UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
DEPARTAMENTO DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

Um sistema de reconhecimento facial escalável baseado em aprendizado profundo para aplicações em tempo real

Vilson Rodrigues Câmara Neto

Natal-RN, Brasil
2023

Vilson Rodrigues Câmara Neto

Um sistema de reconhecimento facial escalável baseado em aprendizado profundo para aplicações em tempo real

Monografia apresentada ao Curso de Bacharelado em Engenharia de Computação do Departamento de Computação e Automação da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Doutor Daniel Sabino Amorim de Araujo

Natal-RN, Brasil
2023

‘

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Câmara Neto, Vilson Rodrigues.
Um sistema de reconhecimento facial escalável baseado em aprendizado profundo para aplicações em tempo real / Vilson Rodrigues Câmara Neto. - 2023.
56 f.: il.

Monografia (graduação) - Universidade Federal do Rio Grande do Norte, Centro de Tecnologia, Bacharelado em Engenharia de Computação. Natal, RN, 2023.
Orientação: Prof. Dr. Daniel Sabino Amorim de Araújo.

1. Sistema de Reconhecimento Facial - Monografia. 2. Aprendizado Profundo - Monografia. 3. Escalabilidade - Monografia. 4. Resiliência - Monografia. I. Araújo, Daniel Sabino Amorim de. II. Título.

RN/UF/BCZM CDU 004.89

Elaborado por Fernanda de Medeiros Ferreira Aquino - CRB-15/301

Vilson Rodrigues Câmara Neto

Um sistema de reconhecimento facial escalável baseado em aprendizado profundo para aplicações em tempo real

Monografia apresentada ao Curso de Bacharelado em Engenharia de Computação do Departamento de Computação e Automação da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Trabalho aprovado. Natal-RN, Brasil, 7 de Dezembro de 2023:

Doutor Daniel Sabino Amorim de Araujo
Orientador

Prof. Dr. Ivanovitch Medeiros Dantas da Silva
Examinador

Prof. Dr. Patrick Cesar Alves Terrematte
Examinador

Natal-RN, Brasil
2023

Dedico este trabalho aos meus pais, Kalyana e Ari, e aos meus familiares que me deram apoio e incentivo durante minha jornada acadêmica.

Agradecimentos

Agradeço aos meus pais e familiares por todo esforço para que eu conseguisse ter a melhor educação possível.

Agradeço aos professores Orivaldo Vieira e Igor Rosberg que acreditaram em mim e me deram a oportunidade de participar do meu primeiro projeto como pesquisador. Agradeço também a Renata Pitta pelos conhecimentos e companhia durante esse projeto.

Agradeço ao professor Daniel Sabino que me acolheu como orientando para o desenvolvimento desse trabalho, além de ser também meu orientador no projeto de pesquisa no MPRN.

Agradeço ao pessoal do MPRN que fizeram desenvolver meu interesse por construção de sistemas.

Agradeço ao professor Ivanovitch Silva pelo esforço em disponibilizar disciplinas relacionadas ao aprendizado de máquina.

Agradeço ao meu primo Mickael Figueredo que foi meu guia profissional durante toda a graduação.

Agradeço a UFRN e demais professores pela sua arte de ensino.

“Não creio que haja uma emoção mais intensa para um inventor do que ver suas criações funcionando. Essas emoções fazem você esquecer de comer, de dormir, de tudo.”

(Nikola Tesla)

Resumo

Este trabalho discute a crescente relevância dos sistemas de vigilância multicâmeras, enfatizando a necessidade de uma identificação eficaz de indivíduos. Os sistemas de reconhecimento facial proporcionam uma identificação e localização rápidas de pessoas, auxiliando na prevenção de incidentes. O reconhecimento facial, particularmente com o uso de técnicas de Aprendizado Profundo, emerge como uma solução promissora. Contudo, desafios como escalabilidade e resiliência ainda persistem. Propõe-se o uso de Bancos de Dados Vetoriais, implementando HNSW, para aprimorar a eficiência na etapa de correspondência de rostos. As etapas de processamento em torno dos modelos foram integradas aos próprios modelos, garantindo o processamento em lote de etapas anteriormente realizadas de maneira serial. Ferramentas de escalabilidade e monitoramento foram implementadas para assegurar a resiliência do sistema. O objetivo é desenvolver um sistema de reconhecimento facial eficiente, resiliente e de baixo custo, fundamentado em Aprendizado Profundo e computação distribuída. Este estudo apresenta uma estrutura que engloba conceitos, métodos de construção do sistema e resultados de testes. Os testes demonstraram que o aumento da demanda por ser resolvido por meio da escalabildade das etapas de processamento. Código e modelos estão disponíveis em <https://github.com/vilsonrodrigues/face-recognition>.

Palavras-chave: Sistema de Reconhecimento Facial. Aprendizado Profundo. Escalabilidade. Resiliência.

Abstract

This work discusses the growing relevance of multi-camera surveillance systems, emphasizing the need for effective identification of individuals. Facial recognition systems detect and quickly identify and locate people, helping to prevent incidents. Facial recognition, particularly using Deep Learning techniques, appears as a promising solution. However, challenges such as scalability and resilience still persist. It proposes the use of Vector Databases, implementing HNSW, to improve efficiency in the face matching stage. The processing steps around the models were integrated into the models themselves, ensuring batch processing of steps previously carried out in a serial manner. Scaling and monitoring tools were solutions to ensure system resilience. The objective is to develop an efficient, resilient and low-cost facial recognition system, based on deep learning and distributed learning. This study presents a structure that encompasses concepts, system construction methods and test results. The tests demonstrated that the increase in demand was resolved through the scalability of the processing steps. Code and models are avaliable at <https://github.com/vilsonrodrigues/face-recognition>.

Keywords: Facial Recognition System. Deep Learning. Scalability. Resilience.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ORT	ONNX Runtime
API	Application Programming Interface
IA	Inteligência Artificial
VM	Virtual Machine
QPS	Queries Per Secound
ID	Identificador
HNSW	Hierarchical Navigable Small World
SSD	Single Shot MultiBox Detector
MLP	Multi-Layer Perceptron
SVM	Support Vector Machine
MTCNN	Multi-task Cascaded Convolutional Networks
NMS	Non-Maximum Supression
K-NN	k-Nearest Neighbors
IiF	Identification-in-Football

Sumário

1 Introdução

À medida que os sistemas de vigilância multicâmeras se expandem rapidamente em todo o mundo, a capacidade de associar pessoas no espaço e no tempo torna-se cada vez mais crucial. Essa habilidade é significativa para uma variedade de aplicações, incluindo segurança pública e forense. Esses sistemas não apenas aumentam a segurança, mas também fornecem uma ferramenta valiosa para a aplicação da lei, permitindo a identificação e localização rápidas de indivíduos em várias câmeras e locais (SG; MC; CC, 2014). A identificação de indivíduos em um grande número de câmeras de vigilância é uma tarefa complexa, que muitas vezes é realizada manualmente por especialistas. Sistemas de reconhecimento automático levam a diminuição da carga de trabalho e dispensam a necessidade de especialistas.

Dentre as atuais técnicas de biometria visual para identificação de pessoas, como corpo inteiro, íris e impressão digital, a aparência facial é considerada uma das mais confiáveis e convenientes para rastreamento e monitoramento de longo prazo. Isso se deve à sua facilidade de captura, a quantidade de informações únicas que cada rosto possui e a sua menor invariância no tempo (CHENG; ZHU; GONG, 2020).

O reconhecimento facial se divide entre duas vertentes: verificação e identificação (AHONEN; HADID; PIETIKÄINEN, 2004). A verificação se dá pela comparação da face recebida com a desejada. Na identificação ocorre a comparação da face recebida com as armazenadas pelo sistema, de forma a identificar dentre elas a mais similar a recebida. Esse trabalho aborda a identificação.

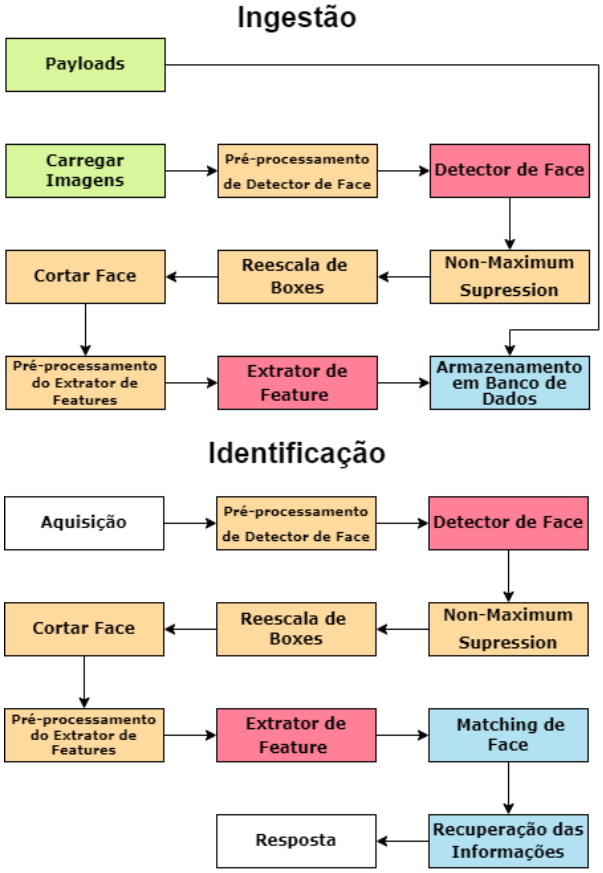
Os sistemas de reconhecimento facial que utilizam técnicas de Aprendizado Profundo (*Deep Learning*) têm demonstrado desempenho superior ao nível humano na maioria dos cenários (WANG; DENG, 2021). No entanto, essas técnicas requerem uma quantidade significativa de recursos computacionais e tempo de processamento.

Embora existam sistemas de reconhecimento facial mais leves usando outras técnicas, eles não são robustos e tendem a falhar quando os rostos no quadro (*frame*) são pequenos ou quando o ambiente do quadro é um pouco escuro (SRIVASTAVA et al., 2017).

Um fluxo de um sistema de reconhecimento facial pode ser visto na Figura 1. A literatura apresenta diversas propostas para reduzir a latência (*latency*) e o consumo de recursos computacionais em aplicações em tempo real (*real-time*). No entanto, as arquiteturas sugeridas enfrentam desafios de escalabilidade e resiliência. A ausência de suporte para processamento em lote (*batch*) impede a manipulação de mais de uma fonte de dados em *real-time*. Além disso, as etapas de pré-processamento e *Non-Maximum Suppression* (NMS) são intrinsecamente sequenciais nesses trabalhos.

O principal desafio desses sistemas é a etapa de *matching* de face, onde ocorre a comparação do vetor de características da face com os armazenados no sistema. Se o

Figura 1 – Diagrama de um fluxo de funcionamento de um sistema de reconhecimento facial, composto por uma etapa de ingestão de dados e uma de identificação.



Fonte: Autoria própria.

conjunto de dados for grande, mesmo comparações realizadas de forma paralela podem ser lentas (MENG et al., 2005). As abordagens que escolhem usar um classificador nesta etapa enfrentam o problema de treinamento contínuo à medida que novas faces são adicionadas ao domínio do sistema.

Bancos de Dados Vetoriais (*Vector Databases*) surgiram para minimizar o tempo de correspondência (*matching*), implementando o algoritmo *Hierarchical Navigable Small World* (HNSW). Eles são capazes de realizar buscas em larga escala na ordem de milissegundos (MALKOV; YASHUNIN, 2018).

A falta de resiliência pode tornar as aplicações pouco confiáveis, o que pode ser problemático em vários aspectos. Uma aplicação não resiliente pode falhar quando confrontada com situações inesperadas ou estressantes, como picos de tráfego, falhas de hardware ou erros de software. Isso pode resultar em interrupções de serviço e perda de dados. Além disso, a falta de resiliência pode levar a custos adicionais. Por exemplo, pode ser necessário investir tempo e recursos significativos para recuperar de falhas e restaurar o serviço normal. Por isso, a implatação de resiliência usando observabilidade e monitoramento se tornam cruciais para garantir um serviço confiável.

1.1 Objetivos

Este trabalho tem como objetivo desenvolver um sistema de reconhecimento facial eficiente, resiliente e de baixo custo, baseado em *Deep Learning*. Utilizando de computação distribuída para paralelizar as tarefas durante o processamento dos dados. O sistema deverá ter duas etapas de funcionamento: ingestão e recuperação dos dados.

Na etapa de ingestão, os dados deverão consumidos, processados e armazenados em um *Vector Database*. Para a etapa de recuperação, uma API no modelo cliente-servidor irá receber imagens e retorna os dados e as localizações das faces correspondentes.

Modelos leves e eficientes devem ser selecionados para construção de um sistema de baixo custo. As etapas de processamento deverão ser escaláveis conforme a quantidade de dados a serem processados aumentar. Os modelos deverão integrar em si próprios as etapas de processamento em que são dependentes. Aceleração de inferência deverá ser utilizada. Ferramentas de monitoramento, observabilidade e resiliência deverão integrar o sistema.

1.2 Estrutura do trabalho

No capítulo 2 ocorre a explicação de conceitos e algoritmos usados em sistemas de reconhecimento facial. No capítulo 3 são descritos conceitos sobre sistemas de aprendizado de máquina e as ferramentas serão utilizadas para construção do sistema proposto. No capítulo 4 são apresentados os métodos usados para construção do sistema. No capítulo 5 são discutidos os resultados dos testes do sistema. E o capítulo 6 apresenta a conclusão do trabalho.

2 Reconhecimento Facial

Este cápitulo busca introduzir e detalhar as etapas de um fluxo de sistema de reconhecimento facial. As técnicas utilizadas na liteturatura são descritas para maior compreensão.

2.1 Fluxo de ingestão e recuperação dos dados

Nas Figura 1 foi exibido o fluxo de funcionamento da ingestão e da recuperação dos dados. A seguir, serão descritos mais detalhes sobre as etapas contidas.

A etapa de carragamento de imagens (*load images*) envolve obter as imagens de alguma fonte de dados. Algumas dessas fontes podem ser diretórios locais, bancos de dados relacionais ou *buckets* em serviços de *cloud*. As dimensões da imagem normalmente são expressas como largura, altura e canais de cores.

A etapa de aquisição (*acquisition*) de imagens na arquitetura de recuperação conta com duas fontes de dados comumente utilizadas. A primeira é a transmissão contínua (*streaming*) de imagens provenientes de uma câmera de vídeo. A segunda fonte é baseada no modelo *client-server*, no qual o usuário carrega uma imagem específica para o servidor e aguarda sua resposta. Essa imagem está formatada como base64 ou ela vem como bytes. Vale ressaltar que o uso de *streaming* também é suportado neste tipo de arquitetura.

