UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

DOUGLAS SOUZA FLÔRES

Detecção de EPIs através de uma CNN

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Luigi Carro

Porto Alegre
2023

AGRADECIMENTOS

Meus profundos agradecimentos a meu pai, Nilton, minha mãe, Regina e a meu irmão Diogo, pessoas sem as quais eu não teria chegado até aqui. Em especial a meus pais por priorizar a educação dos filhos mesmo não tendo tido a mesma oportunidade na vida e por serem exemplos em que eu pude e posso me espelhar.

Também gostaria de agradecer ao pessoal da Vorotech por aceitar participar deste projeto e proporcionar um excelente ambiente de trabalho. Pelas experiências e conhecimentos trocados. Especialmente, agradeço ao Rodrigo por estar sempre disposto a responder minhas dúvidas sobre a área de aplicação do projeto durante todo o percurso.

Finalmente, sou grato ao meu orientador, Prof. Dr. Luigi Carro, pela atenção, orientação e correções de curso durante a jornada. Por prontamente aceitar embarcar neste trabalho e por todo conhecimento passado.

RESUMO

Este trabalho visa realizar a verificação automática do uso de EPIs (Equipamento de Proteção Individual) por parte de trabalhadores de áreas de risco que precisam realizar uma APR (Análise Preliminar de Risco) antes de iniciar a execução do trabalho. Portanto, precisamos detectar quais EPIs uma pessoa está usando, com foco na ausência de uso, através de uma foto. Neste trabalho, o objetivo é aplicar uma CNN (Convolutional Neural Network) ao problema supracitado, avaliando o desempenho e o tempo de execução de cada detecção. Em especial, buscar-se-á obter a resposta em um dispositivo móvel, mesmo que seja necessária a execução na nuvem.

Palavras-chave: CNN. YOLO. YOLOv8. EPI. Detecção de objetos. Computação em nuvem. IoT. Segurança do trabalho.

PPE Detection Using a CNN

ABSTRACT

This work aims to perform an automated verification to check the use of PPEs ("Personal Protective Equipment") by workers in hazardous areas who need to perform an APR ("Preliminary Risk Analysis", free translation) before starting to work. Therefore, we need to detect which PPEs a person is using, focusing on the absence of EPIs, through a photo. In this work, the goal is to employ a CNN (Convolutional Neural Network) to solve the problem previously described, evaluating the performance and the runtime of each detection. Specifically, will be sought to get the answer in a mobile device, even if cloud computing is required.

Keywords: CNN. YOLO. YOLOv8. PPE. Object detection. Cloud computing. IoT. Occupational safety.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AR	Análise de Risco
AP	Average Precision
API	Application Programming Interface
APR	Análise Preliminar de Risco
AWS	Amazon Web Services
CNN	Convolutional Neural Network
CUDA	Compute Unified Device Architecture
ECR	Elastic Container Registry
EPI	Equipamento de Proteção Individual
FaaS	Function as a Service
FN	Falsos Negativos
FP	Falsos Positivos
GPU	Graphics Processing Unit
IoU	Intersection Over Union
mAP	Mean Average Precision
NR	Norma Regulamentadora
PR	Precision-Recall
PWA	Progressive Web App
VN	Verdadeiros Negativos
VP	Verdadeiros Positivos
YOLO	You Only Look Once

SUMÁRIO

1 INTRODUÇÃO

Trabalhos de alto risco geralmente requerem o preenchimento de uma AR ("Análise de Risco") ou APR ("Análise Preliminar de Risco") pelos seus executantes antes que o trabalho seja realizado. A APR tem o intuito de analisar previamente o ambiente e as condições de trabalho, possibilitando o planejamento de medidas de segurança adequadas para controlar e/ou mitigar os riscos, através de um formulário específico de acordo com a natureza do trabalho em questão.

As APRs são exigidas através de Normas Regulamentadoras (NR) que variam de acordo com o tipo de trabalho de alto risco. No âmbito deste trabalho, serão consideradas as NR-10 (Segurança em Instalações e Serviços em Eletricidade) e NR-35 (Trabalho em Altura)[1].

As Normas Regulamentadoras acima citadas não especificam como os formulários de análise de risco devem ser preenchidos e nem mesmo oferecem um modelo de formulário, portanto, essa definição fica a cargo da empresa. Logo, esses formulários podem ser preenchidos em papel, arquivos pdf, planilhas digitais, etc. Porém, hoje já existem diversas aplicações para computador e/ou *smartphones* que visam executar especialmente essa tarefa e agilizar o processo, como as ferramentas APR Digital da Vorotech[2] e APR Digital da Agiliza.ai[3]. Este trabalho, por sua vez, foca nas versões digitais desses formulários e foi desenvolvido dentro da APR Digital da Vorotech.

No caso da Vorotech, se a APR revelar que o serviço se enquadra em uma das NRs, os trabalhadores devem tirar fotos vestindo os equipamentos adequados — especificados pela NR — antes de realizar o serviço e anexá-las ao formulário. Essa medida é requerida para assegurar que o trabalho de risco seja realizado com a devida proteção.

Ainda, todas APRs ficam disponíveis para que o responsável por coordenar os trabalhadores possa verificar se eles tomaram as medidas corretas para realizar o serviço. Entretanto, verificar todas APRs preenchidas em um dia, incluindo todas as fotos, é uma tarefa demorada para um humano e que pode ser acelerada via automação.

---

[1] Disponíveis em: <https://www.gov.br/trabalho-e-previdencia/pt-br/composicao/orgaos-especificos/secretaria-de-trabalho/inspecao/seguranca-e-saude-no-trabalho/ctpp-nrs/normas-regulamentadoras-nrs>. Acesso em: 24 ago. 2023.

[2] Disponível em: <https://www.voro.tech/>. Acesso em: 25 ago. 2023.

[3] Disponível em: <https://agiliza.ai/aprdigital>. Acesso em: 24 ago. 2023.

Fazer a verificação do corretismo das respostas dadas à uma determinada APR é trivial através de métodos computacionais convencionais. Todavia, a análise das fotos tiradas comprovando o uso de EPIs ainda requer total atenção de um humano e, na prática, são usadas somente para documentação do serviço. Caso essas fotos sejam verificadas com frequência — seja por humano ou máquina —, os responsáveis técnicos podem identificar os trabalhadores que estão negligenciando o uso de determinados EPIs — por ignorância ou por inadimplência consciente — e tomar as devidas providências para evitar acidentes graves no futuro.

Esse problema foi identificado ao trabalhar no aplicativo de preenchimento digital de APRs da empresa Vorotech, onde todas as etapas do preenchimento de uma APR são validadas e os erros, caso existam, são notificados aos técnicos de segurança responsáveis. Deste modo, surgiu a ideia de uma solução automatizada de análise das fotos contidas nas APRs para, em um primeiro momento, auxiliar os técnicos responsáveis pela segurança do trabalho. Esta ideia foi levada à empresa Vorotech, que demonstrou interesse em implementá-la no seu próprio negócio. Isso foi um dos catalisadores para levar este projeto adiante.

Assim sendo, pode-se concluir que a demanda pela automatização do processo de verificação de imagens de uso de EPIs é patente. Este trabalho tem como escopo mapear este problema de fiscalização como um problema de visão computacional para a detecção de objetos (EPIs) e adequar a solução para as condições normais de trabalho onde este recurso será utilizado.

2 HISTÓRICO DE DETECÇÃO DE OBJETOS

O problema de detecção de objetos vem sendo abordado há anos através de diversas soluções de *machine learning*.

Em 2001, o método de Haar Cascades foi proposto por VIOLA, P.; JONES (2001), capaz de operar em imagens de 384 por 288 pixel detectando rostos a 15 quadros por segundo em um processador Intel Pentium III. Segundo VIOLA, P.; JONES (2001), quando aplicado à detecção de rostos, esse método consegue atingir menos de 1% de falsos negativos e 40% falsos positivos usando um classificador construído a partir de duas características de Harr.

Mesmo que a acurácia desse método seja menor que a de métodos mais modernos, sua capacidade de ser executado em hardwares modestos o torna um meio viável até hoje quando se é necessário fazer detecção de objetos em dispositivos de baixo poder de processamento. Por exemplo, Haar Cascades foi utilizado para detectar ovos do mosquito *Aegis Aegypti* por ROCHA (2018).

Em 2005, DALAL; TRIGGS (2005) demonstraram que grades de descritores de *Histogram of Oriented Gradient* (HOG) eram mais eficazes que os *feature sets* para detecção humana existentes até aquela época.

Em seguida, houveram diversos métodos de detecção de objetos baseados em *deep learning* utilizando CNNs ("Convolutional Neural Network"). Dentre eles, podemos destacar os seguintes:

- R-CNN (GIRSHICK ET AL., 2014);
- SPP-net (HE ET AL., 2015);
- Fast R-CNN (GIRSHICK, 2015);
- Faster R-CNN (REN ET AL., 2016).

A grande desvantagem das abordagens com CNNs era o tempo de execução. Por exemplo, a R-CNN leva em torno de 50 segundos para analisar uma imagem, já a Faster R-CNN conseguiu reduzir esse tempo para 0,2 segundos, possibilitando detecção de objetos em vídeos de até 5 quadros por segundo. Embora o avanço seja considerável, mesmo a Faster R-CNN ainda não é capaz de ser executada em vídeos comuns — 24 quadros por segundo ou mais — em tempo real.

Em 2016, o método YOLO ("You Only Look Once") foi proposto por REDMON ET AL. (2016), reduzindo o tempo de processamento de imagem para menos de 7 ms. Assim, a partir da concepção da primeira versão da YOLO, detecção de objetos foi capaz de ser

executada a até 155 quadros por segundo. Contudo, esse resultado foi atingido em uma GPU Titan X, ou seja, hardware topo de linha na época.

Em 2023, a YOLOv8, desenvolvida pela Ultralytics, é capaz de fazer inferência em uma imagem em menos de 1ms utilizando uma GPU A100 TensorRT. Até o momento da publicação deste trabalho, ainda não há um artigo publicado sobre a versão 8 da YOLO, mas seus resultados podem ser visualizados em no repositório da Ultralytics[1] e a sua documentação está disponível online[2].

Tendo em vista o histórico discorrido e considerando a preferência por um baixo tempo de inferência, melhorando a experiência do usuário final e/ou reduzindo os custos de operação do projeto, a melhor opção parece ser a YOLOv8.

[1] Disponível em: <https://github.com/ultralytics/ultralytics>. Acesso em: 24 ago. 2023.

[2] Disponível em: <https://docs.ultralytics.com>. Acesso em: 24 ago. 2023.

3 ESPECIFICAÇÃO DO PROJETO

Este Capítulo descreve o ambiente onde o projeto será aplicado e como ele será implementado atendendo aos requisitos.

3.1 Casos de Uso e Condições de Contorno

Antes de tratar da solução do problema, é preciso delimitar as condições materiais e os requisitos que a solução deve atender.

