UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALESSANDRO SOUZA SILVA

DESENVOLVIMENTO DE UMA SOLUÇÃO DE ANÁLISE DE VÍDEO NA BORDA PARA SISTEMAS INTELIGENTES DE VIDEOVIGILÂNCIA EM TEMPO REAL

QUIXADÁ

2021

ALESSANDRO SOUZA SILVA

DESENVOLVIMENTO DE UMA SOLUÇÃO DE ANÁLISE DE VÍDEO NA BORDA PARA SISTEMAS INTELIGENTES DE VIDEOVIGILÂNCIA EM TEMPO REAL

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Dr. Michel Sales Bonfim

QUIXADÁ

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S578d Silva, Alessandro Souza.
Desenvolvimento de um solução de Análise de Vídeo na Borda para Sistemas Inteligentes de Videovigilância em tempo real / Alessandro Souza Silva. – 2021.
75 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2021.
Orientação: Prof. Dr. Michel Sales Bonfim.

1. Vídeos-Análise. 2. Docker (Software). 3. Aprendizagem profunda. I. Título.

CDD 004

ALESSANDRO SOUZA SILVA

DESENVOLVIMENTO DE UMA SOLUÇÃO DE ANÁLISE DE VÍDEO NA BORDA PARA SISTEMAS INTELIGENTES DE VIDEOVIGILÂNCIA EM TEMPO REAL

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em:____/_____/_____.

BANCA EXAMINADORA

Prof. Dr. Michel Sales Bonfim (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Paulo Antonio Leal Rego
Universidade Federal do Ceará (UFC)

Prof. Dr. Pedro Pedrosa Rebouças Filho
Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE)

Prof. Dr. Regis Pires Magalhães
Universidade Federal do Ceará (UFC)

À família, amigos e todo o curso de Ciência da Computação da Universidade Federal do Ceará - Campus Quixadá, que me ajudaram ao longo desta caminhada.

AGRADECIMENTOS

Agradeço à Deus e todas as vibrações positivas que existem no universo.

Agradeço à minha família, pelo amor incondicional, pelo carinho e por todo o apoio oferecido. Em especial, agradeço ao meu irmão Leo, que sempre me guiou em direção às melhores escolhas e nunca desacreditou no meu potencial. Sem você irmão talvez não tivesse conseguido chegar onde estou hoje, muito obrigado.

Agradeço ao meu orientador, professor Michel, por todas as conversas e pela excelente orientação exercida, que contribuíram bastante para o meu amadurecimento acadêmico.

Agradeço aos meus colegas de turma do curso de Ciência da Computação, pelos grupos de estudos, pela diversão, pelos rodízios de *pizza*, pelas longas conversas na fila do ônibus, enfim, por tudo que uma amizade construída na graduação é capaz de oferecer. Em destaque, agradeço pelos momentos que compartilhei ao lado de Diego, Diogo, Zé Gabriel, Carlos Alcides, Claro, Xavier, Severo, Cleison, Lucas e Letícia. Desejo sucesso e saúde a todos.

“You Can Become A Hero.”

(All Might)

RESUMO

A Análise de Vídeo tem desempenhado um papel importante nos mais variados setores de segurança pública, em especial, quando aplicada em Sistemas Inteligentes de Videomonitoramento. Para minimizar problemas envolvendo sobrecarga de transmissão na rede ou alta latência de resposta, a Análise de Vídeo na Borda busca migrar parte da carga de trabalho do processo de Análise de Vídeo para dispositivos próximos à fonte de dados, chamados dispositivos de borda. Assim, este trabalho teve como objetivo desenvolver uma arquitetura de Análise de Vídeo na Borda para sistemas de videomonitoramento em tempo real, com foco na tarefa de detecção e reconhecimento de placas de identificação de carros. Para isso, este trabalho propõe uma divisão do processo de análise em módulos funcionais e independentes, implantados por meio da tecnologia Docker. Além do mais, uma etapa intermediária entre as etapas de detecção e reconhecimento também é proposta, a fim de descartar placas com texto de identificação distorcido e assim evitar que essas placas passem para processos mais robustos, como no caso da etapa de reconhecimento de placas, ou armazenamento na Nuvem. Para esse fim, foram aplicadas técnicas de Aprendizado Profundo em um problema de classificação de placas de acordo com a qualidade de leitura de seus textos de identificação. Experimentos foram conduzidos a fim de selecionar o melhor algoritmo para o modelo de classificação de qualidade, com melhor algoritmo obtendo acurácia de 99,04% no conjunto de validação e 100% em um conjunto pequeno de teste. Além disso, uma aplicação *web* foi desenvolvida para validar a arquitetura proposta, comprovando que a arquitetura cumpre com seus objetivos ao reduzir em média 25,64% do tráfego de rede no seu fluxo de *frames*, devido a um controle de qualidade de resolução, e 56,65% no seu fluxo de placas, devido a etapa de filtragem de placas proposta.

Palavras-chave: Vídeos-Análise. Docker (Software). Aprendizagem profunda.

ABSTRACT

Video Analytics has played an important role in the most varied public security sectors, mainly when applied to Intelligent Video Surveillance Systems. Edge Video Analytics seeks to migrate part of the workload of the Video Analysis process to devices close to the data source, called edge devices, to minimize problems involving transmission overhead on the network or high response latency. Therefore, this work aimed to develop an Edge Video Analytics architecture for real-time video monitoring systems, focusing on detecting and recognizing license plate characters. It proposes a division of the analysis process into functional and independent modules, implemented through Docker technology. Besides, an intermediate step between the detection and recognition steps is also proposed to discard plates with distorted identification text and prevent these plates from moving to more robust processes, as in the case of the plate recognition step or Cloud storage. For this purpose, Deep Learning Techniques were applied in a license plate classification problem according to your identification texts' reading quality. Experiments were conducted in order to select the best algorithm for the classification model, with a better algorithm, obtaining an accuracy of 99.04% in the validation set and 100% in a small test set. In addition, a web application was developed to validate the proposed architecture, proving that the architecture meets its objectives by reducing on average 25.64% of the network traffic in its frames flow, due to a resolution quality control, and 56.65% in its license plate flow, due to the filtering step proposed.

Keywords: Videos-Analysis. Docker (Software). Deep Learning.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

AV	Análise de Vídeo
AVB	Análise de Vídeo na Borda
CNN	*Convolutional Neural Network*
DMC	*Data Machines Corp*
FN	Falsos Positivos
FP	Falsos Negativos
IoT	*Internet of Things*
MCD	Módulo de Captura e Detecção
MF	Módulo de Filtragem
MR	Módulo de Reconhecimento
VP	Verdadeiros Positivos
VN	Verdadeiros Negativos
YAML	*Yet Another Markup Language*

SUMÁRIO

1 INTRODUÇÃO

De acordo com uma pesquisa de 2019 da *Information Handling Services*, estima-se que estejam em funcionamento em todo o mundo 770 milhões de câmeras de videovigilância e que, até 2021, esse valor ultrapasse um bilhão (MARKIT, 2019).

Esse cenário é reflexo da crescente necessidade prática de Sistemas Inteligentes de Videovigilância, sobretudo no setor de segurança pública. A **Análise Inteligente de Vídeo**, ou apenas AV (do português, Análise de Vídeo), busca reconhecer de forma automática eventos temporais e espaciais relevantes nos vídeos. Isso minimiza o envolvimento humano em tarefas típicas de monitoramento como, por exemplo, a detecção ou reconhecimento de faces pelo departamento de polícia, reconhecimento de placas de veículos pelo departamento de transporte e a detecção de chamas pelo corpo de bombeiros (ZHANG *et al.*, 2019). A AV costuma ser implementada com base em algoritmos de Aprendizado Profunda. Isso se deve ao fato das suas estruturas serem reconhecidas como estado da arte nas atividades de detecção e classificação em imagens (RAI *et al.*, 2019).

De maneira geral, os aplicativos de AV possuem como requisitos fundamentais uma baixa latência, o uso eficiente da largura de banda e o alto custo computacional. Além disso, o enorme volume de dados produzido pelas câmeras as tornam a tecnologia mais desafiadora da IoT (do inglês, *Internet of Things*). Somente uma câmera de vídeo é capaz de produzir cerca de dez *gigabytes* de dados por dia, ademais, um estudo da Seagate Technology LLC estima que o volume de dados gerados por dia por sistemas de vigilância no mundo chegue em trinta e cinco mil *petabytes* em 2023. Um cenário ideal provê facilidade de armazenamento desses dados somado ao combate dos desafios de manutenção e operação dos sistemas de videovigilância (ZHANG *et al.*, 2019; SREENU; DURAI, 2019).

Para evitar a sobrecarga de transmissão desses dados na rede, soluções de AV centralizadas e dependentes apenas dos recursos oferecidos pela nuvem tendem a ser substituídas por soluções inspiradas no paradigma de Computação de Borda. A AVB (do português, Análise de Vídeo na Borda) dita que parte da carga de trabalho proveniente do processo de AV seja executada diretamente nos dispositivos de borda, como câmeras inteligentes e servidores de borda (ZHANG *et al.*, 2019). Se por um lado essa técnica garante baixa latência, disponibilidade local, conscientização de contexto/localização, tomada de decisão rápida e privacidade, por outro gera desafios de implantação, limitações de *hardware*, restrições de *design* e custos de infraestrutura e manutenção (KHANN *et al.*, 2019).

Uma técnica viável e interessante para implantação de AVB é a Virtualização Baseada em Contêineres. Em resumo, trata-se de uma virtualização em nível de SO (do português, Sistema Operacional) em que as instâncias geradas compartilham o mesmo *kernel* da máquina *host*. O uso da tecnologia Docker (DOCKER, 2021) como gerenciador de contêineres em aplicativos de borda acarreta uma série de benefícios. Em Ismail *et al.* (2015), os autores avaliaram a viabilidade do uso de contêineres Docker na Computação de Borda, concluindo que esta tecnologia pode ser aplicada para as seguintes funcionalidades: implantação e finalização de serviços; gerenciamento de recursos e serviços; tolerância a falhas; e recursos de armazenamento em cache.

Baseado nisso, este trabalho propõe uma arquitetura de AVB para a atividade de detecção/reconhecimento de placas para uso em sistemas de videovigilância em tempo real. A arquitetura AVB deste trabalho optou por dividir essa atividade de AV em módulos funcionais e independentes, usando contêineres Docker como unidade portadora de tais módulos. Isso, sobretudo, permite que diferentes estágios da análise se adaptem à velocidade e aos requisitos de outros estágios (RICHINS *et al.*, 2020).

Com o objetivo de gerar uma menor sobrecarga de transmissão pela rede, foram implementadas duas funcionalidades primordiais na arquitetura de AVB deste trabalho. A primeira é um controle de qualidade de resolução de vídeo, fundamentado na detecção de placas. A outra funcionalidade é uma etapa extra de filtragem de placas adicionada ao processo de AV da atividade de detecção/reconhecimento de placas.

Dessa maneira, dado um modelo de detecção e outro de reconhecimento prontos, este trabalho investiu na construção e avaliação de um modelo de Aprendizado Profundo capaz de filtrar imagens de placas com boa qualidade e evitar que placas com textos ilegíveis prossigam para a etapa de reconhecimento. Neste trabalho, textos ilegíveis em placas são textos que possuem ao menos um caractere impossível de ser identificado com certeza por seres humanos. Caso contrário, trata-se de textos legíveis. Assim, foi notado que muitas das placas detectadas na etapa de detecção, mais de 50% das detecções, sofrem por distorções causadas pela movimentação rápida dos carros, falta de foco das câmeras e iluminação não favorável do cenário. Isso torna a quantidade de placas detectadas com texto ilegível enorme, o que geraria desperdício de recursos se as mesmas fossem encaminhadas para a etapa de reconhecimento, armazenamento na Nuvem ou quaisquer serviços de processamento mais robustos.

Finalmente, foi conduzido experimentos para comparar quatro algoritmos de Aprendizado Profundo, candidatos a serem usados como modelo de filtragem de placas. Para tanto,

este trabalho utilizou uma série de gravações reais das ruas da cidade de Fortaleza, no estado do Ceará. Além disso, este trabalho desenvolveu uma aplicação *web* para validar a arquitetura AVB proposta.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Propor, implementar e validar uma arquitetura de AVB aplicável a Sistemas Inteligentes de Videovigilância em tempo real, baseando-se no uso de contêineres Docker e técnicas de Aprendizado Profunda, para detecção/reconhecimento de placas de identificação de carros.

1.1.2 Objetivos específicos

a) Modularizar o processo de análise de vídeo em contêineres, permitindo que os módulos atuem como serviços na borda da rede.
b) Aplicar um controle de qualidade de vídeo na borda da rede, baseando-se nos resultados da detecção de placas.
c) Filtrar as placas detectadas com texto ilegível, para que as mesmas não sejam processadas na etapa de reconhecimento.
d) Validar a arquitetura proposta.

O restante deste trabalho está organizado da seguinte maneira. Na Seção 2 são apresentados os principais conceitos para o desenvolvimento deste trabalho, que são a Análise de Vídeo, o paradigma de Computação de Borda, Aprendizado Profunda e a Virtualização Baseada em Contêineres. Na Seção 3 realiza-se a apresentação e discussão dos trabalhos relacionados, com suas semelhanças e diferenças do trabalho aqui proposto. Na Seção 4, a metodologia para o desenvolvimento deste trabalho é apresentada em detalhes. Finalmente, na Seção 9 as considerações finais são dadas junto à propostas de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, são apresentados alguns dos conceitos necessários para o entendimento e desenvolvimento deste trabalho. Na Seção 2.1, a AV no contexto da videovigilância é abordada. Na Seção 2.2, o Aprendizado Profundo é resumido, com foco especial nas Redes Neurais Convolucionais, na técnica de Transferência de Aprendizado e em algumas Métricas de Avaliação em problemas de classificação binária. Na Seção 2.3, o paradigma de Computação de Borda é explorado e comparado com o paradigma de Computação em Nuvem, ao mesmo tempo que a AVB é definida. Por fim, na Seção 2.4, a Virtualização Baseada em Contêineres e o Docker são detalhados.

2.1 Análise de Vídeo

O *Big Data* pode ser entendido como um fenômeno moderno, em que informações são geradas em alto volume, alta velocidade, alta complexidade e alta variedade. Essa situação requer tecnologias capazes de extrair *insights* significativos dos dados para a tomada de decisão. Dois subprocessos principais podem ser identificados na análise de *Big Data*: gerenciamento e análise de dados. Se o gerenciamento de dados foca nas técnicas para adquirir, armazenar, preparar e recuperar os dados, a análise, por outro lado, refere-se às técnicas usadas para analisar e adquirir inteligência a partir do *Big Data* (GANDOMI; HAIDER, 2015).

Essas técnicas de análise de *Big Data* incluem a Análise de Texto, a Análise de Áudio, a Análise de Mídia Social, a Análise Preditiva e a AV, sendo essa última técnica o principal foco deste trabalho (GANDOMI; HAIDER, 2015).

De maneira geral, a AV busca reconhecer, entender ou monitorar os eventos temporais e espaciais relevantes de fluxos de vídeo. Essa técnica pode ser vista como a aplicação de Visão Computacional, Reconhecimento de Padrões e Aprendizado de Máquina no mundo real, pois trabalha fortemente conceitos oriundos dessas áreas. Na literatura, o assunto também é conhecido como Análise de Conteúdo de Vídeo ou Análise Inteligente de Vídeo (GAGVANI, 2009).

2.1.1 Sistemas Inteligentes de Videovigilância

Os Sistemas Inteligentes de Videovigilância são as aplicações dominantes da AV, principalmente os sistemas voltados para o setor de segurança pública (ZHANG *et al.*, 2019). Este trabalho focou nesses tipos de sistema.

Um Sistema Inteligente de Videovigilância é descrito como uma técnica inteligente de processamento de vídeo que monitora objetos persistentes e transitórios em uma ambiente. Esse monitoramento é feito por meio de sensores, por exemplo, câmeras de vigilância. O projeto de um Sistema Inteligente de Videovigilância costuma servir de suporte para o pessoal de segurança, com a análise e processamento em tempo real, e para investigações forenses, com o pós-processamento do vídeo (RAI *et al.*, 2019).

Rai *et al.* (2019) afirma que a AV nesses tipos de sistemas abrangem até as duas seguintes tarefas:

- **Otimização de armazenamento** - O sistema armazena o vídeo somente quando um objeto ou algum movimento é detectado; ou reduz a qualidade do vídeo, taxa de quadros ou resolução, em situações de inatividade ou de não detecção no ambiente filmado.
- **Identificar eventos ameaçadores** - O sistema busca identificar possíveis incidentes de segurança e gerar alertas sobre os mesmos.

Seguindo essas definições de Rai *et al.* (2019), o contexto deste trabalho girou em torno de sistemas de videovigilância com análise e processamento em tempo real, dando ênfase na tarefa de otimização de armazenamento.

2.1.2 Processo geral da Análise de Vídeo

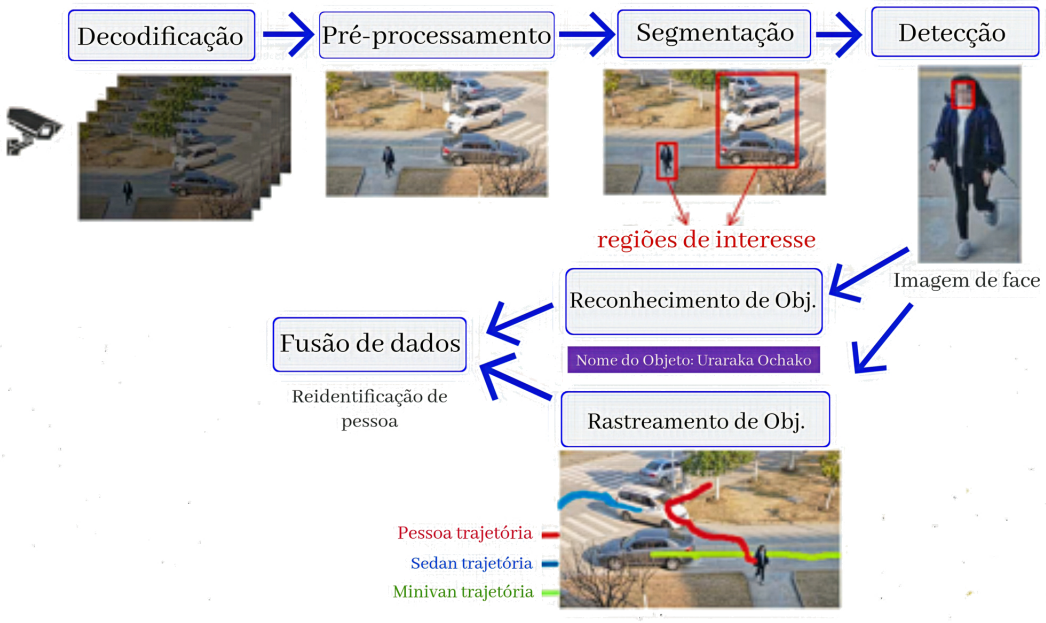
De Zhang *et al.* (2019) e Gagvani (2009) é possível identificar até oito etapas presentes em um processo genérico de AV, também chamado de *pipeline* do processo de AV. A Figura 1 mostra um exemplo prático desse *pipeline*, aplicada no contexto de reconhecimento de faces. As etapas identificadas, resumidamente, são:

- **Decodificação de vídeo** - Transformar os dados originados de *streaming* de vídeo em uma série de imagens ou *frames*;
- **Pré-processamento** - Aplicar procedimentos sobre os *frames*, visando maximizar a qualidade das imagens e diminuir os efeitos gerados por distorções;
- **Segmentação de imagem** - Detectar os pixels em *foreground*, ou regiões alteradas de um *frame*, e conecta-los para formar um conjunto de *blobs* ou regiões de interesse;
- **Detecção de objetos** - Classificar o conjunto de *blobs* em classes gerais, ou seja, classes suficientemente distintas entre si, como pessoa, veículo, edifício ou animal;
- **Reconhecimento de objetos** - Classificar o conjunto de *blobs* em classes semanticamente significantes, permitindo identificar instâncias específicas como um veículo pelo número

da placa ou uma pessoas pelo nome de registro;

- **Rastreamento de objetos** - Estabelecer correspondência entre *blobs* em *frames* sucessivos de vídeo, localizando um ou vários objetos em movimento;
- **Reconhecimento de atividade** - Inferir atividades em uma sequência de vídeo, com base na classificação e rastreamento dos objetos em cena;
- **Fusão de dados** - Integrar vários resultados analíticos de diferentes fontes de vídeo, com o objetivo de obter informações mais valiosas.