As etapas envolvem a preparação das imagens para a detecção de face ou para a extração de *features*. Essa etapa depende de que técnica está sendo executada. As operações comuns envolvem redimensionamento (*resize*), normalização dos canais de cores e aplicação de filtros como escala de cinza (*grayscale*).

Quando houver várias *bounding boxes* geradas , a técnica de *Non-Maximum Supression* deve ser utilizada para restringir a quantidade de ancorâs por área baseado em seu *score* de confiança de que naquela *anchor* existe uma face.

Alguns detectores baseados em *Deep Learning* normalizam as *boxes*, isso traz robustez para os modelos aceitarem imagens das mais diversas proporções e as boxes serem readaptadas conforme a imagem de entrada. Então, logo após a descoberta da localização da face é aplicada a rescala de boxes, onde as boxes se adaptam ao tamanho original da imagem. Após a obtenção das *boxes* na escala da imagem, um corte (*crop*) na imagem original é realizado com base em uma *boxe*.

A etapa de extração de *feature* (*feature extraction*) envolve gerar um vetor representacional de uma face. Esse vetor consegue capturar diversas características de uma face. E será usado para a comparação na etapa de identificação. No *Deep Learning*, esse vetor é conhecido como *embedding*.

Em posse do vetor de *features*, esse dado deve ser persistido em um banco de dados.

Os metadados (ou *payloads*) referente ao vetor de features também deve ser persistido para ocorrer a identificação.

Existem duas abordagens para identificação de uma face. A primeira é o treinamento de algoritmos de classificação, onde dado um conjunto estático de escolhas, o algoritmo deverá prever a classe a qual o vetor de característica tem a maior probabiliade de pertencer. Esse tipo de abordagem é bastante rápida, mas sofre problemas quando uma nova classe (nova identidade) é inserida no sistema, ele deverá ser re-treinado.

A outra abordagem envolve comparação por similaridade ou distância entre pontos. Com essa abordagem é possível realizar a comparação do vetor de característica com todos que são armazenados no banco de dados. A quantidade de faces a serem registrada nessa abordagem é ilimitada do ponto de vista teórico. Mas a comparação vetor-a-vetor pode ser bastante custosa computacionalmente. As métricas de distância/similaridade comuns na literatura para esse tipo de problema são: distância euclidiana, similaridade de cosseno e o produto escalar (WANG et al., 2022).

A distância euclidiana, equação 2.1, é uma medida de distância entre dois pontos em um espaço euclidiano. Em um plano bidimensional, por exemplo, a distância euclidiana entre dois pontos (x1, y1) e (x2, y2) representa o comprimento do caminho mais curto entre dois pontos em um espaço euclidiano, quanto menor a distância, mais próximos estão os pontos.

$$d(P, Q) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{2.1}$$

A similaridade de cosseno, equação 2.2, é uma medida que avalia a similaridade direcional entre dois vetores. Ela é frequentemente usada em espaços vetoriais para comparar a direção de dois vetores, ignorando a sua magnitude. Quanto mais próximo o valor da similaridade de cosseno estiver de 1, mais similares são os vetores. Se o valor for 0, os vetores são ortogonais; se for -1, eles são opostos.

$$\cos(\theta) = \frac{A \cdot B}{||A||||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{2.2}$$

O produto escalar, equação 2.3, é uma operação entre dois vetores que resulta em um número escalar. Ele mede a projeção de um vetor sobre o outro, indicando em que medida os dois vetores estão na mesma direção. Se o produto escalar é zero, os vetores são ortogonais; se for positivo, estão na mesma direção; se for negativo, estão em direções opostas.

$$A \cdot B = \sum_{i=1}^{n} A_i B_i \tag{2.3}$$

O *Hierarchical Navigable Small World* (HNSW) é um algoritmo de indexação de aproximação usado em problemas de pesquisa de vizinhos próximos em conjuntos de dados de alta dimensionalidade (MALKOV; YASHUNIN, 2018). Ele é projetado para

eficientemente encontrar objetos semelhantes em espaços multidimensionais, como ocorre em tarefas de recuperação de informações (*information retrieval*).

O algoritmo HNSW começa com a criação de um grafo que representa a hierarquia dos dados. Os pontos de dados são adicionados à estrutura hierárquica, e conexões são estabelecidas com base na proximidade dos pontos no espaço multidimensional. A estrutura hierárquica permite a navegação eficiente pelos dados em vários níveis de granularidade. Começando de um nó de nível mais alto, a busca se aprofunda progressivamente em direção a regiões mais específicas do espaço.

O termo *Small World* refere-se à propriedade de que mesmo em um espaço de alta dimensionalidade, os pontos podem ser alcançados de forma eficiente por meio de um número relativamente pequeno de conexões. O grafo é construído de maneira a preservar essa propriedade, tornando possível alcançar vizinhos próximos em um número pequeno de etapas.

Para a recuperação a partir de um ponto de consulta, a busca é direcionada pelo grafo para encontrar rapidamente pontos semelhantes no espaço de alta dimensionalidade. O algoritmo HNSW suporta atualizações dinâmicas, permitindo a inclusão ou remoção eficiente de pontos de dados na estrutura existente sem a necessidade de reconstruir todo o índice.

As *Vector Database* que implementam o algoritmo HNSW conseguem realizar uma busca em escala na ordem de milisegundos (QDRANT, 2023a). Em posse da identificação da face, é possível então recuperar outras informações da pessoa. Essa informação pode estar gravada em outro local, por exemplo, um banco de dados relacional ou um diretório local. Nos *Vector Databases* essa informação é registrada no próprio banco como um *payload*.

2.2 Técnicas tradicionais para detecção facial

2.2.1 Viola-Jones

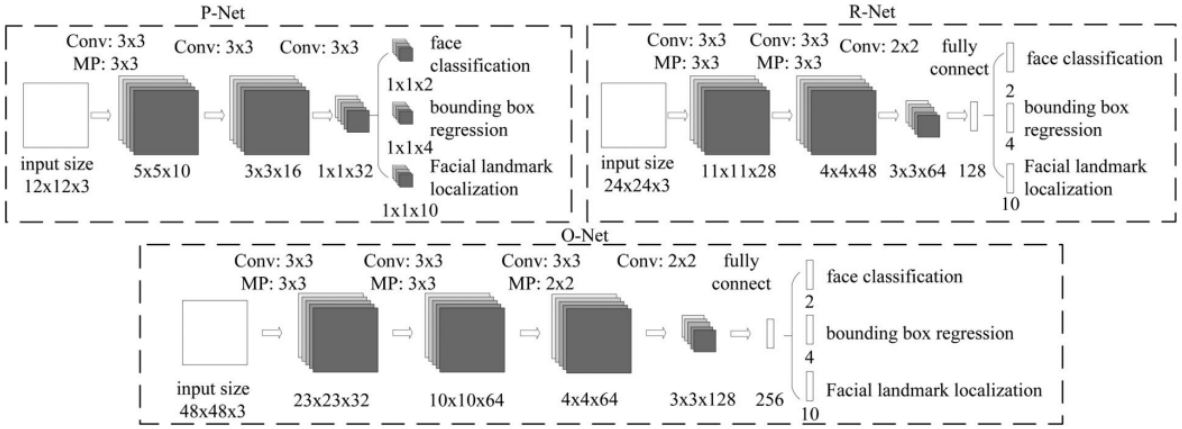
Viola-Jones é algoritmo eficiente de detecção de faces baseada em classificadores em cascata, treinados com características simples, chamadas de *Haar-like features* (VIOLA; JONES, 2001). Essas características são calculadas em regiões retangulares da imagem e são eficientes computacionalmente. As características Haar são padrões retangulares que representam diferenças de intensidade em sub-regiões adjacentes da imagem. Essas características são calculadas rapidamente e são eficientes para a detecção de objetos. Os classificadores são organizados em cascata, onde cada estágio subsequentemente mais complexo é ativado apenas se os estágios anteriores são passados. Isso permite uma rejeição rápida de regiões que não contêm o objeto de interesse.

2.3 Técnicas baseadas em *Deep Learning* para detecção facial

2.3.1 *Multi-task Cascaded Convolutional Networks*

Multi-task Cascaded Convolutional Networks (MTCNN) é um conjunto de modelos de *Deep Learning* do tipo rede neural convolucional (CNN) que foram projetadas para realizar detecção de faces em imagens (ZHANG et al., 2016). A MTCNN opera em três estágios hierárquicos: P-Net, R-Net e O-Net. Juntos, os três modelos possuem um *model size* de 2.174mb. Ele processa 16 imagens por segundo. A arquitetura está descrita na Figura 2.

Figura 2 – Arquitetura de P-Net, R-Net, e O-Net



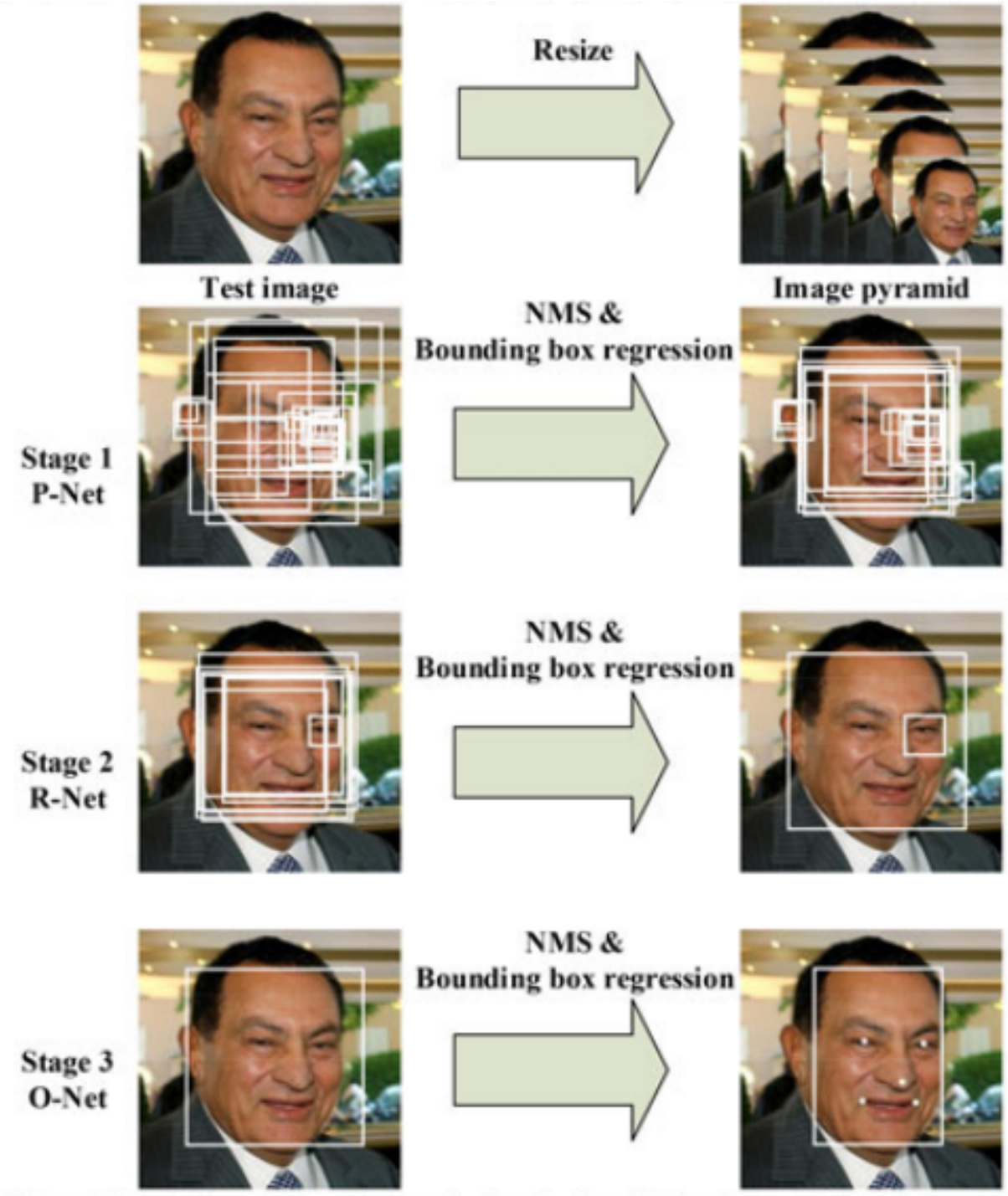
Fonte: (ZHANG et al., 2016)

A saída da MTCNN é composta por 3 saídas. A primeira são caixas delimitadoras (*bounding boxes*) que informam onde estão localizadas as faces na imagem. As *bounding boxes* são normalizadas, isso é feito para garantir a consistência e facilitar a comparação entre diferentes escalas de imagens. A segunda saída são os níveis de confiança (*scores*) que o modelo tem de que aquela caixa gerada pertence a uma pessoa. E a última saída são os pontos-chave faciais como olhos, nariz e boca.

A Figura 3 descreve um processo visual do funcionamento da MTCNN, onde *Non-Maximum Supression* (NMS) é utilizada para reduzir o número de *boxes* propostas por área. O NMS funciona da seguinte forma:

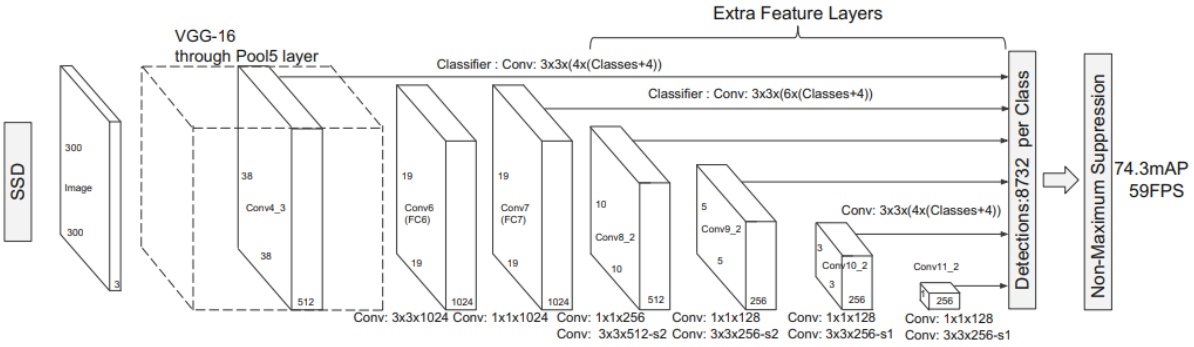
- Ordenação das detecções: As detecções são ordenadas de acordo com a confiança da detecção.
- Seleção da Detecção de Maior Confiança: A detecção com a maior confiança é mantida como uma detecção final.