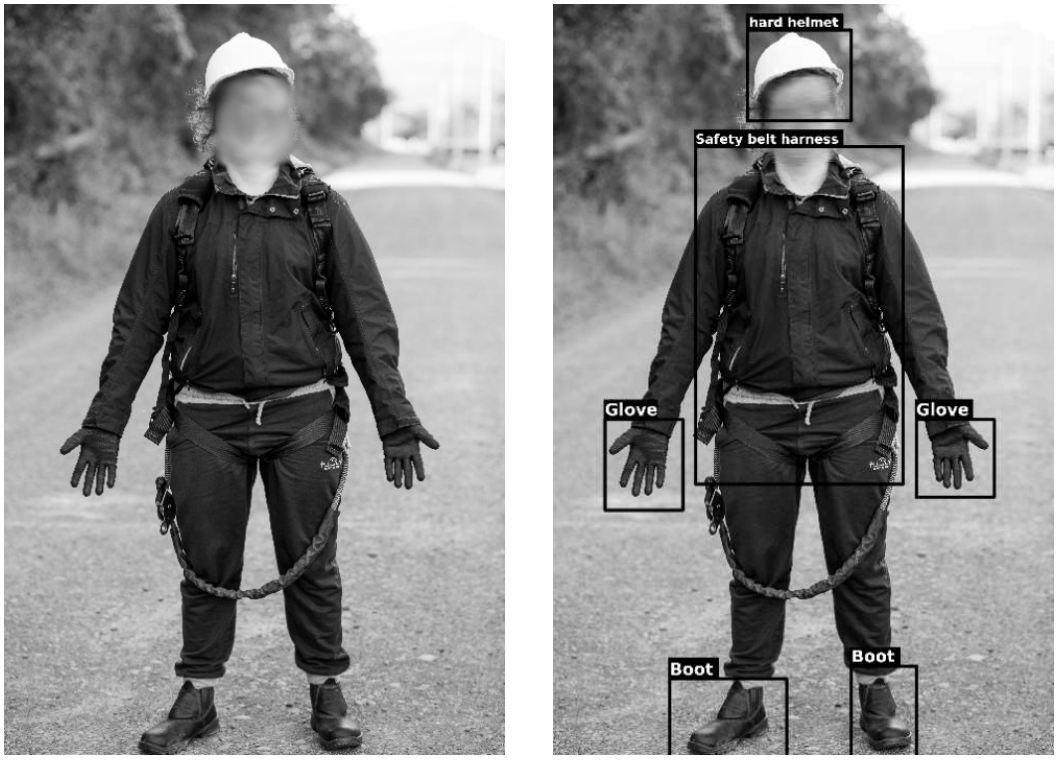
- Os trabalhadores preenchem as APRs nos seus *smartphones* e, caso necessário, tiram as fotos comprovando o uso dos EPIs adequados;
- Como os trabalhadores alvo deste projeto são operadores de instalação e manutenção de telecomunicações, as APRs geralmente são preenchidas em locais remotos que, às vezes, não possuem conexão à internet;
- Execução da detecção de objetos em tempo real não é necessária;
- Após o processamento da imagem, o resultado deve ficar salvo em um banco de dados para acesso posterior.

Também é válido notar que os celulares utilizados para preencher as APRs podem ser de graus variados de poder de processamento, variando desde os aparelhos mais simples até os mais avançados. Consequentemente, a solução deve ser eficiente e capaz de atender o maior número de aparelhos possível.

As fotos-alvo já possuem determinadas regras que são seguidas e devem facilitar o treinamento da YOLO. As fotos devem ser tiradas de corpo inteiro a uma certa distância da câmera. A Figura 3.1a mostra uma foto de referência, enquanto a Figura 3.1b mostra a mesma foto após a detecção de objetos ser executada de forma correta.

Também há um limite de resolução para as imagens que serão fornecidas ao modelo. As fotos capturadas através do aplicativo são redimensionadas para uma largura máxima de 800 pixels e altura máxima de 600 pixels, sempre respeitando a proporção entre largura e altura original da foto. Além disso, as fotos são convertidas para JPG utilizando nível de qualidade de 80% (1% a 100%, onde 100% é uma imagem quase sem compressão). Este processamento é feito sobre a foto para padronizá-las e reduzir o tamanho dos arquivos, diminuindo o tempo de envio das imagens para os servidores e o espaço necessário para as armazenar. Logo, este projeto precisa se adequar a estas condições de qualidade de imagem.

Figura 3.1 – Exemplo de detecção de EPIs almejada no aplicativo APR Digital.



(a) Original (b) Exemplo de Detecção de EPIs

Fonte: O autor, Vorotech. Legenda: À esquerda, um exemplo de foto tirada durante o preenchimento de uma AR. À direita, um exemplo de como este projeto visa mostrar os EPIs detectados.

3.2 Especificação

Para atender aos requisitos supracitados, adotou-se uma abordagem de computação em nuvem para solução do problema. Assim, é possível oferecer suporte a qualquer dispositivo com acesso à internet.

Contudo, como o projeto deve ser operacional inclusive em situações sem acesso à internet, poderá ser implementado um sistema de *caching* que entrará em ação quando o dispositivo estiver *offline*. As requisições armazenadas em cache então devem ser enviadas para nuvem, onde devem ser devidamente processadas e salvas para verificação posterior.

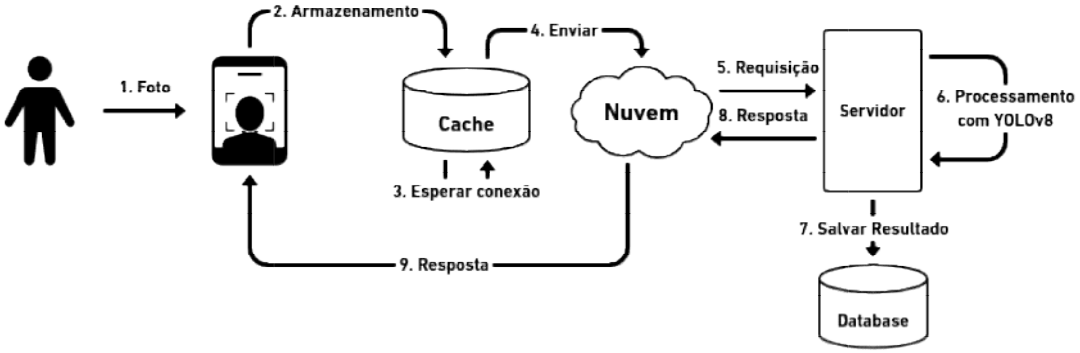
Para efetivamente realizar a detecção dos EPIs, usou-se a YOLOv8, pois até então ela é a versão mais performática.

Conforme a Figura 3.2 demostra, o sistema de detecção de objetos funcionará da seguinte maneira:

1. Antes de concluir o preenchimento da APR, o trabalhador tira uma foto de cada um dos seus colegas e depois tem sua foto tirada por um deles (no mesmo dispositivo);
2. As fotos são então salvas no próprio dispositivo em cache;

3. O aplicativo faz *pooling* por conexão e quando a consegue, avança para o passo 4;
4. As fotos são enviadas para a nuvem em uma requisição de processamento;
5. A requisição chega até o servidor responsável por processar as fotos;
6. As imagens são processadas pela YOLOv8;
7. O resultado é salvo em uma base de dados;
8. O servidor responde à requisição do *smartphone*, avisando sobre o resultado obtido e enviando as respectivas *bounding boxes*.

Figura 3.2 – Arquitetura do sistema de detecção de EPIs



Fonte: O autor.

Entretanto, no escopo deste trabalho assumiu-se que o dispositivo sempre estará online e, portanto, as etapas de *caching* e *pooling* não foram implementadas até o presente momento.

O aplicativo APR Digital da Vorotech é implementado como um PWA utilizando React. Este trabalho foi desenvolvido em uma *branch* do aplicativo. Assim, futuramente a integração do projeto poderá ser realizada de forma simples.

3.2.1 Execução em Nuvem

Este trabalho executou a YOLO em nuvem utilizando o serviço AWS Lambda[1]. Ele é um serviço de *serverless computing*[2] FaaS[3].

---

[1] Disponível em: <https://aws.amazon.com/pt/lambda/>. Acesso em: 24 ago. 2023.

[2] É um modelo de computação em nuvem onde um provedor aloca recursos *on demand* e gerencia toda estrutura de servidores para o cliente. O termo *serverless* se refere ao fato de o cliente não precisar gerir nenhum servidor, mas o serviço de facto é provido através de servidores.

[3] Abreviação de *Function as a Service*. É um modelo de serviço onde um provedor de *serverless computing* fornece execução de código sem armazenamento local de dados.

Dentre suas vantagens, o serviço possui um plano gratuito durante o primeiro ano de uso e mesmo sua versão paga cobra apenas alguns centavos de dólar a cada 1.000 requisições. Além disso, é uma solução escalável, permitindo alocação dinâmica de recursos conforme o escopo do projeto aumenta.

3.2.2 Classes de Objetos

O modelo de detecção de objetos desenvolvido deve detectar o uso adequado dos seguintes EPIs:

- Capacetes de proteção (*hard_helmet*);
- Calçados fechados (*boot*);
- Luvas (*glove*);
- Cinto de segurança tipo paraquedista (*safety_belt_harness*);

A Figura 3.1a mostra um trabalhador vestindo pelo menos um exemplar de cada classe. Todos os EPIs especificados devem ser detectados apenas no caso do seu uso correto. Se qualquer um dos EPIs for detectado enquanto não estiver sendo vestido por alguém, será considerado um erro.

3.3 Ambiente de Desenvolvimento

Para realizar a maior parte do trabalho, utilizou-se um computador desktop com processador AMD Ryzen 5 3600, 16GB de memória RAM DDR4, sem uma GPU CUDA, sistema operacional Ubuntu, utilizando as linguagens Python e TypeScript.

Para gerenciar o banco de imagens e as versões do *dataset* empregadas no treinamento, validação e teste do modelo, utilizou-se a ferramenta Roboflow[1]. Ela permite manter um histórico de versões de um mesmo *dataset*, fazer rotulação de imagens e aplicar métodos de pré-processamento e pós-processamento durante a criação de uma nova versão do *dataset*. Tudo isso em nuvem, além de possuir uma API em Python que permite o download dos *datasets* direto no código, o que facilita a execução das rotinas de treinamento em ambientes remotos. A Roboflow é utilizada por grandes empresas como Intel e Cardinal Health, além de possuir mais de 200 milhões de imagens e mais de 200 mil *datasets* abertos para uso em visão computacional.

[1] Disponível em <https://roboflow.com/>. Acesso em: 24 ago. 2023.

Os treinamentos desempenhados neste trabalho foram realizados através da plataforma Google Colab[1], que oferece o uso gratuito de uma GPU Testa T4 com algumas restrições de tempo de alocação diário.

Assim como já citado na seção 3.2, a YOLOv8 foi utilizada como modelo de detecção de objetos neste trabalho, mais especificamente a implementação da Ultralytics sob a licença AGPL-3.0. Caso o trabalho entre em produção (futuro objetivo) será necessário o uso da licença Entreprise.

Como mencionado na subseção 3.2.1, o serviço de detecção de objetos foi hospedado na AWS Lambda. Além disso, utilizamos o serviço AWS ECR[2] para armazenar a imagem Docker — software de código aberto usado para criação de containers[3] — que é utilizada pela função lambda, conforme detalhado posteriormente neste trabalho.

Finalmente, utilizou-se dois smartphones (LG K10 2017 e SAMSUMG Galaxy A03) para rodar o aplicativo desenvolvido e medir o tempo de resposta do serviço de inferência. Ambos aparelhos possuem processadores octa-core, porém o modelo da LG possui 1 GB de memória RAM enquanto o modelo da SAMSUMG possui 4 GB. Além disso, o aparelho da LG possui uma câmera de 13 MP. Já o aparelho SAMSUMG tem uma câmera de 48 MP.

