Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Multiprocessamento de uma pipeline de inferência para classificação de imagens

Lara Ayumi Nagamatsu

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. João Eduardo Ferreira

São Paulo
2022

Resumo

Lara Ayumi Nagamatsu. **Multiprocessamento de uma pipeline de inferência para classificação de imagens**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Na atualidade, sistemas automatizados tornaram-se uma alternativa que pode substituir a necessidade de monitoramento de vídeo realizado por humanos. Contudo, na condição de monitoramento extenso e ininterrupto, o armazenamento de dados torna-se custoso e, portanto, insustentável. Este trabalho propõe a implementação de um programa de processamento de vídeo de forma paralelizada a fim de se potencializar a velocidade de inferência de classificadores de classes em imagens. Foi realizada a implementação de quatro *pipelines* diferentes cujas velocidades de processamento foram medidas a partir da utilização de vídeos de duração de dez segundos, três minutos e uma hora. O desempenho na medição da velocidade de processamento das *pipelines* apresentou visível diferença, sendo que uma das implementações atingiu um ganho considerável de velocidade em comparação com a implementação básica de apenas uma instância de pré-processamento, uma instância de processo de inferência e uma instância de pós-processamento.

Palavras-chave: multiprocessamento. paralelização. YOLO. inferência.

Abstract

Lara Ayumi Nagamatsu. **Multiprocessing of an inference pipeline for image classification**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Currently, automated systems have become an alternative that can replace the need for video analysis performed by humans. However, in the condition of extensive and uninterrupted monitoring, data storage becomes costly and therefore unsustainable. This thesis proposes the implementation of a system for parallelization of processing of videos in order to speed up the inference process of image classification. Four different pipelines were developed, the speed of inference for each pipeline was measured on videos of duration of ten seconds, three minutes and one hour. The performance in measuring the processing speed of the pipelines showed a visible difference, and one of the implementations achieved a considerable gain in speed compared to the basic implementation of just one pre-processing instance, one inference process instance and one post-processing instance.

Keywords: multiprocessing. parallelization. YOLO. inference.

Lista de abreviaturas

BoF	Bag of Freebies
BoS	Bag of Specials
DTL	Django Template Language
GIL	Global Interpreter Lock
IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo
MOM	Message Oriented Middleware
MVC	Model View Controller
PLN	Processamento de Linguagem Natural
TCC	Trabalho de Conclusão de Curso
USP	Universidade de São Paulo
YOLO	You Only Look Once

Lista de figuras

Sumário

Introdução

Contextualização

De acordo com o site oficial da *Universidade de São Paulo* (2022) (USP), a instituição possui em sua totalidade 42 unidades de pesquisa e ensino, das quais 25 localizam-se na cidade de São Paulo. Segundo a prefeitura da Cidade Universitária Armando Salles Oliveira (2016) (CUASO), o campus da capital possui cerca de 3,6 milhões de metros quadrados em extensão e sedia não só a infraestrutura para ensino superior como também áreas de extensão e espaços de lazer. Dessa forma, o caráter público da faculdade em corroboração à extensa área disponibilizada atraem muitos visitantes que utilizam os espaços da USP para realização de *hobbies* e treinamentos esportivos.

Entretanto, devido à variedade do público frequentador da universidade, nem sempre se consegue manter a harmonia na convivência dentro do campus. Segundo a matéria do Jornal da USP de Erika Yamamoto (2019), o público de ciclistas esportivos frequenta há anos os espaços da CUASO para treinamento competitivo, contudo, esse grupo é historicamente conhecido por já ter causado muitos conflitos devido a desentendimentos com os alunos da universidade e devido a desrespeitarem as leis de trânsito. Consequentemente, em 2019, a prefeitura do campus relatava o recebimento diário de denúncias apresentadas contra os ciclistas esportivos. Assim, após mudanças aplicadas sobre o regimento interno da universidade, foi decidido que o trânsito de ciclistas esportivos seria permitido somente em horários reduzidos e em áreas específicas da universidade.