Figura 3 – Exemplo de funcionamento da MTCNN



Fonte: (ZHANG et al., 2016)

Figura 4 – Arquitetura da SSD



Fonte: (LIU et al., 2016)

- Remoção de Detecções Sobrepostas: As detecções que têm uma sobreposição significativa (IoU - *Intersection over Union*) com a detecção de maior confiança são removidas.
- Repetição: Os passos 2 e 3 são repetidos até que todas as detecções sejam processadas.

2.3.2 FaceBoxes

FaceBoxes é um detector de faces projetado para executar em CPU (ZHANG et al., 2017). Ele tem uma estrutura de rede leve que possibilita operar rapidamente. O modelo utiliza Camadas Convolucionais Rapidamente Digeridas (RDCL, sigla em inglês) e nas Camadas Convolucionais de Escala Múltipla (MSCL, sigla em inglês). O RDCL possibilita a FaceBoxes processar em tempo-real. O MSCL visa enriquecer os campos receptivos e discretizar âncoras em diferentes camadas para lidar com faces de várias escalas. FaceBoxes consegue processa 20 imagens por segundo. O seu *model size* é de 3.966mb.

2.3.3 Ultra LightWeight

Single Shot Multibox Detector (SSD), é um tipo de detector de objetos em imagens que pertence à categoria de detecção de objetos em tempo real (LIU et al., 2016). Ele foi projetado para ser eficiente e preciso, capaz de detectar objetos em diferentes escalas e aspectos em uma única passagem pela rede neural. A arquitetura SSD utiliza uma rede neural convolucional para extrair características da imagem e, em seguida, gera predições de detecção em várias escalas. Na Figura 4 é descrita a arquitetura do SSD.

Ultra LightWeight é um detector de face baseado em SSD (LINZAER, 2023). Seu *model size* é de 1.241mb. Os autores treinaram ele com duas resoluções de imagem: (320, 240, 3) e (640, 480, 3). Ele também possui duas versões de arquitetura: rbf e *slim*. A *slim* é uma versão mais compacta da RBF tendo uma precisão um pouco menor. A saída da rede *boxes* e *scores*. Onde as boxes também são normalizadas. O modelo opera com duas classes de objetos: fundo (*background*) e face.

2.4 Técnicas tradicionais para extração de *features*

2.4.1 *Principal Component Analysis*

Principal Component Analysis (PCA) é uma técnica de redução de dimensionalidade (PEARSON, 1901). O PCA transforma um conjunto de variáveis correlacionadas em um novo conjunto de variáveis não correlacionadas, chamadas de componentes principais, de modo que a maior parte da variabilidade nos dados seja explicada pelos primeiros componentes principais. Ao projetar os dados em um subespaço de menor dimensão, o PCA permite reduzir a complexidade do modelo, mantendo as principais informações. O PCA busca as direções (componentes principais) ao longo das quais os dados têm a maior variabilidade. Os componentes principais são não correlacionados, o que simplifica a interpretação dos padrões presentes nos dados.

2.4.2 *Linear Binary Pattern Histogram*

Esse método une a técnica de *Linear Binary Pattern* (LBP) com o uso de histogramas (OJALA; PIETIKAINEN; MAENPAA, 2002; AHONEN; HADID; PIETIKÄINEN, 2004). O LBP é uma técnica para capturar informações locais sobre a textura em uma imagem. Ele funciona comparando o valor de intensidade de um pixel central com os valores de intensidade de seus *pixels* vizinhos. Essas comparações são usadas para construir um padrão binário que representa a textura local em torno do *pixel* central. O LBPH cria histogramas que contam a frequência dos diferentes padrões binários na imagem. Esses histogramas podem ser usados como um vetor de *features* de uma face.

2.4.3 *Latent Dirichlet Allocation*

Latent Dirichlet Allocation (LDA) (BLEI; NG; JORDAN, 2003) é um método estatístico que funciona encontrando a combinação linear de características que melhor separa duas ou mais classes de objetos ou eventos. No contexto do reconhecimento facial, o LDA é usado para encontrar os vetores no espaço de faces que melhor discriminam entre diferentes faces. Esses vetores formam um subespaço, conhecido como "espaço de faces", no qual as faces são representadas.

2.5 Técnicas baseadas em *Deep Learning* para extração de *features*

2.5.1 FaceNet

FaceNet é um modelo do CNN (SCHROFF; KALENICHENKO; PHILBIN, 2015). A FaceNet é um modelo que foi treinado para aprender a mapear imagens faciais para um espaço euclidiano de dimensão compacta, no qual as distâncias correspondem diretamente a uma medida de similaridade facial. Esse vetor de *features* é conhecido como *embedding*, e na FaceNet possui 128 dimensões. A partir de uma *embedding* é possível realizar tarefas como reconhecimento, verificação e agrupamento facial.

A FaceNet foi estruturada da seguinte maneira: as imagens de entrada com dimensão (224, 224, 3), uma rede convolucional pré-treinada - onde a escolhida pelos autores foi a Inception (SZEGEDY et al., 2015) - e uma normalização L2, que resulta em uma camada de *embedding* como saída do modelo. Essa combinação gera com um *model size* de 89.015mb.

Para o treinamento, os autores utilizaram a função de perda *Triple Loss*. A *Triple Loss* foi projetada para minimizar a distância entre as representações de imagens semelhantes e maximizar a distância entre as representações de imagens diferentes. Por exemplo, dado três imagens: uma imagem âncora (representando a face de uma pessoa), uma imagem positiva (representando outra instância da mesma pessoa), e uma imagem negativa (representando uma pessoa diferente). FaceNet atingiu uma pontuação de 99.63% de precisão no conjunto de dados *Labeled Faces in the Wild* (LFW) (HUANG et al., 2007),

2.5.2 Mobile FaceNet

A Mobile FaceNet é re-implementação da FaceNet para dispositivos móveis e sistemas de tempo-real (CHEN et al., 2018). Ela possui como entrada imagens na proporção (112, 112, 3), as imagens precisam passar por um pré-processamento de subtração de 127 e depois dividido por 128. A MobileNetV2 foi utilizada para extrair as *features* (SANDLER et al., 2018). A dimensão de saída é 128. A Mobile FaceNet possui um *model size* de 3.997mb. Atingindo uma latência de 24ms. Ela atinge uma pontuação de 99.55% no conjunto de dados LFW.

Ela foi treinada usando ArcFace como função de perda (DENG et al., 2022). A ArcFace introduz uma função angular (função de arco) na camada de saída da rede neural. A ideia é que as representações faciais geradas pela rede sejam projetadas em um espaço onde as características de faces semelhantes estão mais próximas, enquanto as características de faces diferentes estão mais afastadas.

A função angular incorpora uma margem angular específica, que controla a dispersão angular entre as representações faciais. A função de perda do ArcFace busca maximizar o cosseno do ângulo entre a representação da classe verdadeira e a representação de

outras classes, enquanto minimiza a variabilidade dentro da mesma classe. Isso resulta em uma melhor separação entre as representações faciais, tornando o modelo mais robusto e discriminativo para tarefas de reconhecimento facial.

2.6 Técnicas classificação de *features*

2.6.1 *Support Vector Machine*

A *Support Vector Machine* (SVM) é um algoritmo de aprendizado de máquina usado tanto para classificação quanto para regressão (VAPNIK; CHERVONENKIS, 2015). SVM é usado para encontrar um hiperplano de decisão que melhor separa os dados em diferentes classes. A margem é a distância entre o hiperplano e os pontos de dados mais próximos de cada classe. SVM busca o hiperplano que maximiza essa margem. Os vetores de suporte são os pontos de dados mais próximos do hiperplano. Para classificar novos pontos de dados, o SVM realiza a verificação de qual lado do hiperplano eles caem. A direção do lado determina a classe à qual o ponto pertence.

2.6.2 *k-Nearest Neighbors*

O *k-Nearest Neighbors* (K-NN) é um algoritmo de aprendizado de máquina utilizado tanto para classificação quanto para regressão (DUDA; HART et al., 1973). Para classificar um novo ponto de dados, o algoritmo identifica os "k"pontos mais próximos a esse ponto no espaço de características. O KNN então realiza uma votação entre os "k"vizinhos para determinar a classe mais comum. O valor de "k"é um hiperparâmetro que precisa ser escolhido. A função de distância comumente usada é a distância euclidiana.

2.6.3 *Multi-Layer Perceptron*

A utilização de *Perceptrons* de Múltiplas Camadas (*Multi-Layer Perceptron* - MLPs) tem se destacado como uma abordagem eficaz para tarefas de classificação em diversos domínios. MLPs são um tipo de rede neural artificial composta por camadas de neurônios, incluindo uma camada de entrada, uma ou mais camadas ocultas e uma camada de saída (RUMELHART; HINTON; WILLIAMS, 1986). Essa arquitetura permite a modelagem de relações complexas nos dados, tornando as MLPs particularmente adequadas para problemas de classificação em que a estrutura subjacente dos dados é intrincada. A capacidade das MLPs de aprender representações hierárquicas e não lineares tem sido fundamental para o sucesso em diversas aplicações, como reconhecimento de padrões, processamento de linguagem natural e visão computacional.

3 Sistemas de Aprendizado de Máquina

Em novembro de 2016, o Google divulgou que introduziu um sistema de tradução multilingual usando redes neurais [1]. Esse foi o primeiro grande sucesso de um modelo de *Deep Learning* em escala (HUYEN, 2022).

3.1 Principais caraceterísticas

Sistemas de Aprendizado de Máquina (*Machine Learning Systems*) são compostos por dados, código, modelos, infraestrutura, lógica de negócios, monitoramento e observabilidade (HUYEN, 2022).

Esses sistemas devem ser confiáveis. Um sistema confiável deve ser capaz de se auto-corrigir diante de uma adversidade. Eles também devem ser escaláveis, onde o sistema ao notar um aumento de tráfego eles tem que ser capazes de se auto-dimensionar horizontalmente para atender a demanda e se o tráfego cair ele deve se auto-dimensionar para poupar o *hardware* utilizado. *ML Systems* também deve ser mantíveis, então, dados, códigos, modelos, dependências e outros artefatos devem ser versionados e documentados para que possam ser reproduzíveis (HUYEN, 2022).

Os sistemas de aprendizado de máquina são comumente divididos entre *online* e *offline* (GÉRON, 2022). Os sistemas *online*, também conhecidos como sistemas de tempo-real, são servidos na internet e esperam receber dados via requisições de clientes. Esse tipo de sistema exige que a latência seja baixa.

Em sistemas do tipo *offline*, os dados já estão consolidados e prontos para serem processados. Esses sistemas operam normalmente em *batch* para que consigam maximizar o *throughput*.

Em sistemas distribuídos modernos, as requisições dos usuários constumam serem processadas em *dynamic batch*. O processamento em *batch* favorece os modelos de Aprendizado de Máquina pois conseguem atingir uma maior eficiência do *hardware* do que se processados serialmente. Essa abordagem faz com que um maior número de usuários possam atentido por intervalo de tempo (ou seja, um maior *throughput*). As requisições são acumuladas em uma fila durante um intervalo pequeno de tempo e então são passadas como um *batch* dinâmico aos serviços. A latência (*latency*) também é inpactada positivamente pois o usuário demorá menos tempo na fila para ser atendido.

[1] Mike Schuster, Melvin Johnson e Nikhil Thorat. "Zero-Shot Translation with Google's Multilingual Neural Machine Translation System". Google AI Blog, 22 de novembro de 2016, https://blog.research.google/2016/11/zero-shot-translation-with-googles.html

3.2 Ferramentas utilizadas

Esse trabalho utilizou um conjunto de ferramentas. Essa seção descreve seus detalhes e funcionamentos.

3.2.1 HTTP

Hypertext Transfer Protocol (HTTP) é o protocolo mais comum na internet. Ele define como as mensagens são formatadas e transmitidas pela internet e especifica a interação entre clientes e servidores da web. O HTTP opera no modelo cliente-servidor, onde um cliente faz uma solicitação e espera por uma resposta do servidor. Cada solicitação do cliente é tratada independentemente, sem conhecimento do estado anterior. Isso significa que o servidor não mantém informações sobre a sessão do cliente entre solicitações. As mensagens HTTP são legíveis para humanos.

O HTTP/2 é a segunda versão do HTTP. Ele foi projetado para ser uma melhoria significativa em relação ao HTTP/1.1 em termos de desempenho e eficiência de comunicação entre clientes e servidores web. Uma das melhorias mais significativas no HTTP/2 é a capacidade de realizar várias solicitações e respostas simultaneamente em um único canal de comunicação. Enquanto o HTTP/1.1 usa texto legível para as mensagens, o HTTP/2 adota a abordagem binária para a serialização de dados. Isso reduz a quantidade de dados transmitidos, melhora a eficiência e facilita o processo de análise por máquinas.

3.2.2 REST

Transferência de Estado Representacional (REST, sigla em inglês) é um estilo de arquitetura para sistemas de hipermídia distribuídos. Uma API REST é uma API web que segue as restrições do estilo de arquitetura REST. As APIs REST não possuem estado, então cada solicitação necessita incluir as informações necessárias para o processamento dela. Os dados são trafegados usando HTTP. Os dados são formatados normalmente como Javascript Object Notation (JSON) ou como Multipart. O Multipart permite o tráfego de arquivos para um servidor HTTP. É usado para fazer o *upload* de arquivos a um servidor web.

3.2.3 gRPC

O gRPC é um framework para a criação de serviços eficientes e rápidos, com suporte a vários tipos de linguagens de programação. Ele utiliza o *Protocol Buffers* (protobuf) como formato para serialização de dados e o HTTP/2 como protocolo de transporte. O gRPC propõe que o cliente interaja com o servidor por meio de chamadas de funções simples, ou seja, de interfaces de códigos geradas automaticamente pela própria aplicação

do gRPC. Nele existe o suporte ao *streaming* bidirecional, o REST implementa apenas solicitações cliente para oservidor.