---

[1] Disponível em <https://colab.research.google.com/>. Acesso em: 24 ago. 2023.

[2] Disponível em <https://aws.amazon.com/pt/ecr/>. Acesso em: 24 ago. 2023.

[3] De forma simples, um container é um sistema capaz de ser rodado dentro de outro sistema. Ele é semelhante a uma máquina virtual, com o diferencial de não ser uma emulação que executa sobre um sistema real, mas sim a criação de um *namespace* independente dentro de um sistema operacional já existente.

4 DESENVOLVIMENTO DO MODELO DE DETECÇÃO DE OBJETOS

Como mencionado nas seções 3.2 e 3.3, a versão 8 da YOLO desenvolvida pela Ultralytics foi usada neste projeto. Contudo, também foi preciso escolher qual modelo de detecção pré-treinado utilizar.

Tabela 4.1 – Desempenho dos modelos pré-treinados da YOLOv8 sobre o dataset COCO val2017

Modelo	*Tamanho (pixels)*	*mAP 50-95*	*Velocidade (ms)*
YOLOv8n	640	37,3	80,4
YOLOv8s	640	44,9	128,4
YOLOv8m	640	50,2	234,7
YOLOv8l	640	52,9	375,2
YOLOv8x	640	53,9	479,1

Fonte: <https://github.com/ultralytics/ultralytics>.

Na Tabela 4.1, o tempo de execução demonstrado em milissegundos (na coluna "Velocidade") foi obtido em uma instância Amazon EC2 P4d, utilizando a CPU ONNX. Não há como saber *a priori* qual dos modelos oferece o melhor balanço entre mAP e velocidade sem saber como o poder de processamento de uma função lambda se compara à máquina usada nos testes da Ultralytics. Então foi adotada uma metodologia *a posteriori*, escolhendo-se um dos modelos, avaliando se ele se encaixou nos pré-requisitos e, a partir disso, estimar qual poderia se sair melhor.

Para guiar o chute inicial, a Tabela 4.2 mostra uma relação de custo-benefício dos modelos da Tabela 4.1 levando em consideração quando milissegundos cada modelo leva para "obter" 1mAP. Notavelmente, o custo-benefício diminui drasticamente conforme os modelos ficam mais exatos e mais custosos. A YOLOv8n apresenta o melhor custo-benefício nestes termos, porém, como ela mostra mAP menor que 40, optou-se por utilizar a YOLOv8s.

Tabela 4.2 – Relação velocidade-mAP da YOLOv8 sobre o dataset COCO val2017

Modelo	*Velocidade/mAP (ms/mAP)*
YOLOv8n	2,155495979
YOLOv8s	2,859688196
YOLOv8m	4,675298805
YOLOv8l	7,092627599
YOLOv8x	8,888682746

Fonte: O autor.

4.1 Método de Construção dos Datasets

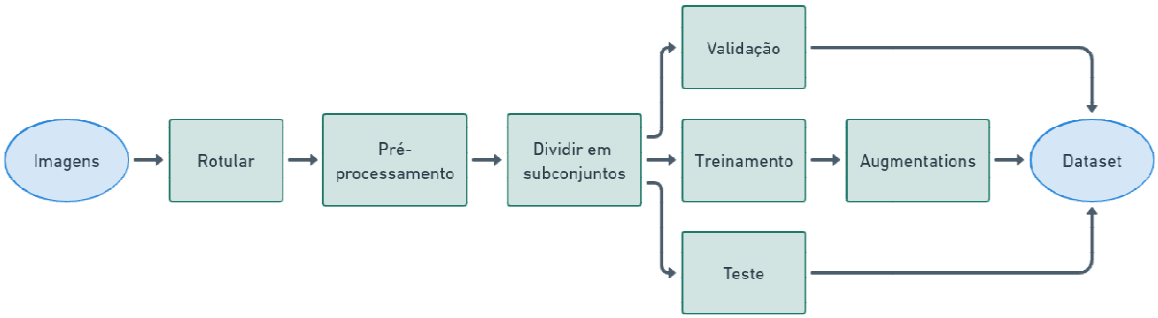
Primeiro, é preciso encontrar as imagens adequadas a compor o *dataset*. Essas imagens devem ser rotuladas de acordo com as classes descritas na subseção 3.2.2. Depois, as imagens precisam ser separadas em subconjuntos de treinamento, validação e teste de forma

estratificada, ou seja, mantendo a mesma proporção de amostras de cada classe através de todos os subconjuntos. Por padrão, os subconjuntos criados neste trabalho seguem a proporção 70%, 20%, 10%, respectivamente. Em seguida, as imagens devem ser submetidas a uma etapa de pré-processamento para uniformizá-las e alinhá-las ao modelo que será treinado. Antes de gerar o *dataset* final, as imagens pré-processadas passam pela etapa de *augmentation,* para criar mais amostras a partir das originais para serem usadas durante o treinamento. A Figura 4.1 mostra o pipeline de criação de um dataset.

Os filtros de pré-processamento são aplicados em todas as imagens do *dataset*. Na descrição de cada versão do *dataset*, são especificados quais foram os filtros de pré-processamento utilizados. Alguns exemplos de operações de pré-processamento são: redimensionamento, filtro de escala de cinzas, ajuste de contraste, recorte da imagem etc.

O processo de *augmentation* consiste em modificar as imagens a fim de criar mais amostras para treinamento. É importante ressaltar que esse processo é aplicado somente ao subconjunto de treinamento para não comprometer as imagens utilizadas na validação e no teste do modelo. Alguns exemplos de operações amplamente utilizadas para aumento do subconjunto de treinamento são: espelhamento vertical ou horizontal, rotação de 90º, recorte da imagem, blur, adição de ruído, variação de saturação, variação de brilho etc. Assim como no caso do pré-processamento, é especificado em cada *dataset* criado quais foram os métodos de *augmentation* utilizados.

Figura 4.1 – Pipeline de criação de um *dataset*



Fonte: O autor.

4.2 Métricas de Avaliação

É de suma importância definir quais métricas serão utilizadas para avaliar se o modelo treinado é bom ou ruim. Dentre elas existem a acurácia e a taxa de erros, demonstradas nas Equações 4.1 e 4.2 (FACELI, K., 2011, p. 153), respectivamente.

$$acc(f) = \frac{VP + VN}{VP + VN + FP + FN} \tag{4.1}$$

$$err(f) = 1 - acc(f) \tag{4.2}$$

Entretanto, a acurácia e a taxa de erros não são adequadas para avaliar o desempenho em *datasets* desbalanceados (FACELI, K., 2011). No Gráfico 4.1 é mostrado o número de instâncias de cada classe por imagem do *dataset* inicial, que é descrito na seção 4.3. Especialmente, as classes *boot* e *safety_belt_harness* possuem um alto desbalanço em favor de exemplos negativos.

Portanto, precisou-se utilizar métricas de avaliação menos abrangentes, como a precisão e a sensibilidade, Equações 4.3 e 4.4 (FACELI, K., 2011, p. 153), respectivamente.

$$prec(f) = \frac{VP}{VP + FP} \tag{4.3}$$

$$rev(f) = \frac{VP}{VP + FN} \tag{4.4}$$

Gráfico 4.1 – Número de instâncias por imagem para cada classe (dataset inicial)

Fonte: O autor. Legenda: Cada barra demonstra o número de instâncias que cada classe possui dividido pelo número total de imagens no dataset.

Já que o objetivo primário deste trabalho é detectar a ausência do uso de EPIs, este projeto é classificado como "orientado a precisão".

4.2.1 Especificidade e Taxa de Falsos Positivos

Outras duas métricas que serão levadas em consideração são a especificidade e a taxa de falsos positivos, Equações 4.5 e 4.6 (FACELI, K., 2011, p. 153), respectivamente.

$$esp(f) = \frac{VN}{VN + FP} \tag{4.5}$$

$$TFP(f) = 1 - spec(f) \tag{4.6}$$

Levando em conta a aplicação deste projeto, ele é “orientado a especificidade”.

4.2.2 Mean Average Precision (mAP)

A *Mean Average Precision* é uma métrica amplamente utilizada para avaliar modelos de detecção de objetos. Ela representa a média aritmética da *Average Precision* (AP) de cada classe, por isso, ela é classificada como uma macro média. Para calcular a AP de uma classe, deve-se determinar o *threshold* de IoU que será levado em consideração. A AP então é simplesmente a área sob a curva PR.

Neste trabalho, utilizou-se um *threshold* de 0.5 para avaliar todos os modelos. Embora muito utilizada, pelos motivos discorridos nesta seção, usou-se a mAP como estimativa inicial de desempenho para depois consultar a precisão e a especificidade de modo a dar o veredito.

4.3 Dataset Inicial

Idealmente, treinar-se-ia o modelo usando a base de fotos capturadas pelo aplicativo APR Digital ao decorrer de todo seu tempo de operação. Mas devido a possíveis complicações jurídicas envolvendo direitos de imagem que poderiam surgir com tal uso das fotos, a Vorotech preferiu não disponibilizar as imagens supracitadas.

Entretanto, a Vorotech forneceu um conjunto de 425 imagens de autoria da própria empresa. Muitas dessas fotos eram duplicatas exatas de outra foto do conjunto e, por conseguinte, foram removidas. Após esse filtro, o conjunto foi reduzido para apenas 190 imagens.

Na etapa de pré-processamento, as imagens foram redimensionadas para 640x640 pixels, esticando a imagem quando necessário para atingir essas dimensões. Na etapa de *augmentation* empregou-se apenas um espelhamento horizontal. Vale notar que isso alterou a proporção de imagens distribuídas entre os subconjuntos de treinamento, validação e teste para 81%, 13% e 6%, respectivamente.

4.3.1 Resultados do Treinamento

O conjunto de imagens descrito foi usado para treinar o modelo YOLOv8s por 25, 30, 35 e 40 gerações. Através da Tabela 4.3, comparando os modelos através de um p-valor de

0.05, poder-se-ia concluir que o modelo de 30 gerações é efetivamente melhor que o modelo de 25 gerações. Porém, devido à fragilidade estatística do método do p-valor, a hipótese nula — nenhum dos modelos é melhor que o outro — será considerada como verdadeira por enquanto.

Tabela 4.3 – Mean average precision dos modelos treinados com o dataset inicial

Número de Gerações	*mAP@50*
25	0,917
30	0,967
35	0,956
40	0,924

Fonte: O autor.

Através da matriz de confusão mostrada na Figura 4.2, nota-se que o modelo de 30 gerações apresentou apenas 4 falsos negativos. Além disso, o modelo não apresentou nenhum falso positivo. Logo, o modelo possui precisão 1 e sensibilidade 0,939.

Figura 4.2 – Matriz de confusão para o modelo de 30 gerações

Matriz de Confusão						
		Verdade				
		boot	glove	hard_helmet	safety_belt_harness	background
Predição	boot	3	0	0	0	0
	glove	0	12	0	0	0
	hard_helmet	0	0	21	0	0
	safety_belt_harness	0	0	0	6	0
	background	0	3	1	0	0

Fonte: O autor.

As métricas de desempenho são excelentes, mas ainda há alguns pontos a se considerar. O conjunto de imagens utilizado para treinamento é pequeno, conta com majoritariamente três pessoas vestindo as mesmas respectivas roupas e EPIs. Combinando isso com o desempenho aparentemente quase-perfeito, temos um forte indicativo de que o modelo sofre de sobre-ajuste (FACELI, K., 2011).