Figura 1 – Processo geral da AV



Fonte: Adaptado de Zhang *et al.* (2019)

Dependendo do domínio da aplicação, algumas dessas atividades não são obrigatórias, além dos casos em que não é necessário seguir estritamente a ordem definida no *pipeline* (GAGVANI, 2009). Contudo, duas ou mais etapas podem ser integradas e executadas simultaneamente, como ocorre em soluções baseadas em algoritmos de Aprendizado Profundo, tema discutido na Seção 2.2 (ZHANG *et al.*, 2019) a seguir.

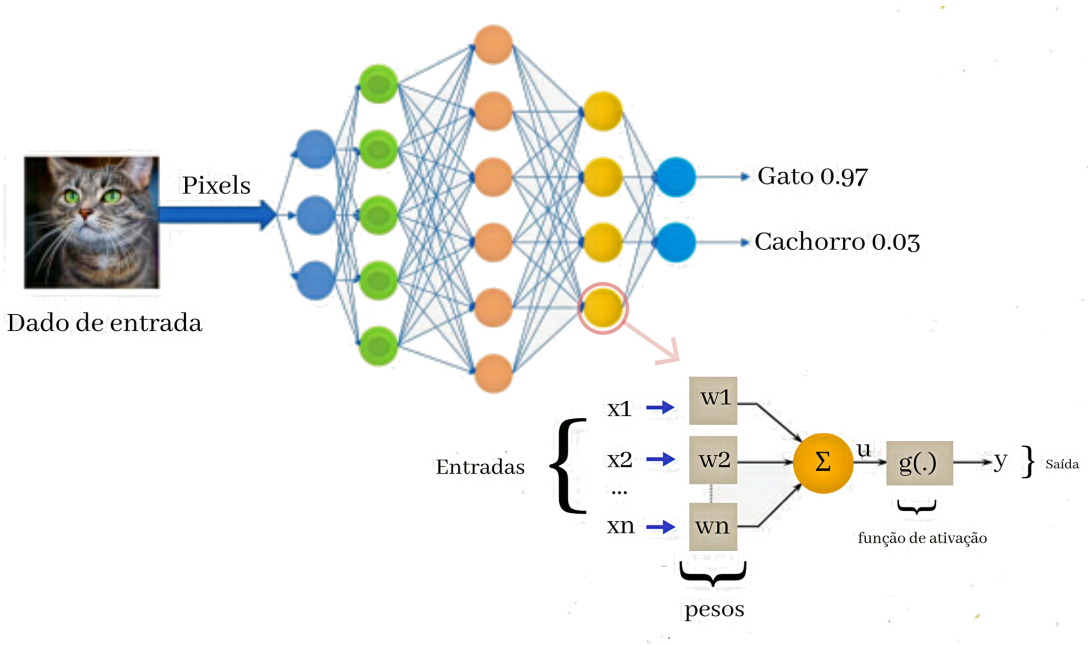
2.2 Aprendizado Profundo

Neste trabalho, as fases do processo de AV, apresentadas na Seção 2.1.2 são implementadas com base em algoritmos de Aprendizado Profundo, subcategoria do Aprendizado de

Máquina do ramo de Inteligência Artificial. A razão disso é o fato do Aprendizado Profundo ser reconhecido como estado da arte nas atividades de classificação e detecção de objetos em domínios variados, entre eles o da videovigilância (RAI *et al.*, 2019).

As Redes Neurais Profundas são construídas utilizando camadas de neurônios matemáticos, que são modelos simplificados do neurônio biológico, que processam os dados de entrada e reconhecem padrões sobre os mesmos (ACADEMY, 2019). A Figura 2 mostra uma Rede Neural Profunda aplicada na atividade de classificação de imagens entre duas classes: gato ou cachorro. Também na Figura 2 é ilustrado um neurônio matemático simplificado. Esse neurônio matemático se resume em um conjunto de sinais de entrada (x1, x2, ..., xn) que são ponderados por pesos sinápticos (w1, w2, ..., wn), também chamados de parâmetros da rede. Em seguida, os valores já ponderados passam por uma combinação linear e por uma função de ativação, sendo essa última responsável por limitar a saída do neurônio em um intervalo de valores (ACADEMY, 2019).

Figura 2 – Rede Neural Profunda simplificada

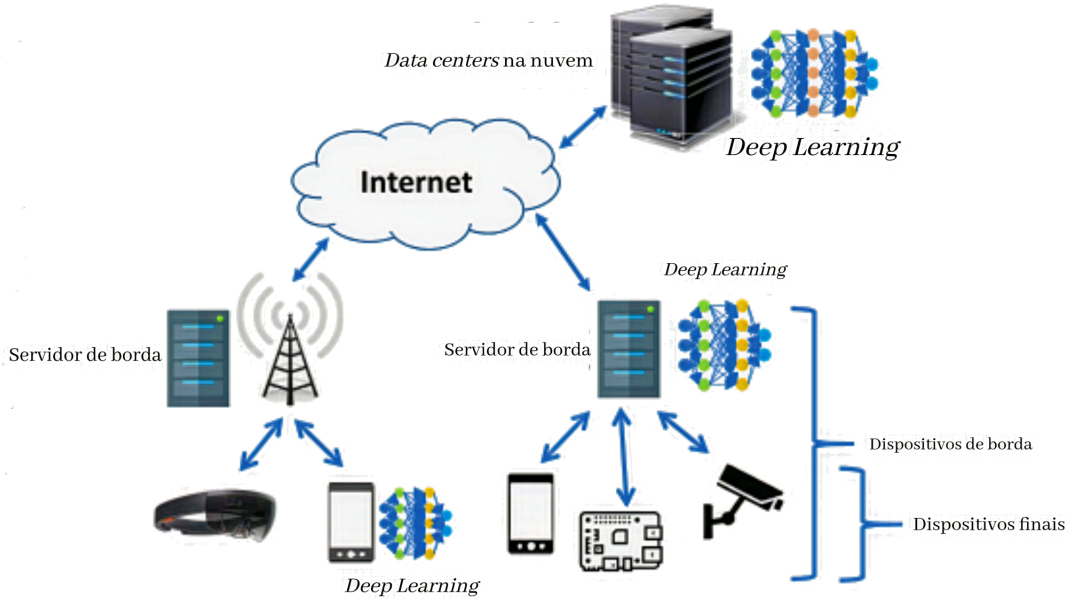


Fonte: Adaptado de Rai *et al.* (2019)

Como mostra a Figura 3, esses modelos de Aprendizado Profundo são também executados na borda da rede, referenciando o que é conhecido como Computação de Borda, conceituado com mais detalhes na Seção 2.3. Nessas circunstancias, dois pontos essenciais devem ser discutidos (RAI *et al.*, 2019):

- **Treinamento** - Para treinamento de um algoritmo de Aprendizado Profundo, intensivos cálculos matriciais são inevitáveis, ocasionando problemas relacionados à latência nos dispositivos finais.
- **Decisões de design** - Existem *tradeoffs* entre as métricas do sistema. A decisão entre um modelo computacionalmente barato ou caro afeta requisitos como a necessidade de precisão e o custo de armazenamento dos parâmetros.

Figura 3 – Redes Neurais Profundas em Computação de Borda



Fonte: Adaptado de Rai *et al.* (2019)

2.2.1 Rede Neural Convolucional

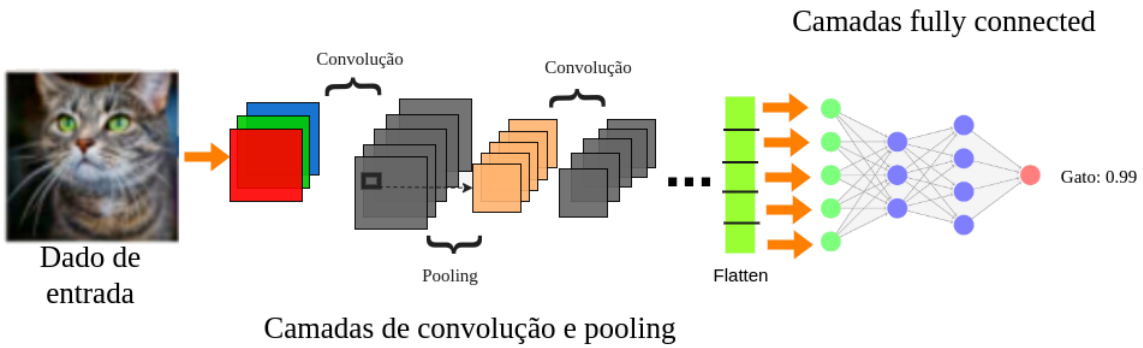
A CNN (do inglês *Convolutional Neural Network*), ou Rede Neural Convolucional, é a estrutura de Aprendizado Profundo definida por este trabalho para a realização da AV e criação do modelo de filtragem. Trata-se de uma classe de algoritmos de Aprendizado Profundo especializada no processamento de dados estruturados em topologia de grade, caso específico dos dados em formato de imagem (SANTOS *et al.*, 2019; ACADEMY, 2019).

Diferente dos outros modelos, a CNN é capaz de encontrar uma hierarquia de recursos por meio de um procedimento denominado convolução, bem como permitir o compartilhamento de parâmetros entre os componentes da rede, tornando sua etapa de treinamento

mais rápida e prática (SANTOS *et al.*, 2019; ACADEMY, 2019). Esse compartilhamento de parâmetros reduz drasticamente o armazenamento exigido por modelos de CNN, logo uma característica a favor da AVB.

A Figura 4 ilustra a arquitetura base de uma Rede Neural Convolucional encontrada com frequência na literatura. Uma CNN profunda consiste em várias camadas convolucionais e camadas de *pooling* agrupadas em módulos, empilhados uns sobre os outros. Tais módulos são seguidos por uma ou mais camadas densas e uma camada de *output* (RAWAT; WANG, 2017).

Figura 4 – CNN simplificada



Fonte: Elaborado pelo autor

A seguir, as principais camadas de uma CNN são resumidas (RAWAT; WANG, 2017; ACADEMY, 2019):

- **Camada Convolucional** - Camada responsável por extrair recursos da imagem por intermédio do aprendizado de múltiplos filtros. Seus neurônios são organizados em mapas de recursos, algo semelhante à matrizes. Assim, as entradas dessa camada são convolvidas com esses filtros para o cálculo de novos mapas de recursos.
- **Camada de Pooling** - Camada responsável por reduzir a dimensão da matriz de mapas de recursos vinda da camada convolucional, por meio da extração de valores mais representativos de uma região. O *pooling* é comumente feito por uma função de média ou soma. Isso diminui o efeito de distorções ou pequenas alterações na imagem.

- **Camada de Flatten** - Camada responsável por preparar as saídas da camada convolucional e de *pooling* para as camadas densas da arquitetura. Transforma a matriz de mapas de recursos em um vetor unidimensional.
- **Camada Densa ou *Fully Connected*** - Camada responsável pelo raciocínio de alto nível da arquitetura. Em geral, todos os neurônios da camada anterior são entradas para cada um dos neurônios da camada atual. Por padrão são as últimas camadas de um Rede Neural Convolucional e responsáveis pela decisão final da rede, chamadas nesse caso de camada de *output* da arquitetura.
- **Camada de Dropout** - Camada responsável por desligar alguns neurônios aleatoriamente, a fim de se evitar sobreajuste do modelo durante a etapa de treinamento.

2.2.2 Transferência de Aprendizado

É fato que as Redes Neurais Convolucionais, assim como os demais modelos de Aprendizado Profundo, são algoritmos com forte dependência de uma quantidade massiva de dados de treinamento. A Transferência de Aprendizado é uma técnica importante para resolver o problema de conjunto de dados de treinamento insuficiente ou demasiadamente pequeno, sendo esse o caso deste trabalho. Nessa técnica, uma arquitetura completa, ou partes da mesma, pode ser retreinada com a adição de novas camadas em um novo conjunto de dados. Isso proporciona uma redução significativa na demanda de dados de treinamento e no tempo de treinamento da nova arquitetura (TAN *et al.*, 2018).

Este trabalho se beneficiou da Transferência de Aprendizado para construção do modelo de filtragem de placas. Dentre as várias arquiteturas famosas para uso nessa técnica, este trabalho optou por selecionar aquelas que tinham um tamanho relativamente pequeno ou poucos parâmetros ou pouca profundidade. Pois, no contexto da AVB, os recursos tendem a ser limitados. Os dados usados como base na decisão podem ser acessados no Anexo A, provinda da documentação da biblioteca Keras para Aprendizado Profundo (KERAS, 2021). Os quatro modelos escolhidos são: Resnet50, MobileNetv2, InceptionV3 e EfficientNetB0. Enquanto as arquiteturas MobileNetv2 e Inception foram escolhidas por serem redes rasas, Resnet50 e EfficientNetB0 foram selecionadas com base no seu tamanho pequeno.

2.2.3 Métricas de Avaliação para problemas de classificação

Por último, o problema de filtragem de placas abordado neste trabalho é também um problema de classificação de placas. Por isso a alternância entre os dois termos para se referir ao mesmo problema no decorrer do texto deste trabalho. Mais especificamente, o problema de classificação de placas deste trabalho tem natureza binária.

Em problemas de classificação binária, as Métricas de Avaliação são aquelas responsáveis por medir o desempenho de um modelo com base no que é conhecido como matriz de confusão. Nessa matriz, são denotados tanto os números de instâncias que são classificadas corretamente nas classes positiva e negativa, chamados VP (do português, Verdadeiros Positivos) e VN (do português, Verdadeiros Negativos), quanto classificadas incorretamente nas classes positiva e negativa, chamados de FP (do português, falsos positivos) e FN (do português, Falsos Negativos). Sendo assim, as principais métricas de avaliação para classificação binária são (HOSSIN; SULAIMAN, 2015):

- **Acurácia** - Mede a proporção de previsões corretas, ou total de verdadeiros positivos e verdadeiros negativos, sobre o número total de instâncias. A Equação 2.1 reflete o cálculo dessa métrica.

$$Acc = \frac{VP + VN}{VP + VN + FP + FN} \tag{2.1}$$

- ***Recall*** - Mede a fração de instâncias positivas que são classificados corretamente, mais precisamente, número de verdadeiros positivos dividido pelo total de verdadeiros positivos e verdadeiros negativos. A Equação 2.2 mostra a fórmula da métrica *recall*.

$$Rec = \frac{VP}{VP + VN} \tag{2.2}$$

- ***Precision*** - Mede as instâncias positivas que estão corretamente previstas a partir do total de previstos em uma classe positiva, mais precisamente, número de verdadeiros positivos dividido pela soma do número de verdadeiros positivos com o número de falsos positivos. Assim, a Equação 2.3 ensina o cálculo de tal métrica.

$$Prec = \frac{VP}{VP + FP} \tag{2.3}$$

- ***F-measure*** - Métrica calculada a partir da média harmônica entre os valores de *recall* e *precision*. A Equação 2.4 descreve a fórmula do *f-measure*.

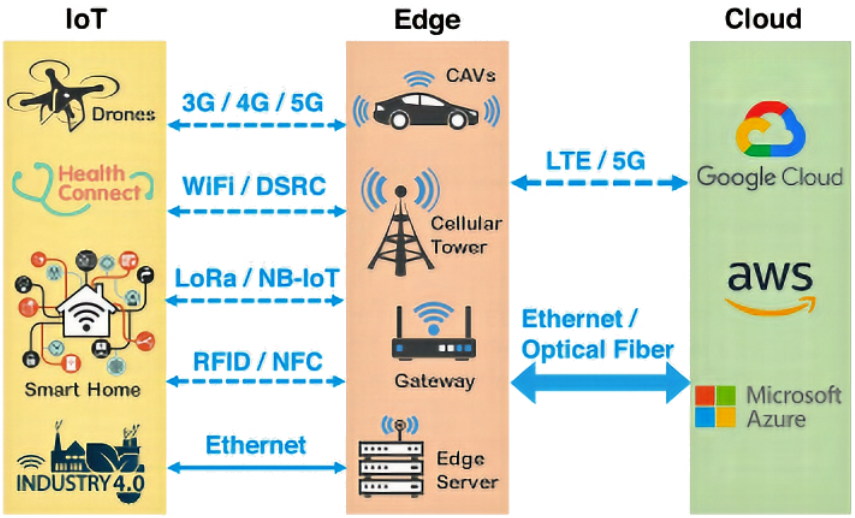
$$FM = \frac{2 * Rec * Prec}{Rec + Prec} \tag{2.4}$$

2.3 Computação de Borda

Outro conceito fundamental deste trabalho é da Computação de Borda. Trata-se de um paradigma emergente que ganhou bastante popularidade tanto no setor acadêmico quanto no industrial. Devido a quantidade exorbitante de aplicativos da IoT e as suas necessidades de baixa latência e resposta rápida, essa abordagem inovou ao direcionar dados, aplicativos e serviços computacionais para a borda da rede (KHANN *et al.*, 2019).