3.2.4 Locust

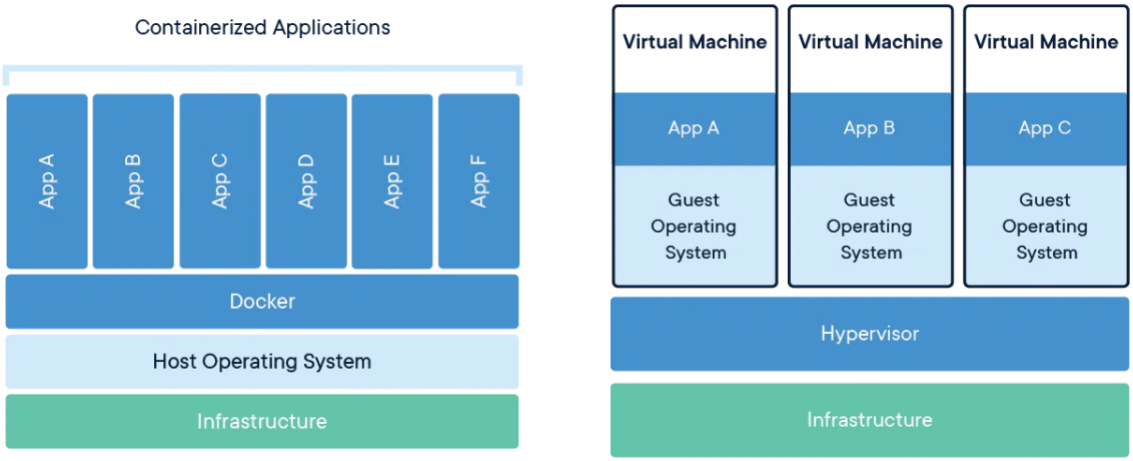
Para medir a escalabilidade de APIs cliente-servidor testes de cargas (*load tests*) podem ser realizados. Esses testes simulam vários clientes realizando requisições ao mesmo tempo. A quantidade de clientes pode variar conforme o tempo. Medidas como latência e requisições por segundo por ser capturadas com esses teste. O Locust é uma ferramenta de *load tests*. Os testes são executados de forma distribuída. Milhões de usuários podem ser simulados ao mesmo tempo.

3.2.5 Docker

Docker é uma plataforma que usa virtualização em nível de sistema operacional para disponibilizar pacotes de *softwares* chamados de *container*. Uma imagem de um *container* Docker é um pacote de *software* leve, independente e executável que inclui tudo o que é necessário para executar um aplicativo: código, tempo de execução, ferramentas do sistema, bibliotecas e configurações do sistema (DOCKER, 2023). Durante o tempo de execução, as imagens de *container* se tornam *containers*.

Tradicionalmente, as aplicações de *software* eram servidas usando *Virtual Machines* (Máquinas Virtuais) (VMs). A principal diferença entre VMs e *containers* é que as VMs virtualizam o *hardware* ao contrarios dos *containers*, que virtualizam o sistema operacional. Na Figura 5 podemos ver um comparativo do esquema de funcionamento entre as duas tecnologias.

Figura 5 – Comparativo do funcionamento entre containers e VMs



Fonte: (DOCKER, 2023)

3.2.6 Prometheus

Prometheus é um kit de ferramentas de alerta e monitoramento de sistemas. Ele coleta e armazena métricas como dados de séries temporais (PROMETHEUS, 2023). O modelo de dados do Prometheus podem ser identificados nome de métrica e pares chave/valor de PromQL, uma linguagem de consulta flexível para aproveitar essa dimensionalidade. Não possui dependência de armazenamento distribuído. Os dados são coletados por um modelo *pull* sobre o HTTP. Esses dados podem ser consumidos por serviços de *dashboards*.

3.2.7 Grafana

O Grafana é um serviço de monitoramento e observabilidade por meio de *dashboards* (GRAFANA, 2023). Ele permite consultar, visualizar, alertar e entender as métricas do sistema. E possui integração com o Prometheus, de forma a consumir as métricas capturadas por meio de solicitações usando o PromQL.

3.2.8 Kubernetes

Kubernetes é uma plataforma de orquestramento de *containers* para automatizar a implantação, o *deployment* e o gerenciamento de aplicativos em *containers* (KUBERNETES, 2023).

3.2.8.1 Principais características

Principais características do Kubernetes:

- **Service discovery e load balancing** O Kubernetes pode expor um *conteiner* usando o nome DNS ou seu próprio endereço IP. Se o tráfego para um *conteiner* for alto, o Kubernetes será capaz de balancear a carga e distribuir o tráfego da rede para que o *deployment* seja estável.
- **Storage orchestration** O Kubernetes permite montar automaticamente um sistema de armazenamento de sua escolha, como armazenamentos locais e *cloud providers*.
- **Automated rollouts e rollbacks** Você pode descrever o estado desejado para seus *conteiners* implantados usando o Kubernetes e pode alterar o estado real para o estado desejado em uma taxa controlada. Por exemplo, você pode automatizar o Kubernetes para criar novos *conteiners* para o *deployment*, remover *conteiners* existentes e adotar todos os seus recursos no novo *conteiner*.
- **Embalagem automática de bin** Você fornece ao Kubernetes um *cluster* de nodes (nós) que ele pode usar para executar tarefas em *conteiners*. Você informa ao Kubernetes quanta CPU e memória (RAM) cada contêiner precisa. O Kubernetes pode acomodar *conteiners* em seus nodes para fazer o melhor uso de seus recursos.

- **Self-healing** O Kubernetes reinicia *conteiners* que falham, substitui *conteiners*, elimina *conteiners* que não respondem à verificação de integridade definida pelo usuário e não os anuncia aos clientes até que estejam prontos para servir.
- **Gerenciamento de Secrets e Configurações** O Kubernetes permite armazenar e gerenciar informações confidenciais, como *Secrets* (senhas), *tokens* OAuth e chaves SSH. Você pode implantar e atualizar *Secrets* e configurações de aplicativos sem reconstruir as imagens de *conteiner* e sem expor segredos na configuração da pilha.

3.2.8.2 Compenentes

Um *cluster* Kubernetes consiste em um conjunto de ***worker machines*** (máquinas de trabalho), chamadas ***nodes***, que executam aplicativos em *conteiners*. Cada *cluster* possui pelo menos um ***worker node***.

Pods são um grupo de um ou mais *conteiners* que podem agrupar diferentes imagens de *conteiner*. O objeto ***Deployment*** gerencia o lançamento de novas versões, ele permite aplicar o *rollout* de versões sem *downtime* ou erros. Em um *Deployment* é definido as Pods *especification* (especificações dos Pods) e as informação de Scale (escala).

Service é um método de expor um aplicativo de rede que está rodando em um ou mais Pods no *cluster*. **ClusterIP** é um objeto do tipo *Service* que expõe em um IP interno do *cluster*, tornando-o acessível apenas de dentro do cluster. Para expor o serviço a internet é necessário um objeto ***Ingress***, ele expõe rotas HTTP e HTTPS de fora do cluster para serviços dentro do cluster. O roteamento de tráfego é controlado por regras definidas nele.

Um objeto ***Job*** cria um ou mais Pods e vai continuar tentando executar os Pods até que um número especificado deles seja encerrado com sucesso.

3.2.8.3 Custom Resources

Custom Resource (Recurso Personalizados) são uma maneira de extender a API do Kubernetes para incluir objetos personalizados que vão além dos tipos de recursos padrão, como Pods, Services e Deployments.

A estrutura de um *Custom Resource* é definida por meio de um *Custom Resource Definition* (CRD), onde é especificado os campos, o esquema de validação e outras propriedades.

Um controlador personalizado deve ser implementado para lidar com os objetos do *Custom Resource*. O controlador monitora eventos no *cluster* Kubernetes e age de acordo com as alterações nos recursos personalizados.

3.2.8.4 Helm

Helm é um gerenciador de pacotes do Kubernetes. Ele fornece a capacidade de fornecer, compartilhar e usar software desenvolvido para Kubernetes (CNCF, 2023).

3.2.9 Python

O Python é uma linguagem de programação versátil e de alto nível, conhecida por sua sintaxe clara e legível. Ela suporta múltiplos paradigmas de programação, incluindo programação funcional, imperativa e orientada a objetos. Sua tipagem dinâmica e natureza interpretada facilitam a prototipação rápida e a iteração.

Uma das principais filosofias do Python é a ênfase na legibilidade do código, tornando-o uma excelente escolha para programadores iniciantes e experientes. Além disso, o Python oferece a capacidade de se integrar com outras linguagens de programação, como C/C++, Fortran e Rust. Esta característica permite que o Python atue como uma interface para estruturas de código otimizadas escritas nessas linguagens.

Essa integração tem sido fundamental para o desenvolvimento de bibliotecas de alto desempenho no Python, permitindo a combinação de fácil prototipação e eficiência computacional. Essas qualidades têm impulsionado o uso do Python na construção de modelos de Machine Learning, solidificando sua posição como uma linguagem de programação dominante neste campo.

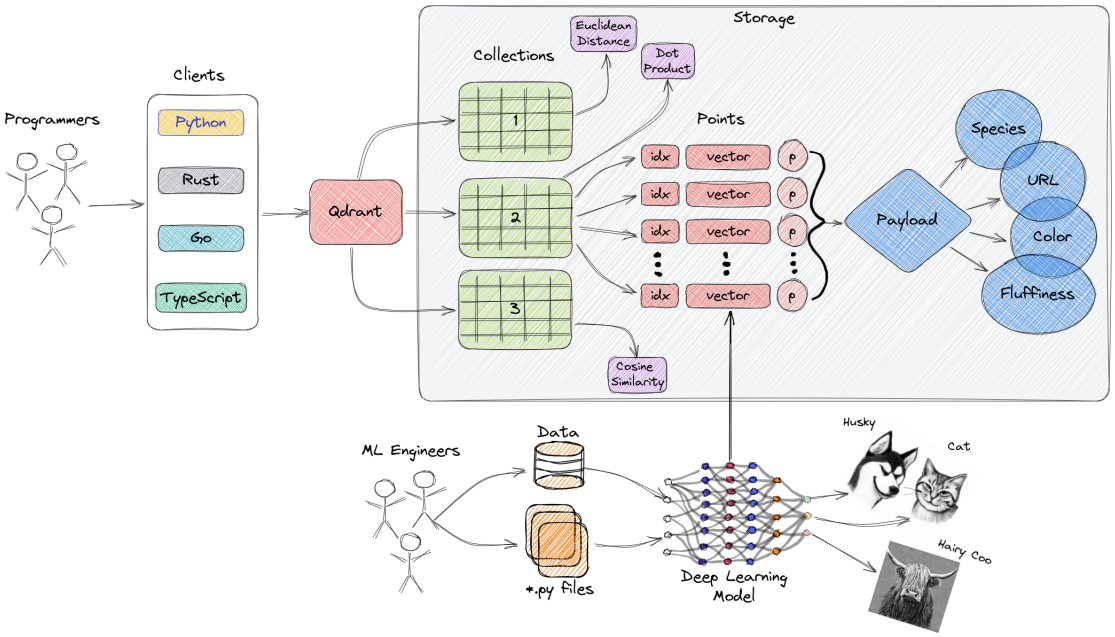
3.2.10 FastAPI

FastAPI é uma estrutura da web de alto desempenho para construir APIs com Python com base em dicas de tipo. Seus principais recursos são sua velocidade, facilidade de codificação e sua robustez. Ele implementa padrões de APIs como o OpenAPI e esquemas JSON (RAMíREZ, 2023).

3.2.11 Qdrant

Qdrant é um *vector database* (banco de dados vetorial). Ele interage com representações geradas por modelos de *Deep Learning*. Essas representações são conhecidas como *embeddings*, um vetor denso de alta dimensão de um certo tipo dado, como texto ou imagem, por exemplo. Nesse contexto, o vetor é uma representação matemática de um ponto, e cada elemento do vetor corresponde a uma *feature* do dado. Nos Vector Database a estrutura é composa por um ID, um *embedding* e por um *payload*, esse conjunto de dados são armazenados em uma *collection* (coleção) (QDRANT, 2023b). Na Figura 6 podemos ver uma representação do funcionamento do Qdrant.

Figura 6 – Representação do funcionamento do Qdrant



Fonte: (QDRANT, 2023b)

Em bancos de dados relacionais que possuem estruturas organizadas em linha-coluna como OLAP e OLTP (SINHA, 2023), as consultas são realizadas com base nos valores das colunas. Os *vector database* são otimizados para armazenar e consultar *embeddings* de forma eficiente. O Qdrant implementa um algoritmo eficiente de indexação chamado *Hierarchical Navigable Small World* (HNSW) (MALKOV; YASHUNIN, 2018), que implementa o algoritmo *Approximate Nearest Neighbors* (Vizinhos Mais Próximos). Com ele, um vetor de *embedding* vira um ponto no espaço vetorial, e permite o cálculo de distância entre pontos próximos. Qdrant implementa três métricas de distância: euclidiana, similaridade de coseno e produto escalar.

Dentre os bancos de dados vetoriais de código aberto, o Qdrant é o que tem as melhores métricas de latência (QDRANT, 2023a), e por isso foi o escolhido.

O termo *Neural Search* foi introduzido pela empresa Jina AI para se referir aos sistemas de recuperação de informação que são alimentados por modelos baseados em *Deep Learning* (JINA, 2023).

3.2.12 Numpy

Numpy é uma biblioteca de computação científica, que permite cálculos de *arrays* multidimensionais e implementa diversas de operações em arrays (NUMPY, 2023).

3.2.13 PyTorch

PyTorch é uma biblioteca para o desenvolvimento de modelos baseados em *Deep Learning*. O PyTorch tem como suas principais características o cálculo otimizado de tensores, a execução dinâmica de grafos computacionais, flexibilidade e a integração com outras bibliotecas (PYTORCH, 2023). Essas caraterísticas tem tornado o PyTorch a principal escolha entre pesquisadores (que disponibilizam seus códigos) para o desenvolvimento de aplicações baseadas em *Deep Learning* (PAPERSWITHCODE, 2023).

3.2.13.1 TorchVision

O TorchVision é uma extensão do PyTorch que possui implementações de algoritmos como *redimensionamento* e normalização de imagens.

3.2.14 ONNX

De acordo com (ONNX, 2023), o ONNX é um formato aberto criado para representar modelos de aprendizado de máquina. ONNX define um conjunto comum de operadores – os blocos de construção de modelos de *Deep Learning* – e um formato de arquivo comum para permitir que os desenvolvedores de IA usem modelos com uma variedade de estruturas, ferramentas, tempos de execução e compiladores.

3.2.14.1 ONNX Simplifier

O ONNX Simflier (ou ONNX-sim) é uma extensão do ONNX que busca aplicar simplificações no grafo dos modelos e substituir redundâncias por saídas constantes (conhecidas como *constant foalding*) (DAQUEXIAN, 2023).

3.2.15 ONNX Runtime

ONNX Runtime (ORT) é uma biblioteca projetada para acelerar a inferência de modelos *Deep Learning* em uma ampla variedade de estruturas, sistemas operacionais e plataformas de hardware (RUNTIME, 2023). Ele fornece suporte a múltiplos sistemas operacionais, linguagens de programação e *hardwares*, usando um único conjunto de API. Além disso ele oferece suporte a diversos *Execution Providers* para acelerador a inferência. O ONNX é o formato de modelo utilizado pelo ONNX Runtime.

