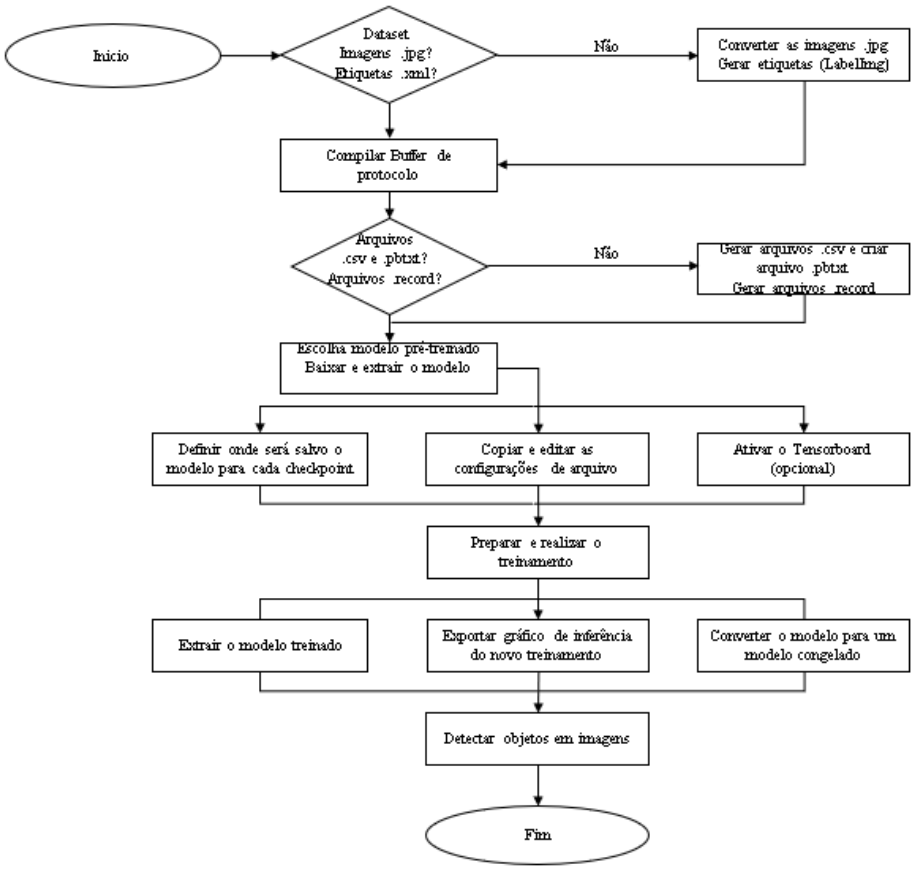
É importante dizer que durante o treinamento, ao não diminuir o valor do *avg loss* ou erro de perda média, deve-se parar o treinamento. Esse valor se encontra na faixa de 0.05 a 3 (dependendo da complexidade dos dados, ele aumenta). Além disso, se aparecer a mensagem –*nan*, significa que o treinamento está ocorrendo com erro e neste caso, deve-se recomeçar o treinamento, fazendo as devidas correções.

O modelo YOLOv3 pode ser avaliado de diversas formas. Uma maneira simples é apresentada pelos autores em [31], uma única linha de código que nos ajuda a determinar as principais métricas de avaliação. Além disso, para podermos avaliar o modelo devemos redefinir o parâmetro *batch* do arquivo ‘.cfg’. Esse parâmetro deve valer 1.

3.6 METODOLOGIA PARA SSD

Nesta seção apresenta-se um passo a passo do desenvolvimento da metodologia utilizada para a preparação de dados, treinamento e avaliação do modelo SSD MobileNet v2. Para entender melhor esse processo, foi construído um fluxograma, observe a Figura 25.

Figura 25 - Fluxograma metodologia SSD MobileNet v2



Fonte: O Autor.

3.6.1 Preparação dos dados

Na preparação dos dados para o modelo SSD MobileNet v2, novamente foi utilizado o *dataset* criado por Douglas H. dos Reis em [12], mas desta vez foi necessário etiquetar todas essas imagens além das imagens obtidas no aumento pois, o modelo SSD não comporta o mesmo tipo de anotações usadas por YOLOv3.

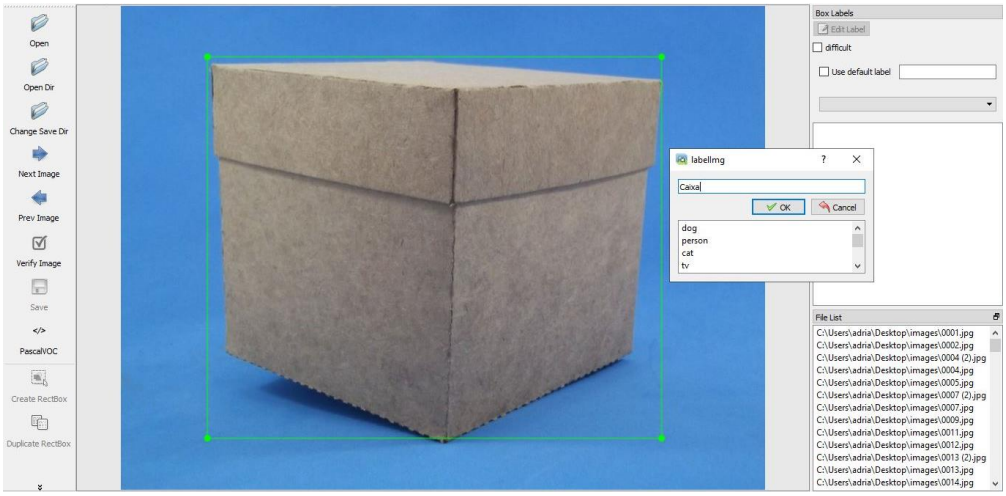
O processo de etiquetagem foi realizado utilizando LabelImg, ela é uma ferramenta simples de se usar.

O único cuidado é que todas as imagens no conjunto de dados devem ter a extensão ‘.JPG’ para não ocorrer erros no treinamento.

O primeiro passo foi colocar todas as imagens dentro uma pasta nomeada ‘imagens’. Depois, criamos outra pasta chamada ‘anotações’, ou seja, nela são salvos todos os arquivos XML que contêm os rótulos de cada uma das imagens.

No terminal do Anaconda3 para Windows 10, instalamos o LabelImg. Logo, executamos um processo análogo ao apresentado em [32]. Por último, com o comando ‘python LabelImg.py’ é aberta a ferramenta para começar a realizar as anotações. Na Figura 26 apresenta-se uma imagem que está sendo etiquetada no LabelImg.

Figura 26 – Etiquetando Imagens com o LabelImg



Fonte: O Autor.

No LabelImg selecionamos a pasta ‘Change Save Dir’ e escolhemos a pasta ‘anotações’. Em seguida, selecionamos ‘Abrir Dir’ e é aqui onde escolhemos a pasta ‘imagens’ que é onde se encontram todas as imagens que serão rotuladas.

Quando realizamos o processo de rotulagem, selecionamos a opção ‘PascalVOC’ que é o formato dos arquivos XML, mas também pode ser selecionada a rotulação YOLO. Quando é mostrada a primeira imagem basta clicar em criar ‘RecBox’, aqui podemos posicionar e arrastar o mouse para criar a caixa delimitadora, ou seja, um retângulo que deve contornar a imagem, em seguida, digitamos o nome da classe do objeto selecionado. Realizada a rotulação na imagem, seleciona-se ‘Next Image’ para assim rotular a próxima e, assim sucessivamente até a última. É recomendado usar os atalhos para acelerar este processo.

3.6.2 Configuração de arquivos

Tendo todos os arquivos de anotações, eles são divididos em duas pastas. Uma para treinamento que contém 80% das anotações, nomeada ‘train_labels’. E, outra com o 20% restante nomeada ‘test_label’ para teste e avaliação, como realizado no YOLOv3. Essa tarefa

de “separação” foi realizada manualmente para que os conjuntos de dados fossem exatamente iguais aos do modelo YOLOv3.

É importante dizer que outros arquivos serão gerados ao decorrer do desenvolvimento da metodologia. À medida que eles forem solicitados irão sendo apresentados os processos utilizados para obter os mesmos, mas resumidamente, os arquivos XML precisam ser convertidos ao formato CSV que por sua vez ajudará a criar os arquivos TFRecords que têm extensão ‘.records’. Esses últimos arquivos são necessários, pois é a maneira como o *TensorFlow* consegue fazer a leitura das informações que se encontram nos arquivos XML.

Além disso, será criado o arquivo ‘.pbtxt’ que é praticamente um equivalente do arquivo ‘obj.names’ do YOLOv3. Por último, também é necessário um arquivo ‘.config’ no qual serão especificados todos os parâmetros necessários para realizar o treinamento do modelo. Esse último arquivo é um equivalente do arquivo ‘.cfg’ do YOLOv3.

3.6.3 Configuração Colab

Analogamente, como realizado para YOLOv3, foi configurado o *Google Colaboratory* para que o acelerador de *hardware* seja uma GPU. Neste caso, usa-se a GPU disponível pelo Google Colab no momento do treinamento. Esclarecendo que entre mais vezes seja utilizada, haverá mais restrições de tempo para seu uso.

Desta vez criamos uma pasta nomeada ‘objeto_deteccion’ e novamente importamos a biblioteca “drive” e montamos o Google Drive para acessar os arquivos do Drive pelo Colab.

3.6.4 Treinamento e Avaliação

Antes de proceder com a metodologia utilizada para a realização do treinamento do modelo SSD MobileNet v2, é necessário atualizar a lista de pacotes e instalar as versões mais recentes no Notebook do Colab. Em seguida, são instaladas diversas bibliotecas necessárias para o treinamento do modelo e para a detecção dos objetos nas imagens, observe o Anexo E. Neste projeto utilizou-se o *TensorFlow* versão 1.15.0, isto devido a uma melhor performance ao trabalhar com o modelo SSD MobileNet v2 no Google Colab.

No final do Anexo E, também é baixado o repositório do *TensorFlow models* na pasta ‘objeto_deteccion’. Nele há diversos projetos inclusive o de detecção de objetos em imagens que vamos utilizar para treinar nosso modelo. Isto é realizado para não ter que treinar o

modelo desde ‘zero’ porque levaria muito tempo. Uma observação importante é que foram utilizados pesos pré-treinados no famoso *dataset MS* COCO.

O seguinte passo, apresentado no Anexo F, é realizar a conversão de arquivos ‘.xml’ para arquivos ‘.csv’ e a criação do arquivo ‘.pbtxt’ que contém o mapeamento dos rótulos das classes. Esse arquivo dirá ao modelo o que é cada objeto, o nome de cada uma das quatro classes terá um número de identificação. Para esta parte da metodologia é necessário realizar uma fusão dos códigos descritos em [33] e [34]. A referência [33] é utilizada para criar o arquivo ‘.pbtxt’ e a [34] para a conversão dos arquivos mencionada no parágrafo anterior. Além disso, é preciso definir o diretório onde o código pode ser executado e realizar as mudanças necessárias adequando para nosso projeto.

No anexo G compilamos os *buffers* de protocolo que são um mecanismo extensível de linguagem neutra e plataforma extensível do Google para serializar dados estruturados - pense em XML, mas menor, mais rápido e mais simples. Definimos como se deseja a estrutura dos dados e depois é gerado um código-fonte que pode ser usado para escrever e ler facilmente esses dados estruturados para uma variedade de linguagens. Além disso, na última linha do Anexo G se executa um teste rápido para confirmar se o construtor de modelo está funcionando corretamente, quando a célula do notebook é executada, ela só deve apresentar o caminho do diretório onde estamos compilando os *buffers*.

O seguinte passo é apresentado no Anexo H, resumidamente é onde será realizada a conversão dos arquivos do tipo ‘.csv’ para arquivos do tipo ‘.record’. Isto porque o *TensorFlow* aceita dados como TFRecords, pois ele é otimizado para trabalhar com eles. Pode-se dizer que os TFRecords são arquivos binários que são executados rapidamente com baixo uso de memória, isso porque não seria possível carregar um *dataset* muito grande na memória. Assim, temos dois arquivos TFRecords, um para teste e outro para treinamento. Para executar esta etapa se tomou como referência os trabalhos disponíveis em [34] e [35].

Nesta parte da metodologia podem surgir alguns erros. Nesse caso, é importante verificar se todos os arquivos XML e CSV foram gerados adequadamente, por exemplo, nenhuma das dimensões das imagens podem apresentar valores zero. Caso aconteça, as imagens devem ser eliminadas.

Para o próximo passo do desenvolvimento da metodologia, é apresentado o ANEXO I, nele é realizada a escolha do modelo pré-treinado. Tomou-se como referência [36], mas nesse trabalho o autor apresenta três possíveis modelos que poderiam ser utilizados. Para nosso caso, só é relevante o SSD MobileNet v2. Na mesma escolha do modelo pode-se definir o tamanho

do *batch* que deixaremos como sendo 16, não foi encontrada recomendação para este valor, só há um default que é 12, mas na execução do treinamento achamos mais apropriado o tamanho 16, pois usando o default o treinamento ficava mais demorado. É importante salientar que ao aumentar o tamanho do *batch* pode-se diminuir o número de *steps*.

No ANEXO J, é baixado e extraído o modelo selecionado anteriormente. Nesse mesmo anexo, temos os 'checkpoints' que são os pontos de verificação. Eles capturam o valor exato de todos os parâmetros usados pelo modelo. Vale salientar que os pontos de verificação não contêm nenhuma descrição do cálculo definido pelo modelo e só são úteis quando o código-fonte que usará os valores de parâmetro salvos está disponível.

No ANEXO K criamos um arquivo que será quase uma cópia do arquivo de configuração para SSD Mobilenet v2 que está disponível em [36], exceto pelas mudanças realizadas para adequá-lo ao nosso trabalho. Para obter esse novo arquivo de configuração, realizamos as modificações que são apresentadas a continuação.