A Figura 5 abstrai um modelo típico de Computação de Borda em três camadas, sendo elas: IoT, *Edge* e *Cloud*. Os protocolos típicos de comunicação entre as camadas são também apresentados na Figura 5 (SHI *et al.*, 2019). A borda da rede é formada pelos recursos de computação e de rede presentes ao longo do caminho entre as fontes de dados e os *data centers* na nuvem (SHI *et al.*, 2016). Para complemento da Figura 5, a Figura 3 também mostra um pouco da Computação de Borda no contexto geral de videovigilância. Nesse cenário, os dispositivos de bordas são as próprias câmeras, as unidades de processamento colocadas próximos às câmeras e os servidores de borda. A comunicação entre as câmeras e a borda é comumente feita por intermédio de redes com fio ou sem fio, como WiFi e LTE (ANANTHANARAYANAN *et al.*, 2017).

Figura 5 – Modelo típico de Computação de Borda em três camadas



Fonte: Shi *et al.* (2019)

2.3.1 Borda vs Nuvem

Pode-se concluir que a Computação de Borda busca apenas complementar a tradicional Computação em Nuvem, aproximando os usuários finais dos seus serviços de computação. Na Computação em Nuvem, serviços de armazenamento e processamento são oferecidos sob demanda aos usuários. Além disso, ela busca otimizar dinamicamente os recursos compartilhados entre vários usuários finais. (KHANN *et al.*, 2019).

Porém, no contexto da IoT, a tradicional Computação em Nuvem tem uma série de deficiências que a limitam (SHI *et al.*, 2019):

- **Latência** - A latência do sistema é maior, visto que os aplicativos apenas enviam os dados ao *data center* e, em seguida, esperam por uma resposta do mesmo. Esse problema pode ser intensificando devido a problemas de rede;
- **Largura de banda** - A largura de banda sofre grande pressão, devido a transmissão de enormes volumes de dados dos dispositivos de borda para a nuvem;
- **Disponibilidade** - É um grande desafio para os provedores manterem os serviços em nuvem sempre disponíveis. Algumas aplicações são parte do cotidiano de vários usuários;
- **Energia** - Os *data centers* consomem bastante energia. Além disso, com a crescente quantidade de computação e transmissão, o consumo de energia restringirá o desenvolvimento de centros de Computação em Nuvem;
- **Segurança e privacidade** - Dados sigilosos são confiados aos provedores, sofrendo com o risco de vazamento de informações privadas dos usuários.

A Computação em Nuvem e a Computação de Borda possuem semelhanças, entretanto a Computação de Borda possui suas próprias características, além daquelas óbvias de melhoria no uso da largura de banda da rede até a nuvem, proximidade e privacidade: (KHANN *et al.*, 2019)

- **Distribuição geográfica densa** - Inúmeras plataformas de computação são implantadas na borda da rede, lidando melhor com as análises em tempo real e em larga escala;
- **Suporte à mobilidade** - Suporte a mobilidade dos dispositivos finais é fornecida, seguindo o princípio da separação da identidade do *host* da identidade local;
- **Reconhecimento de localização** - Permite que os usuários móveis acessem o servidor de borda mais próximo de sua localização atual;
- **Reconhecimento de contexto** - Utilizar informações de contexto dos dispositivos para tomada de decisão, acesso à serviços, melhoria da satisfação e experiência do usuário;

- **Heterogeneidade** - Existem diversas plataformas, arquiteturas, *softwares* e *hardwares* na borda da rede. Essas diferenças resultam em problemas de interoperabilidade e desafios de implantação.

Finalmente, os pontos levantados neste tópico são válidos paras as aplicações de análise de *Big Data*. Além do mais, a AV é considerada mais adequada se sua execução é feita em uma arquitetura de Computação de Borda, dado que é realizada de forma mais rápida e com maior precisão (SHI *et al.*, 2016; KHANN *et al.*, 2019).

2.3.2 Análise de Vídeo na Borda

Basicamente, os aplicativos de AV podem ser executados em uma arquitetura baseada em Computação em Nuvem ou em uma arquitetura baseada em Computação de Borda. Na configuração de AV na Nuvem, as câmeras e os demais sensores têm apenas a função de rotear o fluxo de vídeo capturado para um servidor centralizado e dedicado que, por sua vez, executa todas as etapas apresentadas no processo geral de Análise de Vídeo, como detecção, segmentação e reconhecimento (GANDOMI; HAIDER, 2015).

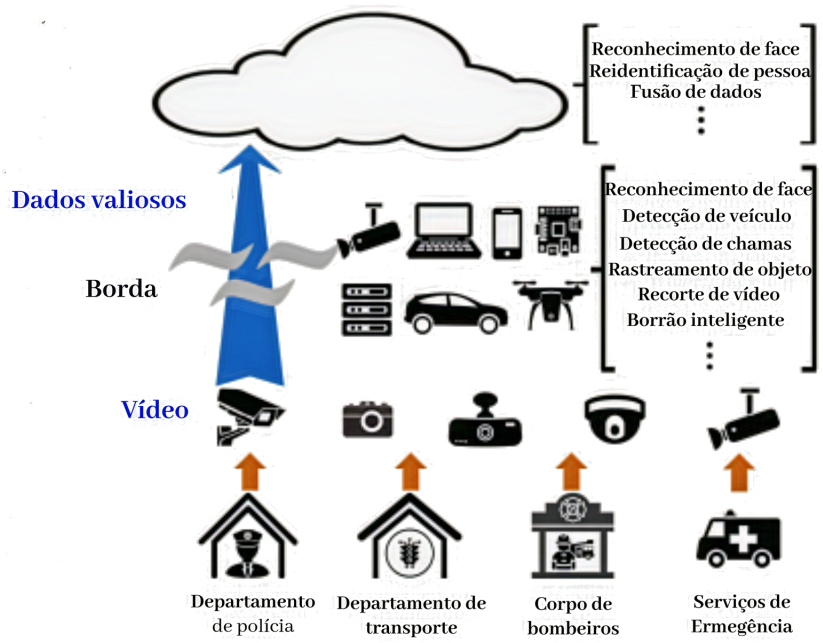
Assim, a AVB dita que parte da carga de trabalho proveniente do processo de AV seja executada diretamente nos dispositivos de borda (ZHANG *et al.*, 2019). Ou seja, a análise é feita localmente e diretamente sobre os dados brutos capturados. A Figura 6 mostra uma visão geral da AVB, ressaltando atividades de AV tipicamente feitas na própria borda e os setores de segurança pública que comumente a utilizam.

2.4 Virtualização Baseada em Contêineres

Para processar a AV nos dispositivos de borda, a técnica de virtualização é essencial. Assim como na definição de Computação na Nuvem, a definição de Computação de Borda implica na necessidade de alguma tecnologia capaz de fornecer uma camada de isolamento e multilocação, com recursos de computação divididos e compartilhados dinamicamente. Nesse sentido, duas tecnologias são comumente usadas, como ilustra a Figura 7 (BERNSTEIN, 2014):

- **Hipervisor** - A máquina *host* pode executar uma ou mais máquinas virtuais, sendo que cada máquina virtual possui um SO inteiro instalado. Assim, sua implantação é ideal quando os aplicativos exigem diferentes SO ou versões de um mesmo sistema.
- **Contêiner** - A máquina *host* pode executar um ou mais contêineres, sendo que os contêi-

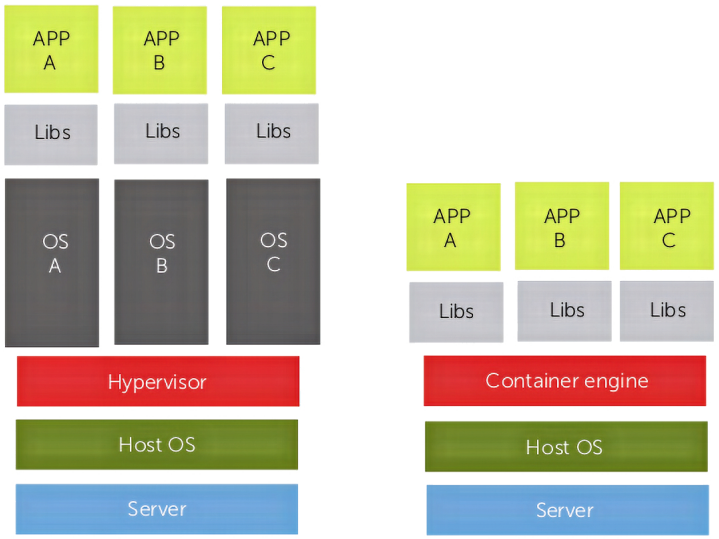
Figura 6 – Visão geral das aplicações de AVB na segurança pública



Fonte: Adaptado de Zhang *et al.* (2019)

neres compartilham o mesmo *kernel* da máquina *host*. Ou seja, suas implantações terão um tamanho significativamente menor que as implantações de hipervisor. Isso permite que até vários contêineres sejam executados em um único *host*.

Figura 7 – Comparação entre a tecnologia de hipervisor e contêiner



Fonte: Bernstein (2014)

Em essência, os hipervisores trabalham com virtualização em nível de *hardware* e os contêineres com virtualização em nível de SO. As máquinas virtuais administradas pelo hipervisor consomem dispendiosamente recursos como RAM e CPU. Por outro lado, com o uso de contêineres esses recursos são muito mais bem utilizados, além dos próprios contêineres serem de fácil criação e destruição. Apesar de compartilharem o *kernel*, os contêineres em execução no mesmo SO possuem sua própria camada de rede abstraída, seus processos e seu próprio mapeamento de volume (MERKEL, 2014).

2.4.1 Docker

Docker é uma plataforma para criar, desenvolver e compartilhar contêineres com as seguintes características: flexibilidade para a implantação de novas funcionalidades, leveza, isolamento de tráfego, portabilidade, fraco acoplamento, escalabilidade para múltiplas fontes, uso eficiente dos recursos computacionais e segurança (DOCKER, 2021). Sua inspiração veio do *Linux Containers*, um conjunto de ferramentas, *API's* e bibliotecas que melhorou as capacidades de uma outra ferramenta chamada *chroot* (BERNSTEIN, 2014).

Os contêineres Docker são criados por meio de imagens bases, que incluem desde os fundamentos de um SO, até uma pilha complexa de aplicativos pré-construída (BERNSTEIN, 2014). O Docker Hub[1] é um dentre os serviços em nuvem disponíveis para armazenamento, compartilhamento e acesso à imagens bases.

A forma eficaz de construção de novas imagens Docker é com o uso de um Dockerfile. Trata-se de um *script* composto por vários comandos e especificações de argumentos que automaticamente executa ações em uma imagem base, adicionando uma nova camada para cada instrução (BERNSTEIN, 2014). Quando um Dockerfile é atualizado e a imagem é reconstruída, as camadas sem alterações são reaproveitadas (DOCKER, 2021).

É possível executar e definir vários contêineres simultaneamente por meio de um arquivo YAML (do inglês, *YAML Ain't Markup Language*). Isso é possível graças à uma ferramenta chamado Docker Compose, sendo ela muito utilizada na criação de ambientes de desenvolvimento e testes automatizados. Alguns recursos que tornam o Compose efetivo são: a possibilidade de vários ambientes isolados em um único *host*; capacidade de preservar dados de volume quando os contêineres são criados; recriação apenas dos contêineres que foram alterados no arquivo; e a definição de variáveis para melhor referenciar outros contêineres (DOCKER,

[1] https://hub.docker.com

2021).

Para facilitar o gerenciamento da comunicação entre contêineres, o Docker oferece uma abstração chamada rede ou rede Docker. Por padrão, o Docker disponibiliza três tipos de rede. A rede *bridge* é a rede padrão e permite que todos os contêineres nessa mesma rede se comuniquem entre si por meio do protocolo TCP/IP. Por outro lado, a rede do tipo *none* isola o contêiner por completo e impede qualquer serviço de rede que o contêiner venha a necessitar. Por fim, existe a rede chamada *host* em que o contêiner é capaz de usar a rede da máquina *host* diretamente. Ou seja, a rede *host* entrega para o contêiner todas as interfaces de rede do *host* e remove o isolamento de rede até então existente entre ambos (DOCKER, 2021).

O Docker também oferece a possibilidade do usuário criar sua própria rede Docker. Cada nova rede é criada a partir de um *drive*. Além dos *drives* das redes padrões, *bridge*, *none* e *host*, existem outros tipos de *drive* como *overlay* e *macvlan* (DOCKER, 2021).

3 TRABALHOS RELACIONADOS

Nesta seção, serão apresentados alguns estudos relacionados à este trabalho.

3.1 *Missing the forest for the trees: end-to-end AI application performance in edge data centers*

Em Richins *et al.* (2020), é descrito um aplicativo de reconhecimento de faces, fundamentado em Tensorflow[1], *Multi-task Cascaded Convolutional Neural Network* e FaceNet[2], para implantação em um *data center* de borda. Nesse trabalho, os autores discutem as implicações da aceleração dada aos algoritmos de Inteligência Artificial na infraestrutura do *data center* como um todo. Mais especificamente, avaliam os efeitos negativos da *IA tax*: termo adotado pelos autores para se referir aos processos não atribuídas aos algoritmos de Inteliência Artificial, como o substrato da comunicação, pré e pós-processamento.

A aplicação de reconhecimento implementada por Richins *et al.* (2020) é dividida em dois estágios independentes: ingestão/detecção, que se responsabiliza pelas etapas de decodificação, segmentação e detecção de faces; e identificação, que executa o reconhecimento de faces. Cada estágio é encapsulado em um contêiner Docker, além da existência de um outro contêiner específico para a comunicação entre esses dois estágios. A comunicação segue o padrão *publish-subscribe* implementado pelo Apache Kafka[3].

Richins *et al.* (2020) realizaram uma avaliação da latência do sistema por meio de um rastreamento do avanço de frames. Também avaliaram o impacto de acelerações teóricas no *kernel* da Inteligência Artificial em diversos graus de aceleração, por meio de uma emulação. Nessa emulação foram analisados apenas a largura de banda de rede e de armazenamento.

A semelhança clara entre o trabalho de Richins *et al.* (2020) e este trabalho é a modularização da Análise de Vídeo em contêineres Docker, com o uso de um contêiner intermediário para comunicação. Richins *et al.* (2020) construiram sua aplicação com TensorFlow, também semelhante à implementação do modelo de filtragem de placas feita neste trabalho.

[1] https://www.tensorflow.org/
[2] https://arxiv.org/pdf/1503.03832.pdf
[3] https://kafka.apache.org

3.2 *A video analytics-based intelligent indoor positioning system using edge computing for IoT*

Em Xie *et al.* (2018), uma arquitetura de *Edge Video Analytics* de quatro camadas de uso genérico é proposta. A arquitetura é avaliada por meio do projeto e implementação de um sistema de posicionamento inteligente de câmeras, com base no reconhecimento facial. O processo de Análise de Vídeo é dividido em módulos funcionais, sendo os contêineres Docker a unidade portadora dos módulos. Três módulos são definidos no sistema de posicionamento: um para segmentação em primeiro plano; um para detecção e reconhecimento de faces; e um para seleção do pixel de recurso, que determina o alvo do posicionamento das câmeras.

Xie *et al.* (2018) avaliaram a precisão do posicionamento das câmeras por meio de uma *Cumulative Distribution Function* e do erro médio da distância. Análises sobre o tempo de processamento de um frame em diferentes resoluções são realizadas.

A semelhança entre este trabalho e o de Xie *et al.* (2018) continua sendo a divisão do processo de Análise de Vídeo em módulos funcionais, executáveis em contêineres Docker. Aliás, a comunicação dos contêineres em Xie *et al.* (2018) não ocorre pelo Apache Kafka, diferente do planejado para este trabalho.

3.3 Enabling GPU-enhanced Computer Vision and Machine Learning research using containers

Em Michel e Burnett (2019) é apresentado uma imagem base de contêiner Docker para suporte aos pesquisadores de Análise de Vídeo, focando nas tecnologias de trabalho usualmente consumidas, como TensorFlow e OpenCV[4], e acesso à aceleração por NVIDIA[5] GPU, através dos *drivers* CUDA e CUDNN. Essa imagem é pública e disponibilizada no Dockerhub pela DMC (do inglês, *Data Machines Corp.*[6]), sendo nomeada de *cudnn_tensorflow_opencv*[7].

Michel e Burnett (2019) testaram essa imagem nos quesitos facilidade de criação de novas imagens e integração de algoritmos de terceiros. Na demonstração, o *framework* Darknet[8] e o algoritmo YOLOv3[9] foram escolhidos para detecção e classificação de objetos em tempo

[4] https://opencv.org/
[5] https://www.nvidia.com
[6] http://www.datamachines.io
[7] https://hub.docker.com/r/datamachines/cudnn_tensorflow_opencv
[8] https://github.com/pjreddie/darknet
[9] https://github.com/madhawav/YOLO3-4-Py

real.

A imagem Docker apresentada em Michel e Burnett (2019) foi fundamental para o desenvolvimento das imagens base deste trabalho, afinal ela forneceu suporta às tecnologias escolhidas para execução da Análise de Vídeo deste trabalho.

3.4 *Edge-centric video surveillance system based on event-driven rate adaptation for 24-hour monitoring*

Em Sakaushi *et al.* (2018) é proposto um sistema de videovigilância em tempo real, aplicado em uma plataforma de *Multi-Access Edge Computing*, com foco na detecção humana, qualidade de vídeo e redução de tráfego na rede, ambos orientados à eventos. O algoritmo *Deep Learning* para detecção empregado é o OpenPose[10]. Os autores implementaram duas funcionalidades excepcionais para a proteção do seu sistema: controle de aprimoramento de imagem e controle de qualidade de vídeo.

O controle de aprimoramento em Sakaushi *et al.* (2018) é definido por um *Support-Vector Machine* treinado para classificar quando uma sequência de vídeo necessita de aprimoramento de imagem ou não, cobrindo essencialmente os casos de pouca iluminação no ambiente de filmagem. O *dataset* de treino foi criado e rotulado pelos próprios autores. Quatro *features* são consideradas no momento da classificação de uma sequência: luminância no espaço de cores; brilho no espaço de cores; *root mean square contrast*; e entropia de uma imagem. O aprimoramento em si é feito em *hardware* pelo Red Super Eye G2[11].

Os autores de Sakaushi *et al.* (2018) também controlam a qualidade da taxa de codificação de vídeo, com base nos resultados do algoritmo de detecção e da necessidade de aprimoramento de imagem. Ou seja, vídeos com maior qualidade são transmitidos na rede apenas em situações de detecção ou aprimoramento, caso contrário, vídeos com menor qualidade são enviados para redução do tráfego na rede.

Sakaushi *et al.* (2018) desenvolveram um dos objetivos deste trabalho: controle de qualidade de vídeo. Contudo, o controle de qualidade neste trabalho atua na resolução do vídeo e não na compactação como em Sakaushi *et al.* (2018).