Em vista disso, foi preciso expandir o conjunto de imagens.

4.4 Incrementando o Dataset

Através da plataforma Roboflow descrita na seção 3.3, 3.000 imagens adicionais foram incorporadas ao conjunto de imagens anterior. Todas elas precisaram ser individualmente avaliadas e rotuladas de acordo com as classes deste projeto. Os subconjuntos de treinamento, validação e teste aqui também foram divididos na proporção 70%, 20%, 10%, respectivamente.

Na etapa de pré-processamento, as imagens apenas foram redimensionadas para 640x640 pixels, esticando-as quando necessário para preencher essa resolução. Na etapa de *augmentation* foi usado somente espelhamento horizontal. Assim, o subconjunto de treinamento foi expandido de 2.233 imagens para 3.900 (um aumento de 75%).

4.4.1 Resultados do treinamento

O modelo YOLOv8s foi treinado por 25, 30, 35 e 40 gerações, agora utilizando a segunda versão do *dataset*, especificada nesta seção. Embora o modelo de 35 gerações tenha apresentado os melhores resultados, eles não passam por um teste de p-valor de 0.05.

De acordo com a matriz de confusão da Figura 4.3, o modelo apresentou precisão de 0,876, o que já é suficiente para a aplicação deste trabalho. Porém, a sensibilidade atingida foi de 0,643 no geral. Analisando a matriz de confusão da Figura 4.3, vê-se que mais da metade dos membros da classe *glove* não foram detectados.

Tabela 4.4 – Mean average precision dos modelos treinados

Número de Gerações	*mAP@50*
25	0,774
30	0,772
35	0,777
40	0,773

Fonte: O autor.

Figura 4.3 – Matriz de confusão do modelo de 35 gerações

Matriz de Confusão						
		Verdade				
		boot	glove	hard_helmet	safety_belt_harness	background
Predição	boot	76	0	0	0	14
	glove	0	53	0	0	14
	hard_helmet	1	0	610	1	12
	safety_belt_harness	0	0	0	33	4
	background	54	60	61	21	0

Fonte: O autor.

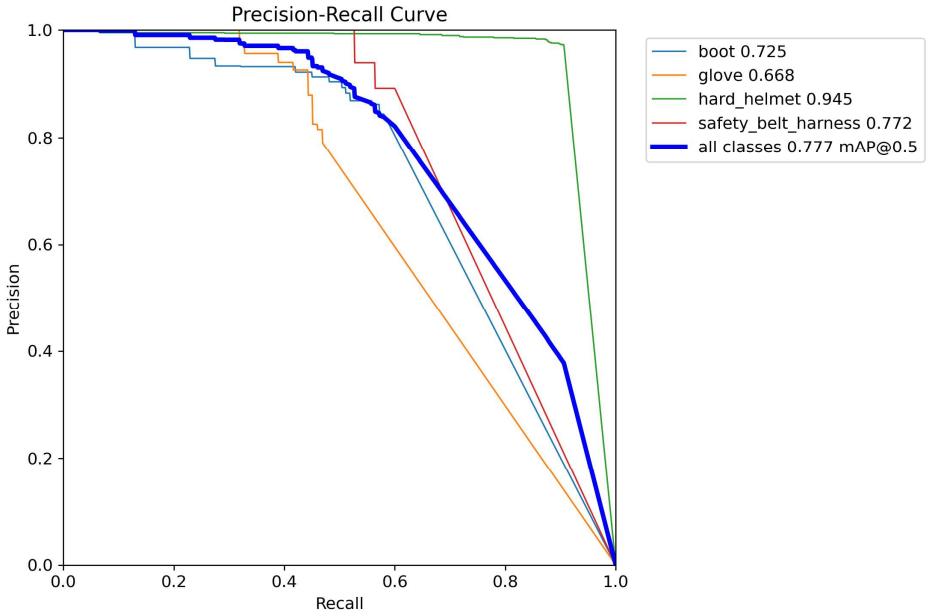
É interessante notar, no Gráfico 4.2, que o desempenho para a classe *hard_helmet* (capacete de proteção) foi muito superior às demais. O motivo provavelmente é o desbalanceamento do *dataset* que contém mais amostras da classe *hard_helmet*.

Adicionalmente, o desempenho para a classe *glove* ficou muito atrás do desempenho para as classes *safety_belt_harness* e *boot*.

É notório que a YOLO possui uma dificuldade maior de detectar objetos pequenos (REDMON, J., 2016) e, coincidentemente, os menores objetos (*glove* e *boot*) tiveram os

piores desempenhos, mesmo havendo mais instâncias de cada um do que de *safety_belt_harness*.

Gráfico 4.2 – Curva PR do modelo de 35 gerações

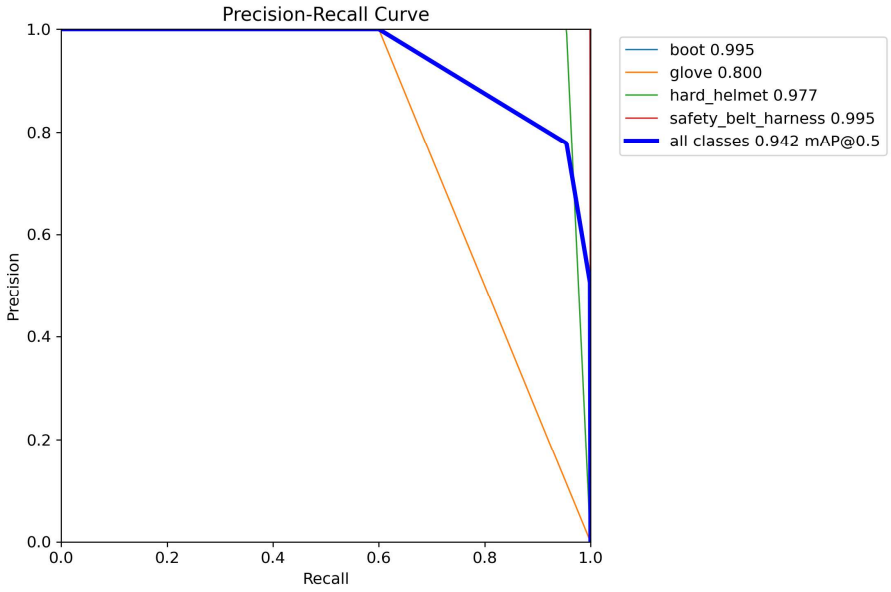


Fonte: O autor.

4.4.2 Comparação com o Modelo Anterior

Antes de resolver os problemas citados em 4.4.1, foi comparada a capacidade de generalização deste modelo com a capacidade do melhor modelo da seção 4.3 afim de avaliar se os modelos anteriores realmente sofrem de sobre-ajuste.

Gráfico 4.3 – Curva PR do modelo desta seção sobre o subconjunto de testes da seção 4.3

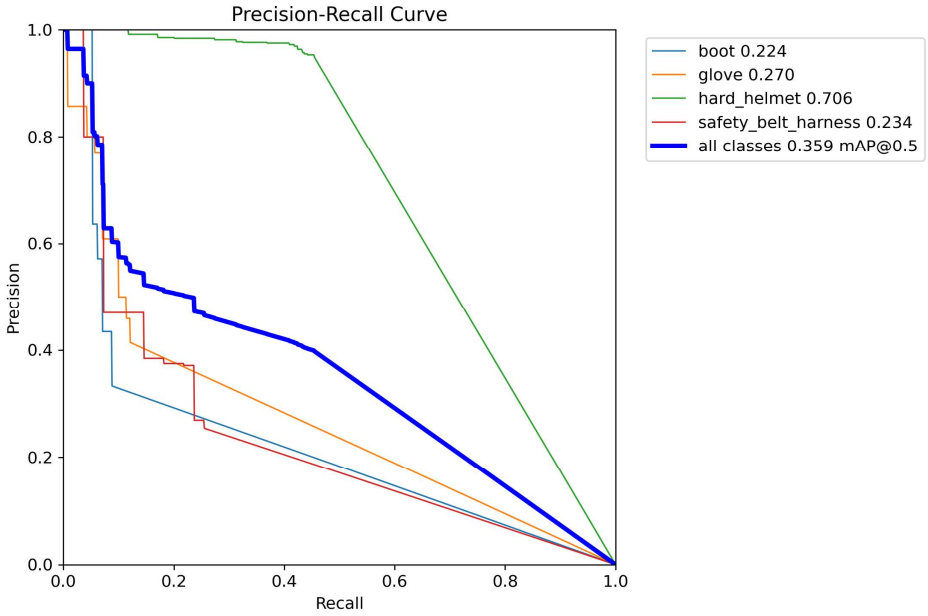


Fonte: O autor.

O modelo desta seção, quando aplicado sobre o subconjunto de testes do *dataset* da seção 4.3 (conforme mostrado no Gráfico 4.3), tem um desempenho pior quando comparado ao modelo da seção 4.3 também sobre o mesmo subconjunto. Mas ainda assim é um desempenho que supera 0.9 mAP.

No entanto, ao avaliar o modelo de 30 gerações da seção 4.3 sobre o subconjunto de testes do *dataset* desta seção, o resultado mostrado no Gráfico 4.4 é de que a mAP caiu para apenas 37% da mAP obtida na Tabela 4.3 (30 gerações) e 46% do mAP do modelo desta seção, que foi demonstrado no Gráfico 4.2. Considerando essa grande diferença, aqui assumiu-se que o modelo desta seção é verdadeiramente melhor que os modelos da seção 4.3 e que aqueles modelos sofrem de sobre-ajuste.

Gráfico 4.4 – Curva PR do modelo anterior sobre o subconjunto de testes do dataset atual



Fonte: O autor.

4.4.3 Influência da Resolução e da Proporção das Imagens

Até agora, a etapa de pré-processamento foi sempre a mesma: redimensionamento para 640x640 pixels esticando a imagem quando necessário para cobrir tal área. Em seguida, analisar-se-á o impacto em desempenho causado pelo aumento da resolução das imagens do *dataset* e do método usado para preencher a grade de 640x640 pixels no caso de imagens com proporções diferentes de 1:1.

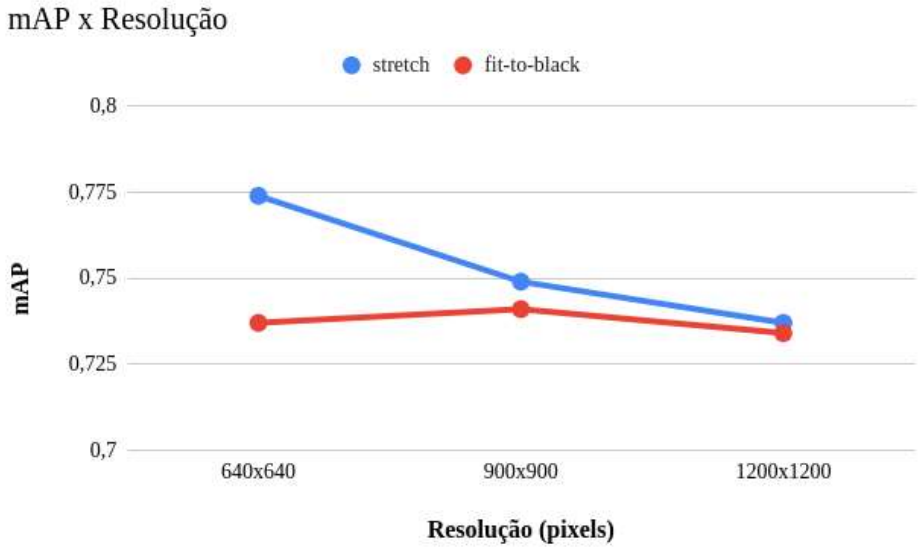
Aqui, duas técnicas foram utilizadas: *stretch*, que consiste em deformar a imagem para preencher a grade (usada até então), e *fit-to-black,* que consiste em fazer a imagem caber dentro da grade sem deformar suas proporções e preencher o restante com a cor preta.