- *Num_classes*: corresponde ao número de classes, o qual será 4.
- Na linha 141, define-se o *batch_size* que corresponde ao tamanho do batch como sendo 16.
- *initial_learning_rate*: define-se como sendo 0.004, e corresponde à taxa de aprendizagem com a qual irá começar o treinamento.
- *Fine_tune_checkpoint*: deve ser colocado o diretório onde são salvos os pontos de verificação.
- *Num_steps*: número de iterações que desejamos realizar. Esse valor foi definido como sendo 120000 para seguir a metodologia aplicada para YOLOv3.
- Na linha 175 define-se o caminho '*input_path*' onde se encontram os arquivos '.record' para treinamento do modelo.
- Na linha 177 define-se o caminho '*label_map_path*' onde se encontra o arquivo '.pbtxt'.
- Analogamente, na linha 190 define-se o caminho '*input_path*' onde se encontram os arquivos '.record' para avaliar o modelo.
- Na linha 192, novamente define-se o caminho '*label_map_path*' para o arquivo '.pbtxt'.

No ANEXO L será definida a pasta de destino para salvar o nosso modelo em cada ponto de verificação enquanto se realiza o treinamento. Além disso, neste anexo também foram colocadas todas as especificações necessárias para a utilização do *Tensorboard*. A ferramenta

do *Tensorflow* chamada *Tensorboard* nos ajuda a entender melhor o que está sendo feito pelo modelo. Ela permite a visualização de qualquer estatística de uma rede neural. Portanto, na segunda parte do ANEXO L, baixamos e descompactamos o *ngrok* que nos permite acessar o *Tensorboard* desde o Google Colaboratory. E, também se cria um link que ao ser acessado permite visualizar os gráficos e parâmetros apresentados no *Tensorboard*, durante o treinamento.

Com os itens descritos anteriormente, podemos realizar o treinamento do SSD, executando o código Python "model_main.py". Nessa mesma célula do notebook também se deve especificar no mínimo, o caminho do arquivo de configuração e indicar o diretório do modelo salvo para que seja utilizado.

Após o treinamento, o próximo passo é exportar ou extrair o gráfico de inferência recém-treinado. Esse gráfico será usado posteriormente para realizar a detecção dos objetos. Para realizar este processo é necessário utilizar o código apresentado no ANEXO M.

Utilizando a primeira parte do código do ANEXO M podemos encontrar o último modelo treinado '*last_model_path*'. Em seguida, executando o código Python 'export_inference_graph.py' converte-se o modelo em um modelo congelado 'frozen_inference_graph.pb', que pode ser usado para inferência. Vale salientar que este modelo congelado não pode ser utilizado para retomar o treinamento. Uma observação importante é que se o *kernel* (da GPU) 'morrer', o treinamento será retomado do último checkpoint. Isto é realizado sempre e quando seja salvo o diretório 'training/' em algum lugar, por exemplo, no Google Drive.

Para poder passar à fase da detecção de objetos nas imagens, além de exportar o modelo também é necessário baixar no Google Colab alguns dos arquivos mencionados anteriormente, o modelo congelado para inferência que foi nomeado 'pb_fname' e o arquivo '.pbtxt', observe o ANEXO N. A continuação, é realizada a detecção de objetos em imagens, observe a última parte do ANEXO N. Isto é executado de acordo ao trabalho apresentado pelo autor em [38]. Primeiro, especificam-se todos os caminhos das imagens nas quais queremos realizar as detecções, para isto cria-se uma pasta chamada 'test' com todas elas. E, depois se apresenta o código que nos ajuda a realizar a detecção propriamente dita, ou seja, nesta etapa estamos testando o modelo.

Para realizar a avaliação do modelo existem várias alternativas, mas a forma mais prática e simples é mostrada aqui, seguindo o roteiro apresentado pelo autor em [40]. Primeiro, é executado o código em Python 'infer_detection.py', que ajuda a gerar um arquivo

‘record detection’ necessário para obtenção da matriz de confusão do modelo. Além disso, deve ser utilizado o modelo congelado e o arquivo ‘.record’ criado para avaliação, ou seja, o arquivo ‘test_labels.record’. E ainda, deve ser criado um diretório para o novo arquivo.

Em seguida, é preciso baixar no Google Drive uma pasta chamada de ‘tf_object_detection_cm-master’ criada pelo autor em [39] porque ela contém o código Python chamado ‘confusion_matriz.py’ que será executado em seguida, para criar efetivamente a matriz de confusão. Para isto é necessário utilizar o arquivo criado anteriormente, além do arquivo ‘label_map.pbtxt’ criado para o treinamento. Vale salientar que o processo aplicado anteriormente também entrega resultados com relação à precisão média de cada uma das classes.

4. APRESENTAÇÃO E DISCUSSÃO DOS RESULTADOS

Neste capítulo são apresentados os resultados obtidos após treinamentos dos modelos YOLOv3 e SSD MobileNet v2. Além disso, para todos os treinamentos, os resultados foram obtidos utilizando a GPU Tesla P100-PCIE-16GB.

4.1 RESULTADOS PARA YOLOv3 SEM AUMENTO

Para avaliação do modelo foi utilizado o último treinamento realizado, com 12000 iterações. Além disso, foram consideradas 200 imagens para avaliação, 50 para cada classe e 10 imagens para realizar detecções. No conjunto de imagens de avaliação, o modelo realiza 546 detecções porque existem imagens que contém vários objetos, inclusive vários da mesma classe.

Com relação aos resultados obtidos para YOLOv3, para cada umas das classes temos as seguintes precisões médias (AP), observe a Tabela 1.

Tabela 1 – Precisões médias de classe YOLOv3 sem aumento

Identificador de Classe	Nome da Classe	Precisão Média
0	Caixa	48.22%
1	Mesa	75.78%
2	Cadeira com rodas	97.02%
3	Cadeira comum	80.32%

Fonte: O Autor.

Ao utilizar o arquivo que contém os melhores pesos, ou seja, maior IOU, pode-se concluir que o modelo identifica com uma boa confiança as classes “Mesa”, “Cadeira com rodas” e “Cadeira comum’, observe as três últimas linhas da Tabela 1. Para a classe “Caixa”, YOLOv3 apresenta uma certa dificuldade para identificar esse objeto. Não tem muita certeza de que o objeto detectado realmente é uma “Caixa”, só 48.22% de confiança.

Além dos resultados para cada classe, na avaliação do modelo são calculadas algumas das métricas mais conhecidas e utilizadas para fazer a devida análise do comportamento do mesmo e por sua vez poder compará-lo com outros modelos. Essas métricas são mostradas a seguir.

- *Precision*

- *Recall*
- *F1-score*

Quanto as métricas obtidas para YOLOv3 sem aumento de dados, temos os seguintes resultados, observe a Tabela 2.

Tabela 2 – Métricas obtidas para YOLOv3 sem aumento

Nome da métrica	Valor
Precision	85%
Recall	67%
F1-score	75%

Fonte: O Autor.

Quando temos um conjunto de dados proporcional, ou seja, com uma distribuição igual das classes, a métrica *precision* nos dá uma boa avaliação do modelo, mas uma maneira mais completa de avaliar o modelo é observando a métrica F1-score. Essa métrica leva em consideração não apenas o número de erros de predição que o modelo comete, mas também considera os tipos de erros cometidos. O maior valor teórico da F1-score é 1, isso indica que o nosso modelo está se desempenhando relativamente bem, pois a F1-score vale 0.75.

Com relação ao conjunto de imagens para fazer detecções, foi criada uma pasta chamada “DetecImg” contendo 10 imagens não conhecidas pelo modelo. Os resultados mais relevantes dos testes realizados para detecção são apresentados a seguir.

Primeiro, apresentamos uma imagem que contém os quatro objetos, observe a Figura 27.

Figura 27 – Predições YOLOv3 sem aumento, em imagem com quatro objetos

Fonte: O Autor.

Percebe-se na Figura 27, que o modelo apresenta um pouco de dificuldade em identificar o objeto “Caixa” como já tinha se notado antes, ele apenas tem 56% de certeza de que detectou uma “Caixa”. Com relação à classe “Mesa” tem 85% de certeza de que detectou uma “Mesa”, porém para o mesmo objeto “Mesa”, o modelo dá uma certeza de 32% de ser uma “Cadeira comum”. Isto poderia ser explicado com o fato de que a mesa parece uma cadeira sem encosto, mas a ideia é que ele não se engane ao realizar essas detecções. Portanto, precisamos melhorar a *performance* do nosso modelo YOLOv3.

Ao realizarmos o teste com uma imagem contendo os objetos “Mesa” e “Caixa”, o modelo não apresenta problemas para detectá-los, observe a Figura 28.

Figura 28 – Predições YOLOv3 sem aumento, em imagem com dois objetos

Fonte: O Autor.

As outras imagens testadas não são apresentadas aqui, mas em resumo apresentaram resultados excelentes (todos os objetos de cada uma das imagens foram detectados com confianças acimas de 99%).

4.2 RESULTADOS PARA SSD MobileNet v2 SEM AUMENTO

Com relação à avaliação do modelo SSD foi utilizado também o último treinamento realizado, com 120000 iterações. Isto porque foi seguida a mesma metodologia utilizada para YOLOv3, seguindo a ideia apresentada pelos autores em [52], nele são realizadas 8000 iterações para YOLOv3 e 80000 para SSD, isto para que o modelo SSD consiga ter uma performance análoga à realizada por YOLOv3, ao fazer as detecções dos objetos nas imagens.

Ao igual que YOLOv3, também foram consideradas 200 imagens para avaliação do modelo, 50 para cada classe.

Com relação aos resultados obtidos para SSD, para cada umas das classes temos as seguintes precisões médias (AP), observe a Tabela 3.

Tabela 3 – Precisões médias de classe SSD sem aumento

Identificador de Classe	Nome da Classe	Precisão Média
2	Caixa	91.11%
1	Mesa	80.44%
4	Cadeira com rodas	86.36%
3	Cadeira comum	75.25%

Fonte: O Autor.

Neste caso, quando o modelo detecta um objeto em uma imagem, consegue fazê-lo com uma boa precisão. Observando a Tabela 3, o SSD apresenta mais dificuldade para identificar o objeto “Cadeira comum”, enquanto YOLOv3 apresentava mais dificuldade para identificar o objeto “Caixa”.

Além dos resultados para cada classe, na avaliação do modelo também são calculadas as métricas mais conhecidas e utilizadas para fazer a devida análise do comportamento do mesmo e por sua vez poder compará-lo com outros modelos, como realizado para YOLOv3.

Quanto às métricas obtidas para SSD sem aumento de dados, temos os seguintes resultados, observe a Tabela 4.

Tabela 4 – Métricas obtidas para SSD sem aumento

Nome da métrica	Valor
Precision	83.29%
Recall	64.59%
F1-score	72.75%

Fonte: O Autor.

Analogamente a YOLOv3, analisou-se a métrica F1-score. Neste caso, vale 0.73 aproximadamente, indicando também um desempenho relativamente bom. Essas métricas foram calculadas com relação a matriz de confusão apresentada na Figura 29. Note que na matriz, a classe 0 representa a classe vazia.

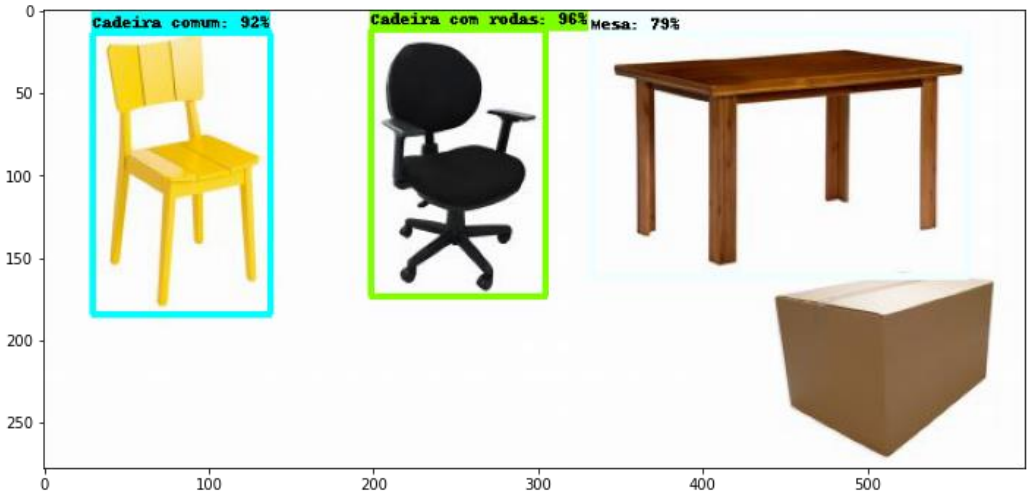
Figura 29 – Matriz de confusão para o modelo SSD sem aumento de dados

Classe	0	1	2	3	4
0	0	2	3	13	5
1	7	37	1	9	1
2	37	1	41	1	2
3	40	6	0	73	1
4	13	0	0	1	57

Fonte: O Autor.

Com relação ao conjunto de imagens para detecção, foi criada uma pasta chamada “test” contendo as mesmas 10 imagens usadas para uma avaliação quantitativa das detecções no modelo YOLOv3. Os resultados mais relevantes desses testes são apresentados a seguir.

Figura 30 – Predições SSD sem aumento, em imagem com quatro objetos

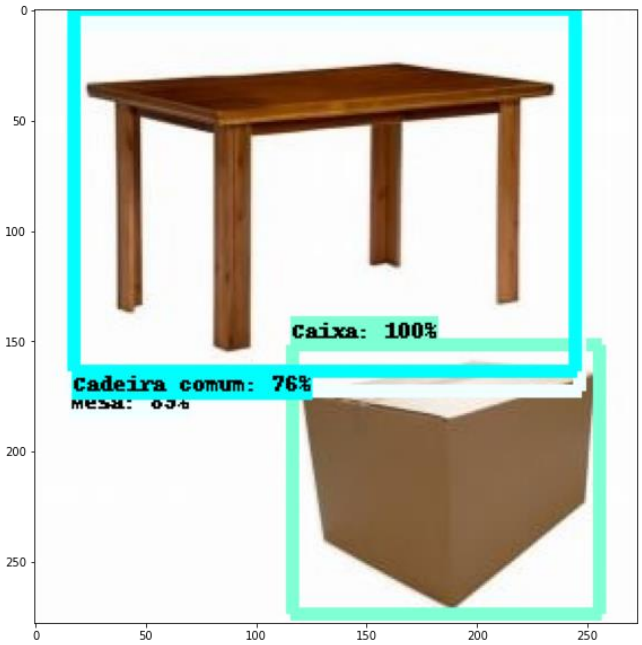


Fonte: O Autor.