3.2.15.1 OpenVINO

Um dos *Execution Providers* suportados pelo ORT é OpenVINO, desenvolvido e mantido pela Intel (OPENVINO, 2023). Ele é um acelerador de inferência em diversos tipos de *hardware*, em especial, se destacando em processores da própria Intel. Os tipos de processadores suportados são:

- Processadores Intel® Core™ de 6ª a 13ª geração;
- Processadores escaláveis Intel® Xeon® de 1ª a 4ª geração.

Uma das maneiras do OpenVINO acelerar a inferência é por meio de redução da precisão do *float*. Os modelos de IA normalmente são desenvolvidos com computação de *float*32, mas para a inferência é possível aplicar a redução para *float*16 sem sofrer com perdas de precisão do modelo.

3.2.16 Ray

Ray é uma biblioteca *open-source* que escala aplicações de *Machine Learning* usando Python. Ela foi projetada para facilitar a criação de aplicativos paralelos e distribuídos de alto desempenho (RAY, 2023).

3.2.16.1 Ray Core

Ray Core fornece um pequeno número de primitivos básicos: *tasks* (tarefas), *actors* (atores) e *objects* (objetos), para construir e escalar aplicações distribuídas.

Ray habilita que funções sejam executadas de forma assíncrona em *workers* (trabalhadores) Python seperados. Essas funções assíncronas do Ray são conhecidas como *tasks*. Ray proporciona que essas ***tasks*** especifiquem seus recursos de requisito de *hardware*, como CPU, GPU e memória RAM, e também de escalabilidade.

Actors são uma extensão da API Ray tasks. Um *actor* é um *worker* com estado. No momento em que um novo *actor* é instanciado, um novo *worker* é criado e os métodos do *actor* são agendados nesse *worker* específico, podendo acessar e alterar o estado desse *worker*. Os requisitos de *hardware* e escalabilidade também são suportados.

Em Ray, as *tasks* e *actors* criam e computam ***objects***. Esses *objects* são conhecidos como *remote objects*, porque eles podem ser armazenados em qualquer lugar em um *cluster* Ray e usamos ObjectRef para nos referirmos a eles. Os *remote objects* são armazenados em cache no armazenamento de objetos de memória compartilhada distribuída de Ray e há um armazenamento de objetos por *node* no *cluster*. Na configuração de *cluster*, um *remote object* pode residir em um ou mais *nodes*, independentemente de quem detém a(s) referência(s) do *object*.

3.2.16.2 Ray Data

Ray Data é uma biblioteca escalonável de processamento de dados de aplicações de *Machine Learning*. Ele proporciona APIs flexíveis e de alto desempenho para dimensionar inferência em *batch* off-line e pré-processamento e ingestão de dados para treinamento de ML. Ray Data usa execução de *streaming* para processar grandes conjuntos de dados de forma eficiente.

3.2.16.3 Ray Serve

Ray Serve é uma biblioteca para construção de APIs de inferência online. Ele é *framework-agnostic*, ou seja, ele permite o uso de várias estruturas de Machine Learning como PyTorch, TensorFlow, ONNX Runtime, entre outros. Ele suporta *dynamic request batching* (lote de requisições dinâmicas), respostas em *streaming*, etc.

Ray Serve é bastante útil para composição de modelos e muitos serviços deles. Permitindo a construção de uma lógica de um serviço de inferência complexo. A forma de construir isso é usando os Deployments do Ray como serviços isolados que são controlados por outro *Deployment*. Esses *Deployments* são escaláveis horizontalmente, ou seja eu posso redimensionar a quantidade ativa de *Deployments* baseado em demanda de serviço.

O decorador do ray *@serve.batch* possibilita uma função acumular em uma fila uma certa quantidade de carga de requisições por um certo intervalo de tempo. O tamanho máximo desse *batch* e o tempo máximo de espera por essas requisições podem ser configurados.

3.2.16.4 Ray Dashboard

Ray fornece um *dashboard* baseado na web para monitorar e depurar os aplicativos Ray. A representação visual do estado do sistema permite aos usuários acompanhar o desempenho dos aplicativos e solucionar problemas. Ele se integra ao Prometheus e o Grafana para coletar métricas extras.

3.2.17 KubeRay

O KubeRay é um *Operator* do Kubernetes. É a maneira recomendada de criar aplicações Ray no Kubernetes. O *Operator* fornece uma maneira nativa do Kubernetes de gerenciar *clusters* Ray. Ele permite que o Ray escale seus *clusters* conforme demanda usando o escalonamento automático do Kubernetes, adicionando ou removendo *pods*.

3.2.17.1 RayCluster

Um RayCluster é um objeto Kubernetes do tipo custom resource, onde é possível descrever o estado de um RayCluster. KubeRay que gerencia os *pods* Ray para atender as especificações do RayCluster.

Um RayCluster pode ser visto como uma coleção de *pods* do Kubernetes, semelhante a um objeto *Deployment* do Kubernetes. A diferença entre eles é que um RayCluster é especializado em executar aplicações Ray.

Um RayCluster consiste de:

- Um ***head pod*** (*pod* principal) que hospeda processos de controle global para o *cluster* Ray. O *head pod* também pode executar tarefas e atores do Ray;

- Qualquer número de ***worker pod*** (*pod* trabalhador) que executa tarefas e atores do Ray. Os *worker pods* são configurados em *worker groups* (grupos de trabalhadores) de *pods*. Cada *worker group* pode ter sua própria configuração. Para cada *worker group* devemos especificar o número de **replicas** de Pods que queremos desse grupo. Também é possível especificar os campos de **minReplicas** e de **maxReplicas**, que são necessários se for desejado o uso de *horizontal scalability*.

  Os códigos das aplicações precisam ser armazenados apenas no *head pod*.

3.2.17.2 RayJob

Com RayJob, KubeRay cria automaticamente um RayCluster e envia um *job* (trabalho) quando o *cluster* está pronto. Um RayJob pode ser configurado para se auto excluir quando tiver sido finalizado.

3.2.17.3 RayService

RayService é composto de duas partes: um RayCluster e de *deployment graphs* do Ray Serve. RayService oferece atualizações com tempo de inatividade zero para RayCluster e alta disponibilidade.

4 Trabalhos Relacionados

A literatura aborda duas maneiras de realizar a identificação de uma face: classificação e por comparação. Diferentes formas foram propostas para a realização de cada etapa.

4.1 Identificação por classificação

Em (SAYPADITH; ARAMVITH, 2018) os autores implementaram um sistema embarcado com um acelerador de hardware NVIDIA Jetson TX2 board. Os autores usaram um detector de face MTCNN. Para extração de features eles usaram a FaceNet baseado em SqueezeNet (IANDOLA et al., 2016), sendo uma versão menor que a original, baseada em Inception, com o propósito de acelerar a inferência. O conjunto de dados de treinamento foi o VGGFace (PARKHI; VEDALDI; ZISSERMAN, 2015). Para realizar um *matching* de identificação eles treinaram uma SVM, se a feature fosse desconhecida ele retornaria exatamente isso, sendo conhecida, ele identificaria que a classe e então buscaria no banco de dados. Como resultados eles conseguiram para imagem com 1 face o tempo de reconhecimento foi de 0.19, para até 8 foi 0.41s. Com uma taxa de erro mínimo de 83.67%.

Uma abordagem análoga foi apresentada por (JOSE et al., 2019), que desenvolveu uma solução incorporada utilizando a placa NVIDIA Jetson TX2. Na implementação, a técnica MTCNN foi empregada para a detecção de faces, enquanto a FaceNet foi utilizada para a extração de características (*features*). Além disso, um classificador foi empregado para determinar a qual classe a face detectada pertence.

Em (ANWAR; NADEEM; TANVIR, 2022) os pesquisadores proporam o uso da FaceBoxes como detector de face e FaceNet como extrator de *features*, e para classificação eles utilizaram uma MLP. O caso de uso também é uma aplicação embarcada, onde dois tipos de *hardware* foram testados: NVIDIA Jetson Nano e TX2. Os modelos foram originalmente desenvolvidos em TensorFlow. A FaceNet foi convertida em ONNX e depois otimizados para o formato *engine* do NVIDIA TensorRT. No Jetson Nano o detector teve uma precisão de 99,88% e o módulo de reconhecimento conseguiu uma precisão de 98,46%. No Jetson TX2 os números foram, respectivamente, 99,89% e 99,42%. Para imagens com uma face o sistema atingia 80ms, para imagens com três faces a latência era de 140ms.

A combinação MTCNN, FaceNet e SVM foi proposta em (ARSENOVIC et al., 2017; EIAMSAARD; BAMRUNGTHAI; JITPAKDEEBODIN, 2021; CHAUDHARI et al., 2023). Nos primeiros casos, os autores exporam o sistema como uma aplicação web. Onde uma câmera captura imagens em tempo real e envia para o serviço web que irá identificar a face com base nas suas classes previamente definidas. No terceiro caso os dados vinham de uma webcam.

Em (BELI; GUO, 2017) um sistema mais simples foi proposto onde ele parte de já ter a face cortada/alinhada. As imagens são carregadas do banco de dados, LBP é utilizado para extração de *features* e o K-NN é responsável por realizar a identificação da face.

A abordagem de MTCNN com FaceNet também foi proposta em (IRBAZ; NASIM; FERDOUS, 2022). Para a classificação das embeddings o K-NN foi utilizado.

4.2 Identificação por comparação

Os autores de (CHOWDHARY et al., 2017) desenvolveram um sistema onde uma webcam vai capturar vídeo em tempo real. As faces são detectadas usando MTCNN. Para extração de *features* foram testados PCA, LDA e LBPH. Com o vetor de características gerado por um dos três algoritmos, é realizada a etapa de *matching* com as faces que estão um banco de dados relacional, no qual tinha 10 registros. As comparações usaram distância euclidiana como métrica de distância. O experimento mostrou que o PCA obteve a maior precisão de reconhecimento.

Em (TEOH et al., 2021) foi proposta a utilização de acelerador de modelos via *software* usando OpenVINO. Um SSD ajustado foi usado para realizar a detecção de faces e uma MobileNet V2 extraia as *embeddings* da face. As faces registradas estão em uma base de dados local. E a medida de similaridade utilizada foi a similaridade de cosseno. Em uma CPU Intel i7 com 6 *cores* de 2.6GHz cada, o sistema atingiu 14ms de latência. Com uma Intel UP Board (CPU para sistemas embarcados), o sistema teve uma latência de 60ms.

(LI; CHA, 2018) Treinaram uma rede neural para gerar *embeddings*, utilizaram distância euclidiana para mensurar a similaridade com as *features* da faces que estão em um banco de dados relacional. O serviço foi exposto como um cliente-servidor, onde métodos *async* foram utilizados para o servidor não ser bloqueante.

Para trabalhar em escala de grandes conjunto de dados, (RANJAN; BEHERA; REZA, 2020) proporam uma abordagem que usa pesquisa parelala. Para detecção de faces usaram um detector baseado em SSD que usa o *backbone* da ResNet10, e a Inception para geração de *embeddings*. Para detecção em *real-time* eles carregavam as imagens de um diretório local e as transformavam em um banco de dados. Usando *threads* eles dividam a carga de busca entre cada uma delas.

Para resolver o problema do tempo de *matching*, (CHUNHONG; GUANGDA; XIAODONG, 2003) proporam usar um sistema distribuído de *cluster* de várias máquinas. Uma abordagem similar é vista em (MENG et al., 2005), onde os autores tinham 2.5mi de faces e usaram 6 máquinas com processadores XEON 2.4GHz, eles obtiveram um tempo de resposta de cerca de 1s. O uso do algoritmo de HNSW foi proposto para acelerar a comparação de similaridade das *embeddings* (LI et al., 2021).

5 Metodologia

Detectada a lacuna na literatura sobre como ter sistemas de reconhecimento facial escaláveis e resilientes, esse trabalho concentra-se em desevolver uma arquitetura que atenda esses requisitos. Para obtenção desses requisitos, o sistema irá usar as técnicas de fusão de etapas, processamento com *batch* dinâmico, uso de banco de dados vetoriais, além de ferramentas de resiliência de serviços.

Para construção desse experimento, os modelos Mobile FaceNet [1] e Ultra LightWeight 320-rbf [2] foram escolhidos dado as suas eficiências computacionais aliado ao alto desempenho na realização em suas determinadas tarefas.

A seguir, foram descritas as etapas realizadas para o desenvolvimento do sistema. A primeira delas explica o aprimoramento dos modelos, onde é discutida as formas de fusão de etapas. Em seguida, é descrito a construção das classes bases. Depois vem a seção de configuração do ambiente. E por fim, como foram feitas as etapas de ingestão e recuperação dos dados referentes a faces.

5.1 Aprimoramento dos modelos

5.1.1 Exportar modelos para ONNX

Ao exportar os modelos de PyTorch para o formato ONNX podemos modificar as entradas dos modelos para aceitar *dynamic axes* nas dimensões de *batch*. Em seguida, foi aplicado o ONNX-sim para obter um modelo com simplifações e remoção de redundâncias na estrutura do grafo.

5.1.2 Fusão de etapas de processamento nos modelos

Os modelos originais foram escritos em PyTorch e possuem como entrada um array no formato (*batch*, *channels*, *height*, *width*). Conforme mostrado na Tabela 1, os modelos têm dimensões de entrada diferentes.

Tabela 1 – Modelos e suas dimensões de entrada

Modelo	Dimensões de entrada
Mobile FaceNet	(N, 3, 112, 112)
Ultra LightWeight	(N, 3, 240, 320)

[1] Implementação da Mobile FaceNet usada: <https://github.com/foamliu/MobileFaceNet>

[2] Implementação do Ultra LightWeight usada: <https://github.com/Linzaer/Ultra-Light-Fast-Generic-Face-Detector-1MB>

5.1.2.1 Construção de modelos de pré-processamento

Ambos os modelos possuem etapas de pré-processamento iguais. Primeiro ocorre o *resize* (redimensionamento) para adequar a imagem de entrada ao dimensão em que o modelo foi treinado. Em seguida, a etapa de *normalize* normalização onde será subtraído um valor de 127 em cada canal de cor e depois será aplicada uma normalização com um fator de escala de 1/128.

Usando as transformações do TorchVision é possível construir um modelo com pré-processamento no PyTorch. O benefício se dá pela realização dessas etapas em *batch*.

Dado que as imagens irão ser processadas estão formatadas como (width, height, channels), podemos então aplicar uma transformação de *permute/transpose* (permutação) nos canais de cores e na dimensão de *batch*, de forma a ficar compatível com a dimensão de entrada dos modelos. Em seguida adicionamos as duas etapas de pré-processamento.