Além disso, a USP desfruta de uma extensa infraestrutura de segurança - segundo Yamamoto (2018), o Centro de Monitoramento Eletrônico da USP tem sob seu domínio cerca de 300 câmeras de alta resolução que cobrem as áreas dos campi da capital e do interior de São Paulo. Disponibilizados pela Guarda Universitária, esses sistemas de monitoramento são compostos por câmeras de segurança situadas em áreas estratégicas do CUASO. O monitoramento em vídeo é feito diariamente, sob todos os horários do dia - dessa forma, a USP armazena uma grande quantidade de vídeos de várias horas de duração.

Os vídeos armazenados pelo Centro de Monitoramento são analisados por um grupo de agentes empregados da USP, contudo, a quantidade de vídeos é abundante e possui duração extensa. Surgiu-se, assim, a demanda por um sistema de monitoramento autônomo das câmeras de segurança. Com isso em mente, Nardi *et al.* (2022) desenvolveram o *OpenImages Cyclists* - um *dataset* de extensão ao *OpenImages*. O *OpenImages* é um *dataset* com mais de 9 milhões de imagens anotadas com caixas delimitadoras. Ele possui cerca de 18 mil imagens de pessoas em bicicletas, porém, na totalidade de classes anotadas do *dataset*, não

existe anotação para a classe 'ciclista'.

Outros *datasets* já preenchem a lacuna da classe 'ciclista' sob seu domínio de dados. Contudo, tais conjuntos possuem poucos dados de imagens anotadas, dos quais o conteúdo mostra-se, em geral, muito limitado - como, por exemplo, imagens sob resolução baixa e com pequena variedade de ângulo de detecção dos dados.

Para o desenvolvimento do *OpenImages Cyclists*, Nardi *et al.* (2022) utilizaram o *framework You Only Look Once (YOLO)* em sua quarta versão (*YOLOv4*) para realizar o processamento dos quadros dos vídeos. O projeto de extensão do *dataset* de ciclistas alcançou ótimos resultados, visto que o grupo conseguiu melhorar consideravelmente o conjunto de dados relacionado às imagens de ciclistas em ambientes diferentes, sob iluminação de diferentes horas do dia e sob ângulos variados de identificação.

Entretanto, a velocidade do processamento dos quadros de vídeo tornou-se um problema para o grupo a partir do momento em que vários vídeos deveriam ser processados ao mesmo tempo, contudo demoravam consideravelmente para que pudesse ser realizada a classificação nessas imagens. A identificação das classes pôde ser feita ainda em tempo real, porém, para um lote de testes grande com vídeos de longa duração, uma maior velocidade de processamento seria ideal. Levantou-se, assim, a necessidade de se construir um ambiente paralelo e distribuído para o processamento das centenas de horas de vídeos coletados diariamente. O desenvolvimento deste Trabalho de Conclusão de Curso se dá a partir desse ponto.

O método escolhido para se agilizar o consumo de vídeos com o *YOLO* foi feito a partir da utilização de paralelização de processos em uma *pipeline*. Uma *pipeline*, na computação, possui o significado de arquitetura de sistema na qual ocorre a alimentação de um processo com a saída de outro processo. Isso envolve, muitas vezes, a paralelização desses processos. Segundo Ramamoorthy e Li (1977), um sistema de *pipeline* envolve a transformação de um processo em vários subprocessos menores a fim de se paralelizar tarefas repetitivas e autônomas. As diferentes abordagens apresentadas neste trabalho demonstram a paralelização de processos de inferência, de leitura de quadros de vídeo e de escrita de quadros de vídeo como arquivos processados.

A linguagem utilizada para se desenvolver o projeto foi *Python* devido a sua popularidade no uso de arcabouços de aprendizado de máquina, devido à interface do *YOLOv4* ser disponibilizada na mesma linguagem e devido à implementação do projeto de Nardi *et al.* (2022) ter sido desenvolvida, também, com *Python*. Contudo, devido à *Global Interpreter Lock (GIL)* da linguagem, a paralelização de processos pode sofrer redução da velocidade a depender do regime de processamento.