Observe na Figura 30, que o modelo SSD não consegue detectar o objeto “Caixa”. Além disso, os outros três objetos são detectados, mas com confianças um pouco menores que as do modelo YOLOv3.

Agora, vamos apresentar outro resultado, que é análogo ao teste realizado para YOLOv3. Neste caso, temos a mesma imagem usada por YOLOv3, observe a Figura 31.

Figura 31 – Predições SSD sem aumento, em imagem com dois objetos



Fonte: O Autor.

Observe na Figura 31, que o modelo SSD ao igual que o YOLOv3 está detectando o objeto “Mesa”, mas ao mesmo tempo, o SSD está se enganando pois também está detectando a “Mesa” como se fosse uma “Cadeira comum” e ainda com uma certeza consideravelmente alta, 76%. Isto indica que o modelo SSD também deve ser melhorado.

Análogo a YOLOv3, para as outras imagens utilizadas para realizar detecções, foram obtidos resultados excelentes (todos os objetos de cada uma das imagens foram detectados com confianças acimas de 99%).

4.3 RESULTADOS PARA YOLOv3 COM AUMENTO

Com relação à avaliação do modelo foi utilizado treinamento realizado, com 12000 iterações. Isto para seguir a metodologia utilizada anteriormente, mas agora incorporando aumento. Além disso, foram consideradas 400 imagens para avaliação, 100 para cada classe e 10 imagens para realizar detecções. Em relação aos resultados obtidos para YOLOv3, para cada umas das classes temos as seguintes precisões médias (AP), observe a Tabela 5.

Tabela 5 – Precisões médias de classe YOLOv3 com aumento

Identificador de Classe	Nome da Classe	Precisão Média
0	Caixa	56.92%
1	Mesa	78.57%
2	Cadeira com rodas	98.83%
3	Cadeira comum	82.89%

Fonte: O Autor.

Na Tabela 5 observa-se que o modelo melhorou com relação aos resultados obtidos para YOLOv3 sem aumento de dados, houve aumento de todas as precisões médias das classes. Ele identifica com uma boa confiança as classes “Mesa”, “Cadeira com rodas” e “Cadeira comum’. Para a classe “Caixa”, YOLOv3 apresenta ainda uma certa dificuldade para identificar o objeto. Não tem muita certeza de que o objeto detectado realmente é uma “Caixa”, só 56.92% de confiança.

Quanto as métricas obtidas para YOLOv3 com aumento de dados, temos os seguintes resultados, observe a Tabela 6.

Tabela 6 – Métricas obtidas para YOLOv3 com aumento

Nome da métrica	Valor
Precision	86%
Recall	75%
F1-score	80%

Fonte: O Autor.

Observe que todas as métricas aumentaram, e essa era a ideia: melhorar o modelo. De novo considerando a métrica *F1-score* para avaliar YOLOv3 e lembrando que essa métrica leva em consideração não apenas o número de erros de predição que o modelo comete, mas também considera tipos de erros cometidos. Para YOLOv3 depois de realizar o aumento de dados, F1-score vale 0.8.

Com relação aos testes realizados, de novo consideramos a pasta “DetecImg” contendo 10 imagens não conhecidas pelo modelo para fazer uma avaliação quantitativa das detecções que realiza o modelo. Os resultados mais relevantes das detecções realizadas são apresentados à continuação.

Figura 32 – Predições YOLOv3 com aumento, em imagem com quatro objetos

Fonte: O Autor.

Percebe-se na Figura 32, que o modelo apresenta uma melhora com relação ao aumento da confiança ao identificar o objeto “Caixa”, 60% de certeza. Com relação à classe “Mesa”, apesar de ter diminuído um pouco a confiança, ele não confunde mais a “Mesa” com a “Cadeira comum”.

Ao realizarmos o teste com uma imagem contendo somente os objetos “Mesa” e “Caixa” como feito anteriormente sem o aumento de dados, o modelo continua a não apresentar nenhum problema como mostra a Figura 33. Ele consegue detectar bem melhor (ao compararmos ao YOLOv3 sem aumento de dados) os dois objetos e com uma boa certeza, 100% a “Mesa” e 92% a “Caixa”.

Figura 33 – Predições YOLOv3 com aumento, em imagem com dois objetos

Fonte: O Autor.

As outras imagens testadas não são apresentadas aqui, mas em resumo apresentaram resultados excelentes (todos os objetos de cada uma das imagens foram detectados com confianças acimas de 99%).

4.4 RESULTADOS PARA SSD MobileNet v2 COM AUMENTO

Com relação à avaliação do modelo SSD foi utilizado também o último treinamento realizado, com 120000 *steps*, seguindo a metodologia utilizada para YOLOv3.

Ao igual que YOLOv3, também foram consideradas 400 imagens para avaliação, 100 para cada classe.

Com relação aos resultados obtidos para SSD com aumento de dados, para cada umas das classes temos as seguintes precisões médias (AP), observe a Tabela 7.

Tabela 7 – Precisões médias de classe SSD com aumento

Identificador de Classe	Nome da Classe	Precisão Média
2	Caixa	91.95%
1	Mesa	88.04%
4	Cadeira com rodas	92.80%
3	Cadeira comum	80.57%

Fonte: O Autor.

Neste caso, o modelo melhora com relação às precisões médias para cada classe. Além dos resultados para cada classe, para avaliação do modelo observe as métricas obtidas na Tabela 8.

Tabela 8 – Métricas obtidas para SSD com aumento

Nome da métrica	Valor
Precision	88.34%
Recall	74.69%
F1-score	80.94%

Fonte: O Autor.

Análogo ao realizado para YOLOv3, analisou-se a métrica F1-score. Neste caso, vale 0.81 aproximadamente, indicando também uma melhora no desempenho do SSD. Essas métricas foram calculadas com relação à matriz de confusão apresentada na Figura 34.

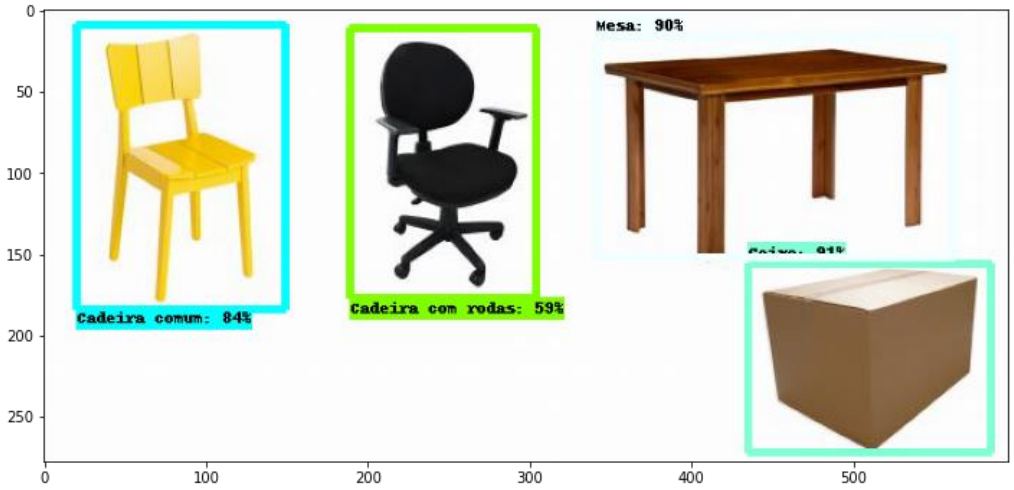
Figura 34 – Matriz de confusão para o modelo SSD com aumento de dados

Classe	0	1	2	3	4
0	0	4	4	17	7
1	10	81	2	14	0
2	51	2	80	2	1
3	41	5	0	141	1
4	12	0	1	1	116

Fonte: O Autor.

Com relação a algumas detecções quantitativas realizadas pelo modelo apresentamos alguns resultados relevantes, observe a Figura 35.

Figura 35 - Predições SSD com aumento, em imagem com quatro objetos

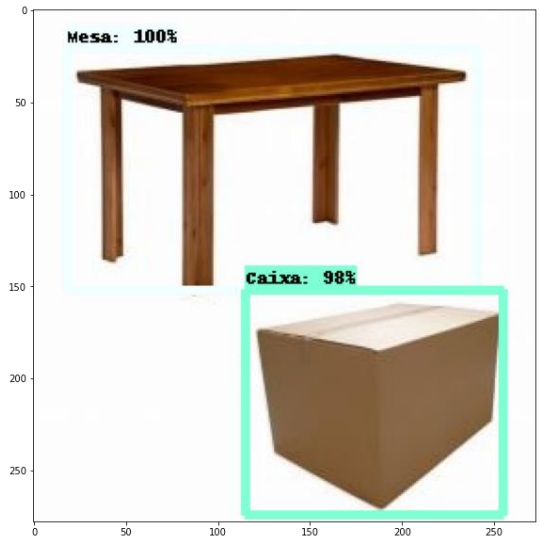


Fonte: O Autor.

Observe na Figura 35, que depois de realizado o aumento de dados, o modelo SSD consegue melhorar e detectar todos os objetos com valores altos de confiança.

Agora, vamos apresentar outro resultado, análogo ao teste realizado para YOLOv3 para uma imagem com dois objetos, observe a Figura 36.

Figura 36 – Predições SSD com aumento, em imagem com dois objetos



Fonte: O Autor.

Observe na Figura 36, que o modelo SSD ao igual que YOLOv3 está detectando corretamente os objetos, com valores altos de confiança, indicando que o modelo também melhorou como pretendido.

Análogo ao realizado para YOLOv3, as outras imagens testadas para detecções quantitativas realizadas pelo modelo, foram conseguidos resultados excelentes (todos os objetos de cada uma das imagens foram detectados com confianças acimas de 99%).

4.5 COMPARAÇÃO E DISCUSSÃO DOS RESULTADOS

O problema de utilizar a métrica *Accuracy* para avaliar os modelos é que ela atribui o mesmo peso para todos os erros e dependendo do caso pode ser que, por exemplo, um erro por falso negativo seja bem mais grave. Assim, temos outras métricas que nos ajudam avaliar melhor os modelos.

A métrica *precision* dá uma ênfase maior para os erros por falso positivo respondendo à pergunta: dos exemplos classificados como positivos quantos realmente são positivos?

Por outro lado, a métrica *recall,* que traduzida ao português é chamada de revocação ou sensibilidade ou taxa de verdadeiros positivos, dá maior ênfase para os erros por falsos negativos respondendo à pergunta: de todos os exemplos que são positivos, quantos foram corretamente classificados como positivos (mesmo que sejam classificados em outra classe)?

A métrica *F1-score* também conhecida como *F-measure*, leva em consideração tanto a métrica *precision* quanto a *recall*. Um modelo que apresenta *F1-score* alto é um modelo que é capaz de acertar suas predições como também consegue recuperar os exemplos da classe de interesse. Essa métrica é um melhor resumo da qualidade do nosso modelo se comparada à acurácia. A *F1-score* é muito boa quando usada em um *dataset* com classes desproporcionadas, mas neste trabalho também pode perfeitamente ser utilizada para termos maior certeza de que os modelos estão sendo bem avaliados. Cabe dizer que em geral, quanto maior o *F1-score*, mais próxima de 1, melhor.

Comparando os modelos com relação ao desempenho. Vamos considerar os resultados das métricas obtidas após treinamento com 12000 iterações para YOLOv3 e com 120000 iterações para SSD Mobilenet v2.

Observe que a exatidão, ou seja, a métrica *precision* é um pouco maior para YOLOv3. Isto significa que quando esse modelo detecta um objeto ele tem mais certeza dessa detecção do que o SSD. Além disso, YOLOv3 consegue detectar mais do que o SSD pois, a métrica *recall* para a YOLOv3 é 67% e 64.59% para o modelo SSD Mobilenet v2. Como consideramos mais interessante comparar os modelos pela métrica *F1-score*, YOLOv3 está se desempenhando melhor, pois a F1-score para esse modelo é 0.75, enquanto para SSD MobileNet v2 é aproximadamente 0.73.

Agora, vamos comparar os modelos quanto ao número de iterações e tempo para execução de um treinamento. Para YOLOv3 é necessário realizar um número bem menor de iterações para obter resultados de detecção semelhantes aos do modelo SSD. Por exemplo, para o YOLOv3 foi necessário realizar 12000 iterações para conseguir detectar os quatro objetos. Por outro lado, o modelo SSD precisa de 120000 iterações para ter resultados semelhantes. Em suma, o número de iterações para SSD é aproximadamente 10 vezes maior se comparado ao número de iterações necessárias para YOLOv3.

Quanto ao tempo de treinamento, YOLOv3 demora muito mais tempo para realizar o treinamento, para o mesmo número de iterações, do que o modelo SSD Mobilenet v2. Por exemplo, para YOLOv3 realizar 1000 iterações precisa mais do que uma hora de treinamento enquanto SSD Mobilenet v2 precisa de no máximo 10 minutos. Isto vale sempre e quando esteja sendo utilizada a mesma GPU.

Outro fato que pode ser analisado é que aparentemente o modelo SSD Mobilenet v2 se mostrava ‘bem melhor’ do que o YOLOv3 se olharmos só os resultados das precisões médias das classes. Isto porque esses valores para SSD são maiores (todos acima de 75% como

mostra a Tabela 4), porém temos que considerar o *recall* médio para cada uma das classes, provavelmente são mais baixos se comparados aos que teria YOLOv3 para cada classe. Os valores do *recall* médio não foram apresentados na avaliação dos modelos, mas certamente esses valores devem ser melhores para YOLOv3.

Lembrando que neste trabalho, consideramos a métrica *F1-score* a mais interessante para avaliar os modelos. Ela novamente será considerada para comparar os modelos após aumento de dados. Neste caso, temos que para YOLOv3, F1-score vale 0.8. Por outro lado, para SSD MobileNet v2, F1-score é aproximadamente igual a 0.81. Isto quer dizer que desta vez o modelo SSD apresentou um desempenho um pouco melhor do que o modelo YOLOv3.