[10] https://github.com/CMU-Perceptual-Computing-Lab/openpose
[11] http://www.infotech-japan.co.jp/_src/sc762/RedSuperEyeG2_EN2017.pdf

3.5 Análise Comparativa

Os trabalhos relacionados expostos inspiraram bastante os objetivos específicos definidos para este trabalho, objetivos esses apresentados previamente na Seção 1.1.2. Na Tabela 1 é possível observar a proximidade dos trabalhos relacionados com os objetivos deste trabalho.

Tabela 1 – Associação dos trabalhos relacionados aos objetivos específicos

Trabalho	**Modularização do processo de análise**	**Algum controle de qualidade baseado em detecção**	**Algum modelo extra adicionado ao processo de análise**
Richins *et al.* (2020)	Sim, dois módulos	Não	Não
Xie *et al.* (2018)	Sim, três módulos	Não	Não
Michel e Burnett (2019)	Não	Não	Não
Sakaushi *et al.* (2018)	Não	Sim, qualidade de compactação	Sim, modelo de controle de aprimoramento
Este trabalho	**Sim, três módulos**	**Sim, qualidade de resolução**	**Sim, modelo de filtragem de placas**

Fonte: Elaborado pelo autor

Outra comparação de suma importância entre os trabalhos relacionados e este trabalho é apresentada na Tabela 2. Os pontos discutidos incluem: a intenção para que a Análise de Vídeo foi implementada; e os tipos de sensores usados para captura de dados. A abreviatura AVB significa Análise de Vídeo na Borda. Este trabalho se sobressai dos de Richins *et al.* (2020), Xie *et al.* (2018) e Michel e Burnett (2019) pelo fato desses estudos não realizarem qualquer controle de qualidade.

Tabela 2 – Visão geral da Análise de Vídeo dos trabalhos relacionados e o deste trabalho

Trabalho	**Propósito da Análise de Vídeo**	**AVB**	**Sensores utilizados**
Richins *et al.* (2020)	Reconhecimento de faces	Sim	Câmeras
Xie *et al.* (2018)	Reconhecimento de faces	Sim	Câmeras
Michel e Burnett (2019)	Detecção e classificação de objetos	Não	Não uitliza
Sakaushi *et al.* (2018)	Reconhecimento humano e controle de qualidade	Sim	Câmeras e sensores modais
Este trabalho	**Reconhecimento de placas e controle de qualidade**	**Sim**	**Câmeras**

Fonte: Elaborado pelo autor

Como última análise comparativa, os trabalhos de Richins *et al.* (2020) e Xie *et al.* (2018) são comparados ao projeto proposto por este trabalho no Quadro 1. Desta vez, a análise comparativa se concentra na divisão do processo de Análise de Vídeo em módulos e no tipo de comunicação entre os contêineres funcionais. É importante ressaltar que o termo comunicação indireta, indicada no Quadro 1, diz respeito somente à abordagem de contêiner intermediário implementada em Richins *et al.* (2020) com Apache Kafka. Logo, uma comunicação direta, no Quadro 1, significa que dois contêineres não dependem desse terceiro contêiner para se

comunicarem. A abordagem de comunicação indireta é melhor do que a abordagem direta, afinal o sistema de fila implementado pelo Apache Kafka garante uma maior independência e melhor divisão de papéis entre os produtores e consumidores no padrão *publish-subscribe* (RICHINS *et al.*, 2020; XIE *et al.*, 2018).

Quadro 1 – Análise comparativa da modularização do processo de análise entre os trabalhos de Richins *et al.* (2020), Xie *et al.* (2018) e este trabalho

Trabalho	**Comunicação**	**Módulo/Estágio 1**	**Módulo/Estágio 2**	**Módulo/Estágio 3**
Richins *et al.* (2020)	Indireta	Decodificação, segmentação e detecção	Reconhecimento	Não se aplica
Xie *et al.* (2018)	Direta	Decodificação, aprimoramento e segmentação	Detecção e Reconhecimento	Extração de pixel de recurso
Este trabalho	**Indireta**	**Decodificação e detecção**	**Filtragem**	**Reconhecimento**

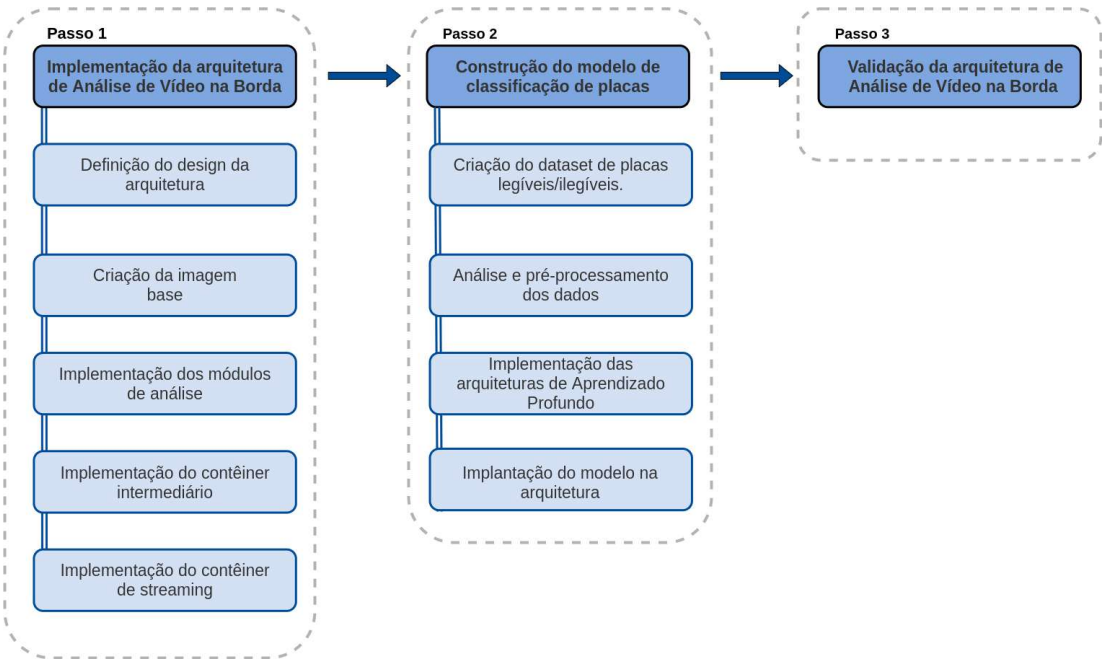
Fonte: Elaborado pelo autor

4 METODOLOGIA

Para que os objetivos propostos neste trabalho fossem alcançados, um conjunto de etapas foram adotadas, conforme ilustrado na Figura 8, descrito nas próximas seções e enumeradas a seguir.

1. Implementação da Arquitetura de AVB:
    a) Definição do *design* da arquitetura;
    b) Criação da imagem base;
    c) Implementação dos módulos de análise;
    d) Implementação do contêiner intermediário ;
    e) Implementação do contêiner de *streaming*.
2. Construção do modelo de classificação de placas:
    a) Criação do *dataset* de placas legíveis/ilegíveis;
    b) Análise e pré-processamento dos dados;
    c) Implementação das arquiteturas de Aprendizado Profundo;
    d) Implantação do modelo na arquitetura.
3. Validação da Arquitetura de AVB.

Figura 8 – Passos metodológicos adotados neste trabalho



Fonte: Elaborado pelo autor

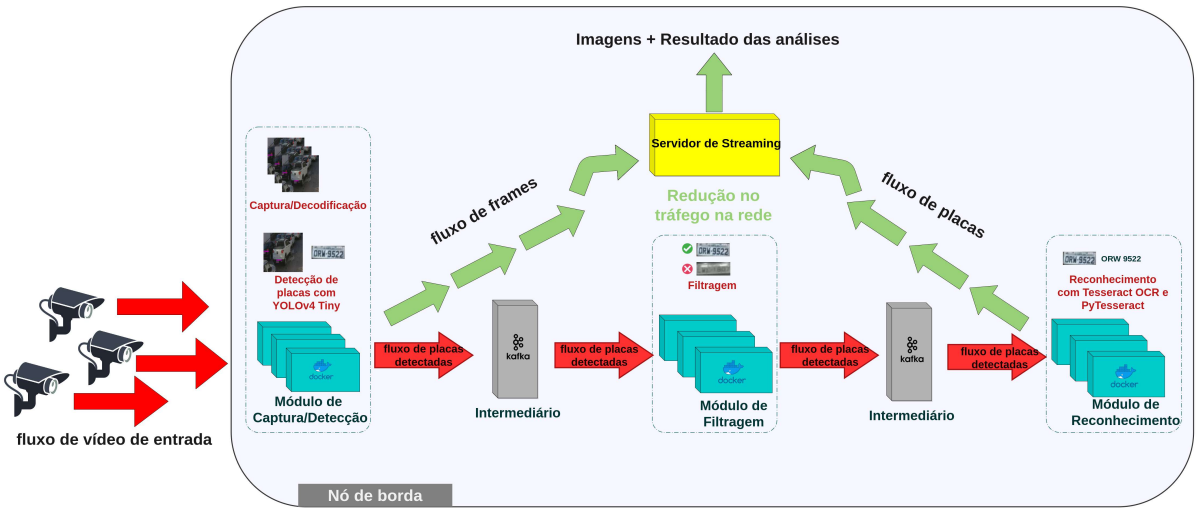
5 ARQUITETURA DE ANÁLISE DE VÍDEO NA BORDA

Nesta seção, o primeiro passo metodológico da Figura 8 é detalhado. Dessa forma, a definição das principais partes e fluxos de dados da arquitetura de AVB deste trabalho são descritos juntos aos seus passos de implementação.

5.1 Definição do *design* da arquitetura

A Figura 9 ilustra a arquitetura de AVB desenvolvida no projeto deste trabalho. No contexto deste trabalho, existem inúmeras câmeras de videovigilância espalhadas geograficamente pela cidade, de tal forma que cada uma delas produz fluxos de vídeo de forma regular e frequente para detecção e reconhecimento de placas na borda. Esses dados são consumidos por um dos nós de borda disponíveis na arquitetura de Computação de Borda. Um nó de borda dispõe de três tipos de contêineres Docker para efetuar seu serviço de análise, a saber, **contêiner de análise** ou **módulo de análise**, **contêiner intermediário** e **contêiner de *streaming***. Esses contêineres são representados na Figura 9 pelas cores azul, cinza e amarelo respectivamente. Portanto, existem três módulos de análise para cada serviço de análise. Em geral, para todos os serviços de análise existem um único contêiner intermediário e um único contêiner de *streaming*.

Figura 9 – Arquitetura de AVB da rede deste trabalho



Fonte: Elaborado pelo autor

Os módulos de análise são os integrantes da arquitetura responsáveis pela execução

das tarefas de detecção/reconhecimento de placas. Por outro lado, o contêiner intermediário e o contêiner de *streaming* atuam no processo de comunicação da arquitetura. Enquanto o contêiner intermediário é responsável pela transmissão de dados entre os módulos de análise, o contêiner de *streaming* fica encarregado de servir os dados para fora da arquitetura.

Ao todo existem três módulos de análise por serviço: MCD (do português, Módulo de Captura e Detecção), MF (do português, Módulo de Filtragem) e MR (do português, Módulo de Reconhecimento). Inicialmente um MCD captura e decodifica o fluxo de vídeo de entrada e, em seguida, executa um modelo de detecção de placas. Esse mesmo módulo tanto envia os *frames* capturados junto com os resultados da detecção para o contêiner de *streaming*, quanto envia a imagem recortada das placas para o próximo contêiner de análise, no caso, o chamado MF. O MCD executa essas duas etapas do processo de AV juntas, são elas decodificação e detecção, pois a decodificação é uma etapa muito simples na arquitetura proposta neste trabalho e a tentativa de separa-las apenas atrasou o fluxo de dados da aplicação.

Por sua vez, o MF é capaz de selecionar apenas os recortes de placa com texto de melhor qualidade para serem processados no MR, descartando os demais. Ao final, o MR executa a etapa de reconhecimento de caracteres nos recortes de placa restantes e envia o recorte junto com o resultado do reconhecimento para o contêiner de *streaming*.

Com isso, percebe-se a existência de dois fluxos de dados para o contêiner de *streaming*, um fluxo de *frames* partindo do MCD e outro fluxo de recortes de placa vindo do MR. Essa escolha de *design* tem o objetivo de tornar o resultado desses módulos mais independentes, além de permitir respostas mais rápidas no monitoramento em tempo real com detecção de placas. Dessa forma, o resultado do MCD pode ser acessado para monitoramento das câmeras sem a necessidade de esperar o resultado do MR que, por seu lado, é mais custoso.

Outra motivação que justifica esses dois fluxos para o contêiner de *streaming* é evitar que um *frame* inteiro percorra até o último módulo de análise ao invés de um recorte de placa. Afinal, a tentativa de guardar vários *frames* inteiros não só exigiu maior poder de armazenamento no próprio nó de borda, quanto também atrasou o fluxo de dados como um todo. Neste último, os módulos de análise necessitam escrever e ler grandes *frames* no contêiner intermediário.

Por fim, com tantas análises sendo realizadas em um nó de borda e o requisito de que essas análises sejam executadas em tempo real, a implementação da arquitetura de AVB deste trabalho necessitou trabalhar no cenário com disponibilidade de GPU.

5.2 Criação da imagem base

Dada a arquitetura AVB, ilustrada na Figura 9, o passo seguinte consiste na criação da imagem base que foi usada para a implementação dos contêineres. Como já dito na Seção 3.3, Michel e Burnett (2019) apresentaram a imagem pública *cudnn_tensorflow_opencv* para suporte à aceleração por GPU das tecnologias Tensorflow e OpenCV na NVIDIA GPU. Assim, nessa imagem da DMC os *drivers* da NVIDIA GPU e CUDNN são instalados no próprio contêiner. Isso somado ao NVIDIA Container Toolkit[1], que permite criar e executar os contêineres Docker acelerados por GPU.

O OpenCV é uma biblioteca de código aberto especializada em Visão Computacional, com centenas de módulos para processamento de imagens, AV, captura de vídeos, codecs de vídeo e entre outros auxiliares. Dentre esses módulos, existe um conjunto de módulos específico para aceleração por GPU chamado OpenCV GPU[2]. Todos os modelos de Aprendizado Profundo utilizados na arquitetura de AVB deste trabalho são executados no módulo do OpenCV GPU, chamado dnn[3], que suporta o uso de modelos de Aprendizado Profundo dos variados *frameworks* conhecidos, como Darknet, Caffe[4] e Tensorflow.

Ademais, para o uso do OpenCV GPU é preciso que o código-fonte seja baixado e compilado com a configuração ideal da NVIDIA GPU. Isso se deve ao fato de que o OpenCV GPU não pode ser instalado por um gerenciador de pacotes comum, pois geralmente não é disponibilizado por ele. Dessa forma, a imagem base de Michel e Burnett (2019) realiza todo esse processo, tornando esse etapa simples.

O Apêndice A mostra o código do Dockerfile usado para a construção da imagem base dos contêineres da arquitetura de AVB deste trabalho com o uso da imagem apresentada em Michel e Burnett (2019), além da instalação das demais dependências para o fluxo de dados da arquitetura.

5.3 Implementação dos módulos de análise

Com a criação da imagem base, o passo subsequente foi criar três novas imagens Docker referentes a cada um dos módulos de análise da arquitetura de AVB deste trabalho.

[1] https://github.com/NVIDIA/nvidia-docker/
[2] https://docs.opencv.org/2.4/modules/gpu/doc/gpu.html
[3] https://docs.opencv.org/master/d2/d58/tutorial_table_of_content_dnn.html
[4] https://github.com/BVLC/caffe

Cada uma dessas novas imagens executa uma aplicação Python para sua respectiva análise. O Apêndice B mostra um *template* para a criação da imagem de um módulo de análise genérico.

Cada aplicação Python pertencente a um dos contêineres da Figura 9, sejam módulos de análise ou não, torna-se configurável por meio de variáveis de ambiente definidas no momento da criação de uma instância. Por exemplo, uma instância do MCD sabe em qual URL de *stream* de vídeo ele deve trabalhar através da variável de ambiente CAMERA_CONFIG_SOURCE. A lista completa de todas as variáveis de ambiente de todos os contêineres da arquitetura de AVB deste trabalho pode ser acessada no Apêndice C.

Seguindo com as implementações, o MCD foi o primeiro módulo de análise a ser desenvolvido e também o módulo com mais funcionalidades atreladas. Além de capturar e decodificar o fluxo de vídeo de entrada por meio do próprio OpenCV, o MCD executa um modelo de detecção de placas de identificação de carros baseado no Yolov4-Tiny[5]. O Yolov4-Tiny é uma versão do Yolov4[6] para cenários de computação limitada, em que há uma troca de desempenho da versão original por velocidade de detecção. Este trabalho buscou apenas utilizar o modelo de detecção de placas, já treinado para o caso de uso específico deste trabalho. Assim, mais detalhes sobre seu processo de criação não faz parte do escopo deste trabalho.

Além de executar o modelo de detecção de placas, o MCD foi implementado para alterar a qualidade do *frame* antes do mesmo ser transmitido para o contêiner de *streaming*. Esse controle de qualidade atua conforme os resultados da detecção. Caso ao menos uma placa seja detectada pelo Yolov4-Tiny no *frame* decodificado, o *frame* é enviado para o contêiner de *streaming* com uma resolução pré-definida no momento da criação da instância do MCD. Caso contrário, ou seja, nenhuma placa no *frame* foi detectada, o OpenCV é usado para diminuir a resolução desse frame dividindo as dimensões pré-definidas por um chamado nível de redução de qualidade, especificado também no momento da criação da instância do MCD.

Passando para a implementação do MF, esse módulo se resume ao uso do modelo de classificação de placas, com implementação detalhada na Seção 6, com o OpenCV GPU. Dessa forma, uma placa classificada como de má qualidade ou ilegível é descartada e não prossegue no fluxo da arquitetura de AVB deste trabalho.

Finalmente, o MR executa um reconhecimento de placas por meio do Tesseract OCR[7]. O Tesseract é relativamente fácil de se instalar e de se utilizar em código Python, além de

[5] https://arxiv.org/abs/2011.04244
[6] https://github.com/kiyoshiiriemon/yolov4_darknet
[7] https://github.com/tesseract-ocr/tesseract

ser uma solução de Aprendizado Profundo de código aberto com boa precisão em ambientes controlados. Assim como o modelo de detecção de placas, a implementação do reconhecimento de placas não faz parte do escopo deste trabalho. Devido a isso, na validação da arquitetura os resultados do modelo de reconhecimento são relativamente baixos. Afinal, nenhum pré-processamento nas placas é realizado ou configuração sofisticada a fim de melhorar ainda mais o desempenho do Tesseract OCR. Portanto, esse módulo está aberto a melhorias em trabalhos futuros assim como a detecção no MCD.