Foi criado um modelo de pré-processamento para cada modelo de inferência. A diferença entre eles se deu apenas pela dimensão de *resize*. Ambos os modelos foram exportados para ONNX, com entradas dinâmicas nas dimensões de *batch*, *height* e *width*.

5.1.2.2 União com os modelos de pré-processamento

Usando a API de merge_models do ONNX podemos unir os modelos de inferência com os modelos de pré-processamento. A união ocorre na camada de saída do modelo de pré-processamento com a camada de entrada modelo de inferência. O modelo ONNX da Mobile FaceNet ficou com um *model size* de 3.980mb e a da Ultra LightWeight com 1.204mb.

5.1.2.3 Concatenação de camadas de Non-Maximum Supression

Modelos detectores de objetos necessitam de operações caras no pós-processamento como a *Non-Maximum-Supression* (NMS) para limitar o número de *boxes* por região de uma imagem. Podemos usar um operador do ONNX para aplicar a NMS.

A saída comum dos detectores de objetos é *logits* e *boxes*. A Ultra LightWeight tem como saída (N x 4420 x 2) para *logits* e (N x 4420 x 4) para *boxes*, onde 4420 é a quantidade de *boxes* geradas. A terceira dimensão de *logits* equivale a probabilidade de ser um *background* ou uma face. Já a terceira dimensão de *boxes* se refere a probabilidade para cada ponto da *anchor*.

O operador de NMS do ONNX espera que seja o *logits* esteja no formato (*batch*, num_classes, *anchors*) e as *boxes* com (*batch*, *anchors*, 4). Um *transpose* é necessário em *logits*.

Além de *logits* e *boxes*, o operador NMS aguarda como entrada a quantidade máxima de *boxes* por classe, o *threshold* de IoU e de *score*. Os valores escolhidos para o experimento são, respectivamente: 200, 0.5 e 0.95. A saída do operador tem como

saída (num_selected_index, 3), sendo os indices selecionados no formato (*batch_index, class_index, box_index*). Usando a API do ONNX, foi concatenado no modelo a operação de *transpose* e de NMS.

O interesse desse detector é que ele apenas informe quais são as *boxes* detectadas das faces. Podemos construir com o PyTorch um modelo que receba as saídas do atual modelo ONNX e as modifique para se adequar a saída ideal.

Primeiro passo é usar a saída do NMS para selecionar apenas as *boxes* que foram selecionados pelo operador, então uma *mask* (máscara) é criada para selecionar apenas onde houve detecção da classe face, que no caso do Ultra LightWeight é a classe 1. A *mask* é aplicada nas boxes que foram selecionadas e na dimensão batch_indices, que informa o *batch* a qual ela pertence. Ao aplicar a *mask*, a informação a qual *batch* pertence a *boxe* é perdida. Por isso é útil retornar além das *boxes* que foram selecionadas também retornar *batch_indices*.

Esse novo modelo de pós-processamento do NMS é exportado de PyTorch para ONNX com *dynamic axes* nas dimensões de *batch* para as três entradas. ONNX-sim foi aplicado nesse modelo.

E então, foi feito a concatenação do modelo Ultra LightWeight com NMS com o modelo de pós-processamento de NMS usando novamente a API de *merge_models* do ONNX.

5.2 Construção de classes base

5.2.1 Inferência acelerada com ONNX Runtime e OpenVINO

Uma classe foi desenvolvida para realizar a inferência usando o ONNX Runtime. O OpenVINO é suportado como *execution provider* para obter a acelaração em CPUs Intel.

As imagens são recebidas como Numpy *arrays*. Então esse *array* é convertido no objeto Ort_Value do ONNX Runtime. Após a inferência ser realizada, os dados são novamente convertidos em Numpy *arrays*. Essa classe foi herdada em outras duas novas, uma para o detector de face e outra para o extrator de *features*. No detector de faces foi incluída uma função que irá receber as duas saídas do modelo, *boxes* e *batch indices*, e então separar as *boxes* por *batch*, a função também se encarrega de verificar se ocorreu uma detecção, se não ocorre a resposta do *batch* deve retornar um Numpy *array* vazio.

5.2.2 Cliente para o Qdrant

A classe de cliente para o Qdrant implementa métodos de inserção e busca em *batch*. Os dados comuns para ambas as funções são os *payloads* e as *embeddings*. Para inserir dados é necessário uma lista de *embeddings* e os *payloads* correspondentes. Para a recuperação é necessário apenas a lista de *embeddings* para obter os *payloads* do Qdrant.

Esse cliente pode operar usando HTTP ou gRPC, sendo o gRPC mais rápido. Parâmetros relacionados a busca no Qdrant são configuráveis, são eles: *top k* e *score threshold*. A métrica de comparação escolhida foi a similaridade de cosseno.

5.2.3 Pós-processamento do detector

O detector de face exige agora apenas duas etapas de pós-processamento: a conversão das *boxes* normalizadas para valores inteiros, que correspondem as proporções da imagem original, e uma função para realizar o *crop* da face na imagem original. Uma classe foi implementada com essas duas funções.

5.3 Configuração do ambiente

A aplicação funciona em dois ambientes. O primeiro deles irá manter o serviço de buscas do Qdrant. A máquina possui 1GB de memória RAM e 0.5 *cores* de CPU. O serviço do Qdrant implementa seu próprio monitorador de métricas, que são latência e o número de requisições por segundo (sigla RPS, em inglês). Esse serviço é implementado usando Helm e Kubernetes. Nesse experimento, foi utilizado o serviço gratuito de *cloud* da Qdrant, que o hospeda.

A segunda máquina pertence ao serviço do KubeRay, que irá gerenciar o RayJob e o RayService. As configurações da máquina de nó único foram: 16GB de memória RAM e 8 *cores* de CPU Intel(R) Xeon(R) CPU com 2.20GHz.

5.4 Conjunto de dados

Esse trabalho introduz um novo conjunto de dados para identificação facial chamado *Identification-in-Football* (IiF - Identificação no Futebol) [3]. Ele é um pequeno conjunto de imagens de jogadores de futebol. As imagens foram obtidas em sites da internet. As proporções das imagens são diversificadas, e todas elas são menores que 1000x1000.

O IiF é uma composição de três conjuntos de dados. Se dividindo em dois grupos, ingestão e recuperação. O conjunto de ingestão possui 32 imagens de diferentes jogadores de futebol profissional. A motivação por trás de gerar esse conjunto é de que os conjunto de dados da literatura possuem apenas uma face por imagem. E como muitos desses jogadores autuaram juntos, foi possível obter algumas imagens onde mais de um deles estão.

Dessa forma, os outros dois conjuntos de dados contêm imagens de jogadores que estão no conjunto de ingestão. Um deles contêm 96 imagens com apenas uma pessoa, e o outro possui 57 imagens com duas pessoas.

[3] Link para o conjunto de dados: <https://huggingface.co/datasets/vilsonrodrigues/identification-in-football/>

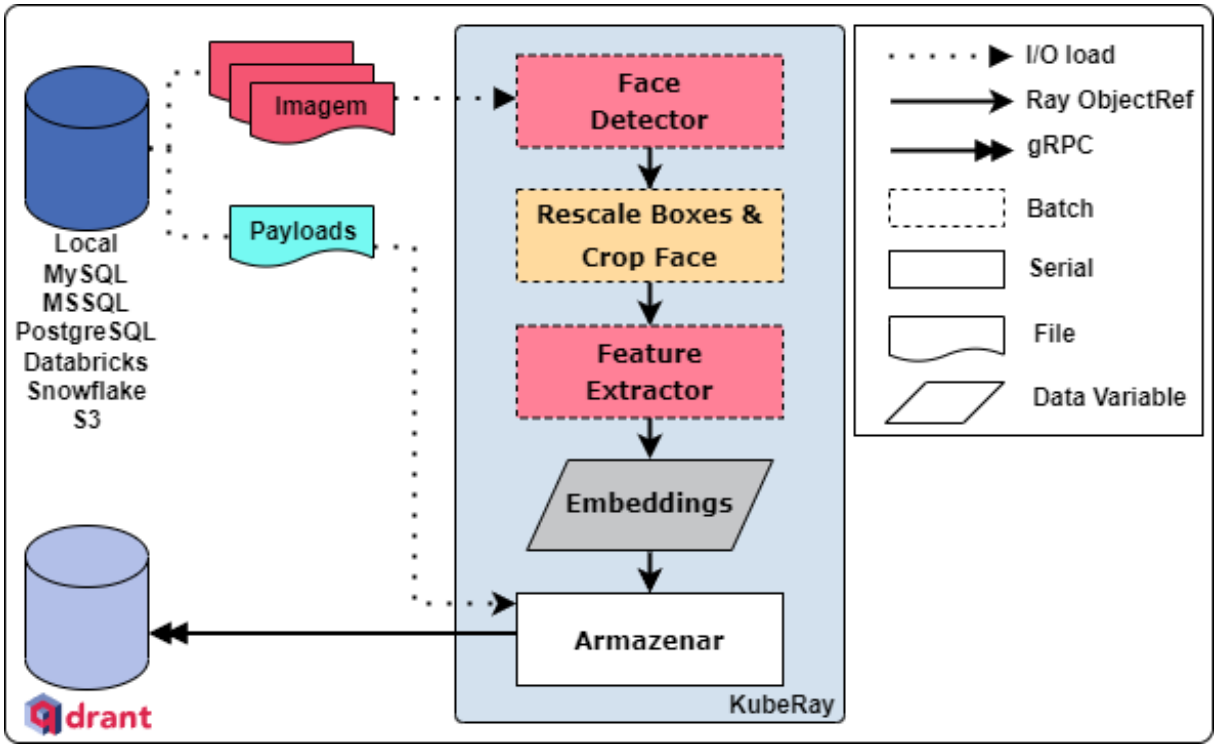


Figura 7 – Pipeline de ingestão dos dados

5.5 Ingestão dos dados

Os dados podem ter diversas fontes, como diretórios locais, bancos de dados ou *cloud*. O fluxo da ingestão dos dados pode ser visto na Figura 7.

5.5.1 Fluxo de processamento dos dados

Neste experimento as imagens do conjunto de dados IiF estão armazenadas localmente. Para o carragamento das imagens a API do Ray Data para ingestão de imagens foi utilizadas. Os metadados (*payloads*) que são os nomes referentes a cada face, serão armazenados no Qdrant como *payloads*.

Para processamento dos *batches* de imagens, a API *map_batches* do Ray Data foi utilizada para distribuir a execução em cada Ray *actor*.

Face Detector é uma classe que herda da classe base de detecção de face. Uma função do tipo *call* do Python, onde ao chamar a classe automaticamente a função é chamada. Dentro dessa função existe uma condicional onde será escolhido se o *batch* irá ou não ser redimensionado (*resized*). Essa opção deverá ser escolhida para casos onde a imagem de entrada pode vir em qualquer proporção. Apesar do modelo aceitar qualquer proporção, ele precisa que o *array* de entrada estejam todos em uma mesma proporção. Após a inferência do modelo, as *boxes* normalizadas são separadas por *batch* e então são retornadas junto com o *batch* de imagens de entrada.

Uma classe de pós-processamento do *Face Detector* foi implementada a partir da

classe base de pós-processamento de face. Em uma função *call* é recebido um *batch* de imagens e as *boxes*, a função de *rescale boxes* e a de *crop face* são aplicadas para cada imagem que a função irá receber. A função então retorna as faces.

A classe de ***Feature Extractor*** herda da classe base de extração de *features*. A função *call* recebe um *batch* de faces. As faces cropadas podem vir em qualquer proporção, então um *resize* é aplicado para padronizar as faces antes da inferência do extrator de *features*. Após a inferência, o modelo retorna as *embeddings* de cada face. As *embeddings* então são convertidas de Numpy *array* para uma lista de *floats*.

Por fim, os *embeddings* gerados irão ser inseridos no Qdrant junto aos *payloads* usando uma instância da classe de cliente para o Qdrant desenvolvida anteriormente.

5.5.2 *Deploy* com RayJob

Os códigos foram empactados em container Docker então submetidos ao Kubernetes via RayJob para a realização o processamento dos dados. A integração com o Prometheus e Grafana permite a visualização das métricas via Ray Dashboard.

5.6 Recuperação dos dados

Para a etapa de recuperação (*retrieval*) dos dados, uma API *WebServer* foi desenvolvida para a partir da imagem fornecida pelo cliente, ocorra a recuperação dos dados baseados na face detectada.

5.6.1 Construção dos *Deployments*

A etapa de recuperação foi divida em 5 subetapas, sendo cada uma um Ray Serve Deployment. Foram eles: ***Neural Search***, ***PostProcessing Face Detector***, ***Face Detector***, ***Feature Extractor*** e ***Retrieval Pipeline***. Cada *Deployment* tem sua configuração de escala.

Os *Deployments* que suportam *batch* funcionam com duas funções assíncronas, uma irá receber a carga de um objeto de um *Deployment* e então irá repassar para a função de *handle batch*. Os *Deployments* que não tem suporte a *batch* implementam funções assícronas individuais.

Os *Deployments* que realizam inferência possuem uma função de *warmup*. Essa função faz com o que o modelo seja carregado para a memória e o compile (se o *backend* do OpenVINO estiver ativo). Isso faz com que não tenha *delay* nas primeiras requisições que o servidor receber.

5.6.1.1 *Face Detector*

O *Deployment Face Detector* herda da classe de detecção de face e opera em *batch*. A função de carga de carga do modelo irá receber um Numpy *array*. A função de *handle batch* também tem a mesma condicional de redimensionamento, onde será escolhido se o *batch* irá ou não ser redimensionado. Em casos onde o *batch* irá ser igual a 1, se torna desncessário aplicar o *resize*. Após a inferência do modelo, as *boxes* normalizadas são separadas por *batch*. Se não houver detecções, um *array* vazio (*empty*) deverá ser retornado.

5.6.1.2 *Feature Extractor*

O *Deployment Feature Extractor* funciona de maneira similiar ao anterior. Ele herda da classe de extração de *features* e também opera em *batch*. Ela recebe como entrada um Numpy *array* que representa uma face.

A condição de *resize* também é implementada e possui as mesmas condições da anterior. Após a face ser selecionada a proporção não é garantida e o *resize* para obter uma mesma proporção se torna necessário. Também se torna opcional para quando a aplicação estiver com *batch* igual a 1.

Após a inferência, o modelo retorna as *embeddings* de cada face. As *embeddings* então são convertidas de Numpy *array* para uma lista de *floats*.

5.6.1.3 *Post Processing Face Detector*

O *Deployment Post Processing Face Detector* herda a classe de pós-processamento de faces. Ele opera de forma serial. Esse *Deployment* implementa duas funções assíncronas.