Observe que a exatidão, ou seja, a métrica *precision* desta vez é um pouco maior para SSD MobileNet v2. Isto significa que quando esse modelo detecta um objeto ele tem mais certeza dessa detecção do que o YOLOv3. Por outro lado, YOLOv3 consegue detectar mais do que o SSD pois, a métrica *recall* para YOLOv3 é 75% se comparado a 74.69% do modelo SSD Mobilenet v2.

Além disso, ao compararmos os modelos quanto ao número de iterações e tempo para execução de um treinamento, pode-se concluir exatamente o mesmo que foi apresentado anteriormente para os modelos sem aumento de dados.

Em resumo, na Tabela 9 é apresentada uma compilação dos resultados obtidos.

Tabela 9 – Resumo dos resultados obtidos

Modelo	Métricas sem *data augmentation*	Métricas com *data augmentation*	Nome da Métrica
YOLOv3	85.00%	86.00%	*Precision*
	67.00%	75.00%	*Recall*
	75.00%	80.00%	*F1-Score*
SSD MobileNet v2	83.29%	88.34%	*Precision*
	64.59%	74.69%	*Recall*
	72.75%	80.94%	*F1-Score*

Fonte: O Autor.

5. CONCLUSÕES

A execução deste projeto permitiu o conhecimento e aprofundamento de conceitos importantes dentro da visão computacional. O processo desenvolvido principalmente para o modelo SSD Mobilenet v2 foi bem completo e inclui todos os passos necessários para o treinamento da maioria de modelos para detecção de objetos em imagens.

Os resultados com relação aos dois modelos (YOLOv3 e SSD MobileNet v2) treinados, poderiam ter sido melhores se houvéssemos trabalhado com um *dataset* que contivesse mais imagens, mas a ideia era trabalhar com base no *dataset* criado por Douglas H. dos Reis em [12]. Os desenvolvedores geralmente trabalham com conjuntos de dados maiores ou bem maiores, em alguns casos, e isso para garantir melhor desempenho dos modelos.

Além disso, devem-se considerar as restrições do *Google Colaboratory* e exatamente por conta dessas restrições é que foram escolhidos os modelos: SSD Mobilenet v2, que é um modelo que o *Google Colaboratory* suporta. E, o modelo YOLOv3 com as devidas modificações na Darknet, realizadas por Alexey em [22].

Tendo em conta as considerações mencionadas anteriormente, e após avaliação de cada um dos modelos, e comparação para cada um dos casos: sem aumento de dados e com aumento, concluímos que os dois basicamente apresentam um equilíbrio quanto ao desempenho, pois o YOLOv3 conseguiu se sair um pouco melhor do que SSD sem o aumento de dados (F1-score para YOLOv3 de 0.75 e para SSD MobileNet v2, 0.73). Por outro lado, o SSD conseguiu se sair um pouco melhor que o YOLOv3 com o aumento de dados (F1-score para YOLOv3 de 0.80 e para SSD MobileNet v2, aproximadamente 0.81), indicando que o aumento de dados favorece mais o modelo SSD.

Outra conclusão importante é que ao compararmos os resultados obtidos com os resultados do artigo que apresentamos em [52], observa-se que é importante a maneira como são rotuladas as imagens. Neste trabalho, as métricas ficaram mais próximas para os modelos sem aumento, pois se utilizou a ferramenta Roboflow para seguir as rotulações realizadas pelo autor em [12].

REFERÊNCIAS

[1] SILVA, A.; CINTRA, M. E. **Reconhecimento de padrões faciais: Um estudo.** Universidade Federal Rural do Semi-Árido. Disponível em: <https://pdfs.semanticscholar.org/aa94/f214bb3e14842e4056fdef834a51aecef39c.pdf>. Acesso em: 20 de abril de 2020.

[2] LECUN, Y. et all. **Aprendizagem profunda**. Nature 521**,** 436-444. 2015. Disponível em: <https://doi.org/10.1038/nature14539>. Acesso em: 25 de abril de 2020.

[3] JAIN, A. K, MAO, J., MOHIUDDIN, K.M. **Artificial neural network: a tutorial.** IEEE Computer, v. 29, n. 3, p. 56-63, 1996. Acesso em: 20 de maio de 2020.

[4] VARGAS, Ana C. G. et. All. **Um Estudo sobre Redes Neurais Convolucionais e sua Aplicação em Detecção de Pedestres**. UFF, Niterói, Brasil.

[5] REDMON, J.; FARHADI, A. **You Only Look Once: Unified, Real-Time Object Detection**. Conference on Computer Vision and Pattern Recognition - CVPR, 2016.

[6] REDMON, J.; FARHADI, A. **YOLOv3: An Incremental Improvement**, 2018. Disponível em: <http://resources.dbgns.com/study/ObjectDetection/YOLOv3.pdf>. Acesso em: 23 de maio de 2020.

[7] WEI, L. et. all. **SSD: Single Shot MultiBox Detector**, 2016. Disponível em: <https://arxiv.org/pdf/1512.02325.pdf>. Acesso em 10 de junho de 2020.

[8] SZEGEDY, C. et. all. **Scalable, High-Quality Object Detection**, 2015. Disponível em: <https://arxiv.org/abs/1412.1441>. Acesso em: 10 de junho de 2020.

[9] ERHAN, D. et all. **Scalable object detection using deep neural networks**, 2014. Disponível em: < https://arxiv.org/abs/1312.2249>. Acesso em: 21 de junho de 2020.

[10] GAO, H. **Understand single shot multibox detector (SSD) and implement it in pytorch**, 2018. Disponivel em: <https://medium.com/@smallfishbigsea/understand-ssd-and-implement-your-own-caa3232cd6ad>. Acesso em: 15 Março 2020.

[11] HUI, J. **SSD object detection: Single Shot Mutibox Detector for real-time processing**, 2018. Disponível em: <https://medium.com/@jonathan_hui/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06>. Acesso em: 5 de julho de 2020.

[12] DOS REIS H., Douglas. **Software de reconhecimento de objetos utilizando imagens RGBD do Kinect e sua aplicação na navegação de um robô móvel.** 2019. Trabalho de

conclusão do curso de Engenharia de controle e automação - Universidade Federal de Santa Maria, Santa Maria, 2019.

[13] GOOGLE COLABORATORY. **Welcome To Colaboratory**. 2020. Disponível: <https://colab.research.google.com/notebooks/intro.ipynb>. Acesso: 28 de abril de 2020.

[14] NVIDIA. **CUDA Zone**. 2020. Disponível em: < https://developer.nvidia.com/cuda-zone>. Acesso: 30 de abril de 2020.

[15] NVIDIA. **cuDNN**. 2020. Disponível em: <https://developer.nvidia.com/cudnn>. Acesso: 30 de abril de 2020.

[16] NUMPY. **NumPy**. Disponível em: <http://www.numpy.org/>. Acesso em: 6 de abril de 2020.

[17] MATPLOTLIB. **Matplotlib: Python plotting**. Disponível em: <https://matplotlib.org/>. Acesso em: 7 de maio de 2020.

[18] OPENCV. **OpenCV library**. Disponível em: <https://opencv.org/>. Acesso em: 8 de maio de 2020.

[19] READMON, J. **Darknet: Open Source Neural Networks in C**. Disponível em: <https://pjreddie.com/darknet/>. Acesso em: 15 de abril de 2020.

[20] ALEXEY AB. **Repositório contido no Github: Yolo_mark**. 2017. Disponível em: <https://github.com/AlexeyAB/Yolo_mark>. Acesso em: 23 de maio de 2020.

[21] READMON, J. **Repositório contido no Github: darknet**. Disponível em: https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg>. Acesso em: 6 de maio de 2020.

[22] ALEXEY AB. **Repositório contido no Github: Darknet**. 2017. Disponível em <https://github.com/AlexeyAB/darknet/>. Acesso em: 10 de julho de 2020.

[23] READMON, J. **Darknet: Open Source Neural Networks in C**. Disponível em: <https://pjreddie.com/darknet/yolo>. Acesso em: 3 de abril de 2021.

[24] GONCHAROV, I. **Repositório do Github: YOLOv3-GoogleColab**. Disponível em: <https://github.com/ivangrov/YOLOv3-GoogleColab>. Acesso em: 5 de maio de 2020.

[25] GENZ R. **Repositório do Github: How to train your own dataset with YOLOv3 using Darknet on Google Colaboratoy.** 2020. Disponível em: <https://github.com/robingenz/object-detection-yolov3-google-colab>. Acesso em: 12 de maio de 2020.

[26] HUANG, J. et al. **Speed/accuracy trade-offs for modern convolutional object detectors**. 2017. Disponível: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). [S.l.: s.n.], 2017. p. 3296–3297. ISSN 1063-6919.

[27] HUYNH, S. **How to install LabelLimg in Windows with Anaconda?** 2020. Disponível em: https://medium.com/@sanghuynh_73086/how-to-install-labelimg-in-windows-with-anaconda-c659b27f0f>. Acesso em: 15 de agosto de 2020.

[28] Wikipédia Foundation Inc (WIKIPÈDIA). **Anaconda (Python distribution)**. 2020. Disponível em: <https://en.wikipedia.org/wiki/Anaconda_(Python_distribution)>. Acesso em: 26 de agosto de 2020.

[29] Data Science Academy. **O que é o TensorFlow machine intelligence platform?** 2020. Disponível em: http://datascienceacademy.com.br/blog/o-que-e-o-tensorflow-machine-intelligence-platform/>. Acesso em: 01 de dezembro de 2020.

[30] Pypi.org. **TensorFlow-Slim: A lightweight library for defining training and evaluating complex models in TensorFlow**. 2020. Disponível em: https://pypi.org/project/tf-slim/>. Acesso: 21 de agosto de 2020.

[31] PRIVALOV, V. **Training AlekseyAB YOLOv3 on own dataset in Google Colab**. 2020. Disponível em: <https://medium.com/@vovaprivalov/training-alekseyab-yolov3-on-own-dataset-in-google-colab-8f3de8105d86>. Acesso em: 10 de outubro de 2020.

[32] TZUTALIN, D. **Repositório do Github: tzutalin/LabelImg**. 2020. Disponível em: <https://github.com/tzutalin/labelImg>. Acesso em 16 de agosto de 2020.

[33] SINJAB, A. **Step by Step: Build Your Custom Real-Time Object Detector**. 2019. Disponível em: <https://towardsdatascience.com/detailed-tutorial-build-your-custom-real-time-object-detector-5ade1017fd2d>. Acesso em: 19 de agosto de 2020.

[34] JOSÉ, I. **Custom object detection for non-data scientists – Tensorflow.** 2019. Disponível em: https://towardsdatascience.com/custom-object-detection-for-non-data-scientists-70325fef2dbb>. Acesso em: 22 de agosto de 2020.

[35] SINLAB, A. **Notebook do Colab: weapon_detection_BL_ipynb**. 2019. Disponível em: <https://colab.research.google.com/github/AlaaSenjab/-Tutorial-Tensorflow_Object_Detection_API_On_Custom_Dataset/blob/master/weapon_detection_BL.ipynb#scrollTo=t9C3L_r4Pi6m>. Acesso em: 18 de julho de 2020.

[36] NELSON, J. **Notebook do Colab: Roboflow-tensorflow-object-detection-mobilenet-colab.ipynb**. 2020. Disponível em: <https://colab.research.google.com/drive/1wTMIrJhYsQdq_u7ROOkf0Lu_fsX5Mu8a#scrollTo=CjDHjhKQofT5>. Acesso em: 23 de agosto de 2020.

[37] PKULZC**. Repositório do Github: tensorflow/models**. 2018. Disponível em: <https://github.com/tensorflow/models/blob/master/research/object_detection/samples/configs/ssd_mobilenet_v2_coco.config>. Acesso em: 24 de agosto de 2020.

[38] NELSON, J. **Training a TensorFlow MobileNet Object Detection Model with a Custom Dataset.** 2020. Disponível em: <https://blog.roboflow.com/training-a-tensorflow-object-detection-model-with-a-custom-dataset/>. Acesso em: 23 de agosto de 2020.

[39] VALDERRAMA, S. **Repositório do Github: Confusion Matrix in Object Detection with TensorFlow**. 2018. Disponível em: <github.com/svpino/tf_object_detection_cm>. Acesso em: 04 de dezembro de 2020.

[40] VALDERRAMA, S. **Confusion Matrix in Object Detection with TensorFlow**. 2018. Disponível em: <https://towardsdatascience.com/confusion-matrix-in-object-detection-with-tensorflow-b9640a927285>. Acesso em: 04 de dezembro de 2020.

[41] LOKHMOTOV, A. **Omni-benchmarking Object Detection**. 2019. Disponível em:<https://towardsdatascience.com/omni-benchmarking-object-detection-b390cc4114cd>. Acesso em: 18 de dezembro de 2020.

[42] Y. Liu, X. Sun, J. H. L. Pang, **"IVSP '20: Proceedings of the 2020 2nd International Conference on Image, Video and Signal Processing"**, pp. 33-38, 2020. Disponível em: <https://doi.org/10.1145/3388818.3388827>. Acesso em: 10 de janeiro 2021.

[43] R. Anhu, N. Xiaotong, B. Jingjing, **"Application of YOLOv3 in road traffic detection"**. 14th IEEE International Conference on Electronic Measurement & Instruments (ICEMI): pp. 1731-1734, 2019 DOI: 10.1109/ICEMI46757.2019.9101888, 2019.

[44] X. Li, M. Tian, S. Kong, L. Wu, J. Yu, **"A modified YOLOv3 detection method for vision-based water surface garbage capture robot"**. International Journal of Advanced Robotic Systems: Vol. 17, Issue 3, May-June 2020. Disponível em: <https://doi.org/10.1177/1729881420932715>. Acesso em: 11 de janeiro de 2021.

[45] J. D. Kelly, J. D. Hedengren, **"A steady-state detection (SSD) algorithm to detect non-stationary drifts in processes"**, Journal of Process Control, Vol. 23, Issue 3, 2013, pp. 326-331, ISSN 0959-1524, Disponível em: <https://doi.org/10.1016/j.jprocont.2012.12.001>. Acesso em: 12 de janeiro de 2021.