O GIL é um *mutex* encontrado dentro do código da linguagem *Python* escrita em *C*, o *CPython*. Segundo Bernal (2017), o *mutex* é uma implementação de acesso a seções de código compartilhada de forma a permitir a exclusividade de um acesso por vez. Na linguagem *Python*, o GIL é utilizado para impedir que ocorram problemas em casos específicos de acesso de variáveis. Devido ao contador de referências para gerenciamento de memória em *multithread*, por exemplo, sem o GIL, casos de acesso de variáveis já não utilizadas na memória por uma *thread* seriam possíveis. Dessa forma, a paralelização de programas em *Python* torna-se um problema no caso de implementação em *multithread* - exceto em

casos de programas que são muito dependentes de dispositivos de entrada e saída (*I/O heavy*) e de programas que têm forte uso de computação com matrizes. Assim, a escolha da implementação multiprocesso com estratégia de filas pode se tornar uma alternativa mais interessante.

A utilização de apenas uma *thread* na linguagem *Python* pode não ser suficiente para saturar uma GPU, o que a torna ociosa dependendo do consumo de quadros de vídeo no pré-processamento e pós-processamento das imagens. Assim, faz-se necessário um esquema de filas que maximize o abastecimento da inferência a ser realizada.

A partir dessas informações, foi realizada a implementação de um programa que paraleliza o processo de detecção do *YOLO*. O sistema proposto neste trabalho possui uma *pipeline* de alimentação de quadros de vídeo para várias filas, as quais repassam os quadros para processos que realizam a inferência do *YOLO* a fim de se realizar a detecção das classes em tempo real. Após esse processo, os quadros processados são, finalmente, combinados para se realizar o armazenamento de um novo vídeo.

O arcabouço de *software* proposto neste trabalho pode ser estendido para contemplar outras ferramentas de *deep learning*, as quais requerem processamento massivo de dados gerados continuamente.

Objetivos

O objetivo deste trabalho é disponibilizar uma ferramenta que potencialize o abastecimento de quadros de vídeo para a realização do processo de inferência na detecção de classes delimitadoras em imagens. O processo de inferência foi realizado a partir da utilização do arcabouço *YOLO*, que realiza teste e treinamento de detecção de classes por caixas delimitadoras em vídeos e imagens.

Capítulo 1

Método e Técnicas

O trabalho realizado para o desenvolvimento deste TCC envolve o conhecimento de três frentes específicas: multiprocessamento, aprendizado de máquina e o arcabouço *Django*, acompanhado do uso do *middleware Celery*. Abaixo encontram-se os conhecimentos gerais para contextualização das técnicas utilizadas.

1.1 Django

Django é um arcabouço para desenvolvimento em *Python* que visa a rápida implementação e disponibilização em produção de aplicações web. Segundo Foundation (2022), *Django* é uma opção escalável, segura e versátil, que se baseia em três camadas: o *model*, a *view* e o *template*. Essas três partes modelam-se a partir da arquitetura do padrão de projeto *Model View Controller* (*MVC*), apresentado em 1.1.