5.4 Implementação do contêiner intermediário

A lógica de construção do contêiner intermediário teve como base o uso do Apache Kafka, assim como no trabalho de Richins *et al.* (2020). O uso do Apache Kafka como coordenador é justificado pelo seu baixo uso de recursos, seu sistema de filas e seu uso comum para *streaming* de dados (RICHINS *et al.*, 2020). O Kafka implementa um *streaming* de eventos de ponta a ponta por meio do padrão *publish–subscribe*, na qual clientes podem ler, gravar e processar fluxos de eventos em um *cluster* ou servidor. Essa comunicação entre clientes e *clusters* é através do protocolo TCP/IP.

O contêiner intermediário é simbolicamente um *cluster* do Kafka e sua dependência, o Apache Zookeeper[8]. O Kafka utiliza o Zookeeper basicamente para sincronizar as suas configurações. No Apêndice D, o arquivo YAML para levantamento dos serviços que formam o contêiner intermedário é mostrado. As imagens Docker utilizadas foram a confluentinc/cp-kafka[9] e a confluentinc/cp-zookeeper[10], ambas disponíveis no Docker Hub.

Para que os módulos de análise pudessem se comunicar por intermédio do Kafka, o seu cliente Python foi usado. O Kafka trabalha com três conceitos fundamentais, sendo eles: tópicos, que são as regiões de armazenamento no *cluster* referentes a um dos fluxos; produtores, responsáveis por inserirem dados em fluxos de diferentes tópicos; e consumidores, capazes de lerem e processarem os dados de um único tópico. Assim, o cliente Python do Kafka permitiu a criação de instâncias para os papéis de produtor e consumidor nos três módulos de análise.

Portanto, um único serviço de análise na arquitetura de AVB deste trabalho precisaria de dois tópicos no *cluster* do Kafka. No primeiro tópico, o MCD faz papel de produtor e envia uma mensagem para o MF que, por sua vez, atua como consumidor do tópico. A mensagem

[8] https://zookeeper.apache.org/
[9] https://hub.docker.com/r/confluentinc/cp-kafka/
[10] https://hub.docker.com/r/confluentinc/cp-zookeeper

enviada é no formato JSON[11], contendo o recorte de placa convertido em vetor de caracteres e o tempo de captura do *frame*. No segundo tópico, o MF assume o papel de produtor e reenvia a mensagem recebida para o MR, tendo esse último adotado o papel de consumidor do tópico.

Vale ressaltar que, no Kafka, quando um dado é consumido em um tópico, o mesmo é excluído do *cluster*. Por isso, quando o MF decide não passar adiante um recorte de placa, por meio do modelo de classificação de placas discutido na Seção 6, o mesmo cumpre o seu papel de descarte de placas com texto ilegível.

Além disso, os tópicos de comunicação entre contêineres de análise deveriam ter nomes únicos. Para que isso fosse possível, os tópicos foram montados a partir das variáveis de ambiente NVR_ID e CAMERA_CONFIG_ID que identificam juntas um fluxo de vídeo de entrada na arquitetura. À vista disso, a comunicação entre uma instância do MCD e sua respectiva instância do MF ocorria pelo tópico com nome queueing.CAPTURA_DETECCAO_to_FILTRAGEM concatenado com os valores das duas variáveis de ambiente citadas anteriormente. O mesmo ocorria na comunicação entre a instância do MF com o MR, porém o prefixo do tópico era alterado para queueing.FILTRAGEM_to_RECONHECIMENTO.

5.5 Implementação do contêiner de *streaming*

Por último, o contêiner de *streaming* foi implementado para enviar o fluxo de *frames* e o fluxo de placas do processo de análise realizado na arquitetura de AVB deste trabalho para um cliente web escrito em JavaScript. Isso se dá por meio do uso do Python WebSockets como protocolo de comunicação. Dessa forma, o contêiner *streaming* funciona como um servidor construído em Flask-SocketIO[12] que atua como um misto entre um servidor HTTP e um servidor SocketIO.

A inspiração dessa dinâmica veio de um tutorial online[13] que mostra como construir um servidor de *streaming* de vídeo simples para uma aplicação de AV. Aplicar algo semelhante na arquitetura de AVB deste trabalho garantiria tanto uma independência entre os módulos de análise e o servidor de *streaming*, quanto uma forma mais fácil de validar a solução.

Caso o foco da arquitetura de AVB deste trabalho fosse enviar apenas os resultados para outra aplicação Python, a conexão TCP aberta pelo Apache Kafka seria suficiente. Porém, este trabalho optou por algo mais geral e focou nas páginas web que trabalham majoritariamente

[11] https://www.json.org/json-en.html
[12] https://flask-socketio.readthedocs.io/en/latest/
[13] https://learn.alwaysai.co/build-your-own-video-streaming-server-with-flask-socketio

com JavaScript.

Logo, o MC e o MR atuam como clientes de análise do servidor implantado no contêiner de *streaming*. Um objeto do cliente Python WebSocketIO é instanciado em cada cliente de análise. Como já mencionado, no MCD o cliente envia o fluxo de *frames* no formato JPEG junto com o tempo de captura para o servidor, enquanto que no MR o cliente envia os recortes de placa com o tempo de captura e o texto resultante do reconhecimento.

O ideal é que o servidor possua também clientes *web* para consumo dos resultados e exposição dos mesmos por meio de uma página *web*. Por exemplo, no processo de validação da arquitetura, comentada na Seção 7, um cliente *web* foi construído para lidar com um serviço de análise e exibir o fluxo de *frames* e o fluxo de placas dos dois clientes de análise.

6 MODELO DE CLASSIFICAÇÃO DE PLACAS

Como mencionado na Seção 5.3, o MF emprega um modelo de classificação de placas, também conhecido como modelo de filtragem de placas, no meio da arquitetura de AVB deste trabalho. A principal motivação da existência desse modelo é que os recortes de placa resultantes do MCD sofrem por distorções que retiram, sobretudo, a legibilidade do texto dessas placas. A fim de evitar que placas com texto ilegível e sem chances de reconhecimento sejam processadas no MR, o segundo passo metodológico deste trabalho foi dedicado à construção desse modelo de filtragem. Para isso, arquiteturas de Aprendizado Profundo foram construídas e avaliadas com base nos algoritmos Resnet50, MobileNetv2, InceptionV3 e EfficientNetB0, apresentados brevemente na Seção 2.2.2 sobre Transferência de Aprendizado.

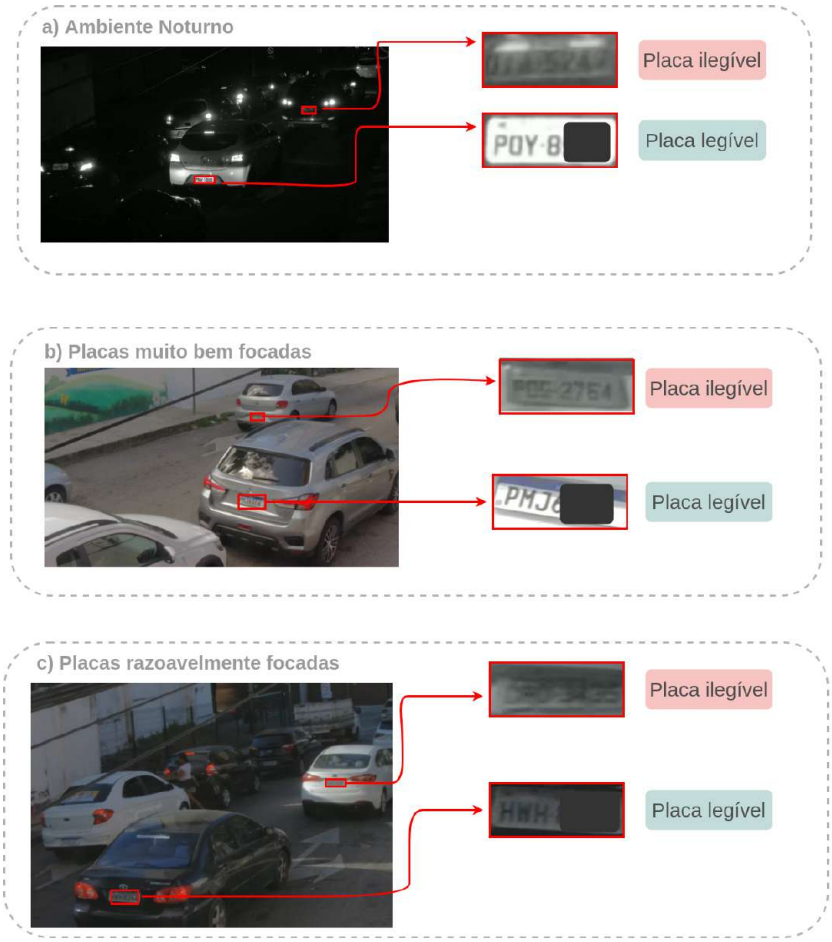
6.1 Criação do *dataset* de placas legíveis/ilegíveis

Nesta etapa, os dados foram coletados de gravações em vídeo das ruas da cidade de Fortaleza, no estado do Ceará. Cada vídeo possui duração aproximada de uma hora. Visando fornecer uma amostra significativa, os vídeos escolhidos cobriram três cenários distintos. O primeiro cenário foi o ambiente noturno, na qual se observou muitas placas com texto legível graças a iluminação dos faróis. O segundo cenário coberto foi de um ambiente em que as placas estavam muito bem focadas pelas câmeras, onde notou-se muitas placas com texto legível de boa qualidade. Por fim, o último cenário foi aquele de placas razoavelmente focadas pelas câmeras, em que ocorreu um misto de placas com texto ilegível e placas com texto legível, mas de qualidade razoável. A Figura 10 faz alusão a esses cenários encontrados.

Assim, os vídeos foram submetidos ao modelo Yolov4-Tiny para detecção de placas, o mesmo usado no MCD na arquitetura de AVB deste trabalho. Dessa forma, os recortes de placa foram coletados, cada recorte com dimensão de 200x80 *pixels*. Dessas placas detectadas, muitas eram de textos ilegíveis ou placas repetidas. Esse último caso é consequência dos semáforos próximos aos locais de gravação que permitiam que os mesmos carros ocupassem as imagens por mais de um minuto. Por isso, houve a necessidade de que todas as placas detectadas fossem analisadas e comparadas uma a uma.

Após isso, 520 recortes de placas foram selecionados para compor o *dataset* para a criação do modelo de classificação de placas deste trabalho. Essas 520 placas foram classificadas em duas classes. Uma classe titulada como legível, representando as placas com texto legível

Figura 10 – Coleta de dados para o modelo de classificação de placas



Fonte: Elaborado pelo autor

de qualidade boa e média, e outra classe chamada ilegível, representando aquelas placas com texto ilegível. Essa classificação é ilustrada também na Figura 10. Outra informação importante é que as 520 imagens foram selecionadas com foco em um *dataset* com equilibrio de amostras entre classes, ou seja, um número de imagens similar em ambas as classes. Ao todo, existem 220 imagens da classe legível e 300 imagens da classe ilegível. O Apêndice E mostra cinco exemplos de cada classe.

6.2 Análise e pré-processamento dos dados

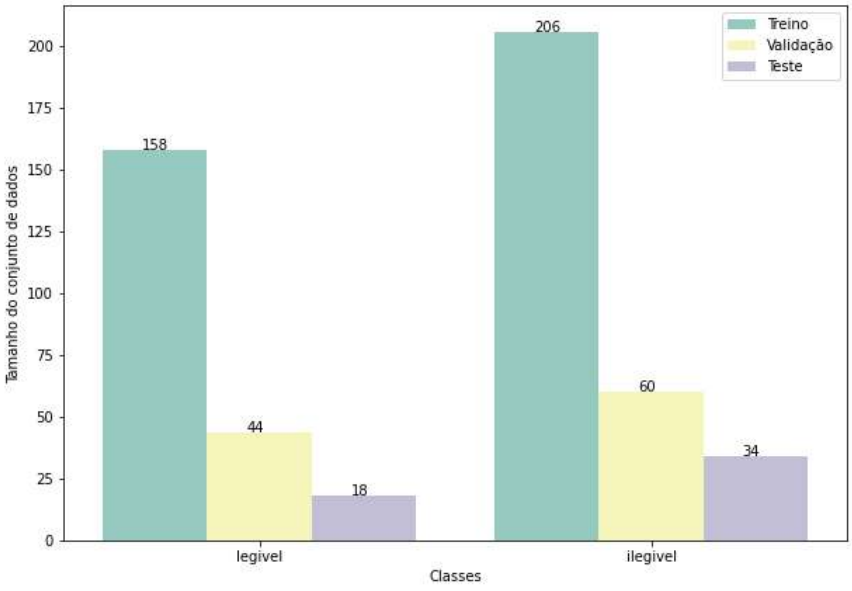
Com a criação do *dataset*, a próxima etapa foi analisar e preparar os dados para a construção das arquiteturas de Aprendizado Profundo. A princípio, foi realizado um *upload* do *dataset* para a plataforma online Roboflow[1]. O Roboflow auxilia na organização e na preparação

[1] https://roboflow.com/

dos dados, facilitando atividades de anotação de classes, pré-processamento, *data augmentation*, compartilhamento de *datasets* e entre outras atividades. Assim, no momento do *upload* do *dataset* para a plataforma foi possível fazer a separação aleatória dos dados em 70% para conjunto de treino, 20% para conjunto de validação e 10% para conjunto de teste. O próprio Roboflow prover um *link* para *download* do *dataset* em uma estrutura de arquivos adequada para um problema de classificação de imagens.

Desses três conjuntos de dados criados, o conjunto de treino é utilizado para de fato treinar o modelo de Aprendizado Profundo. Por outro lado, o conjunto de validação serve para avaliar o aprendizado do modelo ao longo do seu treinamento. Ao contrário do conjunto de validação, o conjunto de teste é um conjunto a parte para avaliar o modelo com dados desconhecidos após o treinamento, simulando algo próximo do que seria o desempenho do modelo em um cenário real. Na Figura 11 é possível observar a distribuição das imagens nesses conjuntos de dados, na qual 364 imagens são utilizadas para treinamento, 104 imagens para validação e 52 imagens para teste.

Figura 11 – Distribuição de imagens nos conjuntos de dados treino/validação/teste



Fonte: Elaborado pelo autor

Em seguida, todos as imagens de placas foram redimensionadas para 224x224 *pixels* de forma a adequá-las à camada de entrada das arquiteturas de Aprendizado Profundo do

Resnet50, MobileNetv2, InceptionV3 e EfficientNetB0. Além disso, cada *pixel* das imagens foi dividido por 255, sendo esse um passo de pré-processamento padrão quando se trabalha com imagens.

6.3 Implementação das arquiteturas de Aprendizado Profundo

O ambiente de implementação dos modelos de Aprendizado Profundo utilizado neste trabalho foi o *Google Colaboratory*[2]. O *Google Colaboratory* utiliza um ambiente de *notebooks* Jupyter[3] que é executado na nuvem. Esses *notebooks* são aplicações *web* que contém ao mesmo tempo código interativo e textos explicativos. Nessa plataforma, nenhuma configuração do ambiente é necessária e o auxílio de GPU para o treinamento das arquiteturas é gratuito. O ambiente usado no *Google Colaboratory* tinha disponível um processador Intel Xeon 2.30GHz, pouco mais de 12GB de memória e uma placa de vídeo NVIDIA Tesla P100-PCIE.

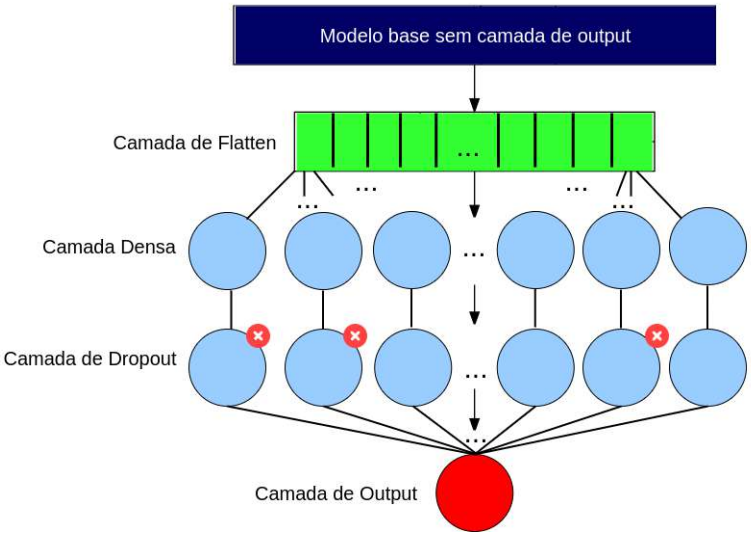
Utilizou-se o Keras para prover uma implementação rápida e simplificada das arquiteturas de Aprendizado Profundo. O Keras é uma API de Aprendizado Profundo de alto nível incorporada no TensorFlow 2.0 ou maior, contando com inúmeros blocos de construção para camadas, funções de ativação e otimizadores.

Seguindo com as fases de implementação das arquiteturas de Aprendizado Profundo, elas se embasam na técnica de Transferência de Aprendizado, explicada na Seção 2.2.2. Por conseguinte, como já explicado na Seção 2.2.2, as arquiteturas Resnet50, MobileNetv2, InceptionV3 e EfficientNetB0 foram testadas como modelo base nessa técnica. Assim sendo, foi retirada a última camada dessas arquiteturas, a camada de *output* referente ao problema de classificação original daquela arquitetura, e adicionadas novas camadas como indicado na Figura 12. As camadas adicionadas foram, nessa ordem: uma camada de *flatten*; uma camada densa com 256 neurônios; uma camada de *dropout* para 20% dos neurônios; e uma camada Densa com um único neurônio como camada de *output*.

Outras variações foram investigadas, como aplicação de camadas de *pooling* e mais inserções de camadas densas seguidas por camadas de *dropout*. Porém, as camadas de *pooling* atrapalharam o aprendizado do modelo e seu uso sem a camada de *flatten* ocasionou um imenso problema de sobreajuste de dados. A explicação mais plausível disso seria o fato da camada de *pooling* suavizar o efeito de distorções (RAWAT; WANG, 2017). Como o problema de

[2] https://colab.research.google.com/
[3] https://jupyter.org/

Figura 12 – Padrão das arquiteturas de Aprendizado Profundo para a classificação de placas



Fonte: Elaborado pelo autor

classificação de placas deste trabalho tem foco na distorção das placas, logicamente esse tipo de camada atrapalharia nos resultados. Por fim, o uso de mais de uma camada densa não trouxe melhoras no desempenho do modelo durante o treinamento, apenas atraso para se atingir uma boa avaliação e uma maior quantidade de parâmetros para se armazenar na arquitetura.

A camada densa mostrada na Figura 12 foi projetada com a função de ativação ReLU (do inglês, *Rectified Linear Unit*). Uma função de ativação é a unidade responsável pela transformação não-linear nos dados de entrada de um neurônio. Dentre as funções de ativações, a ReLu é a mais utilizada em imagens e simplesmente retorna zero caso a entrada seja negativa ou o valor original caso contrário (ACADEMY, 2019).