[46] Y. Li, H. Huang, Q. Xie, L. Yao, Q. Chen, **"Research on a Surface Defect Detection Algorithm Based on MobileNet-SSD"**, Applied Sciences, 8(9):1678, 2018. Disponível em: <https://doi.org/10.3390/app8091678>. Acesso em: 12 de janeiro de 2021.

[47] J. HUI, **"Object detection: speed and accuracy comparison (Faster R-CNN, R-FCN, SSD, FPN, RetinaNet and YOLOv3)"**, 2018. Disponível em: <https://jonathan-hui.medium.com/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcnssd -and-yolo-5425656ae359>. Acesso: 15 de maio de 2020.

[48] BUSIREDDY, C. **"Is YOLO really better than SSD?"**, 2019. Disponível em: <https://www.linkedin.com/pulse/yolo-really-better-than-ssd-chandrakala-busireddy>. Acesso em: 20 de maio de 2020.

[49] KURDTHONGMEE, W. **"A comparative study of the effectiveness of using popular DNN object detection algorithms for pith detection in cross-sectional images of parawood"**. Disponível em: <https://www.cell.com/heliyon/fulltext/S24058440(20)30325X?_returnURL= https%3A%2F%2Flinkinghub.elsevier.com%2Fretrieve%2Fpii%2FS240584402030325X%3Fshowall%3Dtrue>. Acesso em: 15 de janeiro de 2021.

[50] HERAS, J. **Repositório contido no Github: CLoDSA**. 2021. Disponível em <https://github.com/joheras/CLoDSA>. Acesso em: 22 de outubro de 2021.

[51] ROBOFLOW. Disponível em: < https://roboflow.com/>. Acesso em: 20 de agosto de 2021.

[52] RIOS, A. C. et al. **"Comparison of the YOLOv3 and SSD MobileNet v2 Algorithms for Identifying Objects in Images from an Indoor Robotics Dataset"**, 14th IEEE International Conference on Industry Applications, ISBN 978-1-6654-4118-6. 2021.

[53] BROWNLEE, J. **"ImageDataGenerator how to configure image data augmentation in keras"**. 2019. Disponível em: <https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/>. Acesso em: 20 de julho de 2021.

[54] SANDLER, M. et al. **"MobileNetV2: Inverted Residuals and Linear Bottlenecks"**, The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 4510-4520. 2018. Disponível em: < https://arxiv.org/abs/1801.04381>. Acesso em: 29 de julho de 2021.

ANEXO A - ARTIGO

Comparison of the YOLOv3 and SSD MobileNet v2 Algorithms for Identifying Objects in Images from an Indoor Robotics Dataset

Adriana Carrillo Rios[1], Douglas Henke dos Reis[1], Rodrigo Mattos da Silva[1], Marco Antonio de Souza Leite Cuadros[2], Daniel Fernando Tello Gamarra[1]
[1]Federal University of Santa Maria, Santa Maria, Brazil
[2]Federal Institute of Espírito Santo, Sierra - ES, Brazil
(E-mail: adriana_carrillo15@hotmail.com, douff.reis@gmail.com, rodrigomattos13@gmail.com, marcoantonio@ifes.edu.br, Daniel.gamarra@ufsm.br)

Abstract: The YOLO and SSD algorithms are tools widely used for detecting objects in images or videos. This is due to the speed of detection and good performance in the identification of objects. This article presents a comparison of the YOLOv3 and SSD MobileNet v2 algorithms for identifying objects in images through simulations, the dataset used is an indoor robotics dataset. In order to reach the objective, several training sessions were carried out to analyze the behavior of each model when detecting objects in images. After analyzing the results, a better performance of the YOLOv3 model was observed, although this model takes more time to complete the training for the same number of steps compared to the SSD MobileNet v2 model. It is worth mentioning that this work presents for the first time a comparison between the SSD MobileNet v2 and YOLOv3 algorithms.

Keywords: *Object Recognition, Artificial Intelligence, Computer Vision, Yolov3, SSD, MobileNet v2*.

1. INTRODUCTION

The accelerated growth of information, increases the need of development of efficient techniques that help to extract useful information from the data, and thus, obtain better use of devices and technologies. In this scenario, artificial intelligence (AI) emerges as a tool to understand and solve problems joining large amounts of data in systems, helping in the inference of important characteristics and as a consequence requiring less effort from the systems and devices and less storage and processing capacity.

An important application of artificial intelligence is the recognition of objects in images, which already has several uses, for example, in the area of security and autonomous vehicles. Among the models that help to detect objects, there are two state of the art algorithms the YOLO (*You Only Look Once*) and SSD (*Single Shoot Multibox Detection*) algorithms. The versions chosen for analysis of these algorithms were the YOLOv3 and SSD MobileNet v2 because they need less computational power. In addition, they present a good performance regarding the quality and speed of the detections. It is worth mentioning that this work presents for the first time a comparison between the SSD MobileNet v2 and YOLOv3 algorithms.

This work aims to compare both architectures (YOLO and SSD) for detecting objects in images. So, it is necessary to train the YOLOv3 and SSD MobileNet v2 algorithms, test the training for validation and compare the structures using the metrics for model evaluation.

The algorithms comparison was developed using a dataset created by Dos Reis et al in [1] and [2]. This dataset contains images of the four classes of objects that the models will detect in the images.

The article is divided into five sections. After a brief introduction, the second section describes some related works, the third section describes the computational resources and the methodologies used for the implementation of the models, the fourth section shows the experimental results, and the conclusions are summarized in the last section.

2. RELATED WORK

Yang et al. used yolov3 in [3] to detect and monitor crucial components of the rail system. These components are applied to operating rails and with the detection is possible to avoid failures, helping preventive maintenance. Anhu at el. in [4] proposed a method of detecting vehicles with different types of traffic intersections in real time for traffic flow. They use the YOLOv3 convolutional neural network model as the basis of the study.

Some studies also present the YOLOv3 model as a basis for improving the performance of robots, as presented by Li et al. in [5], a waste detection method based on YOLOv3, which allows the detection of objects in real time and of high precision in dynamic aquatic environments.

Kelly et al. in [6] used the SSD model for detecting windows or intervals of a continuous process operating in a state of steadiness, this application is useful especially when steady-state models are being utilized to optimize the process or plant in real-time. Li et al. in [7] aimed to achieve the accurate and real-time detection of surface defects using the SSD model and as the base network structure MobileNet. Then, a method of detecting surface defects was proposed based on the MobileNet SSD.

Several studies comparing the different models for

detecting objects in images are found in the literature, but not comparing the YOLOv3 and SSD MobileNet v2 architectures.

A very complete comparative analysis of the different models can be found in [8]. They summarize results of individual articles so that they can be viewed together. The aim is to present various points of view in a context, with the idea of better understanding the performance scenario of each model. Furthermore, the authors point out that the models are often implemented in the most varied environments and a direct comparison between them cannot be made.

Bussyreddy in [9] also analyzes and compares several architectures derived from the YOLO and SSD models, but the performance of the SSD MobileNet v2 model is not presented.

Kurdthongmee in [10] analyzes and compares the performance of the YOLOV3 and SSD MobileNet v1 models. The results presented by the author show that the MobileNet SSD model surpasses YOLOv3 in terms of detection rate and average location error. However, YOLOv3 gives better results when standard deviation, maximum and minimum location errors are considered.

3. THEORETICAL BACKGROUND

3.1 YOLOv3

YOLO creates an S×S grid over each input image. Each of the squares formed by the grid is called a cell, each of the cells created provides a fixed number of bounding boxes (anchor boxes) for an object [13]. Usually, YOLO doesn't predict the absolute coordinates of the bounding box's center, so for each bounding box, the model will compensate at the output, the confidence, the values that represent the center of the bounding box in relation to the limits of the grid cell in the input image, the width and height of the box and the conditional class probabilities. The confidence would be the probability that a box contains an object and how accurate is the bounding box, that is

$$\Pr(Object) * IoU_{Predict}^{truth} \quad (1)$$

Here, ground truth bounding box represents the desired output of an algorithm over an input, predict bounding box represents a rectangle region generated from model detector that indicates the location of the object predicted. The Intersection over Union (IoU) is an evaluation metric used to measure the percentage overlapping between the ground-truth bounding box and the predicted bounding box. On the other hand, the conditional class probability is the probability that an object could belong to $class_i$ and is given by

$$\Pr(class_i \mid Object) \quad (2)$$

The sigmoid function, in YOLOv3 predicts the coordinates of the center of the box in relation to the location of the filter application, and the width and height of the box as displacements of the cluster's centroid [11]. The following relations are used by the model to determine the elements mentioned.

$$b_x = \sigma(t_x) + c_x \quad (3)$$

$$b_y = \sigma(t_y) + c_y \quad (4)$$

$$b_w = p_w \cdot e^{t_w} \quad (5)$$

$$b_h = p_h \cdot e^{t_h} \quad (6)$$

The values b_x and b_y are the coordinates of the center; b_w and b_h the width and height of the forecast box. Furthermore, t_x, t_y, t_w, t_h are the values that the network generates, i.e. the network predicts 4 coordinates for each bounding box. For example, a prediction of t_x=1 would shift the box to the right by the width of the anchor box, a prediction of t_x=-1 would shift it to the left by the same amount [12].

The coordinates of the upper left corner of the grid are c_x and c_y. And, the dimensions of the anchors for the box are p_w and p_h (the anchor boxes are defined only by width and height).

YOLOv3 works with 3 different scales, at each scale predicts 3 anchor boxes. The model applies the k-means cluster algorithm to determine the priors (anchors). There are pre-select 9 clusters that are split up evenly across three scales. Each group is assigned to a specific feature map above in detecting objects. In addition, YOLOv3 uses the network Darknet-53 for performing feature extraction. Darknet-53 uses successive 3×3 and 1×1 convolutional layer, but it has some shortcut connections [14].

3.2 SSD MobileNet v2

The SSD model used is based on MobileNet v2, which extracts the characteristics. Then, the image goes through convolutional layers that reduce the size, so that objects can be detected at different scales [15]. The SSD applies small convolutional filters to resource maps to predict scores and class offsets for a fixed set of the standard bounding box. In this case, the grid is created on each feature map and not on the network entry image as performed by YOLOv3.

It is possible to calculate the scales of the standard boxes using the equation described below.

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m-1}(k-1), k \in [1, m] \quad (7)$$

For every k, s_k has five non-square proportions. Different aspect ratios for the default boxes are required, and denote them as a_r. The width and height also are calculated (see relationships in 8) for the default boxes, denoted as w_{ka} and h_{ka}, respectively.

$$a_r \in \left\{1, 2.3.\frac{1}{2}, \frac{1}{3}\right\}, w_k^a = s_k\sqrt{a_r}, h_k^a = \frac{s_k}{\sqrt{a_r}} \quad (8)$$

These width and height values allow us to determine five non-square bounding boxes. Now, for the 1: 1 ratio, the scale is given by equation 9. In this case, we have a square bounding box, but we can also have four boxes, eliminating 3 and 1/3.

$$s_k' = \sqrt{s_k s_{k+1}} \quad (9)$$

The model uses depthwise separable convolutions in its composition, that is, it separates a standard convolution into convolution filters for each input channel [16], and subsequently applies a 1×1 convolution, called a pointwise convolution, to combine the output of the depthwise convolution.

In addition, instead of using absolute coordinates for the box location, the bounding box forecasts are relative to the standard bounding boxes in each cell or location (Δcx, Δcy, w, h), that is, the compensations for the standard box at each location [17].

The improvements brought by the SSD MobileNet v2 are mainly two, the so-called reverse residual networks, responsible for connecting the initial layers of the model directly with the final layers [18]. This allows the model to use the pattern that starts with convolutions of a large number of channels, reduces the dimensions in the intermediate layers and at the end expands, reducing the amount of weights needed to be stored by the model. The other improvement is the use of the ReLU6 activation function, this to code. This was done following the procedure adopted ensure floating point precision [19].

4. EXPERIMENTAL SETUP

4.1 Software Architecture

It is known that the base network of YOLOv3 is Darknet-53, that is written in C and CUDA language. In addition, Darknet53 has support for using NVIDIA GPUs, so it became necessary to download and install Darknet and the NVIDIA CUDA®deep neural network library [cuDNN].

The SSD MobileNet v2 uses TensorFlow and TFSlim algorithms, which are libraries that make machine learning faster and easier. It is important to say that TFSlim is a lightweight library for defining, training and evaluating complex models in TensorFlow.

The TensorFlow tool called Tensorboard can be optionally used, it helps us for a better understanding of what is being done by the model and allows the visualization of any statistics of a neural network.

OpenCV will also be used to help implement both models. It is an open source, multiplatform computer vision library, which is widely used in applications that run in real time, for this reason it is implemented for pattern recognition for both YOLOv3 and SSD MobileNet v2.

4.2 Dataset

The dataset for the implementation of the models is a set of images created by Dos Reis et al in [1] and [2]. This dataset was designed for the YOLOv2 model. It contains 1120 images and labels. The data is divided in four folders according to the type of labeled object (Box, Table, Chair with wheels, Common chair). YOLOv3 do not need to label because the labels are compatible with those used by YOLOv2.

On the other hand, for the SSD algorithm it is not necessary to separate the images in folders, but they must have the same format so that there are no errors during the training. In addition, it is necessary to do the labeling of the images manually, using the LabelImg tool, thus obtaining XML type files. So, the files obtained are manually separated in 'train_labels.xml' and 'test_labels.xml' as done in YOLOv3, in such a way that the training files are exactly the same, but now they are labels for the SSD model.

Figure 1 shows an image that contains all four classes of images of the dataset.

Fig.1.Image with the four objects of the dataset.

4.3 Methodology for YOLOv3

The YOLOv3 model essentially needs six files, 'obj.names', 'obj.data', 'yolov3.cfg', 'test.txt', 'train.txt' and 'darknet53.conv.74'. The first one contains all the objects names to be detected in the order that they were placed in the dataset. The second one contains the number of classes and directories of the files 'train.txt', 'test.txt', 'obj.names' and 'backup'; the latter directory is created in the event that training should be resumed. The file 'yolov3.cfg' corresponds to the model configuration file, it contains the structure of the model, it is where the parameters for training are defined. The files 'train.txt' and 'test.txt' correspond to 80% of the tags for training and 20% of the tags for validation, respectively. And finally, we have the file 'darknet53.conv.74' which corresponds to the pre trained weights for the YOLOv3 model, in this case, it is interesting to use pre-trained weights to decrease considerably the model's training time.