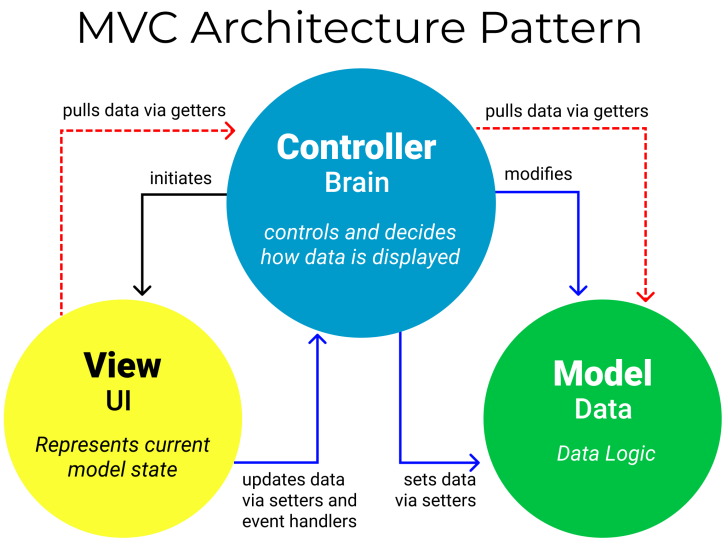

Figura 1.1: *Representação do MVC, de Hernandez (2021)*

Similarmente ao *MVC*, a camada *model* armazena a representação dos dados contidos dentro do programa, assim, dispõe de funções que são mapeadas para tabelas de um banco de dados relacional. Contudo, as outras duas camadas são diferentes. O *template* do *Django* é equivalente à *view* do *MVC*: no *template* ocorre a formatação da página para o usuário a partir da estruturação da *Django Template Language* (*DTL*), que carrega o *HTML* de forma dinâmica para a página. Enquanto isso, a *view* do arcabouço é responsável por receber as requisições do usuário, processá-las e devolver uma resposta, porém não é equivalente ao *controller* do *MVC*, visto que o próprio *Django* é responsável pelo mapeamento de interações do *template* às funções *views* correspondentes.

Devido à estrutura mencionada acima, *Django* é uma opção completa pois assegura a implementação tanto da parte do *front-end* quanto da parte do *back-end* para a arquitetura do programa. Dessa forma, o arcabouço não requer a utilização de outros arcabouços para que haja a apresentação e integração completa dos dados de um programa.

1.2 Celery

Celery (2022) é um *middleware* orientado a mensagens, ou seja, *Message Oriented Middleware* (*MOM*). Um *middleware* constitui uma ferramenta intermediária para outros programas cujo objetivo principal é realizar a comunicação das aplicações entre si pela transferência de dados. O *Celery* realiza a transferência de tarefas de forma distribuída e pode processar uma grande quantidade de mensagens em tempo real ou de forma planejada. Seu funcionamento ocorre a partir de um gerenciador de mensagens que pode ser escolhido pelo desenvolvedor - o banco de dados chave-valor *Redis* e o *RabbitMQ* são duas alternativas popularmente aplicadas para uso em combinação com o *Celery*.

O *Celery* é um dos programas mais populares para a realização de requisições assíncronas e distribuídas em tempo real que possui implementação em *Python*, porém o protocolo pode ser aplicado a uma implementação desenvolvida em qualquer outra linguagem. Para sua implementação, *Celery* utiliza a biblioteca *Billiard* (2022) que constitui fundamentalmente uma versão antiga da biblioteca *multiprocessing* do *Python*.

A implementação de programas que fazem uso do *Celery* baseia-se no desenvolvimento de tarefas, *tasks* do *Celery*, enviadas para trabalhadores, *workers*, realizarem. A partir de um *broker* de mensagens escolhido, o *Celery* realiza a transferências de mensagens para uma fila. Dessa fila, o programa administra a repassagem das mensagens para os trabalhadores disponíveis que se encontram em espera de novas tarefas. Esses trabalhadores realizam o processamento necessário da tarefa até que ela seja finalizada. Um sistema que se utiliza do *Celery* pode possuir múltiplos trabalhadores e múltiplas filas de tarefas.

Em geral, o *Celery* é utilizado acompanhado do arcabouço *Django* para apresentação de dados assíncronos em tempo real.