Ainda na camada densa, a regularização L2 é disposta para tentar combater o sobreajuste dos modelos. A arquitetura ResNet50 foi a que mais se beneficiou de tal mecanismo. O fator de regularização foi fixado em 0,001.

Citando os detalhes da camada de *output*, a mesma detém exclusivamente um neurônio devido ao fato de se tratar de um problema de classificação de duas classes: legível e ilegível. Isso permite reduzir o problema a um caso de classificação binária. Por isso, a função de ativação *sigmoid* foi recorrida nessa camada, afinal, essa função modela melhor esse

comportamento ao assumir valores entre 0 e 1. Com a *sigmoid*, a saída da camada de *output* consiste na probabilidade de uma amostra ser da classe legível (ACADEMY, 2019).

Os modelos são compilados utilizando o otimizador Adam, uma versão superior do gradiente descendente que auxilia no ajuste dos parâmetros ao decorrer da fase de treinamento das arquiteturas de Aprendizado Profundo. A taxa de aprendizado foi estipulado em 0,0001 para esse otimizador. Outros hiperparâmetros decididos nesse momento foram: o número de épocas, ou número de iterações de treinamento, com valor máximo igual a 100; e o tamanho do lote com valor 30, referindo-se ao número de exemplos de treinamento usados em uma iteração.

Para se prevenir contra o sobreajuste, a técnica da parada precoce, ou *early stopping*, foi aplicada. Na parada precoce, o número ideal de épocas de treinamento é encontrado por intermédio de um monitoramento do desempenho do modelo em cada época. Assim, se porventura esse desempenho não apresentar melhoras ao longo de um valor pré-definido de épocas, o treinamento é encerrado (ACADEMY, 2019). No Keras, a parada precoce é implementada como uma *callback* fornecida como parâmetro de treinamento. Logo, ficou definido um intervalo de 15 épocas, chamado de paciência da parada precoce, para a melhora de no mínimo 0,001 na acurácia do conjunto de validação ao decorrer da fase de treinamento dos modelos.

A Tabela 3 resume os parágrafos anteriores ao mostrar os valores de hiperparâmetros e das demais configurações de treinamento dos modelos de Aprendizado Profundo construídos. Todos esses valores foram definidos após testes com outros valores, durante o próprio treinamento das redes.

Tabela 3 – Hiperparâmetros e outras configurações de treinamento das arquiteturas de Aprendizado Profundo experimentadas

Hiperparâmetro/Configuração	**Valor**
Tamanho do lote	30
Taxa de aprendizado	0,0001
Número de épocas	100
Paciência da parada precoce	15
Mínimo de melhora da parada precoce	0,001

Fonte: Elaborado pelo autor

Apenas as quatro últimas camadas das quatro arquiteturas baseadas no ResNet50, InceptionV3, MobileNetv2 e EfficientNetB0 foram treinadas, justamente as camadas adicionadas por este trabalho. Afinal, detectou-se a presença de sobreajuste ao treinar todas as camadas das arquiteturas. Nem mesmo a técnica de *data augmentation* foi eficiente para lidar com essa situação, descartando a possibilidade do responsável ser a baixa quantidade de dados de treino.

Vale salientar que as quatro arquiteturas bases foram baixadas diretamente pelo Keras, já treinadas com os pesos do *dataset* ImageNet[4]. O ImageNet é um vasto *dataset* com milhões de imagens distribuídas em milhares de classes. Logo, tornou-se curioso saber como essas arquiteturas iriam se comportar no *dataset* construído neste trabalho em um problema cujo foco era, de certo modo, na classificação da qualidade das imagens.

6.4 Implantação do modelo na arquitetura

Com base nos experimentos realizados sobre as arquiteturas de Aprendizado Profundo construídas na subseção anterior, uma dentre as quatro arquiteturas foi escolhida para ser implantada no MF da arquitetura de AVB deste trabalho, apresentada anteriormente na Figura 9. O resultado que motivou a escolha pode ser acessado na Seção 8.1.

Para que o modelo de classificação de placas seja executado no módulo dnn do OpenCV GPU, torna-se necessário exportar o gráfico do modelo em um arquivo *protobuf*, com extensão .pd. Além dele, um outro arquivo de configuração extra baseado no .pd, com extensão .pbtxt, é requisitado. Criar esses arquivos se resumiu em congelar o modelo e em seguida otimizar o mesmo. No processo de congelamento, as variáveis do modelo foram convertidas em constantes e salvas em seu gráfico. Para otimizar o modelo, partes do modelo que eram necessárias apenas no treinamento foram deletadas, como no caso da camada de *dropout*. Tudo isso foi feito utilizando funções do próprio Tensorflow.

Com esses arquivos, o MF se tornou capaz de executar o modelo de Aprendizado Profundo em sua aplicação, tornado assim a arquitetura de AVB apta para a fase de validação, descrita na Seção 7 seguinte.

[4] http://www.image-net.org/

7 VALIDAÇÃO

A seguir, o cenário de validação da arquitetura de AVB deste trabalho, aquela detalhada na Figura 9, é descrito. A arquitetura teve todos os seus contêineres executados em uma máquina para a validação da arquitetura em si, ou seja, executados para demonstrar que a solução implementada neste trabalho de fato funciona e cumpre integralmente seus objetivos específicos, listados na Seção 1.1.2. Para isso, uma máquina exclusiva de teste foi utilizada para fazer alusão a um servidor de borda. Trata-se de um *notebook* Acer Aspire Nitro 5, com processador Intel Core I5-9300H, 8GB de memória RAM, uma GPU NVIDIA GeForce GTX 1050 e um SO Ubuntu 18.04. O servidor de borda possuía instalado o Docker versão 20.10.3 e o Docker- Compose versão 1.26.2.

Vale ressaltar que, no processo de validação, este trabalho considerou apenas um fluxo de vídeo por vez no servidor de borda. Ou seja, somente um serviço de análise foi requisitado ao servidor de borda durante os testes. Isso ocorreu em virtude da implementação feita do contêiner de *streaming* até o momento da escrita deste trabalho. Em suma, esse contêiner se encontra limitado a um cliente *web* capaz de exibir somente um *feed* de vídeo com suas placas detectadas. A evolução desse contêiner de *streaming* para lidar com múltiplos clientes *web* e, por consequência, com múltiplos fluxos de vídeo fica como trabalho futuro.

O Apêndice F mostra em detalhes a interface do cliente *web* do contêiner de *streaming*. Em geral, essa interface é dividida em duas partes: um *feed* de vídeo que mostra o resultado do fluxo de *frames* oriundo do MCD; e um painel com os resultados vindos do MR, listando todas as placas detectadas com seus respectivos textos reconhecidos.

Durante o processo de validação, um fluxo de vídeo de entrada foi reproduzido a partir de vídeos armazenados localmente no próprio servidor de borda. Para esse fim, bastou aplicar o mapeamento de volume oferecido pelo Docker. Assim, uma pasta com vídeos de gravação das ruas de Fortaleza, no estado do Ceará, era compartilhada com o MCD. Logo, o valor da variável de ambiente CAMERA_CONFIG_SOURCE recebia o caminho do vídeo que se desejava aplicar o processo de análise da arquitetura de AVB deste trabalho. Vídeos representativos dos três cenários apresentados na Seção 6.1 foram testados.

Finalmente, uma rede Docker chamada *arquitetura-network* foi criada para a comunicação entre os contêineres. Essa rede é derivada do *drive* do tipo *bridge*. Devido ao fato do Docker-Compose ainda não possuir suporte ao NVIDIA Container Toolkit, cada contêiner da arquitetura de AVB deste trabalho foi criado um a um por linhas de comando no terminal. Porém,

visando uma maior facilidade de entendimento do leitor, o Apêndice D mostra um arquivo YAML para levantamento dos contêineres do experimento sem aceleração por GPU.

Ao todo, 6 cenários de experimentação distintos foram verificados. São eles:

- **Cenário noturno** - Execução da arquitetura de AVB em vídeo com ambiente noturno, sem controle de qualidade de resolução e sem filtragem de placa;
- **Cenário noturno com otimizações** - Execução da arquitetura de AVB em vídeo com ambiente noturno, com controle de qualidade de resolução e com filtragem de placa;
- **Cenário com foco alto** - Execução da arquitetura de AVB em vídeo com ambiente de placas muito bem focadas, sem controle de qualidade de resolução e sem filtragem de placa;
- **Cenário com foco alto e otimizações** - Execução da arquitetura de AVB em vídeo com ambiente de placas muito bem focadas, com controle de qualidade de resolução e com filtragem de placa;
- **Cenário com foco razoável** - Execução da arquitetura de AVB em vídeo com ambiente de placas razoavelmente focadas, sem controle de qualidade de resolução e sem filtragem de placa;
- **Cenário com foco razoável e otimizações** - Execução da arquitetura de AVB em vídeo com ambiente de placas razoavelmente focadas, com controle de qualidade de resolução e com filtragem de placa;

8 RESULTADOS

Nesta seção são comentados os resultados atingidos durante as etapas de criação do modelo de classificação de placas, Seção 6, e validação da arquitetura de AVB, Seção 7.

8.1 Resultados dos experimentos nos modelos de Aprendizado Profundo para a tarefa de classificação de placas

Os quatro modelos construídos na Seção 6.3, aqueles baseados na arquitetura do ResNet50, InceptionV3, MobileNetv2 e EfficientNetB0, foram avaliados pelas métricas de avaliação citadas na Seção 2.2.3, a saber, acurácia e *f-measure*. Uma terceira métrica também foi consultada, a chamada *log loss*. A *log loss* mede a incerteza da previsão do modelo apoiada no quanto a previsão varia do valor real da classe. Na Tabela 4, todos os resultados das métricas de avaliação na etapa de treinamento e validação dos modelos construídos são expostos.

Tabela 4 – Acurácia, *log loss*, e *f-measure* das arquiteturas de Aprendizado Profundo ao final das épocas de treinamento

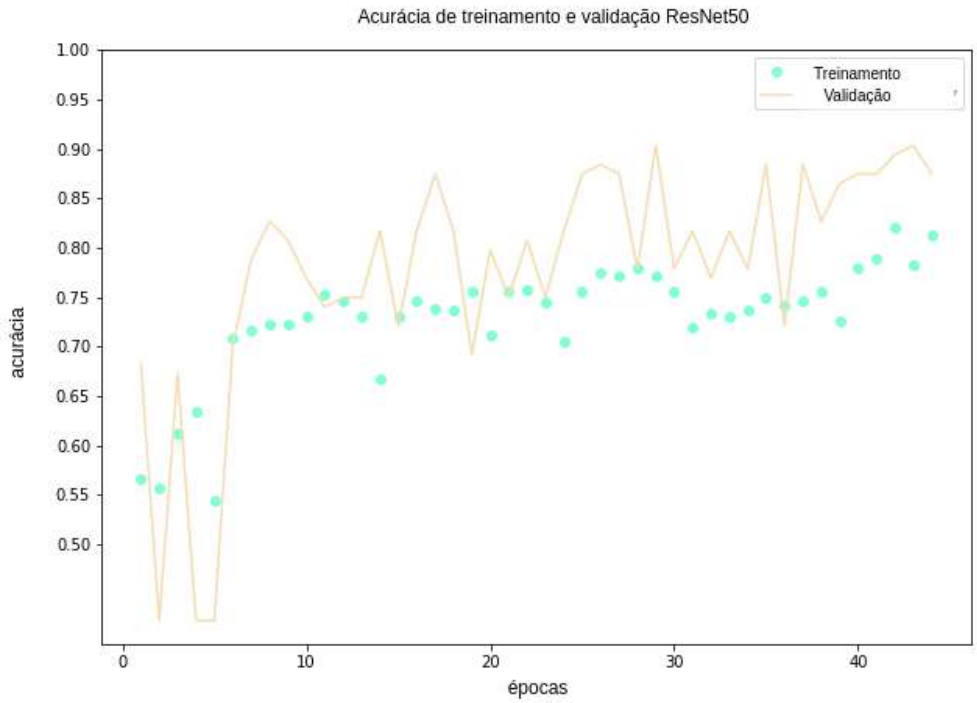
Arquitetura base	**Acurácia Treino**	**Acurácia Validação**	***Log loss* Treino**	***Log loss* Validação**	***F-measure* Treino**	***F-measure* Validação**	**Épocas**
ResNet50	0,8478	0,8750	0,3965	0,3737	0,8168	0,8395	44
InceptionV3	1,0000	0,9712	0,0465	0,1680	1,0000	0,9670	21
MobileNetv2	**1,0000**	**0,9904**	0,0303	0,0488	1,0000	0,9887	21
EfficientNetB0	0,5770	0,5769	0,7198	0,7235	$0{,}0000e^{+0}$	$0{,}0000e^{+0}$	16

Fonte: Elaborado pelo autor

A arquitetura baseada no ResNet50 teve um resultado mediano, 84,70% de acurácia no conjunto de treino e, de maneira surpreendente, 87,50% de acurácia no conjunto de validação, indicando que o modelo se saiu melhor nesse último conjunto de dados. Também foi o modelo que mais precisou de épocas de treino, foram 44 épocas para alcançar esse desempenho. A Figura 13 mostra o gráfico de comportamento da acurácia do modelo ao longo das épocas de treino. Reparou-se que, conforme o modelo melhorava sua acurácia no conjunto de treinamento, uma ocilação ocorria na acurácia do conjunto de validação. Isso fica evidente na Figura 13, em que a acurácia de validação diminuía drasticamente formando picos no gráfico.

O segundo modelo avaliado, InceptionV3, obteve acurácia máxima no conjunto de treino e a segunda melhor acurácia no conjunto de validação com seus 97,12% de acertos. Diferente do que aconteceu com o ResNet50, o modelo alcançou desempenho excelente já nas primeiras épocas, parando seu treinamento em 21 épocas graças a técnica da parada precoce.

Figura 13 – Acurácia do modelo de classificação de placas com a arquitetura Resnet50 ao longo das épocas de treino



Fonte: Elaborado pelo autor

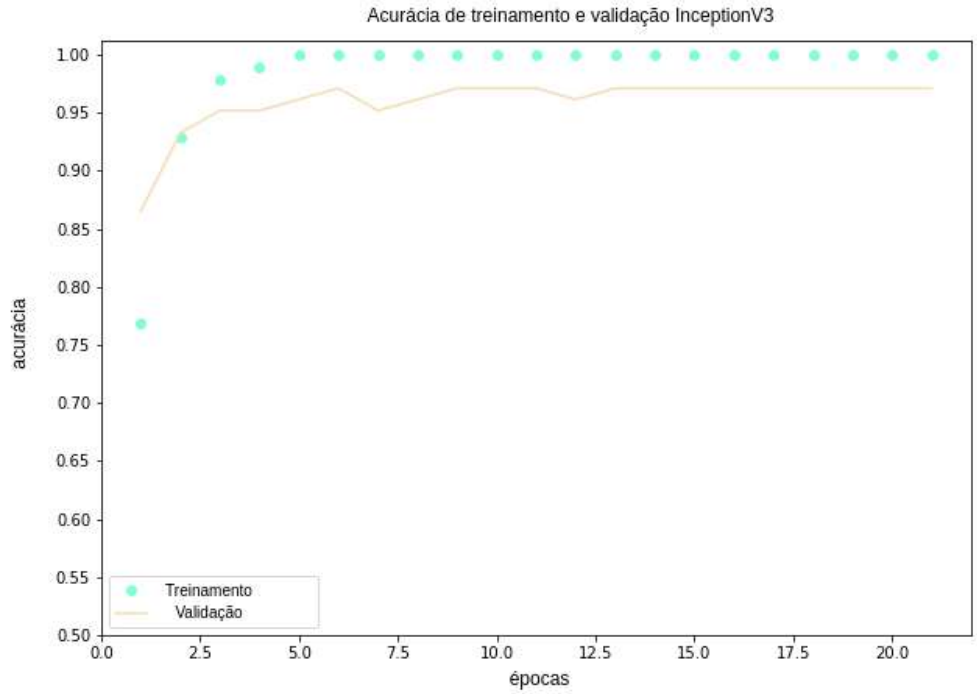
Isso mostra que a classificação foi bem mais fácil para esse modelo do que para o anterior. O gráfico da Figura 14 ilustra a acurácia do modelo ao decorrer das épocas de treinamento.

O MobileNetv2 foi a terceira arquitetura avaliada, atingindo resultados ainda melhores do que o InceptionV3. Com 100% de acurácia no conjunto de treino e 99,04% no conjunto de validação, alcançando o melhor resultado entre as quatro arquiteturas experimentadas. Como mostra o gráfico da Figura 15, esse modelo alcançou um ótimo desempenho tão rapidamente quanto o InceptionV3.

A última arquitetura experimentada, o EfficientNetB0, não conseguia aumentar seu desempenho ao decorrer das épocas de treinamento. Por isso, em 16 épocas consecutivas o modelo não conseguiu evoluir a acurácia de 57,70%. Este trabalho não investiu em melhorar tal performance devido aos bons resultados alcançados pelas três arquiteturas anteriores. Além disso, torna-se curioso notar que os modelos mais profundos, ResNet50 e EfficientNetB0, obtiveram um desempenho pior que os modelos mais rasos, InceptionV3 e MobileNetv2.

Ao fim da fase de treinamento, os dois melhores modelos foram submetidos a

Figura 14 – Acurácia do modelo de classificação de placas com a arquitetura InceptionV3 ao longo das épocas de treino



Fonte: Elaborado pelo autor

avaliação com os dados de teste. A Tabela 5 resume a repercussão desses modelos no novo conjunto de dados. Observa-se que o MobileNetv2 continua na liderança, obtendo 100% de acurácia no conjunto de teste. Lembrando que fatores como conjunto de teste com apenas 52 amostras de dados e conjunto de teste formado por amostras muito semelhantes, seja pelo ângulo das placas ou pela quantidade pequena de ruas de onde foram coletados os dados, devem ter ajudado o modelo a atingir um resultado de 100% no conjunto de teste.