In the configuration of the parameters it is important to consider that the parameter 'max_batches' is directly related to the number of iterations that will be made in the training, the recommended number in [20] is (2000 * number of classes). The number of filters must be defined according to the relation, filters = (classes + 5) * 3.

It was followed the same procedure shown in [16] for training the YOLOv3, synchronizing the local machine with Google Drive to increase the available GPU time. Then, configure Google Colaboratory and finally, place the files mentioned above in a synchronized Google Drive folder.

The training of the algorithm must be stopped, when the average loss (‘avg loss’) does not decrease the value range of 0.05 to 3 (depending on the

complexity of the data, it increases). For model validation, you can choose between the various '.weights' files obtained in the training. For this choice, the highest IoU and mAP (*mean average precision*) should be considered [21].

4.4 Methodology for the SSD MobileNet v2

The SSD MobileNet v2 methodology carry out the training is in the flowchart of Figure 2. In summary, TensorFlow needs '.record' files because they offer a format that allows an efficient reading of the data [22]. These files are obtained through conversions using code. This was done following the procedure adopted in [22] and [23].

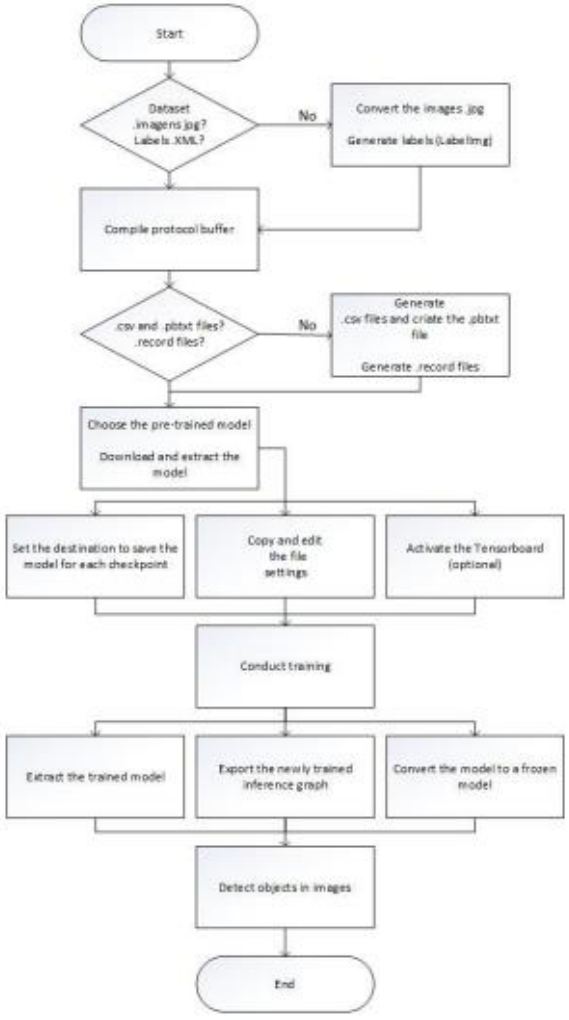


Fig.2 Flowchart showing the implementation of the SSD MobileNet v2 model.

In addition, the '.pbtxt' file is created, which is practically an equivalent of the 'obj.names' file used in YOLOv3. The '.config' file is also required in which all the parameters necessary to perform the model training will be specified to perform the model training. This last file is an equivalent of the 'YOLOv3.cfg' file. In this configuration file, it is important to highlight the parameters 'Num_steps' and the checkpoint paths, which capture the exact value of all parameters used by the model. It is worth noting that the checkpoints do not contain any description of the calculation defined by the model and are only useful when the source code that will use the saved parameter values is available. In addition the paths of the '.pbtxt', '.config', 'train_labels.record' and 'test_labels.record' files are defined. These last two correspond to the files obtained after the conversions mentioned above.

A pre-trained model is also used here, like YOLOv3 do not start training from scratch. Then there is the latest trained model 'last_model_path'. and executing the Python code 'export_inference_graph.py' the model is converted into a frozen model 'frozen_inference_graph.pb' that can be used for inference.

It is important to say that both models were pretrained on the COCO dataset object detection task.

5. RESULTS

5.1 Results of the SSD and the YOLOv3

The indoor robotic dataset created in our laboratory was used as a testbed for the object identification algorithms, Table I shows the results of the average accuracy of each class obtained using the YOLOv3 algorithm with 8000 steps and the default learning rate (0.001). Also, Table II shows the values related to accuracy of each of the four classes, using the SSD model. In this case, using 80000 steps and the initial learning rate, 0.004.

A fact that can be analyzed with the results obtained and which are shown in Tables I and II, is that apparently the SSD MobileNet v2 model was shown to be 'better' than YOLOv3. This is because the values of the average precision of each of the classes are high (all above 83% as shown in Table II). However, the average accuracy of the classes is not enough to evaluate models in which we are using a dataset with disproportionate classes like ours.

TABLE I - Class Results for YOLOv3

Class identifier	Class Name	Average Accuracy
0	Box	74.94%
1	Table	72.33%
2	Chair with wheels	95.59%
3	Common Chair	71;03%

TABLE II– Class Results for SSD MobileNet v2

Class Identifier	Class Name	Average Accuracy
0	Chair with wheels	94.33%
1	Common chair	83.80%
2	Box	93.54%
3	Table	89.33%

5.2 Results of Comparison of SSD and YOLOv3

The results for validation the YOLOv3 and the SSD models with 1000 and 50000 iterations (respectively),

are shown in Table III. The test was performed on an image with the four objects as shown in Figure 1. YOLOv3 is able to detect all objects even though the 'Box' is detected with low certainty. In contrast, the SSD MobileNet v2, even with 50000 iterations, cannot detect the 'Table' class, it can be observed that all the three objects with the exception of the table class that is shown in Figure 3.

Fig.3.Detection SSD MobileNet v2 using 50000 steps.

TABLE III - Class Probability

Class Name	YOLOv3 1000 steps	SSD MobileNet v2 50000 steps
Common Chair	98%	86%
Chair with wheels	79%	41%
Table	75%	-
Box	45%	97%

It was necessary to increase the number of iterations for both models, for YOLOv3 we use the author recommendation in [24], 8000 iterations and thus increase accuracy in the detection of the 'Box' object. For SSD MobileNet v2 there is no recommendation, the option was to increase the number of steps (80000) in such a way that it was possible to detect the 'Table' object. The results obtained were very satisfactory, and are shown in Table IV.

TABLE IV - Class Probability

Class name	YOLOv3 8000 steps	SSD MobileNet v2 80000 steps
Common Chair	100%	100%
Chair with wheels	91%	95%
Table	97%	98%
Box	100%	83%

An important observation is that both models performed well in detecting objects for images with one, two and three objects. YOLOv3 model required 8000 steps and the SSD MobileNet v2 model, 80000 steps.

Fig.4.Detection YOLOv3 'Common chair' and 'Chair with wheels'.

TABLE V - Metrics for models

Metric	YOLOv3	SSD MobileNet v2
Accuracy	72.70%	59.70%
Precision	86%	88.98%
Recall	74%	62.84%
F1-score	79%	73.66%

The Table V shows the metrics obtained for each of the models.

The fact that the accuracy is low does not necessarily mean that the model is bad. It is worth remembering that precision does not adequately evaluate a model with a data set with disproportionate classes, as is our case.

Here, it is more interesting to evaluate the model by the F1-score metric as it is more precise. The F1 score is very good when used in a dataset with disproportionate classes like ours. F1-score 73.66% is not an excellent assessment, but it is an indication that the SSD model performs a good part of the detections with good accuracy, but the performance of the YOLOv3 model is better since the F1-score 79% is closer to 100%.

Comparing the models in terms of the number of iterations and time to perform a training, it can be said that for YOLOv3 it is necessary to perform a much smaller number of iterations to obtain detection results similar to the SSD MobileNet v2 model. For example, for YOLOv3 it was necessary to perform 8000 iterations to be able to detect the four objects with high certainty in the detections. Now, for the SSD model, it was necessary to perform 80000 iterations, 10 times more than those performed in YOLOv3.

Fig.5.Detection SSD MobileNet v2 'Common chair' and 'Chair with wheels'

Another item to be analyzed is the training time, YOLOv3 takes much longer to carry out the training, for the same number of iterations, then the SSD MobileNet v2 model. For example, for YOLOv3 to perform 1000 iterations it takes more than an hour of training while SSD MobileNet v2 needs a maximum of 10 minutes.

6. CONCLUSIONS

This work presents the necessary steps for training the YOLOv3 algorithm and the SSD algorithm using MobileNet v2 to detect objects in an indoor robotics dataset.

Strictly considering the results of the evaluated metrics, it is concluded that the model that managed to

obtain a better performance was YOLOv3.

The accuracy is slightly higher for SSD MobileNet v2. This means that when the model detects an object it is more confident of that detection than YOLOv3. However, YOLOv3 is able to detect more than the SSD because the recall metric for YOLOv3 is 74% compared to 62.84% for the SSD MobileNet v2 model.

Overall, the YOLOv3 model shows better indexes, such as the F1-score of 79%. Whereas for the SSD MobileNet v2 is 73.66%. The F1-score is closer to '1' it means that the model is performing better. Although the YOLOv3 algorithm takes more time for training, it needs a much smaller number of training steps. In addition, YOLOv3 is a much simpler algorithm to be implemented.

It is worth mentioning that this work presents for the first time a comparison between the SSD MobileNet v2 and YOLOv3 algorithms.

In a future work we plan to use the trained models to identify objects and construct a semantic map for a mobile robot application [25].

REFERENCES

[1] D. H. Reis, D. Welfer, M. A. S. L. Cuadros, D. F. T. Gamarra, "Mobile Robot Navigation Using an Object Recognition Software with RGBD Images and the YOLO Algorithm", Applied Artificial Intelligence, JCR, Vol. 33, pp. 1290-1305, 2019.

[2] D. H. Reis, D. Welfer, M. A. S. L. Cuadros, D. F. T. Gamarra, "Object Recognition Software Using RGBD Kinect Images and The YOLO Algorithm for Mobile Robot Navigation", in: The International Conference on Intelligent Systems Design and Applications (ISDA), 2019.

[3] Y. Liu, X. Sun, J. H. L. Pang, "IVSP '20: Proceedings of the 2020 2nd International Conference on Image, Video and Signal Processing", pp. 33-38, 2020. Available in: <https://doi.org/10.1145/3388818.3388827>. Accessed on: 10 January 2021.

[4] R. Anhu, N. Xiaotong, B. Jingjing, "Application of YOLOv3 in road traffic detection". 14th IEEE International Conference on Electronic Measurement & Instruments (ICEMI): pp. 1731-1734, 2019 DOI: 10.1109/ICEMI46757.2019.9101888, 2019.

[5] X. Li, M. Tian, S. Kong, L. Wu, J. Yu, "A modified YOLOv3 detection method for vision-based water surface garbage capture robot". International Journal of Advanced Robotic Systems: Vol. 17, Issue 3, May-June 2020. Available in: <https://doi.org/10.1177/1729881420932715>. Accessed on: 11 January 2021.

[6] J. D. Kelly, J. D. Hedengren, "A steady-state detection (SSD) algorithm to detect non-stationary drifts in processes", Journal of Process Control, Vol. 23, Issue 3, 2013, pp. 326-331, ISSN 0959-1524, Available in: <https://doi.org/10.1016/j.jprocont.2012.12.001>. Accessed on: 12 January 2021.

[7] Y. Li, H. Huang, Q. Xie, L. Yao, Q. Chen, "Research on a Surface Defect Detection Algorithm Based on MobileNet SSD", Applied Sciences, 8(9):1678, 2018. Available in: <https://doi.org/10.3390/app8091678>. Accessed on: 12 January 2021.

[8] J. HUI, "Object detection: speed and accuracy comparison (Faster R-CNN, R-FCN, SSD, FPN, RetinaNet and YOLOv3)", 2018. Available in: <https://jonathan hui.medium.com/objectdetection-speed-and-accuracy comparison-faster-r-cnn-rfcnssd -and-yolo-5425656ae359>. Accessed on: 15 may 2020.

[9] C. Busireddy, "Is YOLO really better than SSD?", 2019. Available in: <https://www.linkedin.com/pulse/yolo-reallybetter-than-ssd-chandrakala-busireddy>. Accessed on: 20 may 2020.

[10] W. Kurdthongmee, "A comparative study of the effectiveness of using popular DNN object detection algorithms for pith detection in cross-sectional images of parawood". Available in: <https://www.cell.com/heliyon/fulltext/S24058440(20)30325X?_returnURL=https%3A%2F%2Flinkinghub.elsevier.com%2Fretrieve%2Fpii%2FS240584402030325%3Fshowall%3Dtrue>. Accessed on: 15 January 2021.

[11] J. Redmon, A. Farhadi, "YOLOv3: An Incremental Improvement", 2018. Available in: <https://arxiv.org/abs/1804.02767>. Accessed: 10 June 2020.

[12] J. Redmon, A. Farhadi, "Yolo9000: Better, faster, stronger. In Computer Vision and Pattern Recognition (CVPR)", 2017 IEEE Conference on, pages 6517–6525. IEEE, 2017.

[13] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, "You only look once: Unified, real-time object detection", in: CVPR, 2016.

[14] J. Hui, "Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3", 2018. Available in: <https://jonathanhui.medium.com/real-time-object-detection with-yolo-yolov2-28b1b93e2088#b8ad>. Accessed on: 13 July 2020.

[15] L. Wei et. al, "SSD: Single Shot MultiBox Detector", 2016. Available in: <https://arxiv.org/pdf/1512.02325.pdf>. Accessed on: 10 June 2020.

[16] C.-Y. Fu, W. Liu, A. Ranga, A. Tyagi, A. C. Berg, "Dssd: Deconvolutional single shot detector", in: arXiv preprint arXiv:1701.06659, 2017.