A primeira função é a de *rescale boxes*, ela espera como carga a imagem original e as *boxes* normalizadas. Após a aplicação da função de conversão de *boxes* para absolutas, elas são retornadas.

A segunda função de realiza o *crop* da face recebe as boxes absolutas e a imagem original. Essa função usa um gerador, onde cada face cropada na imagem será retornada imediatamente. Isso foi implementado pois quando referência desse objeto for repassada a outro *Deployment* o método *await* não irá ser necessário já que as faces irão ser repassadas uma por vez.

5.6.2 *Neural Search*

Esse *Deployment* herda da classe de cliente do Qdrant e opera em *batch*. A função de carga irá receber um *embedding* e irá retornar os *payloads* correspondentes.

5.6.3 *Retrieval Pipeline*

O Deployment de *Retrieval Pipeline* é responsável por gerenciar as chamadas para os demais *Deployments*. Ele é implementado como uma API em FastAPI. Ele irá receber requisições HTTP do usuário (*user*) contendo uma imagem que será carregada como um Numpy *array*.

O fluxo deste *Deployment* é implementado da seguinte maneira: a imagem é passada para o *Deployment Face Detector*. Um *await* é aplicado ao Ray ObjectRef da resposta. Ocorre uma verificação se houve detecção. Se não houver, retorne um dicionário com os campos vazios para *payloads* e *boxes*. Se houver uma detecção repasse a resposta para a função de *rescale boxes* do *Deployment Post Processing Face Detector* junto com a imagem em *array*. Então repasse o Ray ObjectRef para a função de *crop face* do *Deployment Post Processing Face Detector* junto a imagem em *array*, a resposta será um Ray ObjectRef *generator*.

Usando um *async for* cada Ray ObjectRef da face é repassado para o *Deployment* de *Feature Extractor*, e a sua resposta é repassada ao *Deployment* de *Neural Search*. O resultado disso é armazenado em uma lista. O método *gather* da biblioteca *asyncio* é usado para juntar as respostas dos *payloads*. Por fim, retorne as *boxes* e os *payloads*.

5.6.4 Configurações de escala dos *Deployments*

As configurações de escala de cada Deployment foram definidas em 2. Essa definição habilita ao contralador de réplicas do Ray Serve operar e gerenciar elas. Os três *Deployments* que operam em *batch* tiveram o tempo máximo de espera definido em 30ms.

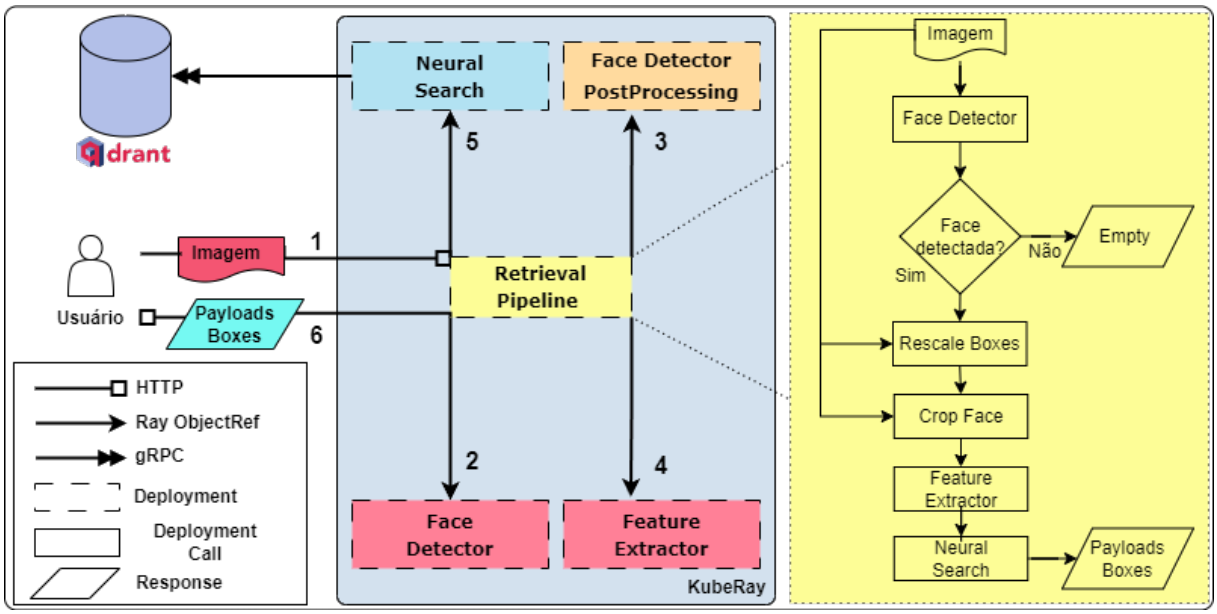
Tabela 2 – Configurações de escala dos Deployments

Deployment	**Max Replicas**	**Min Replicas**	**Initial Replicas**	**Max Batch Size**
Face Detector	4	1	1	16
Feature Extractor	4	1	1	16
Post Processing Face Detector	32	1	16	-
Neural Search	4	1	1	16
Retrieval Pipeline	1	1	1	-

5.6.5 *Deploy* com RayService

Os códigos e os modelos são empacotados em um *container* Docker. Então usando o KubeRay realizamos o *Deploy* da aplicação com a API do RayService. A integração automática com Prometheus e Grafana faz o Ray Dashboard receber as métricas da aplicação. A Figura 8 resume a aplicação.

Figura 8 – Fluxo de funcionamento da API de recuperação dos dados com etapas de funcionamento ordenadas



Fonte: Autoria própria

5.6.6 Testes de carga

O Locust foi utilizado para realizar testes de carga na API. Foram dois experimentos realizados. Cada experimentou utilizou um conjunto de dados de identificação do IiF. Os experimentos começam com quatro clientes conectados e crescem linearmente a cada segundo, até atingir o limite estabelecido. Os valores testados de limites foram 16, 32, 64 e 128 clientes conectados. O intervalo de teste carga foi de 40s. Ao fim, as métricas de latência e requisições por segundo são geradas pelo Locust. A latência mediana foi escolhida para análise já que as requisições iniciais ou de bastante carga podem gerar *ourliers*.

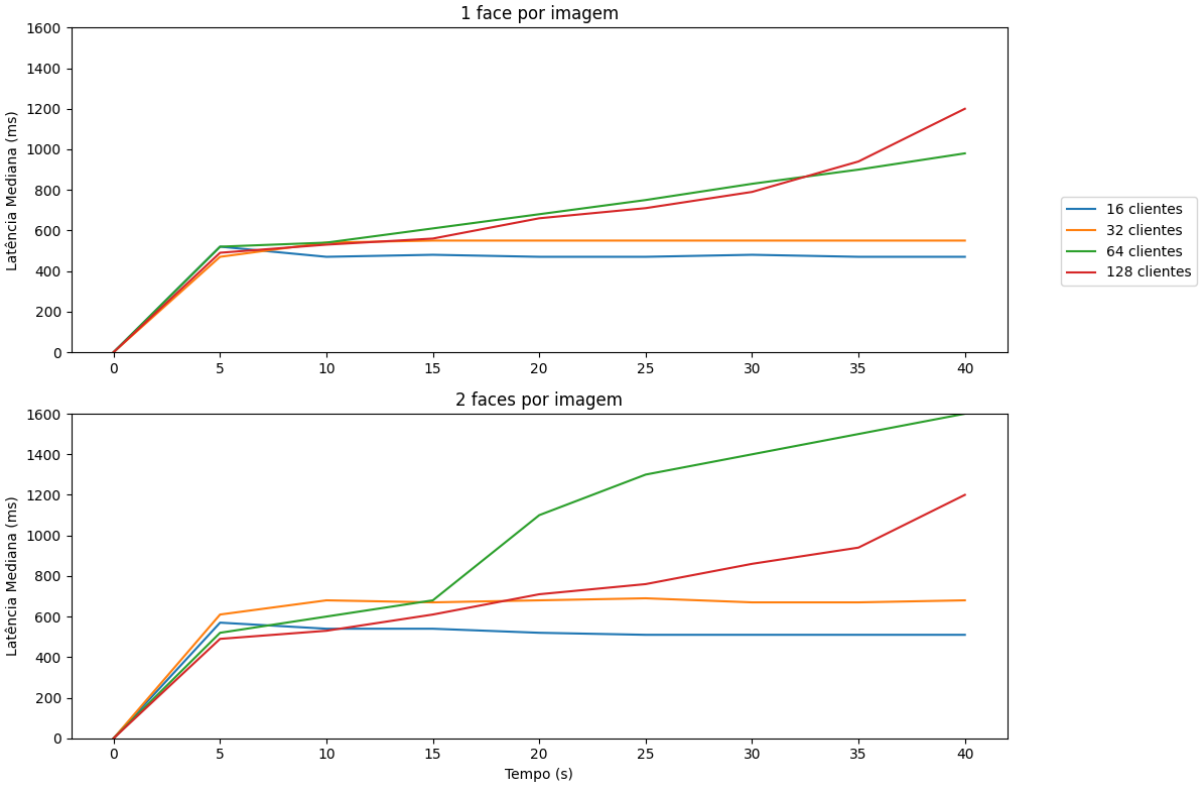
6 Resultados

Os dois experimentos de teste de carga geraram métricas importantes, como latência e quantidade de requisições por segundo.

Na Figura 9, no primeiro cenário, onde cada imagem continha apenas uma face, o sistema manteve uma latência estável de aproximadamente 500ms para até 32 clientes conectados. No entanto, com 64 clientes conectados, houve um aumento constante na latência mediana, chegando a cerca de 950ms ao final do teste. Considerando que o tamanho máximo do lote foi de 16 e até 4 réplicas dos serviços foram permitidas, era esperado um aumento maior na latência mediana com 128 clientes conectados.

No segundo cenário, com duas faces por imagem, a latência mediana do sistema permaneceu semelhante à primeira situação para até 16 clientes fazendo requisições. Quando o número de clientes aumentou para 32, o tempo de resposta mediano foi ligeiramente superior a 600ms. Com 64 clientes conectados, o sistema começou a ter dificuldades, chegando a ultrapassar os 1200ms.

Figura 9 – Comparativo do número de clientes *versus* a latência mediana do sistema medida em segundos



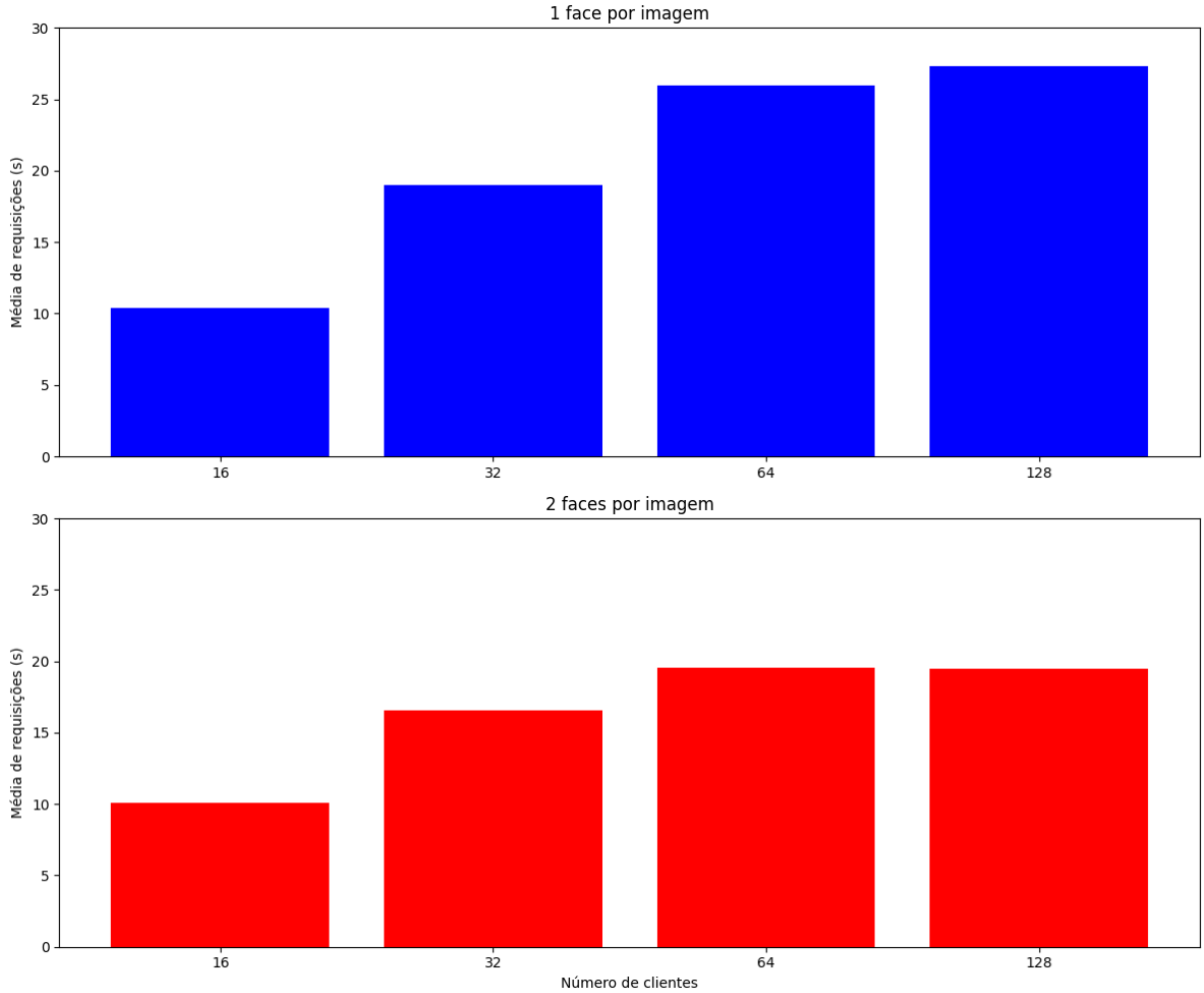
Fonte: Autoria própria

A Figura 10 mostra a quantidade média de requisições por segundo. Apesar de

terem uma latência mediana menor, 16 clientes conectados entregam pouco mais que 10 requisições por segundo.

O aumento no número de clientes conectados permite que o sistema atinja uma maior quantidade de requisições por segundo. No caso de duas faces por imagem, o sistema sofre uma pequena perda em comparação ao caso de uma única face. A partir de 64 clientes, o sistema estagna para até 4 réplicas dos serviços que processam em lote. Portanto, um aumento no número de réplicas é necessário para que o sistema continue escalando para atender a demandas maiores.