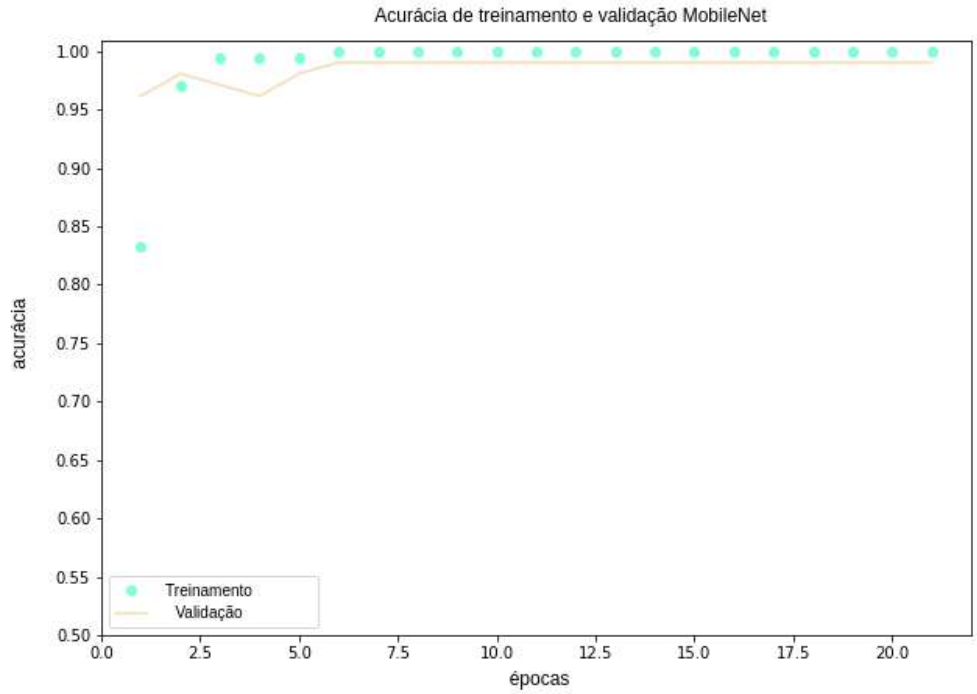
Tabela 5 – Acurácia, *log loss* e *f-measure* das duas melhores arquiteturas de Aprendizado Profundo no conjunto de dados de teste

Arquitetura base	**Acurácia**	***Log loss***	***F-measure***
InceptionV3	0.9615	0,1104	0,9473
MobileNetv2	**1,0000**	0,0339	1,0000

Fonte: Elaborado pelo autor

Com base nesses resultados, a arquitetura baseada no MobileNetv2 foi implantada no MF da arquitetura de AVB deste trabalho, como direcionado na Seção 6.4, e utilizada na etapa de validação, que inclusive tem seus resultados considerados a seguir.

Figura 15 – Acurácia do modelo de classificação de placas com a arquitetura MobileNetv2 ao longo das épocas de treino



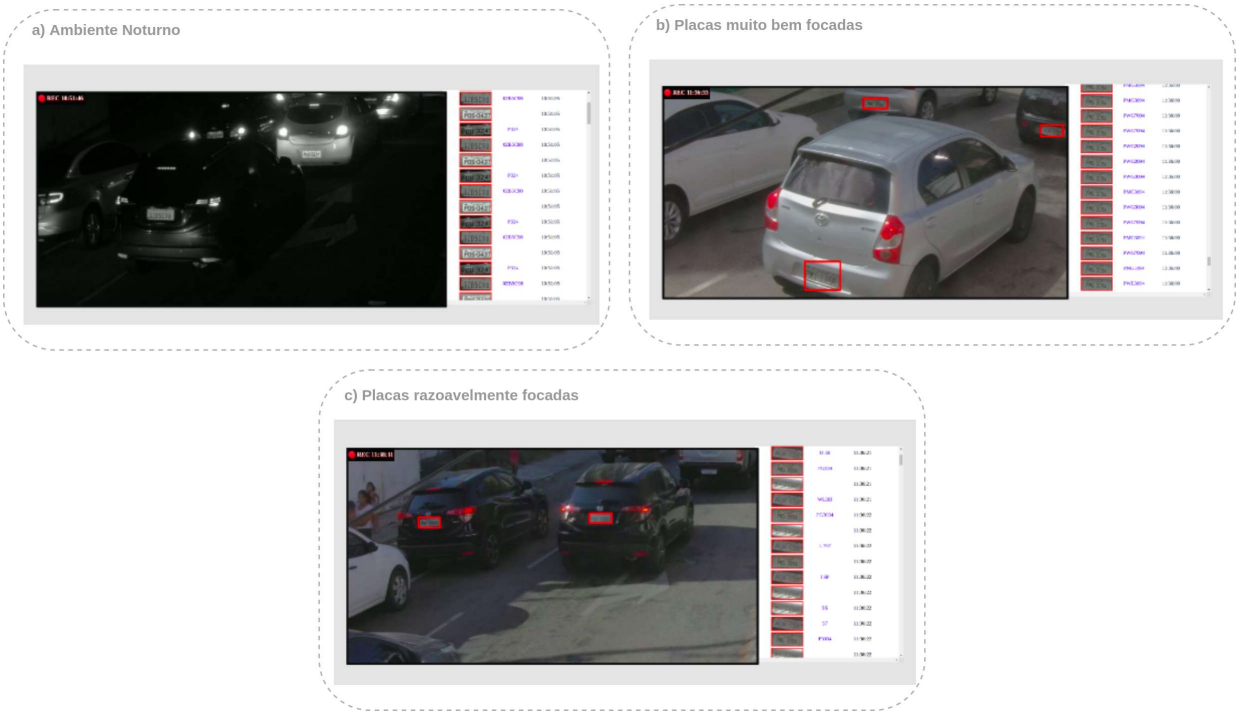
Fonte: Elaborado pelo autor

8.2 Resultados da validação da arquitetura de AVB

Os resultados da validação, com cenário descrito na Seção 7, são apresentados nas Figuras 16, 17 e 18. A Figura 16 faz alusão aos testes feitos nos três cenários, (a) ambiente noturno, (b) placas muito bem focadas pela câmera e (c) placas razoavelmente focadas. O próprio arquivo do Apêndice D demonstra que o objetivo de modularizar o processo de AV foi cumprido, pois o MCD, o MF e o MR são serviços separados e independentes no servidor de borda.

A Figura 17 mostra um exemplo de funcionamento do controle de qualidade durante a etapa de validação da arquitetura de AVB deste trabalho. Neste cenário, duas situações diferentes de uma mesma gravação são destacadas do *feed* de vídeo do cliente *web*: a) *frame* do momento em que a rua capturada pela câmera estava vazia e sem qualquer detecção, assim, o MCD reduziu a qualidade de resolução desse *frame* por um fator de 7; b) *frame* do momento exato em que um veículo aparece na rua e sua placa é detectada, fazendo com que o *frame* ficasse com resolução padrão de 1200x800 *pixels*. Isso mostra que o controle de qualidade de resolução está presente na arquitetura de AVB deste trabalho.

Figura 16 – Validação da arquitetura de AVB nos três cenários: ambiente noturno, placas muito bem focadas e placas raozavelmente focadas



Fonte: Elaborado pelo autor

Por outro lado, a Figura 18 mostra um exemplo de funcionamento da filtragem de placas. De forma proposital, o exemplo da Figura 18 apresenta o momento exato em que somente uma placa é detectada. Não só isso, a placa detectada se encontra borrada devido à falta de foco da câmera e à iluminação do ambiente. Os carros que aparecem nesse *frame* estão parados em frente a um semáforo, portando a placa borrada foi detectada várias vezes. A Figura 18 mostra que: a) sem o uso do MF, essa placa com texto ilegível foi submetida ao processo de reconhecimento e, logo após, enviada para exibição no cliente *web*; b) com o MF a placa em questão é detectada, porém não é enviada até o cliente *web* por meio do MR da arquitetura, ou seja, a placa sequer chegou à fase de reconhecimento na arquitetura de AVB deste trabalho. Assim, o último objetivo específico foi cumprido, logo, todos os objetivos do trabalho foram alcançados.

Em geral, observou-se certas limitações da arquitetura. Algumas configurações, como número de quadros por segundo e dimensão dos *frames* transmitidos, tiveram que assumir pequenos valores para evitar a alta latência de resposta durante a exibição do fluxo de *frames* no cliente *web*. Sendo mais preciso, a variável de ambiente CAMERA_CONFIG_FPS deveria receber no máximo o valor 15, enquanto as variáveis CAMERA_FRAME_WIDTH e

Figura 17 – Exemplo do funcionamento do controle de qualidade na visão do cliente *web*

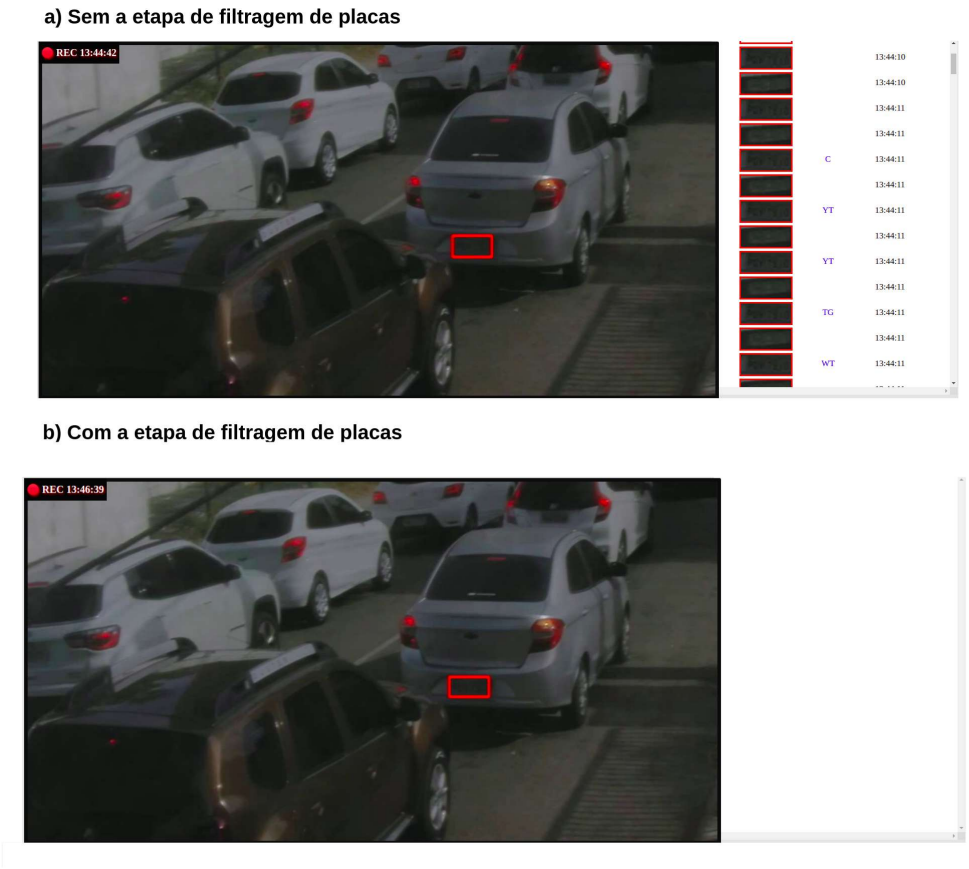
Fonte: Elaborado pelo autor

CAMERA_FRAME_HEIGHT não poderiam ultrapassar o valor 900.

Outro ponto é a latência de resposta no fluxo de placas recebido pelo cliente *web*. Notou-se que as placas demoravam até quinze minutos para aparecerem no painel de placas, se comparado ao momento de sua detecção exibida no *feed* de vídeo. As razões disso são óbvias. Primeiro, muitas placas são capturadas em um único *frame*, ao mesmo tempo que, apenas uma placa por vez é processada no MF e no MR. Segundo, o modelo de filtragem do MF é um modelo de Aprendizado Profundo que não passou por grandes otimizações em seu tamanho, logo, ainda possui um tempo considerável para efetuar sua tarefa de classificação em uma amostra de entrada.

Por fim, a Tabela 6 apresenta as quantidades de *frames* e recortes de placa, em GB, que o cliente *web* recebeu nos 6 cenários de experimentação, cenários esses explicados na Seção 7. Esses resultados são referentes à entrada de um fluxo de vídeo por aproximadamente 30 minutos e com 8 quadros por segundo na arquitetura de AVB deste trabalho. Além disso, foi definido um nível de redução de qualidade igual à 7, para os cenários que executam o controle de qualidade, e uma resolução padrão de 1200x800 *pixels* para os *frames* transmitidos. Dos resultados da Tabela 6, a otimização de armazenamento gerou uma redução de 22,3%, 12,9% e 41,73% nos cenários

Figura 18 – Exemplo do funcionamento da etapa de filtragem de placas na visão do cliente *web*



Fonte: Elaborado pelo autor

noturno, com foco alto e com foco razoável, respectivamente, na quantidade de *frames* enviados. Por outro lado, a otimização proporcionou uma redução de aproximadamente 43,75%, 28,73% e 97,48% nos cenários noturno, com foco alto e com foco razoável, respectivamente, dessa vez na quantidade de placas transmitidas. Assim, uma redução média de 25,64% no fluxo de *frames* e de 56,65% no fluxo de placas da arquitetura foram notadas. Com isso, observa-se que a arquitetura, com seu controle de qualidade e sua filtragem de placas em execução, conseguiu reduzir a quantidade de dados transmitido pelos dois fluxos de dados direcionados para o cliente *web* que, por consequência, reduziu o tráfego da rede nesses fluxos. Dadas essas últimas conclusões, o trabalho segue para suas considerações finais.

Tabela 6 – Quantidades de *frames* e de recortes de placa recebidas pelo cliente *web* nos cenários de experimentação

Cenário de experimentação	**Quantidade de *frames***	**Quantidade de placas**
Noturno	24,40GB	2,88GB
Noturno com otimizações	**18,96GB**	**1,62GB**
Com foco alto	24,28GB	2,75GB
Com foco alto e otimizações	**21,15GB**	**1,96GB**
Com foco razoável	32,50GB	2,77GB
Com foco razoável e otimizações	**18,94GB**	**0,07GB**

Fonte: Elaborado pelo autor

9 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, realizou-se a implementação e validação de uma arquitetura de AVB para a execução de serviços de análise para a tarefa de detecção/reconhecimento de placas. A arquitetura era composta por três partes essenciais. São elas: três contêineres de análise por serviço de análise, chamados de MCD, MF e MR; um contêiner intermediário para comunicação entre módulos de análise de diferentes serviços de análise; e um contêiner de *streaming* para expor os resultados dos serviços de análise ao exterior da arquitetura.

A tecnologia Docker foi usada para implantação dessas três partes da arquitetura, enquanto o OpenCV com aceleração por GPU foi empregado para realização da Análise de Vídeo com Aprendizado Profundo. Essas tecnologias, somado ao uso tanto do Kafka quanto do Flask-SocketIO para fluxo de dados, formaram o alicerce da arquitetura de AVB implementada neste trabalho.

O MCD emprega um modelo de detecção de placas fundamentado no Yolov4-Tiny, além de utilizar os resultados de detecção desse modelo para controlar a qualidade de resolução do fluxo de *frames* da arquitetura. Enquanto isso, o MR aplica o reconhecimento de caracteres nas placas detectadas por meio do Tesseract OCR e envia um fluxo de recorte de placas em paralelo ao fluxo de *frames* da arquitetura.

Este trabalho investiu em um novo estágio do processo de Análise de Vídeo implementado através do MF. O MF atua como um intermediário entre o MCD e o MR, sendo responsável por executar um modelo de filtragem de placas. Nesse modelo, recortes de placa com texto ilegível vindas do MCD são descartados antes mesmo de chegarem ao MR. Para esse fim, este trabalho optou por arquitetar modelos de Aprendizado Profunda para a tarefa de classificação de placas. A classificação é segundo a legibilidade do texto de identificação dos recortes de placa.

Assim, recorreu-se à técnica de Transferência de Aprendizado com os modelos Resnet50, MobileNetv2, InceptionV3 e EfficientNetB0 para a construção das arquiteturas candidatas ao MF. Com posse de gravações em vídeo das ruas da cidade de Fortaleza, no estado do Ceará, este trabalho montou um *dataset* pequeno composto por 520 amostras de recorte de placa. Dentre as quatro estruturas, o MobileNetv2 foi escolhido após alcançar uma acurácia de 99,04% sobre o conjunto de validação e 100% sobre um conjunto pequeno de teste.

Finalmente, todos os objetivos específicos deste trabalho foram alcançados após uma etapa de validação da arquitetura. Os resultados puderam ser vistos por meio de uma

página *web* capaz de capturar o fluxo de *frames* e o fluxo de placas servido pelo contêiner de *streaming* da solução. Assim, os objetivos alcançados foram: modularizar o processo de detecção/reconhecimento de placas executado na arquitetura de AVB, comprovado pela existência do MCD, MF e MR como contêineres Docker independentes; realizar um controle de qualidade de resolução, sendo isso uma das funcionalidade do MCD; e, por fim, realizar uma filtragem de placas, sendo essa a função do MF.

Por este trabalho possuir um escopo amplo, são inúmeras as possibilidades de melhorias e trabalhos futuros. Realizar uma análise de desempenho da arquitetura, levando em consideração o consumo de processador, o uso de memória e o tráfego da rede, seria um exemplo de trabalho futuro. Afinal, este trabalho acredita que a análise de desempenho seja crucial para avaliar a escalabilidade tal como a eficiência da arquitetura. Esses fatores definiriam o quão bem a solução se daria no contexto da Computação de Borda.

Além disso, testar outros modelos de detecção de placas no MCD e evoluir, ou também variar, a tecnologia usada para reconhecimento no MR fica como trabalho futuro. Uma análise de desempenho com uma variedade de modelos em ambos os módulos seria um trabalho futuro rico e satisfatório de realizar.

Também é possível que a otimização do modelo de filtragem de placas implementado neste trabalho seja um trabalho futuro bastante necessário, a fim de diminuir a latência de resposta causada por esse modelo no MR. Converter para Tensorflow Lite, aplicar a técnica de poda combinada com quantização pós-treinamento são exemplos de meios oferecidos pelo próprio Tensorflow para diminuição do tamanho do modelo sem perca substancial do seu desempenho.

Espera-se que os resultados obtidos por este trabalho agreguem algo ao projeto "ID Digital"da Secretaria da Segurança Pública e Defesa Social do Ceará. Este trabalho enfatizou solucionar questões associadas à manutenção e operação de Sistemas Inteligentes de Vídeo Vigilância. Assim, espera-se que tudo que foi desenvolvido neste trabalho contribua na construção de sistemas de videovigilância mais seguros, com baixa latência de resposta, com um menor uso da largura de banda e com melhor aproveitamento dos recursos oferecidos pela Computação de Borda.

REFERÊNCIAS

ACADEMY, D. S. **Deep Learning Book**. 2019. Disponível em: http://deeplearningbook.com.br. Acesso em 6 mai. 2020.

ANANTHANARAYANAN, G.; BAHL, P.; BODIK, P.; CHINTALAPUDI, K.; PHILIPOSE, M.; RAVINDRANATH, L.; S., S. Real-time video analytics: the killer app for edge computing. **Computer**, IEEE, [S.I], v. 50, n. 10, p. 58–67, 2017.

BERNSTEIN, D. Containers and cloud: from lxc to docker to kubernetes. **IEEE Cloud Computing**, IEEE, [S.I], v. 1, n. 3, p. 81–84, 2014.

DOCKER. **Docker Documentation**. 2021. Disponível em: https://docs.docker.com. Acesso em 30 mar. 2021.