[17] J. Hui, "SSD object detection: Single Shot Mutibox Detector for real-time processing", 2018. Available in: <https://medium.com/@jonathan_hui/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06>. Accessed on: 5 July 2020.

[18] A. Howard et al., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", 2017. Available in: <https://arxiv.org/pdf/1704.04861.pdf>. Accessed on: 2 august 2020.

[19] M. Sandler et al., "MobileNetV2: Inverted Residuals And Linear Bottlenecks", 2019. Available in: <https://arxiv.org/pdf/1801.04381.pdf>. Accessed on: 15 July 2020.

[20] D. Ibañez, "How to train YOLOv3 Using Darknet on Colab", 2019. Available in: <http://blog.ibanyez.info-/blogs/coding/20190410-run-a-google-colab-notebook totrain-yolov3-using-darknet-in/>. Accessed on: 12 may 2020.

[21] I. Goncharov, "Repository of GitHub: YOLOv3-GoogleColab". Available in: <https://github.com/ivangrov/YOLOv3-GoogleColab>. Accessed on: 5 May 2020.

[22] A. Sinlab, "Weapon_detection_BL_ipynb". Available in: <https://colab.research.google.com/github/AlaaSenjab/Tutorial_Tensorflow_Object_Detection_API_On_Custom_Dataset/ blob/master/weapon_detection_BL.ipynb>. Accessed on: 18 July 2020.

[23] J. Nelson, "Training a TensorFlow MobileNet Object Detection Model with a Custom Dataset". Available in:

<https://blog.roboflow.com/training-a-tensorflow objectdetection-model-with-a-custom-dataset/>. Accessed on: 23 August 2020
[24] Alexey (AlexeyAB), “Darknet, 2017. How to train (to detect your custom objects)”. Available in: <https://github.com/AlexeyAB/darknet>. Accessed on: 20 June 2020.
[25] R. M. Silva, T. R. Garcia, M. A. S. L. Cuadros, D. F. T. Gamarra, “Comparison of a Trajectory Controller based on Fuzzy Logic and Backstepping Using Image Processing for a Mobile a Mobile Robot. In: The International Conference on Intelligent Systems Design and Applications (ISDA), 2019. The International Conference on Intelligent Systems Design and Applications (ISDA), 2019.

ANEXO B – CÓDIGO YOLOv3

```
# Instalação e atualização de bibliotecas
!apt-get update
!apt-get upgrade
!apt-get install -y build-essential
!apt-get install -y cmake git libgtk2.0-dev pkg-config libavcodec-
dev libavformat-dev libswscale-dev
!apt-get install -y libavcodec-dev libavformat-dev libswscale-d
!apt-get install -y libopencv-dev
!apt-get install -y g++-5
!apt-get install -y gcc-5

# Verificar se a GPU está selecionada como acelerador de Hardware
!/usr/local/cuda/bin/nvcc --version
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if not '/device:GPU:0' in device_name:
    print('\nERROR: GPU is not selected as hardware accelerator!')
else:
    print(device_name)

#Descompactar os arquivos da cuDNN desde o Drive, diretamente
para a máquina virtual do CUDA
!tar -xzvf /content/gdrive/MyDrive/YOLOV3/cuDNN/cudnn-11.1-linux-x64-
v8.1.1.tgz -C /usr/local/
!chmod a+r /usr/local/cuda/include/cudnn.h

# Clonar e construir a Darknet
!git clone https://github.com/AlexeyAB/darknet
%cd darknet
!ls
!sed -i 's/OPENCV=0/OPENCV=1/g' Makefile
!sed -i 's/GPU=0/GPU=1/g' Makefile
!make

# Montar o Google Drive
from google.colab import drive
drive.mount('/content/gdrive')

!ls "/content/gdrive/MyDrive/"

#Baixar arquivos
def imShow(path):
  import cv2
  import matplotlib.pyplot as plt
  %matplotlib inline
```

```
  image = cv2.imread(path)
  height, width = image.shape[:2]
  resized_image = cv2.resize(image,(3*width, 3*height), interpolation =
 cv2.INTER_CUBIC)

  fig = plt.gcf()
  fig.set_size_inches(18, 10)
  plt.axis("off")
  #plt.rcParams['figure.figsize'] = [10, 5]
  plt.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
  plt.show()

def upload():
  from google.colab import files
  uploaded = files.upload()
  for name, data in uploaded.items():
    with open(name, 'wb') as f:
      f.write(data)
      print ('saved file', name)
def download(path):
  from google.colab import files
  files.download()

# Começar um novo treinamento
!./darknet detector train /content/gdrive/MyDrive/YOLOV3/obj.data /cont
ent/gdrive/MyDrive/YOLOV3/cfg/yolov3.cfg /content/gdrive/MyDrive/YOLOV3
/darknet53.conv.74 -dont_show -map

# Continar o treinamento
!./darknet detector train /content/gdrive/MyDrive/YOLOV3/obj.data /cont
ent/gdrive/MyDrive/YOLOV3/cfg/yolov3.cfg /content/gdrive/MyDrive/YOLOV3
/backup/yolov3_last.weights -dont_show -map

# Fazer avaliação
!./darknet detector map /content/gdrive/MyDrive/YOLOV3/obj.data /conten
t/gdrive/MyDrive/YOLOV3/yolov3.cfg /content/gdrive/MyDrive/YOLOV3/backu
p/yolov3_last.weights

# Realizando teste
!./darknet detector test /content/gdrive/MyDrive/YOLOV3/obj.data /conte
nt/gdrive/MyDrive/YOLOV3/yolov3.cfg /content/gdrive/MyDrive/YOLOV3/back
up/yolov3_last.weights /content/gdrive/MyDrive/YOLOV3/DetecImg/imagem2.
jpg
imShow('predictions.jpg')
```

ANEXO C – DATA AUGMENTATION I

```
!pip install clodsa
from clodsa.techniques.techniqueFactory import createTechnique
import cv2
import numpy as np
img = cv2.imread("/content/drive/MyDrive/MESAS_AUM/3577.jpg")

t = createTechnique("change_to_hsv",{})
img1 = t.apply(img)
cv2.imwrite("change_to_hsv.jpg",img1)
from google.colab import files
files.download('change_to_hsv.jpg')

t = createTechnique("change_to_lab",{})
img1 = t.apply(img)
cv2.imwrite("change_to_lab.jpg",img1)
from google.colab import files
files.download('change_to_lab.jpg')

t = createTechnique("dropout",{"percentage":0.05})
img1 = t.apply(img)
cv2.imwrite("dropout.jpg",img1)
from google.colab import files
files.download('dropout.jpg')

t = createTechnique("flip",{"flip":0})
img1 = t.apply(img)
cv2.imwrite("flip.jpg",img1)
from google.colab import files
files.download('flip.jpg')

t = createTechnique("gaussian_noise", {"mean" : 0,"sigma":10})
img1 = t.apply(img)
cv2.imwrite("gaussian_noise.jpg",img1)
from google.colab import files
files.download('gaussian_noise.jpg')

t = createTechnique("invert",{})
img1 = t.apply(img)
cv2.imwrite("invert.jpg",img1)
from google.colab import files
files.download('invert.jpg')

t = createTechnique("resize", {"percentage" : 0.4,"method":"INTER_NEARE
ST"})
img1 = t.apply(img)
```

```
cv2.imwrite("resize.jpg",img1)
from google.colab import files
files.download('resize.jpg')

t = createTechnique("salt_and_pepper", {"low" : 0,"up":25})
img1 = t.apply(img)
cv2.imwrite("salt_and_pepper.jpg",img1)
from google.colab import files
files.download('salt_and_pepper.jpg')

t = createTechnique("shift_channel", {"shift":0.2})
img1 = t.apply(img)
cv2.imwrite("shift_channel.jpg",img1)
from google.colab import files
files.download('shift_channel.jpg')

t = createTechnique("shearing", {"a":0.5})
img1 = t.apply(img)
cv2.imwrite("shearing.jpg",img1)
from google.colab import files
files.download('shearing.jpg')

t = createTechnique("translation", {"x":30,"y":30})
img1 = t.apply(img)
cv2.imwrite("translation.jpg",img1)
from google.colab import files
files.download('translation.jpg')

t = createTechnique("sharpen", {})
img1 = t.apply(img)
cv2.imwrite("sharpen.jpg",img1)
from google.colab import files
files.download('rotate.jpg')

t = createTechnique("rotate", {"angle" : 45})
img1 = t.apply(img)
cv2.imwrite("rotate.jpg",img1)
from google.colab import files
files.download('rotate.jpg')

t = createTechnique("rotate", {"angle" : 30})
img1 = t.apply(img)
cv2.imwrite("rotate.jpg",img1)
from google.colab import files
files.download('rotate.jpg')
```

ANEXO D – DATA AUGMENTATION II

```
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
from google.colab import files

import cv2
import os

def load_images_from_folder(images):
  imagens = []
  for filename in os.listdir(images):
    img = cv2.imread(os.path.join(images,filename))
    data = img_to_array(img)
    # expand dimension to one sample
    samples = expand_dims(data, 0)
    # create image data augmentation generator
    #Zoom
    datagen = ImageDataGenerator(zoom_range=[0.7,0.8])
    #Brilho
    #datagen1 = ImageDataGenerator(brightness_range=[0.7,0.8])
    #Translação vertical
    #datagen2 = ImageDataGenerator(height_shift_range=0.5)
    #Translação Horizontal
    #datagen3 = ImageDataGenerator(width_shift_range=[-200,200])
    #Reflexão no eixo Y
    #datagen4 = ImageDataGenerator(horizontal_flip=True)

    # prepare iterator
    it = datagen.flow(samples, batch_size=1)
    #it1 = datagen1.flow(samples, batch_size=1)
    #it2 = datagen2.flow(samples, batch_size=1)
    #it3 = datagen3.flow(samples, batch_size=1)
    #it4 = datagen4.flow(samples, batch_size=1)

    if img is not None:
      imagens.append(img)

    for i in range(2):
      pyplot.plot(110 + 1 + i)
      batch = it.next()
      image = batch[0].astype('uint8')
      pyplot.imshow(image)
      #pyplot.savefig(filename, format='jpg')
      #files.download(filename)
```

```
        pyplot.show()
        #batch = it1.next()
        #image = batch[0].astype('uint8')
        #pyplot.imshow(image)
        #pyplot.show()

        #batch = it2.next()
        #image = batch[0].astype('uint8')
        #pyplot.imshow(image)
        #pyplot.show()

        #batch = it3.next()
        #image = batch[0].astype('uint8')
        #pyplot.imshow(image)
        #pyplot.show()

        #batch = it4.next()
        #image = batch[0].astype('uint8')
        #pyplot.imshow(image)
        #pyplot.show()

  return imagens
load_images_from_folder('/content/drive/MyDrive/data/Cadeiras')
```

ANEXO E – PARTE I CÓDIGO SSD MobileNet v2

```
# Montando o Google Drive
from google.colab import drive
drive.mount('/content/gdrive')

!sudo apt-get update
!sudo apt-get upgrade

!sudo apt-get install tree
!tree /content/gdrive/MyDrive/objeto_deteccion/

# Instalação de Bibliotecas

!apt-get install -qq protobuf-compiler python-pil python-lxml python-tk
!pip install -qq Cython contextlib2 pillow lxml matplotlib pycocotools
!pip install jupyter
!pip install --upgrade tf_slim

#instalar biblioteca
%cd /content/gdrive/MyDrive/objeto_deteccion/models/research
!pip install lvis

from __future__ import division, print_function, absolute_import
import pandas as pd
import numpy as np
import csv
import re
import cv2
import os
import glob
import xml.etree.ElementTree as ET
import io
!pip install tensorflow-gpu==1.15.0
import tensorflow as tf
from PIL import Image
from collections import namedtuple, OrderedDict
import shutil
import urllib.request
import tarfile
from google.colab import files


# Clonar o modelo a ser treinado

%cd /content/gdrive/'My Drive'/objeto_deteccion
!git clone --quiet https://github.com/tensorflow/models.git
```

ANEXO F – PARTE II CÓDIGO SSD MobileNet v2

```
# Conversão de '.xml' para '.csv' e criação do arquivo '.pbtxt'

%cd /content/gdrive/'My Drive'/objeto_deteccion/models/research/object_
detection

def xml_to_csv(path):
  classes_names = []
  xml_list = []

  for xml_file in glob.glob(path + '/*.xml'):
    tree = ET.parse(xml_file)
    root = tree.getroot()
    for member in root.findall('object'):
      classes_names.append(member[0].text)
      value = (root.find('filename').text,
               int(root.find('size')[0].text),
               int(root.find('size')[1].text),
               member[0].text,
               int(member[4][0].text),
               int(member[4][1].text),
               int(member[4][2].text),
               int(member[4][3].text))
      xml_list.append(value)
  column_name = ['filename', 'width', 'height', 'class', 'xmin', 'ymin'
, 'xmax', 'ymax']
  xml_df = pd.DataFrame(xml_list, columns=column_name)
  classes_names = list(set(classes_names))
  classes_names.sort()

  label_map_path = os.path.join('/content/gdrive/My Drive/objeto_detecc
ion/data/', "label_map.pbtxt")
  pbtxt_content = ""

  for i, class_name in enumerate(classes_names):
    pbtxt_content = (
        pbtxt_content
        + "item {{\n    id: {0}\n    name: '{1}'\n}}\n\n".format(i + 1,
 class_name)
    )
  pbtxt_content = pbtxt_content.strip()
  with open(label_map_path, "w") as f:
    f.write(pbtxt_content)

  return xml_df, classes_names
```

```
def main():
  for folder in ['train_labels','test_labels']:
    image_path = '/content/gdrive/My Drive/objeto_deteccion/data/' + fo
lder
    xml_df, classes = xml_to_csv(image_path)
    xml_df.to_csv(('/content/gdrive/My Drive/objeto_deteccion/data/' +
folder + '.csv'), index=None)
    print('Successfully converted xml to csv.')

main()
```