Para isso, treinou-se um modelo por 25 gerações para as resoluções de 640x640, 900x900 e 1200x1200 pixels. Além disso, para cada resolução foi gerado um *dataset* com redimensionamento por *stretch* e outro com redimensionamento por *fit-to-black*.

Contraintuitivamente, o Gráfico 4.5 mostra que o aumento de resolução quando usado *stretch* nas imagens foi prejudicial ao desempenho do modelo. Com o método *fit-to-black*, o desempenho parece não ter sido impactado pela variação de resolução.

Na Tabela 4.5, encontram-se os resultados em mAP de cada modelo de cada *dataset*. Embora as diferenças sejam mínimas, o melhor modelo (640x640, *stretch*) se saiu 5,45% melhor que o pior modelo (1200x1200, *fit-to-black*).

Gráfico 4.5 – Variação de mAP de acordo com a variação da resolução de entrada.



Fonte: O autor. Legenda: Em azul, mAP dos datasets com *stretch*. Em vermelho, mAP dos datasets com *fit-to-black*.

Tabela 4.5 – Desempenho do modelo de acordo com a resolução e a proporção das imagens

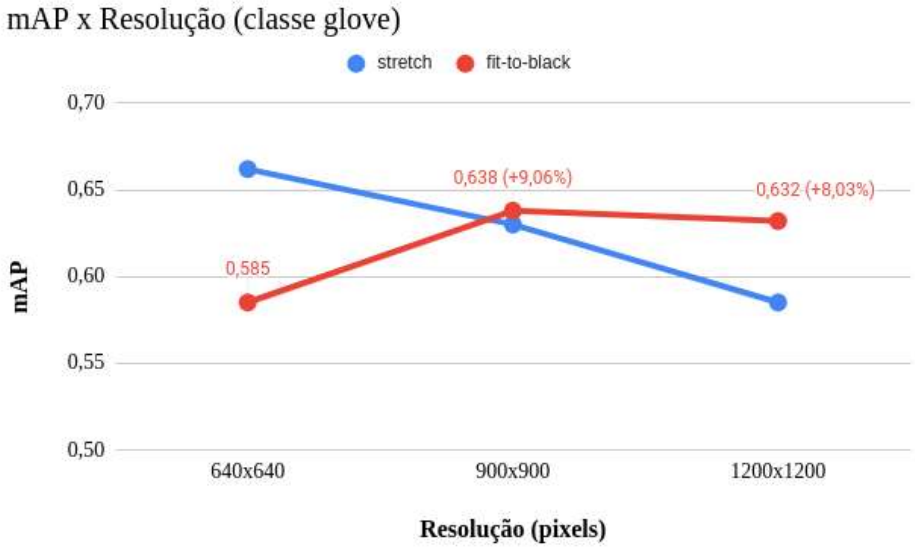
Resolução (pixels)	*Proporção*	*mAP 50-95*
640x640	stretch	0,774
640x640	fit-to-black	0,737
900x900	stretch	0,749
900x900	fit-to-black	0,741
1200x1200	stretch	0,737
1200x1200	fit-to-black	0,734

Fonte: O autor.

Contudo, no momento o objetivo é aumentar o desempenho principalmente na classe *glove*. O Gráfico 4.6 mostra uma tendência de subida no mAP dos modelos treinados com imagens *fit-to-black*. Porém, mesmo com o incremento expressivo de desempenho neste caso, ainda não foi possível atingir o desempenho do primeiro modelo (640x640 pixels, *stretch*).

Assim sendo, variação da resolução ou da proporção das imagens não resultou em ganho de desempenho.

Gráfico 4.6 – Variação de mAP de acordo com a variação da resolução de entrada na classe *glove*.



Fonte: O autor.

4.5 Tiling

Como o pior desempenho do modelo é sobre os menores objetos (*glove* e *boot*), tentou-se dividir cada imagem do *dataset* em quatro, redimensionando cada pedaço para o tamanho da imagem original durante a etapa de pré-processamento. Essa técnica, chamada de *tiling*, foi utilizada no trabalho "The Power of Tiling for Small Object Detection" (UNEL, F. O. ET AL, 2019) para detecção de pedestres e veículos por um *micro aerial vehicle* (MAV) com imagens de alta resolução.

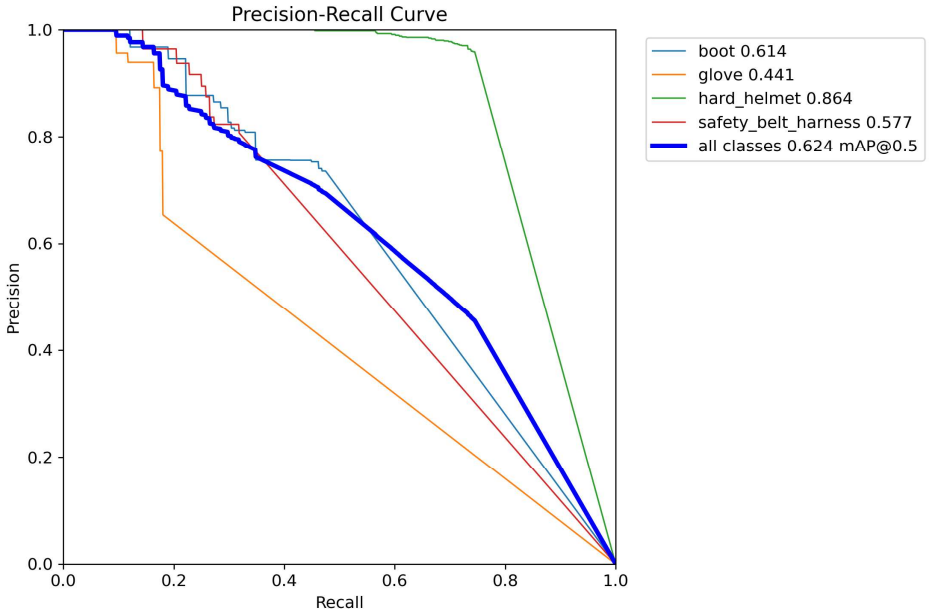
Ainda na etapa de pré-processamento, as imagens foram redimensionadas para 900x900 pixels (*stretch*). Como o número de imagens foi quadruplicado através do *tiling*, não foi aplicada nenhuma técnica de *augmentation* ao *dataset* para diminuir o tempo de treinamento.

4.5.1 Resultado

Conforme o Gráfico 4.7, o resultado foi uma degradação de AP em todas classes.

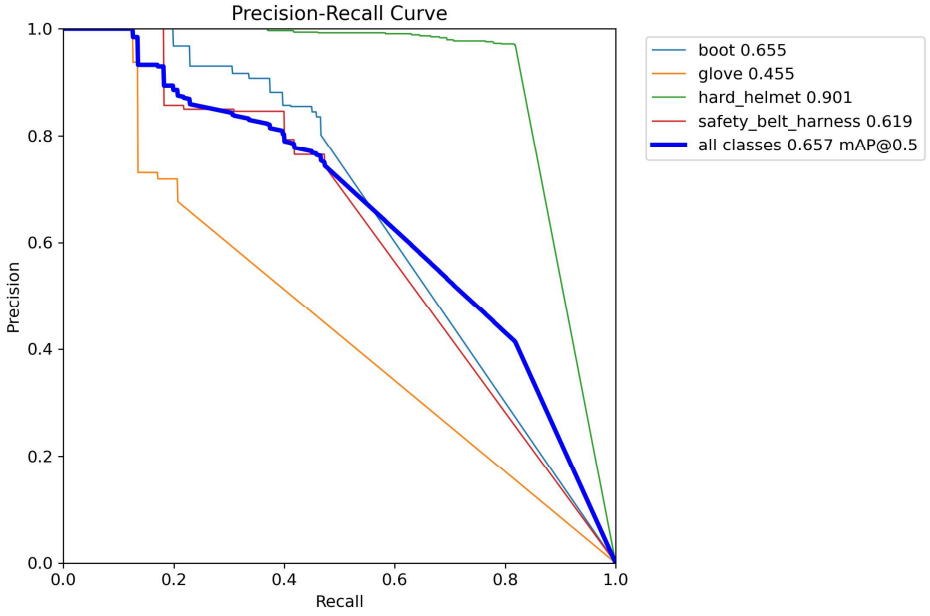
Ao testar este novo modelo contra o subconjunto de testes do *dataset* da seção 4.4, o desempenho geral (mAP) subiu em 5,3%. O resultado pode ser visto no Gráfico 4.8. No caso específico da classe *glove*, o ganho de desempenho foi insignificante (3,17%).

Gráfico 4.7 – Curva PR



Fonte: O autor.

Gráfico 4.8 – Curva PR contra subconjunto de testes da seção 4.4.



Fonte: O autor.

Em suma, a queda de desempenho foi tão grande em relação ao modelo da seção 4.4 que descartou-se o uso de *tiling* neste trabalho. Isso pode ter ocorrido pelo fato de as imagens do *dataset* não possuírem alta resolução, o que pode ser investigado no futuro.

4.6 Ajustes no Banco de Imagens

Embora a heterogeneidade aumente o poder de generalização, no âmbito deste trabalho, algumas imagens do *dataset* estão muito longe do uso real do nosso modelo.

As Figuras 4.4 e 4.5 mostram dois exemplos de imagens que se encaixam nesses parâmetros. Embora ambas as imagens contenham exemplares dos classes-alvo deste projeto, é visível que a dificuldade de detecção de uma luva em qualquer uma delas é uma tarefa muito mais difícil do que na imagem de referência da Figura 3.1a.

Figura 4.4 – Exemplo 1 de imagem com objetos pequenos que fogem do escopo do projeto

Fonte: Betterteam (2023).

Figura 4.5 – Exemplo 2 de imagem com objetos pequenos que fogem do escopo do projeto

Fonte: AFP (2019).

Sob a suposição de que a heterogeneidade das imagens no *dataset* estão impedindo o modelo de conseguir um desempenho maior e considerando-se que a detecção de objetos tão longe do plano da câmera não é necessária neste projeto, imagens como as das Figuras 4.4 e 4.5 foram removidas do *dataset*.

Em seguida, também foram removidas todas as imagens com resolução abaixo de 640x640 pixels, visto que o projeto final não trabalhará com imagens menores que isso. Depois, foram removidas várias imagens que continham apenas rótulos da classe *hard_helmet*, já que os modelos anteriores já apresentaram desempenho acima de 0,9 AP para esta classe e, com menos imagens, menor o tempo de treinamento.

Por último, foram adicionadas mais 832 imagens ao *dataset* respeitando as seguintes condições:

1. Resolução superior ou igual a 640x640 pixels;
2. Há pessoas na imagem, contendo o corpo inteiro das mesmas;
3. Se a pessoa está vestindo um capacete, ela também deve estar usando um dos outros EPIs (*glove*, *boot* ou *safety_belt_harness*). Essa condição existe para não aumentar a proporção da classe *hard_helmet* no *dataset*, que já possui amostras suficientes;

   Após os ajustes mencionados, o banco de imagens passou a ter 3.657 imagens.