GAGVANI, N. Challenges in video analytics. In: BRANISLAV, K.; S., B. S.; SEK, C. (Ed.). **Embedded Computer Vision**. London: Springer London, 2009. cap. 12, p. 237–256.

GANDOMI, A.; HAIDER, M. Beyond the hype: big data concepts, methods, and analytics. **International Journal of Information Management**, ScienceDirect, [S.I], v. 35, p. 137–144, abr. 2015.

HOSSIN, M.; SULAIMAN, M. A review on evaluation metrics for data classification evaluations. **International Journal of Data Mining Knowledge Management Process**, [S.I], v. 5, p. 01–11, 03 2015.

ISMAIL, B. I.; GOORTANI, E. M.; KARIM, M. B. A.; TAT, W. M.; SETAPA, S.; LUKE, J. Y.; HOE, O. H. Evaluation of docker as edge computing platform. In: **2015 IEEE Conference on Open Systems (ICOS)**. [S.l.: s.n.], 2015. p. 130–135.

KERAS. **Keras Documentation**. 2021. Disponível em: https://keras.io/. Acesso em 30 mar. 2021.

KHANN, W. Z.; AHMED, E.; HAKAK, S.; YAQOOB, I.; AHMED, A. Edge computing: a survey. **Future Generation Computer Systems**, ScienceDirect, [S.I], v. 97, p. 219–235, 2019.

MARKIT, I. **Security technologies top trends for 2020**. 2019. Disponível em: https://ihsmarkit.com/. Acesso em 6 jun. 2020.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux journal**, [S.I], v. 2014, n. 239, p. 2, 2014.

MICHEL, M.; BURNETT, N. Enabling gpu-enhanced computer vision and machine learning research using containers. In: WEILAND, M.; JUCKELAND, G.; ALAM, S.; JAGODE, H. (Ed.). **High Performance Computing**. Cham: Springer International Publishing, 2019. p. 80–87.

RAI, M.; HUSAIN, A. A.; MAITY, T.; YADAV, R. K. Advance intelligent video surveillance system (aivss): A future aspect. In: NEVES, A. J. R. (Ed.). **Intelligent Video Surveillance**. Rijeka: IntechOpen, 2019. cap. 3.

RAWAT, W.; WANG, Z. Deep convolutional neural networks for image classification: A comprehensive review. **Neural Computation**, [S.I], v. 29, n. 9, p. 2352–2449, 2017.

RICHINS, D.; DOSHI, D.; BLACKMORE, M.; NAIR, A. T.; PATHAPATI, N.; PATEL, A.; DAGUMAN, B.; D., D.; ILLIKKAL, R.; LONG, K.; ZIMMERMAN, D.; REDDI, V. Missing the forest for the trees: End-to-end ai application performance in edge data centers. In: **2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2020. p. 515–528.

SAKAUSHI, A.; KANAI, K.; KATTO, J.; TSUDA, T. Edge-centric video surveillance system based on event-driven rate adaptation for 24-hour monitoring. In: **2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)**. [S.l.: s.n.], 2018. p. 651–656.

SANTOS, G. N. P. d.; FREITAS, P. A. d.; BUSSON, A.; LIVIO, A.; COLCHER, S.; MILIDIú, R. Métodos baseados em deep learning para análise de vídeo. In: **Minicursos do XXV Simpósio Brasileiro de Sistemas Multimídia e Web**. [S.l.]: SBC, 2019. cap. 4, p. 119.

SHI, W.; CAO, J.; ZHANG, Q.; LI, Y.; XU, L. Edge computing: vision and challenges. **IEEE Internet of Things Journal**, IEEE, [S.I], v. 3, n. 5, p. 637–646, 2016.

SHI, W.; PALLIS, G.; XU, Z. Edge computing [scanning the issue]. **Proceedings of the IEEE**, IEEE, [S.I], v. 107, n. 8, p. 1474–1481, 2019.

SREENU, G.; DURAI, M. A. S. Intelligent video surveillance: a review through deep learning techniques for crowd analysis. **Journal of Big Data**, [S.I], v. 6, p. 48, ago. 2019.

TAN, C.; SUN, F.; KONG, T.; ZHANG, W.; YANG, C.; LIU, C. A survey on deep transfer learning. In: **Artificial Neural Networks and Machine Learning – ICANN 2018**. [S.l.]: Springer International Publishing, 2018. p. 270–279.

XIE, Y.; HU, Y.; CHEN, Y.; LIU, Y.; SHOU, G. A video analytics-based intelligent indoor positioning system using edge computing for iot. In: **2018 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)**. [S.l.: s.n.], 2018. p. 118–1187.

ZHANG, Q.; SUN, H.; WU, X.; ZHONG, H. Edge video analytics for public safety: a review. **Proceedings of the IEEE**, IEEE, [S.I], v. 107, n. 8, p. 1675–1696, 2019.

APÊNDICE A – DOCKERFILE PARA A CRIAÇÃO DA IMAGEM BASE DOS CONTÊINERES DOCKER

No Código-fonte 1 é apresentado a imagem base dos contêineres Docker da arquitetura de AVB deste trabalho. É possível constatar pela palavra-chave FROM que a imagem de fato foi construída da imagem apresentada em Michel e Burnett (2019). Observe que a *tag* da imagem de Michel e Burnett (2019) diz quais versões do NVIDIA CUDNN, Tensorflow e Opencv são instalados: versão 10.2, versão 2.3.0 e versão 4.4.0 respectivamente. Em seguida, são executados os comandos para instalação das dependências do contêiner intermediário e do contêiner de *streaming*.

Código-fonte 1 – Dockerfile para criação da imagem base

```
1  FROM datamachines/cudnn_tensorflow_opencv:10.2_2.3.0_4.4.0-20200803
2
3  #Instalando dependencias do conteiner intermediario
4  RUN pip install kafka-python==2.0.1
5
6  #Instalando dependencias do conteiner de streaming
7  RUN pip install requests websocket-client Click==7.0 dnspython==1.16.0 \
8      eventlet==0.25.1 Flask==1.1.1 Flask-SocketIO==4.2.1 greenlet==0.4.15 \
9      itsdangerous==1.1.0 Jinja2==2.11.1 MarkupSafe==1.1.1 monotonic==1.5 \
10     python-engineio==3.11.2 python-socketio==4.4.0 six==1.14.0 Werkzeug==1.0.0
11
12 RUN apt-get install -y netbase
13
14 RUN apt-get -o Dpkg::Options::="--force-confmiss" install --reinstall netbase
```

APÊNDICE B – *TEMPLATE* DE DOCKERFILE PARA A CRIAÇÃO DA IMAGEM BASE DE UM MÓDULO DE ANÁLISE

No Código-fonte 2 é apresentado um Dockerfile molde para a criação da imagem Docker de um módulo de análise qualquer. Na mesma pasta do Dockerfile existem um arquivo Python chamado app.py acompanhado de outros arquivos necessários para a execução desse arquivo, por exemplo, aqueles referentes ao modelo de Aprendizado Profundo empregado em tal módulo. A exceção à regra é o MR que precisa das dependências do Tesseract OCR, bastando executar também os comandos comentados entre as linhas 7 e 10.

Código-fonte 2 – *Template* de Dockerfile para criação da imagem de um módulo de análise genérico

```
1  FROM imagem-base:latest
2
3  #Copia tudo que for necessario para a execucao da aplicacao
4  ADD ./ ./
5
6  #Comandos exclusivos para o modulo de reconhecimento
7  #RUN add-apt-repository ppa:alex-p/tesseract-ocr
8  #RUN apt-get update
9  #RUN apt install -y tesseract-ocr
10 #RUN pip install pytesseract
11
12 #Executa a aplicacao Python
13 CMD [ "python3", "app.py" ]
```

APÊNDICE C – TABELA DE VARIÁVEIS DE AMBIENTE DOS CONTÊINERES DA ARQUITETURA DE ANÁLISE DE VÍDEO NA BORDA DESTE TRABALHO

A Tabela 7 lista as variáveis de ambiente para configuração dos contêineres da arquitetura de AVB deste trabalho. Com o Docker, essas variáveis podem ser informadas no momento da criação de uma instância. A primeira coluna informa qual contêiner do valor daquela variável, com o nome da variável informado na coluna do meio. A última coluna informa a configuração feita por aquela variável de ambiente.

Tabela 7 – Variáveis de ambiente dos módulos de análise

Cônteiner(es)	**Nome da variável**	**Configuração**
Todos	NVR_ID	Identifica a NVR que aquele serviço é associado
Todos	CAMERA_CONFIG_ID	Identifica a câmera que aquele serviço é associado
Captura/Detecção	CAMERA_CONFIG_SOURCE	Informa a URL do fluxo de vídeo de entrada
Captura/Detecção	CAMERA_CONFIG_FPS	Informa quantos quadros por segundo serão capturados do fluxo de vídeo de entrada
Captura/Detecção	CAMERA_FRAME_WIDTH	Define a largura padrão do frame enviado para *streaming*
Captura/Detecção	CAMERA_FRAME_HEIGHT	Define a altura padrão do frame enviado para *streaming*
Captura/Detecção	CAMERA_FRAME_LEVEL_REDUCE	Define o fator de divisão das dimensões de um frame, caso não haja detecções
Captura/Detecção	YOLO_CONFIDENCE	Define qual o valor mínimo de confiança aceito na detecção de placas
Filtragem	MODEL_CONFIDENCE	Define qual o valor mínimo de confiança aceito na filtragem de placas
Todos	KAFKA_BROKER_URL	Identifica o IP e a porta do *cluster* do Kafka na rede
Captura/Detecção e Reconhecimento	STREAMING_SERVER_API_HOST	Identifica o IP do contêiner de *streaming*
Captura/Detecção e Reconhecimento	STREAMING_SERVER_API_PORT	Identifica a porta do contêiner de *streaming*

Fonte: Elaborado pelo autor

APÊNDICE D – EXEMPLO DE ARQUIVO YAML PARA LEVANTAMENTO DOS CONTÊINERES DA ARQUITETURA DE AVB DESTE TRABALHO

O Código-fonte 3 apresenta o arquivo YAML para levantamento dos contêineres para um serviço de análise na arquitetura de AVB deste trabalho, aquela ilustrada na Figura 9. Porém, as instâncias que são criadas por meio deste arquivo não se beneficiam da aceleração por GPU, devido à falta de suporte pelo Docker-Compose. Além disso, esse código-fonte orienta bem como foram criadas as instâncias para a etapa de validação da arquitetura na Seção 7.

Código-fonte 3 – Arquivo YAML para levantamento dos contêineres sem aceleração por GPU

```yaml
version: "3.3"

services:

    # Conteiner de streaming
    streaming:
      image: server:versaoTCC
      container_name: streaming
      environment:
        STREAMING_SERVER_API_HOST: streaming
        STREAMING_SERVER_API_PORT: 5001
      ports:
        - "80:5001"

    # Conteiner intermediario
    zookeeper:
      image: confluentinc/cp-zookeeper:latest
      container_name: zookeeper
      environment:
        ZOOKEEPER_CLIENT_PORT: 2181
        ZOOKEEPER_TICK_TIME: 2000

    broker:
      image: confluentinc/cp-kafka:latest
      container_name: broker
      depends_on:
        - zookeeper
      environment:
        KAFKA_BROKER_ID: 1
        KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
        KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:9092

    # MCD
    mcd_NVR1_CAM1:
      image: captura_deteccao:versaoTCC
```

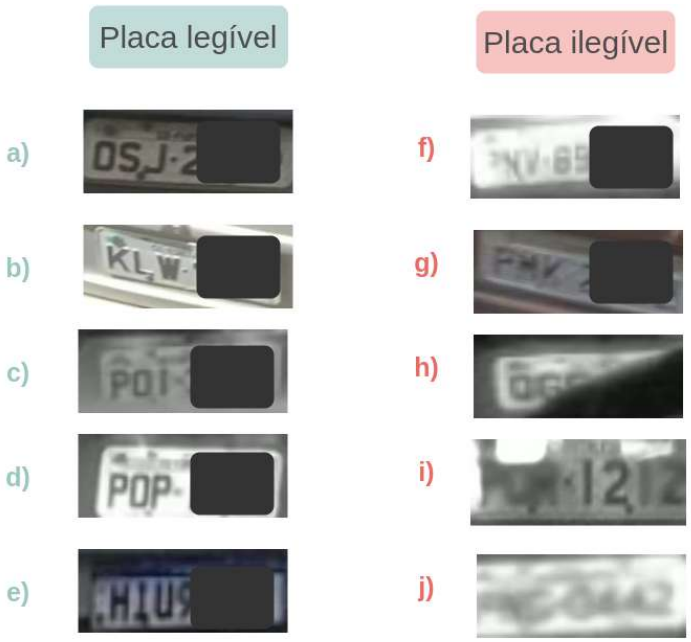
```
36        container_name: mcd_NVR1_CAM1
37        depends_on:
38          - broker
39          - streaming
40        environment:
41          NVR_ID: NVR1
42          CAMERA_CONFIG_ID: CAM1
43          KAFKA_BROKER_URL: broker:9092
44          CAMERA_CONFIG_SOURCE: /usr/videos/cam.avi
45          CAMERA_CONFIG_FPS: 15
46          CAMERA_FRAME_WIDTH: 1200
47          CAMERA_FRAME_HEIGHT: 800
48          CAMERA_FRAME_LEVEL_REDUCE: 7
49          YOLO_CONFIDENCE: 0.7
50          STREAMING_SERVER_API_HOST: streaming
51          STREAMING_SERVER_API_PORT: 5001
52        volumes:
53          - /media/alessandro/videos:/usr/videos
54
55      # MF
56      mf_NVR1_CAM1:
57        image: filtragem:versaoTCC
58        container_name: mf_NVR1_CAM1
59        depends_on:
60          - broker
61          - mcd_NVR1_CAM1
62        environment:
63          NVR_ID: NVR1
64          CAMERA_CONFIG_ID: CAM1
65          KAFKA_BROKER_URL: broker:9092
66          MODEL_CONFIDENCE: 0.7
67
68      # MR
69      mr_NVR1_CAM1:
70        image: reconhecimento:versaoTCC
71        container_name: mr_NVR1_CAM1
72        depends_on:
73          - broker
74          - streaming
75          - mf_NVR1_CAM1
76        environment:
77          NVR_ID: NVR1
78          CAMERA_CONFIG_ID: CAM1
79          KAFKA_BROKER_URL: broker:9092
80          STREAMING_SERVER_API_HOST: streaming
81          STREAMING_SERVER_API_PORT: 5001
82
83  # Rede Docker do tipo bridge
84  networks:
```

```
85      default:
86          external:
87              name: arquitetura-network
```

APÊNDICE E – EXEMPLOS DE PLACAS DO *DATASET* DE PLACAS LEGÍVEIS/ILEGÍVEIS

A Figura 19 ilustra exemplos de placas do *dataset* construído na Seção 6.1. Em a), b), c), d), e e) são mostradas placas com texto legível. Em f), g), h), i) e j) são mostradas placas com texto ilegível. Algumas das placas foram censuradas nessa figura, a fim de evitar qualquer risco de identificação das placas pelo leitor.

Figura 19 – Exemplos de placas do *dataset* construído neste trabalho

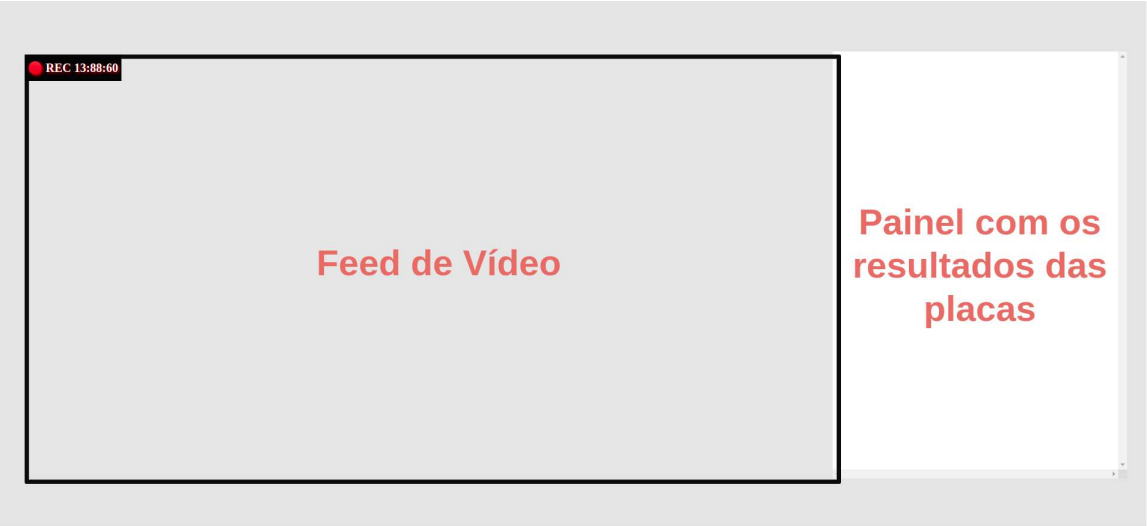


Fonte: Elaborado pelo autor

APÊNDICE F – INTERFACE DO CLIENTE *WEB*

A Figura 20 ilustra a organização do cliente *web* do contêiner de *streaming* para a etapa de validação da arquitetura de AVB deste trabalho, citada na Seção 7. Esse cliente *web* possui duas partes principais. Primeiro, ele contém uma região destinada ao fluxo de *frames* recebido do MCD para exibição do vídeo e das detecções, chamado de *feed* de vídeo. Exatamente ao lado do *feed* de vídeo, existe um painel destinado a expor os resultados do fluxo de placas vindo do MR.

Figura 20 – Visão geral do cliente *web*



Fonte: Elaborado pelo autor

ANEXO A – TABELA DE INFORMAÇÃO DAS ESTRUTURAS DE APRENDIZADO PROFUNDO DA DOCUMENTAÇÃO DO KERAS

Este anexo é uma tabela originada da documentação do Keras (KERAS, 2021). No anexo, são apresentados alguns algoritmos de Aprendizado Profundo disponíveis na API do Keras. Assim, os modelos são listados com informações sobre tamanho, quantidade de parâmetros, acurácias nos conjuntos de validação do ImageNet, e profundidade, respectivamente. Foi esse o anexo utilizado para selecionar as arquiteturas de Aprendizado Profundo para o experimento de construção do modelo de classificação de placas, na Seção 6.

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-
EfficientNetB0	29 MB	-	-	5,330,571	-
EfficientNetB1	31 MB	-	-	7,856,239	-
EfficientNetB2	36 MB	-	-	9,177,569	-
EfficientNetB3	48 MB	-	-	12,320,535	-
EfficientNetB4	75 MB	-	-	19,466,823	-
EfficientNetB5	118 MB	-	-	30,562,527	-
EfficientNetB6	166 MB	-	-	43,265,143	-
EfficientNetB7	256 MB	-	-	66,658,687	-