ANEXO G – PARTE III CÓDIGO SSD MobileNet v2

```
%cd /content/gdrive/'My Drive'/objeto_deteccion/models/research

!protoc object_detection/protos/*.proto --python_out=.

os.environ['PYTHONPATH'] += ':/content/gdrive/My Drive/objeto_deteccion
/models/research/:/content/gdrive/My Drive/objeto_deteccion/models/rese
arch/slim/'

!python object_detection/builders/model_builder_test.py
```

ANEXO H – PARTE IV CÓDIGO SSD MobileNet v2

```
from __future__ import division
from __future__ import print_function
from __future__ import absolute_import

import os
import io
import sys
import pandas as pd
import tensorflow as tf

from PIL import Image
from object_detection.utils import dataset_util
from collections import namedtuple, OrderedDict

%cd /content/gdrive/My Drive/objeto_deteccion/models/

DATA_BASE_PATH = '/content/gdrive/My Drive/objeto_deteccion/data/'
image_dir = DATA_BASE_PATH +'images/'

def class_text_to_int(row_label):
  if row_label == 'Cadeira com rodas':
    return 1
  elif row_label == 'Cadeira comum':
    return 2
  elif row_label == 'Caixa':
    return 3
  elif row_label == 'Mesa':
    return 4
  else:
    None


def split(df, group):
  data = namedtuple('data', ['filename', 'object'])
  gb = df.groupby(group)
  return [data(filename, gb.get_group(x)) for filename, x in zip(gb.gro
ups.keys(), gb.groups)]


def create_tf_example(group, path):
  with tf.gfile.GFile(os.path.join(path, '{}'.format(group.filename)),
'rb') as fid:
    encoded_jpg = fid.read()
  encoded_jpg_io = io.BytesIO(encoded_jpg)
  image = Image.open(encoded_jpg_io)
  width, height = image.size
```

```
  filename = group.filename.encode('utf8')
  print(filename)
  image_format = b'jpg'
  xmins = []
  xmaxs = []
  ymins = []
  ymaxs = []
  classes_text = []
  classes = []

  for index, row in group.object.iterrows():
    xmins.append(row['xmin'] / width)
    xmaxs.append(row['xmax'] / width)
    ymins.append(row['ymin'] / height)
    ymaxs.append(row['ymax'] / height)
    classes_text.append(row['class'].encode('utf8'))
    classes.append(class_text_to_int(row['class']))

  tf_example = tf.train.Example(features=tf.train.Features(feature={
      'image/height': dataset_util.int64_feature(height),
      'image/width': dataset_util.int64_feature(width),
      'image/filename': dataset_util.bytes_feature(filename),
      'image/source_id': dataset_util.bytes_feature(filename),
      'image/encoded': dataset_util.bytes_feature(encoded_jpg),
      'image/format': dataset_util.bytes_feature(image_format),
      'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
      'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
      'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
      'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
      'image/object/class/text': dataset_util.bytes_list_feature(classe
s_text),
      'image/object/class/label': dataset_util.int64_list_feature(class
es),
  }))

  return tf_example


def main():
  for csv in ['train_labels', 'test_labels']:
    writer = tf.python_io.TFRecordWriter(DATA_BASE_PATH + csv + '.recor
d')
    path = os.path.join(image_dir)
    examples = pd.read_csv(DATA_BASE_PATH + csv + '.csv')
    grouped = split(examples, 'filename')

    for group in grouped:
```

```
      tf_example = create_tf_example(group, path)
      writer.write(tf_example.SerializeToString())
    writer.close()
    output_path = os.path.join(os.getcwd(), DATA_BASE_PATH + csv + '.re
cord')
    print('Successfully created the TFRecords: {}'.format(DATA_BASE_PAT
H + csv + '.record'))
main()
```

ANEXO I – PARTE V CÓDIGO SSD MobileNet v2

```
MODELS_CONFIG = {
        'ssd_mobilenet_v2': {
        'model_name': 'ssd_mobilenet_v2_coco_2018_03_29',
        'pipeline_file': 'ssd_mobilenet_v2_coco.config',
        'batch_size': 16
    }
}

selected_model = 'ssd_mobilenet_v2'

# Name of the object detection model to use.
MODEL = MODELS_CONFIG[selected_model]['model_name']

# Name of the pipeline file in tensorflow object detection API.
pipeline_file = MODELS_CONFIG[selected_model]['pipeline_file']

# Training batch size fits in Colab's GPU memory for selected model

batch_size = MODELS_CONFIG[selected_model]['batch_size']
```

ANEXO J – PARTE VI CÓDIGO SSD MobileNet v2

```
%cd /content/gdrive/'My Drive'/objeto_deteccion/models/research

import os
import shutil
import glob
import urllib.request
import tarfile

MODEL_FILE = MODEL + '.tar.gz'
DOWNLOAD_BASE = 'http://download.tensorflow.org/models/object_detection
/'
DEST_DIR = '/content/gdrive/My Drive/objeto_deteccion/models/research/p
retrained_model'

if not (os.path.exists(MODEL_FILE)):
  urllib.request.urlretrieve(DOWNLOAD_BASE + MODEL_FILE, MODEL_FILE)

tar = tarfile.open(MODEL_FILE)
tar.extractall()
tar.close()

os.remove(MODEL_FILE)
if (os.path.exists(DEST_DIR)):
  shutil.rmtree(DEST_DIR)
os.rename(MODEL, DEST_DIR)


# Checkpoint

fine_tune_checkpoint = os.path.join(DEST_DIR, "model.ckpt")
fine_tune_checkpoint
```

ANEXO K – PARTE VII CÓDIGO SSD MobileNet v2

```
import os

pipeline_fname = os.path.join('/content/gdrive/My Drive/objeto_deteccio
n/models/research/object_detection/samples/configs', pipeline_file)

assert os.path.isfile(pipeline_fname), '`{}` not exist'.format(pipeline
_fname)

pipeline_fname

%%writefile pipeline_fname
```

ANEXO L – PARTE VIII CÓDIGO SSD MobileNet v2

```
%cd /content/gdrive/'My Drive'/objeto_deteccion/models/research

import os

model_dir = 'training/'

!rm -rf {model_dir}
os.makedirs(model_dir, exist_ok=True)


#Especificações que nos permite utilizar o Tensorboard

!wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
!unzip -o ngrok-stable-linux-amd64.zip


LOG_DIR = model_dir
get_ipython().system_raw(
    'tensorboard --logdir {} --host 0.0.0.0 --port 6006 &'
    .format(LOG_DIR)
)
get_ipython().system_raw('./ngrok http 6006 &')


! curl -s http://localhost:4040/api/tunnels | python3 -c \
    "import sys, json; print(json.load(sys.stdin)['tunnels'][0]['public
_url'])"
```

ANEXO M – PARTE IX CÓDIGO SSD MobileNet v2

```
%%capture

import re
import numpy as np

output_directory = './fine_tuned_model'

lst = os.listdir(model_dir)
lst = [l for l in lst if 'model.ckpt-' in l and '.meta' in l]
steps=np.array([int(re.findall('\d+', l)[0]) for l in lst])
last_model = lst[steps.argmax()].replace('.meta', '')
last_model_path = os.path.join(model_dir, last_model)


!python3 /content/gdrive/'My Drive'/objeto_deteccion/models/research/ob
ject_detection/export_inference_graph.py \
    --input_type=image_tensor \
    --pipeline_config_path='pipeline_fname' \
    --output_directory={output_directory} \
    --trained_checkpoint_prefix={last_model_path}


#Exportando o modelo para um modelo congelado

import os
pb_fname = os.path.join(os.path.abspath(output_directory), "frozen_infe
rence_graph.pb")
assert os.path.isfile(pb_fname), '`{}` not exist'.format(pb_fname)

#Arquivos necessários na máquina local

files.download(DATA_BASE_PATH + 'label_map.pbtxt')

from google.colab import files
files.download(pb_fname)
```

ANEXO N – PARTE X CÓDIGO SSD MobileNet v2

```
# Imagens que desejamos utilizar para fazer as detecções

import os
import glob

PATH_TO_CKPT = pb_fname

PATH_TO_LABELS = "/content/gdrive/My Drive/objeto_deteccion/data/label_
map.pbtxt"

PATH_TO_TEST_IMAGES_DIR =  os.path.join('/content/gdrive/My Drive/objet
o_deteccion/data/', "test")

assert os.path.isfile(pb_fname)
assert os.path.isfile(PATH_TO_LABELS)
TEST_IMAGE_PATHS = glob.glob(os.path.join(PATH_TO_TEST_IMAGES_DIR, "*.*
"))
assert len(TEST_IMAGE_PATHS) > 0, 'No image found in `{}`.'.format(PATH
_TO_TEST_IMAGES_DIR)
print(TEST_IMAGE_PATHS)


#Código que realiza a detecção de objetos em imagens

%cd /content/gdrive/'My Drive'/objeto_deteccion/models/research/object_
detection

import numpy as np
import os
import six.moves.urllib as urllib
import sys
import tarfile
import tensorflow as tf
import zipfile

from collections import defaultdict
from io import StringIO
# This is needed to display the images.
%matplotlib inline
import matplotlib.pyplot as plt
from PIL import Image

# This is needed since the notebook is stored in the object_detection f
older.
sys.path.append("..")
```

```
from object_detection.utils import ops as utils_ops
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as vis_util

detection_graph = tf.Graph()
with detection_graph.as_default():
  od_graph_def = tf.GraphDef()
  with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
    serialized_graph = fid.read()
    od_graph_def.ParseFromString(serialized_graph)
    tf.import_graph_def(od_graph_def, name='')


label_map = label_map_util.load_labelmap(PATH_TO_LABELS)
categories = label_map_util.convert_label_map_to_categories(label_map,
max_num_classes=4, use_display_name=True)
category_index = label_map_util.create_category_index(categories)


def load_image_into_numpy_array(image):
  (im_width, im_height) = image.size
  return np.array(image.getdata()).reshape((im_height, im_width, 3)).as
type(np.uint8)

# Size, in inches, of the output images.
IMAGE_SIZE = (12, 8)


def run_inference_for_single_image(image, graph):
  with graph.as_default():
    with tf.Session() as sess:
      # Get handles to input and output tensors
      ops = tf.get_default_graph().get_operations()
      all_tensor_names = {output.name for op in ops for output in op.ou
tputs}
      tensor_dict = {}
      for key in ['num_detections', 'detection_boxes', 'detection_score
s',
                  'detection_classes', 'detection_masks']:
        tensor_name = key + ':0'
        if tensor_name in all_tensor_names:
          tensor_dict[key] = tf.get_default_graph().get_tensor_by_name(
tensor_name)
      if 'detection_masks' in tensor_dict:
        # The following processing is only for single image
        detection_boxes = tf.squeeze(tensor_dict['detection_boxes'], [0
])
        detection_masks = tf.squeeze(tensor_dict['detection_masks'], [0
])
```

```
        # Reframe is required to translate mask from box coordinates to
 image coordinates and fit the image size.
        real_num_detection = tf.cast(tensor_dict['num_detections'][0],
tf.int32)
        detection_boxes = tf.slice(detection_boxes, [0, 0], [real_num_d
etection, -1])
        detection_masks = tf.slice(detection_masks, [0, 0, 0], [real_nu
m_detection, -1, -1])
        detection_masks_reframed = utils_ops.reframe_box_masks_to_image
_masks(
          detection_masks, detection_boxes, image.shape[0], image.shape
[1])
        detection_masks_reframed = tf.cast(tf.greater(detection_masks_r
eframed, 0.5), tf.uint8)
        # Follow the convention by adding back the batch dimension
        tensor_dict['detection_masks'] = tf.expand_dims(detection_masks
_reframed, 0)
      image_tensor = tf.get_default_graph().get_tensor_by_name('image_t
ensor:0')

      # Run inference
      output_dict = sess.run(tensor_dict, feed_dict={image_tensor: np.e
xpand_dims(image, 0)})

      # all outputs are float32 numpy arrays, so convert types as appro
priate
      output_dict['num_detections'] = int(output_dict['num_detections']
[0])
      output_dict['detection_classes'] = output_dict['detection_classes
'][0].astype(np.uint8)
      output_dict['detection_boxes'] = output_dict['detection_boxes'][0
]
      output_dict['detection_scores'] = output_dict['detection_scores']
[0]
      if 'detection_masks' in output_dict:
        output_dict['detection_masks'] = output_dict['detection_masks']
[0]
  return output_dict


for image_path in TEST_IMAGE_PATHS:
  image = Image.open(image_path)
  print(image_path)
  # the array based representation of the image will be used later in o
rder to prepare the
  # result image with boxes and labels on it.
  image_np = load_image_into_numpy_array(image)
```

```
  # Expand dimensions since the model expects images to have shape: [1,
 None, None, 3]
  image_np_expanded = np.expand_dims(image_np, axis=0)
  # Actual detection.
  output_dict = run_inference_for_single_image(image_np, detection_grap
h)
  # Visualization of the results of a detection.
  vis_util.visualize_boxes_and_labels_on_image_array(
    image_np,
    output_dict['detection_boxes'],
    output_dict['detection_classes'],
    output_dict['detection_scores'],
    category_index,
    instance_masks=output_dict.get('detection_masks'),
    use_normalized_coordinates=True,
    line_thickness=8)
  plt.figure(figsize=IMAGE_SIZE)
  plt.imshow(image_np)
  plt.show()

# Avaliar o modelo parte I

!python /content/gdrive/MyDrive/objeto_deteccion/models/research/object
_detection/inference/infer_detections.py \
  --
input_tfrecord_paths='/content/gdrive/MyDrive/objeto_deteccion/data/tes
t_labels.record' \
  --
output_tfrecord_path=/content/gdrive/MyDrive/objeto_deteccion/data/pred
icao \
  --
inference_graph='/content/gdrive/MyDrive/objeto_deteccion/frozen_infere
nce_graph.pb'

# Avaliar o modelo parte II

!python /content/gdrive/MyDrive/objeto_deteccion/tf_object_detection_cm
-master/confusion_matrix.py \
  --
detections_record=/content/gdrive/MyDrive/objeto_deteccion/data/predica
o \
  --
label_map='/content/gdrive/MyDrive/objeto_deteccion/data/label_map.pbtx
t' \
  --
output_path='/content/gdrive/MyDrive/objeto_deteccion/data/confusion_ma
trix.csv
```