4.6.1 Gerando um Novo Dataset

Com o banco de imagens alterado, um novo *dataset* foi criado. Na etapa de pré-processamento as imagens apenas foram redimensionadas para 900x900 pixels (*stretch*). Na etapa de *augmentation*, foram utilizadas rotação entre -15º e +15º e adição de ruído em até 5% dos pixels. Após o aumento, o subconjunto de treinamento passou a contar com 7.700 imagens, triplicando seu tamanho.

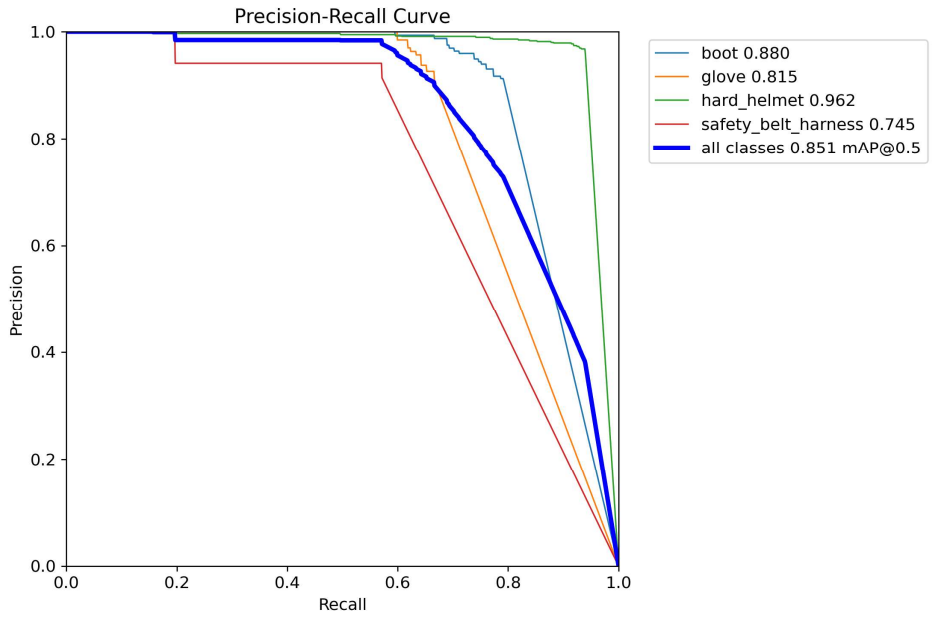
Espera-se que os ajustes feitos no banco de imagens aliados ao uso de mais técnicas de *augmentation* para ampliar o conjunto de informações que o modelo dispõe para treinamento aumentem o desempenho em todas as classes.

4.6.2 Resultados

O Gráfico 4.9 mostra AP de 0,815 para a classe *glove* após um treinamento realizado por 30 gerações. Comparando ao melhor resultado anterior, que foi demonstrado no Gráfico 4.2, houve um aumento de 22%.

Através da Figura 4.6 obteve-se que a sensibilidade na classe *glove* foi de 0,67, enquanto na Figura 4.3, obteve-se sensibilidade de 0,47 também na mesma classe. Nesse caso, ocorreu um aumento de 42,6%.

Gráfico 4.9 – Curva PR.



Fonte: O autor.

Figura 4.6 – Matriz de confusão.

Matriz de Confusão

		Verdade				
		boot	**glove**	**hard_helmet**	**safety_belt_harness**	**background**
Predição	**boot**	178	0	0	0	17
	glove	0	138	0	0	14
	hard_helmet	2	0	574	2	13
	safety_belt_harness	0	0	0	32	3
	background	45	69	37	22	0

Fonte: O autor.

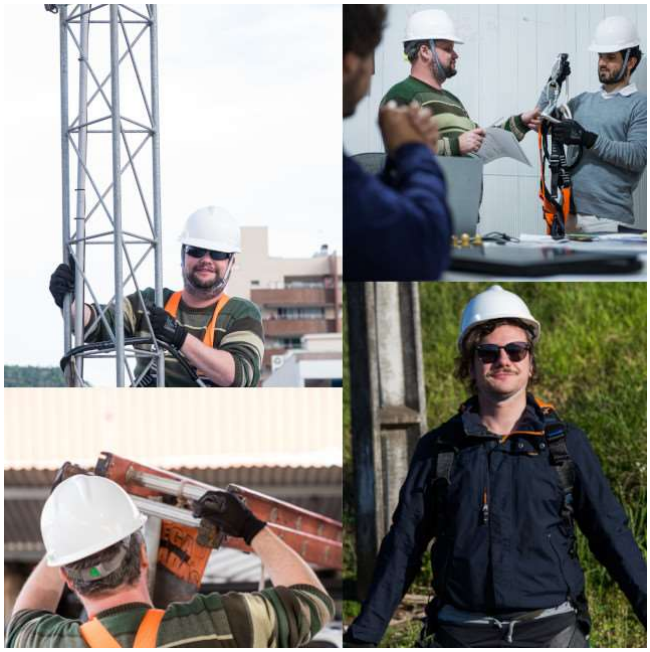
4.7 Utilizando Mosaicos

Em 2020, a técnica de criação de mosaicos para *augmentation* de dados de treinamento foi introduzida em "YOLOv4: Optimal Speed and Accuracy of Object Detection" (BOCHKOVSKIY ET AL, 2020), onde é detalhado os ganhos de desempenho observados ao usar essa técnica.

O método de mosaico consiste em criar uma nova imagem no subconjunto de treinamento ao combinar 4 outras imagens do mesmo subconjunto, sempre mantendo a proporção dos objetos. A Figura 4.7 mostra o exemplo de uma imagem criada através do método de mosaico.

Além do método de mosaicos, os métodos de rotação entre -15º e +15º e adição de ruído em até 5% dos pixels foram mantidos na etapa de *augmentation*.

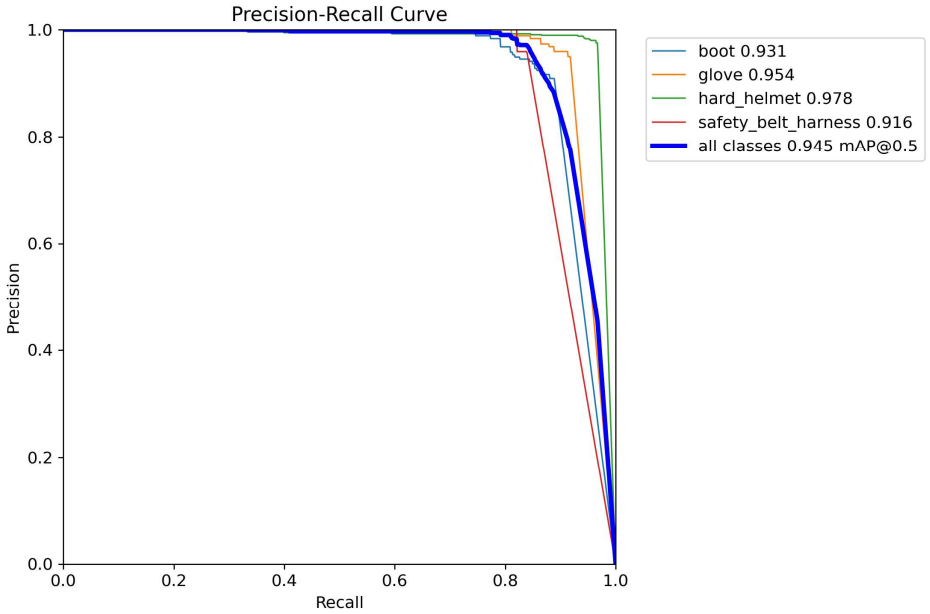
Figura 4.7 – Exemplo de mosaico.



Fonte: O autor, Vorotech.

4.7.1 Resultados

Gráfico 4.10 – Curva PR após aplicação do método de mosaico.

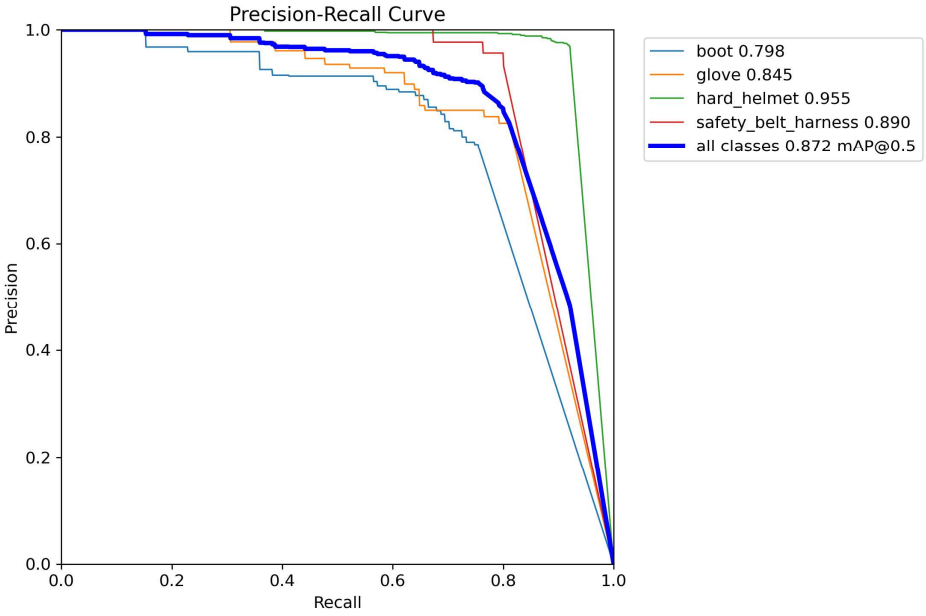


Fonte: O autor.

De acordo com o Gráfico 4.10, o modelo atingiu mAP de 0,945. Esse valor se equipara aos resultados obtidos na Tabela 4.3, se tornando o melhor resultado deste trabalho até então. Em relação ao mAP do modelo da seção 4.6, ocorreu um aumento de 11,04%.

O modelo atual, quando validado no subconjunto de testes de um dos *datasets* da seção 4.4 (900x900 pixels, *stretch*), apresenta o resultado mostrado no Gráfico 4.11. Logo, mesmo incluindo as imagens removidas na seção 4.6, este modelo se saiu melhor que os modelos da seção 4.4.

Gráfico 4.11 – Curva PR do modelo treinado utilizando o método de mosaico sobre o subconjunto de testes da versão anterior do dataset.



Fonte: O autor.

Para validar este modelo com mais precisão, poderia-se utilizar a técnica de k-fold cross validation, mas isso ficará para o futuro. Também poderia-se utilizar técnicas como nested cross-validation para otimizar os hiperparâmetros, porém, os resultados obtidos até o momento já são satisfatórios para o escopo deste projeto, então faremos isso futuramente.

5 ESTRUTURA CLIENTE-SERVIDOR

Conforme discutido na seção 3.2, é preciso criar uma estrutura de cliente-servidor para fornecer o serviço de detecção de EPIs.