Figura 10 – Comparativo do número de clientes *versus* a média de requisições atendidas por segundo



Fonte: Autoria própria

Os *logs* de observabilidade do sistema gerados pelo Ray mostram que o serviço que apresenta o maior gargalo é o serviço de comparação de faces. Embora o Qdrant esteja realizando essa etapa em cerca de 3ms, segundo suas métricas, o tempo de transporte até a máquina do Qdrant ainda é um problema, mesmo utilizando o gRPC.

Embora mais clientes conectados gerem melhores números de requisições atendidas

por segundo, o tempo de espera pela resposta começa a aumentar. Esse *trade-off* deve ser considerado dependendo do foco da demanda do sistema.

Este estudo não se estende à verificação da precisão dos modelos no processo de identificação porque os modelos utilizados para reconhecimento facial já foram testados no padrão da literatura, o LFW. O conjunto de dados proposto aqui não poderia servir de base para validar o desempenho porque é curto e poderia gerar resultados enviesados. Portanto, é importante lembrar que a eficácia de um sistema de reconhecimento facial não depende apenas da sua capacidade de processar um grande número de requisições, mas também da precisão dos modelos utilizados.

7 Conclusão

Este estudo explorou o fortalecimento da prevenção de crimes através do uso de sistemas de reconhecimento facial com reconhecimento automático, com foco em experimentar métodos para aumentar a escalabilidade e atender a um maior número de serviços de vigilância. Os resultados indicam que um sistema capaz de escalar proporcionalmente ao aumento da demanda oferece uma solução eficaz para atender a um volume maior de requisições.

No contexto dos sistemas de reconhecimento facial, este trabalho contribuiu com a integração do processamento nos modelos para permitir o processamento em lote de etapas anteriormente realizadas de forma sequenciais. Além disso, demonstrou-se que os Bancos de Dados Vetoriais podem atenuar os desafios da etapa de correspondência e que o uso de computação distribuída em *clusters* Kubernetes pode construir sistemas robustos e resilientes, mesmo em hardwares mais modestos.

Para futuras pesquisas, propõe-se uma análise mais aprofundada para determinar a melhor relação entre o tamanho dos lotes dinâmicos nos *Deployments* e o número de clientes. A consideração do uso de aceleradores de *hardware* também poderia ser uma estratégia viável para atender a um número ainda maior de clientes.

Referências

AHONEN, T.; HADID, A.; PIETIKÄINEN, M. Face recognition with local binary patterns. In: SPRINGER. *Computer Vision-ECCV 2004: 8th European Conference on Computer Vision, Prague, Czech Republic, May 11-14, 2004. Proceedings, Part I 8*. [S.l.], 2004. p. 469–481. Citado 2 vezes nas páginas 16 e 25.

ANWAR, A.; NADEEM, S.; TANVIR, A. Edge-ai based face recognition system: Benchmarks and analysis. In: *2022 19th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. [S.l.: s.n.], 2022. p. 302–307. Citado na página 39.

ARSENOVIC, M. et al. Facetime—deep learning based face recognition attendance system. In: IEEE. *2017 IEEE 15th International symposium on intelligent systems and informatics (SISY)*. [S.l.], 2017. p. 000053–000058. Citado na página 39.

BELI, I. L. K.; GUO, C. Enhancing face identification using local binary patterns and k-nearest neighbors. *Journal of Imaging*, v. 3, n. 3, 2017. ISSN 2313-433X. Disponível em: <https://www.mdpi.com/2313-433X/3/3/37>. Citado na página 40.

BLEI, D. M.; NG, A. Y.; JORDAN, M. I. Latent dirichlet allocation. *Journal of machine Learning research*, v. 3, n. Jan, p. 993–1022, 2003. Citado na página 25.

CHAUDHARI, V. et al. Real time face recognition based attendance system using multi task cascaded convolutional neural network. In: *2023 International Conference on Emerging Smart Computing and Informatics (ESCI)*. [S.l.: s.n.], 2023. p. 1–6. Citado na página 39.

CHEN, S. et al. *MobileFaceNets: Efficient CNNs for Accurate Real-Time Face Verification on Mobile Devices*. 2018. Citado na página 26.

CHENG, Z.; ZHU, X.; GONG, S. Face re-identification challenge: Are face recognition models good enough? *Pattern Recognition*, Elsevier, v. 107, p. 107422, 2020. Citado na página 16.

CHOWDHARY, O. A. et al. Automated attendance management system using face recognition. 2017. Disponível em: <http://ir.aiktclibrary.org:8080/xmlui/bitstream/handle/123456789/2054/aiktcDspace2054?sequence=1&isAllowed=y>. Citado na página 40.

CHUNHONG, J.; GUANGDA, S.; XIAODONG, L. A distributed parallel system for face recognition. In: *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*. [S.l.: s.n.], 2003. p. 797–800. Citado na página 40.

CNCF. *Helm Project Journey Report*. 2023. Helm. Disponível em: <https://www.cncf.io/reports/helm-project-journey-report/>. Acesso em: 22 de setembro de 2023. Citado na página 33.

DAQUEXIAN. *ONNX Simplifier*. 2023. Wiki do abnTeX2. Disponível em: <https://github.com/daquexian/onnx-simplifier>. Acesso em: 25 de novembro de 2023. Citado na página 35.

DENG, J. et al. Arcface: Additive angular margin loss for deep face recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Institute of Electrical and Electronics Engineers (IEEE), v. 44, n. 10, p. 5962–5979, out. 2022. ISSN 1939-3539. Disponível em: <http://dx.doi.org/10.1109/TPAMI.2021.3087709>. Citado na página 26.

DOCKER. *What is Container?* 2023. Docker. Disponível em: <https://www.docker.com/resources/what-container>. Acesso em: 21 de setembro de 2023. Citado na página 30.

DUDA, R. O.; HART, P. E. et al. *Pattern classification and scene analysis*. [S.l.]: Wiley New York, 1973. v. 3. Citado na página 27.

EIAMSAARD, K.; BAMRUNGTHAI, P.; JITPAKDEEBODIN, S. Smart inventory access monitoring system (siams) using embedded system with face recognition. In: *2021 18th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. [S.l.: s.n.], 2021. p. 1–4. Citado na página 39.

GÉRON, A. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. [S.l.]: "O'Reilly Media, Inc.", 2022. Citado na página 28.

GRAFANA. *Grafana*. 2023. Grafana. Disponível em: <https://github.com/grafana/grafana>. Acesso em: 21 de setembro de 2023. Citado na página 31.

HUANG, G. B. et al. *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments*. [S.l.], 2007. Citado na página 26.

HUYEN, C. *Designing machine learning systems*. [S.l.]: "O'Reilly Media, Inc.", 2022. Citado na página 28.

IANDOLA, F. N. et al. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. Citado na página 39.

IRBAZ, M. S.; NASIM, M. A. A.; FERDOUS, R. E. Real-time face recognition system for remote employee tracking. In: SPRINGER. *Proceedings of the International Conference on Big Data, IoT, and Machine Learning: BIM 2021*. [S.l.], 2022. p. 153–163. Citado na página 40.

JINA. *What is Neural Search?* 2023. Jina. Disponível em: <https://jina.ai/news/what-is-neural-search-and-learn-to-build-a-neural-search-engine/>. Acesso em: 28 de novembro de 2023. Citado na página 34.

JOSE, E. et al. Face recognition based surveillance system using facenet and mtcnn on jetson tx2. In: *2019 5th International Conference on Advanced Computing Communication Systems (ICACCS)*. [S.l.: s.n.], 2019. p. 608–613. Citado na página 39.

KUBERNETES. *Kubernetes documentation*. 2023. Kubernetes. Disponível em: <https://kubernetes.io/docs/home/>. Acesso em: 21 de setembro de 2023. Citado na página 31.

LI, Q. et al. Video face recognition system: Retinaface-mnet-faster and secondary search. In: SPRINGER. *Advances in Information and Communication: Proceedings of the 2021 Future of Information and Communication Conference (FICC), Volume 2*. [S.l.], 2021. p. 625–636. Citado na página 40.

LI, Y.; CHA, S. Implementation of robust face recognition system using live video feed based on cnn. *arXiv preprint arXiv:1811.07339*, 2018. Citado na página 40.

LINZAER. *Ultra-lightweight face detection model*. 2023. <https://github.com/Linzaer/Ultra-Light-Fast-Generic-Face-Detector-1MB>. Citado na página 24.

LIU, W. et al. Ssd: Single shot multibox detector. In: SPRINGER. *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. [S.l.], 2016. p. 21–37. Citado na página 24.

MALKOV, Y. A.; YASHUNIN, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, IEEE, v. 42, n. 4, p. 824–836, 2018. Citado 3 vezes nas páginas 17, 20 e 34.

MENG et al. A high performance face recognition system based on a huge face database. In: *2005 International Conference on Machine Learning and Cybernetics*. [S.l.: s.n.], 2005. v. 8, p. 5159–5164 Vol. 8. Citado 2 vezes nas páginas 17 e 40.

NUMPY. *NumPy documentation*. 2023. Numpy. Disponível em: <https://numpy.org/doc/stable/>. Acesso em: 21 de setembro de 2023. Citado na página 34.

OJALA, T.; PIETIKAINEN, M.; MAENPAA, T. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *IEEE Transactions on pattern analysis and machine intelligence*, IEEE, v. 24, n. 7, p. 971–987, 2002. Citado na página 25.

ONNX. *About*. 2023. ONNX. Disponível em: <https://onnx.ai/about.html>. Acesso em: 21 de setembro de 2023. Citado na página 35.

OPENVINO. *OpenVINO Docs*. 2023. OpenVINO. Disponível em: <https://docs.openvino.ai/2023.1/home.html>. Acesso em: 21 de setembro de 2023. Citado na página 35.

PAPERSWITHCODE. *Paper Implementations grouped by framework*. 2023. Wiki do abnTeX2. Disponível em: <https://paperswithcode.com/trends>. Acesso em: 21 de setembro de 2023. Citado na página 35.

PARKHI, O. M.; VEDALDI, A.; ZISSERMAN, A. Deep face recognition. *British Machine Vision Conference (BMVC)*, 2015. Disponível em: <http://www.robots.ox.ac.uk/~vgg/data/vgg_face/>. Citado na página 39.

PEARSON, K. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, Taylor & Francis, v. 2, n. 11, p. 559–572, 1901. Citado na página 25.

PROMETHEUS. *Overview*. 2023. Prometheus. Disponível em: <https://prometheus.io/docs/introduction/overview/>. Acesso em: 21 de setembro de 2023. Citado na página 31.

PYTORCH. *Pytorch*. 2023. Wiki do abnTeX2. Disponível em: <https://pytorch.org/>. Acesso em: 21 de setembro de 2023. Citado na página 35.

QDRANT. *Benchmarks*. 2023. Qdrant. Disponível em: <https://qdrant.tech/benchmarks/>. Acesso em: 19 de novembro de 2023. Citado 2 vezes nas páginas 21 e 34.

QDRANT. *Documentation*. 2023. Qdrant. Disponível em: <https://qdrant.tech/documentation/>. Acesso em: 21 de setembro de 2023. Citado 2 vezes nas páginas 33 e 34.

RAMíREZ, S. *FastAPI*. 2023. Wiki do abnTeX2. Disponível em: <https://fastapi.tiangolo.com/>. Acesso em: 21 de setembro de 2023. Citado na página 33.

RANJAN, A.; BEHERA, V. N. J.; REZA, M. *A Parallel Approach for Real-Time Face Recognition from a Large Database*. 2020. Citado na página 40.

RAY. *What is Ray Core?* 2023. Ray. Disponível em: <https://docs.ray.io/en/latest/ray-core/walkthrough.html>. Acesso em: 13 de novembro de 2023. Citado na página 36.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. *Nature*, Nature Publishing Group, v. 323, n. 6088, p. 533–536, 1986. Citado na página 27.

RUNTIME, O. *ONNX Runtime Docs*. 2023. ONNX Runtime. Disponível em: <https://onnxruntime.ai/docs/>. Acesso em: 21 de setembro de 2023. Citado na página 35.

SANDLER, M. et al. Mobilenetv2: Inverted residuals and linear bottlenecks. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2018. p. 4510–4520. Citado na página 26.

SAYPADITH, S.; ARAMVITH, S. Real-time multiple face recognition using deep learning on embedded gpu system. In: IEEE. *2018 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. [S.l.], 2018. p. 1318–1324. Citado na página 39.

SCHROFF, F.; KALENICHENKO, D.; PHILBIN, J. Facenet: A unified embedding for face recognition and clustering. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2015. p. 815–823. Citado na página 26.

SG, S.; MC, M.; CC, L. *The Re-identification Challenge, in: Person re-identification*. [S.l.]: London: Springer, 2014. 1–20 p. Citado na página 16.

SINHA, T. *OLAP vs. OLTP: What's the Difference?* 2023. Olap-oltp-diff. Disponível em: <https://www.ibm.com/blog/olap-vs-oltp/>. Acesso em: 21 de setembro de 2023. Citado na página 34.

SRIVASTAVA, A. et al. A survey of face detection algorithms. In: IEEE. *2017 International Conference on Inventive Systems and Control (ICISC)*. [S.l.], 2017. p. 1–4. Citado na página 16.

SZEGEDY, C. et al. Going deeper with convolutions. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2015. p. 1–9. Citado na página 26.

TEOH, S. K. et al. Face detection and face re-identification system using deep learning and openvino. In: *2021 2nd International Conference on Artificial Intelligence and Data Sciences (AiDAS)*. [S.l.: s.n.], 2021. p. 1–5. Citado na página 40.

VAPNIK, V. N.; CHERVONENKIS, A. Y. On the uniform convergence of relative frequencies of events to their probabilities. In: *Measures of complexity: festschrift for alexey chervonenkis*. [S.l.]: Springer, 2015. p. 11–30. Citado na página 27.

VIOLA, P.; JONES, M. Rapid object detection using a boosted cascade of simple features. In: IEEE. *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*. [S.l.], 2001. v. 1, p. I–I. Citado na página 21.

WANG, M.; DENG, W. Deep face recognition: A survey. *Neurocomputing*, v. 429, p. 215–244, 2021. ISSN 0925-2312. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0925231220316945>. Citado na página 16.

WANG, X. et al. A survey of face recognition. *arXiv preprint arXiv:2212.13038*, 2022. Citado na página 20.

ZHANG, K. et al. Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE signal processing letters*, IEEE, v. 23, n. 10, p. 1499–1503, 2016. Citado 2 vezes nas páginas 22 e 23.

ZHANG, S. et al. Faceboxes: A cpu real-time face detector with high accuracy. In: *IJCB*. [S.l.: s.n.], 2017. Citado na página 24.