5.1 Aplicação Cliente

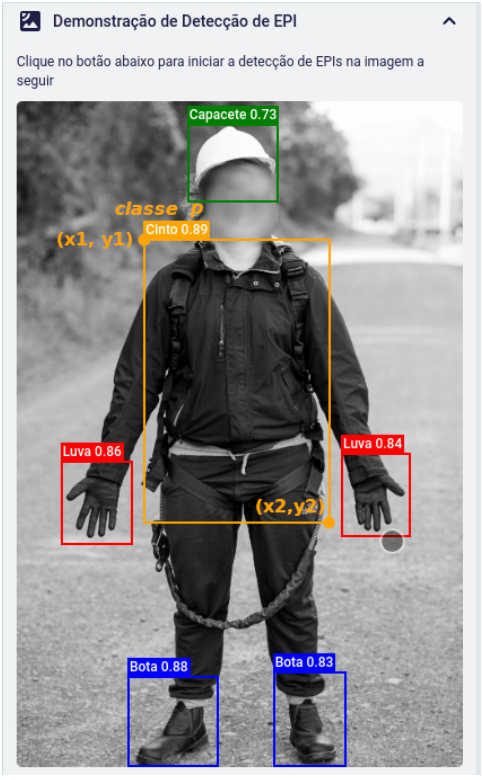
Uma *branch* do aplicativo APR Digital da Vorotech foi criada para implementar o serviço de detecção de EPIs. Assim, uma vez que o projeto estiver concluído será simples de colocá-lo em produção.

5.1.1 Adicionando Suporte a Bounding Boxes

As fotos tiradas pelos usuários são salvas em um objeto. Para que possa-se salvar os dados relativos à detecção de EPIs juntamente das imagens, é necessário modificar esse objeto, adicionando um novo atributo que contém a lista de *bounding boxes* relativas à imagem.

Uma *bounding box* foi modelada como um objeto possuindo 6 atributos [*x1, y1, x2, y2, p, c*]. Onde [*x1, y1*] são as coordenadas relativas do canto superior esquerdo, [*x2, y2*] são as coordenadas relativas do canto inferior direito, *p* é a confiança da predição e *c* é a classe de objeto detectado. A Figura 5.1 mostra os pontos denotados por [*x1, y1, x2, y2*].

Figura 5.1 – Exemplo de detecção de EPIs no aplicativo APR Digital.



Fonte: O autor, Vorotech.

A classe *c* é um número entre 0 e 3 (intervalo fechado), representando uma das 4 classes do modelo de detecção de EPIs. Todavia, o usuário final deve ser capaz de ver o nome da classe e não seu número. Para isso utilizou-se um dicionário de TypeScript que ao receber um dos números referentes a uma classe retorna um objeto contendo o nome em português do objeto e a cor que sua *bounding box* deve ter. Assim, é possível renderizar as *bounding boxes* com uma cor para cada classe. O dicionário garante que no futuro seja fácil alterar este mapeamento para suprir novas demandas.

Todos os componentes do aplicativo que renderizam imagens também foram alterados para mostrar as *bounding boxes* caso elas existam. A Figura 5.1 mostra o resultado final da renderização das *bounding boxes*.

5.1.2 Requisição HTTP

Assim que uma foto é tirada por um usuário, uma requisição HTTP deve ser realizada para rodar o modelo desenvolvido ao longo do Capítulo 4. Essa requisição contém apenas o arquivo da foto no seu corpo. A seção 5.2 aborda como este arquivo é manipulado no servidor.

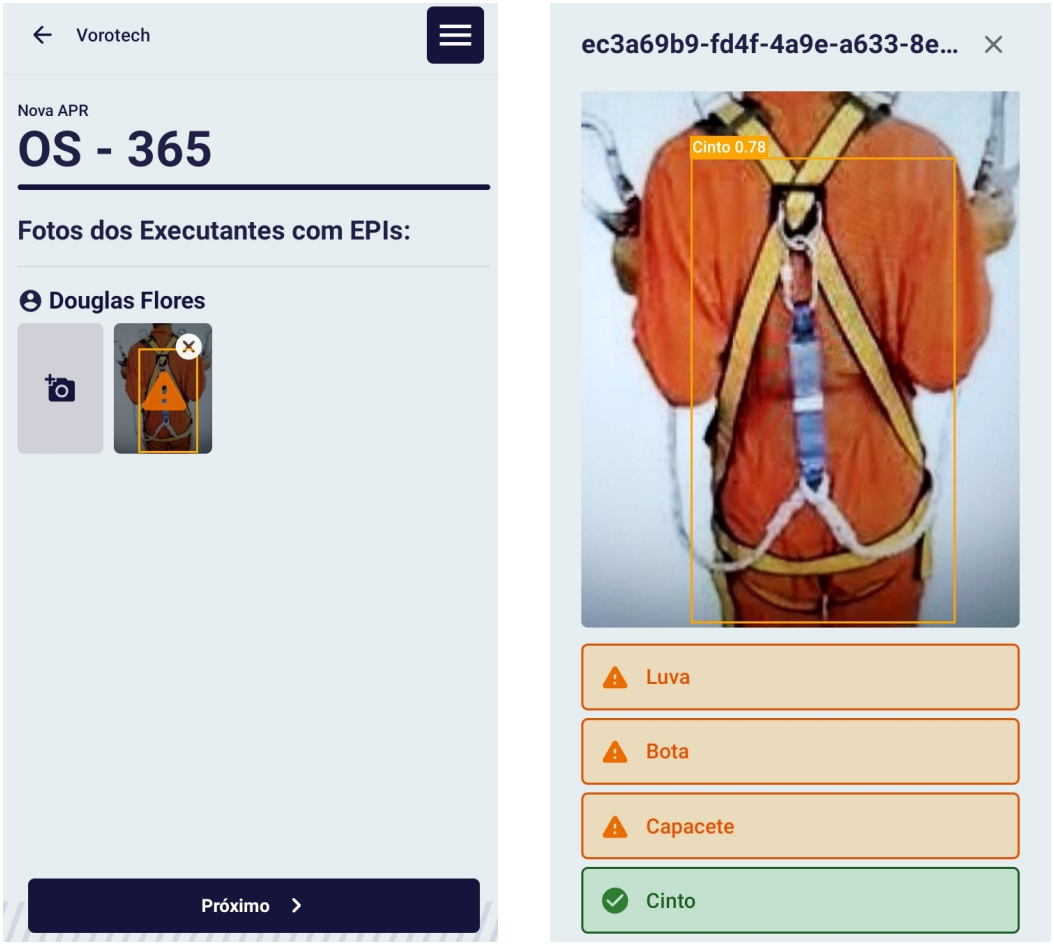
O servidor, então, retorna uma lista de *bounding boxes* que usamos conforme descrito na subseção 5.1.1. A resposta da requisição HTTP não é recebida, a foto exibe uma animação de carregamento.

5.1.3 Experiência de Usuário

Quando o resultado da detecção de EPIs não contém os representantes adequados de cada classe, a foto fica marcada conforme a Figura 5.2a. Ao expandir a imagem, pode-se ver o resultado com mais detalhes, como demonstrado na Figura 5.2b.

Devido ao fato de o modelo de detecção de objetos não ser imune a falhas, o preenchimento do formulário pode continuar independentemente do resultado da detecção de EPIs. De todo modo, a imagem é salva na base de dados remota já com os dados das *bounding boxes*. Futuramente, será possível utilizar esses dados para gerar alertas aos supervisores.

Figura 5.2 – Exemplo da detecção de EPIs no aplicativo.



(a) **(b)**

Fonte: O autor, Vorotech. Legenda: À esquerda, marcação de alerta quando não são encontrados todos os devidos EPIs. À direita, tela de detalhes da foto.

5.2 Função Lambda

A função lambda funciona como uma espécie de servidor. Aqui o termo "servidor" é utilizado de forma coloquial, pois o serviço Lambda da AWS que foi utilizado é uma solução de *serverless computing*, conforme explicado na subseção 3.2.1. Contudo, do ponto de vista de um cliente ele funciona como um servidor HTTP qualquer.

5.2.1 Lambda Handler

Lambda handler é a função que será executada quando a função lambda for ativada. A Figura 5.3 mostra o pseudocódigo que é executado.

Nas linhas 1 e 2, a foto enviada via HTTP é recuperada. Na linha 3 o modelo da YOLOv8s com os pesos desenvolvidos no Capítulo 4 é carregado. Na linha 4 a predição é realizada usando o modelo carregado e a foto de entrada. Na linha 5 as *bounding boxes* são

extraídas do resultado da predição e na linha 6 as *bounding boxes* são enviadas como resposta à requisição.

Figura 5.2 – Pseudocódigo da função lambda handler.

Pseudocódigo Lambda_Handler
1. R←*requisicao_http*;
2. I ← PegarImagem(R);
3. M ← CarregarModeloYOLO();
4. R ← Predicao(M, I);
5. BBoxes ← PegarBboxes(R);
6. **retornar** Bboxes;

Fonte: O autor.

5.2.2 Criação da Função lambda

Uma função lambda pode ser criada facilmente através do painel da AWS. Contudo, algumas configurações devem ser definidas especialmente para este trabalho. O serviço lambda normalmente é utilizado para funções com baixo tempo de execução, por isso por padrão ele possui um *timeout* de 5 segundos. Como a detecção de objetos é uma tarefa mais demorada, o *timeout* foi redefinido para 60 segundos. As permissões de execução também foram reconfiguradas para permitir que qualquer requisitante autenticado com protocólo HTTPS possa fazer requisições à lambda sem a necessidade de autenticação adicional com chaves AWS.

Além disso, é preciso definir quanto poder de processamento cada instância lambda pode ter acesso. A AWS vincula o poder de processamento com o tamanho de memória RAM alocada. Ou seja, quanto mais memória RAM for alocada para função lambda, mais núcleos de CPU também são alocados. Segundo a AWS, cada 1.769 MB de memória RAM alocada também reserva o equivalente a um vCPU (CPU virtual) embora não haja mais especificações sobre a vCPU utilizada. O valor padrão desta configuração é 128 MB, mas essa quantia não foi suficiente para rodar a inferência antes do *timeout*. Por isso, este valor foi incrementado para 512 MB. Aqui é importante levar em consideração que o serviço Lambda cobra por vCPU, então idealmente devemos manter esse número o mais baixo possível.

Por fim, é necessário carregar a função *lambda handler* demonstrada na subseção 5.2.1 para dentro da função lambda recém-criada. Entretanto, aqui encontrou-se uma limitação das funções lambda, que permitem que apenas pacotes de até 250 MB de código sejam carregados. O código final da *lambda handler* deste projeto juntamente a todas as bibliotecas necessárias atingiu um total de 2,5 GB, ou seja, uma ordem de grandeza maior que o limite imposto.

Para contornar essa limitação, utilizou-se o AWS ECR para carregar um container Docker que contém a *lambda handler* mostrada na seção 5.2.1 juntamente a todas as bibliotecas e dependências necessárias. Essa imagem Docker foi, então, vinculada à função lambda criada nesta subseção.

5.2.2.1 Imagem Docker

Para este trabalho, a imagem Docker criada foi carregada no ECR, que se encarrega de criar um container a partir dessa imagem sempre que necessário.

Uma imagem Docker é gerada a partir de um ou mais dockerfiles. Para tornar o processo mais modular, neste trabalho criou-se a imagem final a partir de dois dockerfiles, ambos disponíveis nos apêndices. A vantagem deste método é evitar gastar tempo de processamento recriando partes da imagem que já existem, além de economizar espaço em disco.

5.2.3 Análise de Tempo de Resposta

Com a função lambda no ar e o aplicativo APR Digital finalizado, é possível medir o tempo entre a captura da foto no aplicativo e o recebimento do resultado da inferência (tempo de resposta). Neste caso, os alvos de análise são: tempo de execução da função lambda e taxa de transferência da conexão à internet do aparelho que está rodando o aplicativo.

5.2.3.1 Tempo de Execução da Função Lambda

A AWS possui um serviço chamado CloudWatch que permite ver quanto tempo uma função lambda ficou ativa, desde sua inicialização até o seu retorno. De acordo com a Tabela 5.1, o tempo médio de execução da função lambda foi de 6,85 segundos. Nota-se também um alto desvio padrão de 1,71 segundos.

5.2.3.2 Tempo de Resposta com Internet 4G

A seguir, realizou-se uma medida de tempo *end-to-end* em um celular com conexão 4G. Para capturar estes dados foi utilizado um cronômetro com precisão de 10ms ativado manualmente. O intuito deste teste é ter uma estimativa de quanto tempo o usuário irá esperar até receber o resultado da inferência e, pelo fato de a execução da função lambda estar na casa dos segundos não precisamos de uma precisão maior que essa.

Tabela 5.1 – Tempo de execução da função lambda

Requisição	*Tempo de Execução (ms)*
1	9.222
2	5.054
3	6.301
4	6.014
5	6.267
6	6.194
7	5.049
8	6.369
9	10.212
10	7.834
Média	6.851,6
Desvio Padrão	1711,68

Fonte: O autor.

Considerando o tempo de transferência em uma rede 4G, obteve-se um tempo médio de 7 segundos, conforme a Tabela 5.2. Isso significa que em um cenário de conexão 4G, o tráfego de rede equivale em média a 2,66% do tempo de resposta. Nota-se que o desvio padrão foi amenizado em relação ao da Tabela 5.1, levando a crer que mais amostras precisam ser testadas futuramente para obter-se uma estimativa mais precisa do real desvio padrão.

Tabela 5.2 – Tempo de resposta com internet 4G

Amostra	*Tempo de Resposta (ms)*
1	9.000
2	7.470
3	6.810
4	6.590
5	7.010
6	6.420
7	6.540
8	6.570
9	6.710
10	7.270
Média	7.039
Desvio Padrão	767,38

Fonte: O autor.

5.2.3.3 Tempo de Resposta com Internet 3G

Também foi simulado o efeito de uma conexão 3G lenta através das ferramentas de desenvolvedor do navegador Google Chrome. Nestas condições, o navegador limita a largura de banda para 400Kbps e adiciona 2 segundos de latência de requisição. Conforme a Tabela

5.3, houve um aumento do tempo de resposta para 8,6 segundos. Assim, o tempo de tráfego de rede passa a ser em média 20% do tempo de resposta. Aqui o desvio padrão foi semelhante ao da Tabela 5.1.

Tabela 5.3 – Tempo de resposta com internet 3G lenta (400Kbps)

Amostra	*Tempo de Resposta (ms)*
1	9.510
2	13.470
3	7.890
4	7.310
5	7.460
6	7.390
7	8.500
8	7.260
9	8.800
10	8.070
Média	8.566
Desvio Padrão	1873,553961

Fonte: O autor.

5.2.3.4 Tempo de Inicialização da Função Lambda

Outro fator importante que contribui para o tempo de execução da função lambda e, consequentemente, para o tempo de resposta é o tempo de inicialização da função lambda. Como explicado, a função lambda é executada sob demanda e por isso deve executar todo o código do *lambda handler* sempre que for invocada. Isso quer dizer que o modelo deve ser carregado toda vez que uma requisição a função lambda for realizada. Em um servidor tradicional poderia-se carregar o modelo apenas uma vez ao iniciar o servidor e apenas rodar a inferência quando requisitado.

Em média, o tempo de inferência corresponde a 73,97% do tempo de execução (Tabela 5.4). Portanto, conclui-se que haveria um ganho significativo de desempenho se fosse possível manter o modelo em memória entre uma requisição e outra.

5.2.3.5 Conclusão

O tempo de execução da função lambda foi o gargalo do tempo de resposta, então os esforços futuros devem ser direcionados a redução do mesmo, seja através de um modelo mais simples e leve, do aumento de recursos alocados por instância da função lambda ou até mesmo do aluguel de um servidor dedicado para atender às requisições.

Tabela 5.4 – Relação entre tempo de inferência e tempo de execução

Amostra	*Tempo de Inferência (ms)*	*Tempo de Execução (ms)*
1	4.042	5.558,38
2	3.738,5	5.148,92
3	4.663,3	6.104,35
4	4.599	6.012,79
5	4.878	6.327,92
6	4.955,5	6.697,12
7	4.546	6.036,27
8	4.622,4	6.055,78
9	4.979,1	6.413,02
10	5.199,2	8.128,6
Média	4.622	6.248,32
Desvio Padrão	442,52	789,74

Fonte: O autor.

6 PLANOS PARA O FUTURO

No momento, o sistema de alerta aos gerentes no caso de captura de fotos com irregularidades ainda não foi implementado e, provavelmente, será o próximo passo do projeto.

Também será interessante treinar o modelo usando as imagens já capturadas pelo aplicativo APR Digital, adicionando-as ao *dataset* atual e vendo como isso impacta o desempenho e o tempo de inferência.

A remoção da dependência de conexão à internet para execução da YOLO é mais um objetivo no longo prazo. Para lidar com as diferenças de poder de processamento entre os celulares dos clientes, uma opção pode ser criada no aplicativo para escolher entre execução remota ou local. Ou também escolher entre um modelo mais acurado e outro menos custoso, como a Tinier-YOLO-V3 demonstrada por (FANG W. ET AL, 2019), por exemplo, que é capaz de executar uma inferência a cada 40ms em um Jetson TX1.

A Ultralytics fornece outros modelos pré-treinados que prometem menor tempo de execução, conforme mostrado na Tabela 4.1. Logo, também seria interessante ver como eles se comparam ao modelo desenvolvido aqui nos quesitos de tempo de inferência e de mAP, avaliando se a troca é vantajosa ou não. Outra opção é buscar por serviços online que dão acesso à GPUs CUDA para reduzir o tempo de inferência.

No âmbito do *dataset*, também será considerado adicionar uma classe *person* para detectar pessoas e usar a *bounding box* dessa classe para verificar quais *bounding boxes* das outras classes estão contidas nela e assim ser capaz de determinar se todas as pessoas na foto estão usando os EPIs adequados.

7 CONCLUSÃO

Neste trabalho, explicou-se a ideia de empregar um método de visão computacional para detectar o uso de EPIs em fotos tiradas como parte do preenchimento de ARs já em uso hoje através do aplicativo APR Digital da Vorotech.

Utilizando a YOLOv8 e o serviço AWS Lambda, foi possível gerar um modelo de detecção de EPIs e hospedar esse modelo em nuvem, disponibilizando o serviço *on demand* com tempo de resposta médio inferior a 10 segundos mesmo em conexões precárias de 400Kbps.

Diversos métodos de pré-processamento e *augmentation* foram utilizados nos *datasets* em busca do melhor desempenho. Ao reavaliar as condições em que o modelo será empregado e readaptar o dataset de acordo, o desempenho de 0,777 mAP subiu para 0,851 mAP. Quando analisamos a classe *glove* (a de menor AP) isoladamente, obteve-se um salto de 0,668 AP para 0,815 AP, surpreendentes 22%. Ainda, ao utilizar mosaicos como método de *augmentation* na criação do *dataset* foi possível atingir 0,945 mAP, um aumento de mais 11%.

Por fim, no Capítulo 6 são discutidas possíveis melhorias para o projeto e quais são as próximas etapas a serem tomadas.

REFERÊNCIAS

Monografia no todo

FACELI, K.; LORENA, A.C.; GAMA, J.; CARVALHO, A.C.P. **Inteligência Artificial:** Uma Abordagem de Aprendizado de Máquina. Rio de Janeiro: LTC, 2011.

Dissertações, teses, trabalhos individuais, etc.

DALAS, N. E TRIGGS. **Histograms of oriented gradients for human detection.** 2005. In: 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05).IEEE.

GIRSHICK, R. **Fast r-cnn.** 2015. In: 2015 Proceedings of the IEEE International Conference on Computer Vision (ICCV).IEEE.

GIRSHICK, R., DONAHUE, J., DARRELL, T., AND MALIK, J. **Rich feature hierarchies for accurate object detection and semantic segmentation.** 2014. In: 2014 IEEE Conference on Computer Vision and Pattern Recognition. IEEE.

REDMON, J., DIVVALA, S., GIRSHICK, T., AND FARHADI, A. **You Only Look Once: Unified, Real-Time Object Detection.** 2016. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).IEEE.

ROCHA, C. D. F. **Aplicação do algoritmo haar cascade em um sistema embarcado para detecção de ovos do mosquito aedes aegypti em palhetas de ovitrampas.** 2018. Trabalho de Conclusão de Curso (Tecnólogo em Análise e Desenvolvimento de Sistemas) – Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, Natal, 2018.

VIOLA, P. JONES. **Robust real-time face detection.** 2001. In: Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001. IEEE.

Artigo de periódico

HE, K., ZHANG, X., REN, S., AND SUN, J. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognitiont. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Seoul, v. 37, n. 9, p. 1904-1916, set. 015). IEEE.

Em meio eletrônico

BOCHKOVSKIY, J., WANG, S.Y. AND LIAO, H.Y.M. **YOLOv4: Optimal Speed and Accuracy of Object Detection.** 2020. Disponível em: <https://arxiv.org/abs/2004.10934>. Acesso em: 24 ago. 2023.

FANG, W., WANG, L., REN, P. **Tinier-YOLO: A Real-Time Object Detection Method for Constrained Environments.** 2019. IEEE. Disponível em < https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8941141>. Acesso em: 24 ago. 2023.

UNEL, F. O., ÖZKALAYC, B. O., ÇIGLA, C. The Power of Tiling for Small Object Detection. Proceedings of the **IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops**, 2019, p. 0-0. Disponível em: < https://openaccess.thecvf.com/content_CVPRW_2019/papers/UAVision/Unel_The_Power_of_Tiling_for_Small_Object_Detection_CVPRW_2019_paper.pdf>. Acesso em: 24 ago. 2023.

APÊNDICE A: DOCKERFILE CUSTOMISADO PARA YOLOV8

```
# Define custom function directory
ARG FUNCTION_DIR="/function"
FROM python:3.9 as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}

# Install the function's dependencies
RUN pip install \
      --target ${FUNCTION_DIR} \
      awslambdaric \
      torch==1.9.0+cpu torchvision==0.10.0+cpu –f
https://download.pytorch.org/whl/torch_stable.html \
      ultralytics==8.0.122

# Use a slim version of base Python image to reduce the final image size
FROM python:3.9-slim
RUN apt update && apt install -y --no-install-recommends \
      libgl1 \
      libglib2.0-0 \
      && apt clean \
      && rm -rf /var/lib/apt/lists/*

# Include global arg in this stage of the build
ARG FUNCTION_DIR
# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}
```

APÊNDICE B: DOCKERFILE PARA LAMBDA HANDLER

```
ARG FUNCTION_DIR="/function"
FROM yolov8:0.6
COPY . ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/local/bin/python", "-m", "awslambdaric" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "lambda_function.handler" ]
```