1.3 Aprendizado de Máquina

Segundo a Universidade de Columbia (2023), a Inteligência Artificial (IA) é uma área de estudo em que sistemas são desenvolvidos a fim de se emular o comportamento humano,

mais especificamente as capacidades de aprendizado e resolução de problemas. O aprendizado de máquina é uma subárea da IA na qual um programa replica esse comportamento de aprendizado: dados são alimentados para o treinamento de um modelo a fim de se aumentar sua acurácia. Existem diferentes modelos matemáticos que podem ser utilizados para realizar a inferência desses sistemas, dentre eles modelos de aprendizagem supervisionados, modelos não-supervisionados, modelos de aprendizagem por reforço e aprendizagem constrastiva. Em geral, o aprendizado de máquina possui a fase de treinamento, de elaboração do modelo de aprendizado, de definição de hiperparâmetros e de pré e pós-processamento de dados. Essas fases possuem, também, subetapas. Mais especificamente no treinamento, existem três, as quais se definem por treinamento, teste e validação. Nessas subetapas do treinamento, são utilizados conjuntos de dados diferentes de um mesmo *dataset* a fim de se acompanhar a evolução do treinamento e, eventualmente, interrompê-lo para se evitar casos de *overfitting*. Existem também outras formas de se contornar casos de aprendizado enviesado, como *batch normalization* e utilização de funções de ativação e de perda apropriadas, como a ReLU e a de entropria cruzada, respectivamente.

Essa área de estudo pode ser aplicada para diferentes esferas, como no reconhecimento de voz, no Processamento de Linguagem Natural (PLN), em *chatbots* e na área de visão computacional.

1.4 *You Only Look Once (YOLO)*

Aprendizado profundo trata-se de um subtipo de aprendizado de máquina específico para redes neurais que é amplamente utilizado para reconhecimento de imagens. A partir disso, uma rede neural é um modelo de aprendizado de máquina com várias camadas e chama-se dessa forma pela similaridade com o neurônio dos mamíferos, que possui um comportamento de ativação de outros neurônios em cascata, como exposto por Norman (2014). A primeira rede neural desenvolvida é o *Perceptron* de Rosenblatt, como exposto em 1.2. Assim como neurônios que repassam informação de uma ramificação para outra, a rede neural o faz com camadas das quais se utiliza de pesos e perdas para determinar valores dos neurônios.

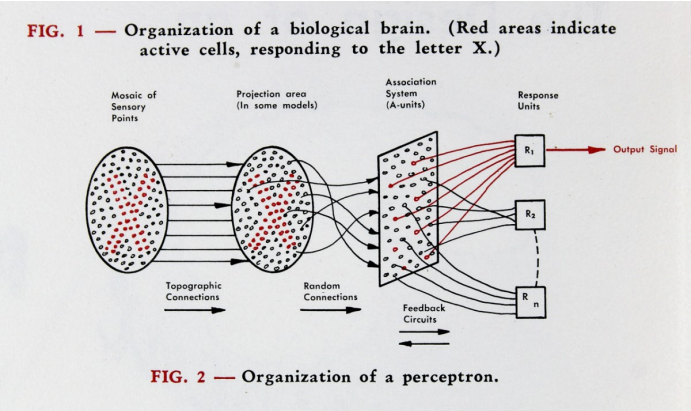

Figura 1.2: *Perceptron de Rosenblatt (1958)*

Em geral, a utilização de redes neurais tem se tornado cada vez mais frequente, contudo,

nem sempre isso foi verdade. A arquitetura das redes neurais clássicas possui um desafio - o gradiente descendente não-equilibrado. Normalmente, esse problema origina-se devido ao aparecimento de valores muito pequenos nas primeiras camadas das redes neurais, que tornam o valor do gradiente descendente muito baixa - esse é a questão do gradiente descendente que desaparece, segundo IBM (s.d.). Isso faz com que o aprendizado das redes neurais ocorra de forma muito lenta ou que seja quase inexistente, situação que pode ser observada com redes neurais de camadas muito profundas. Outro problema relacionado é o gradiente descendente que explode, cujos valores se tornam muito altos a ponto de não serem representáveis computacionalmente. Para contornar esses problemas, He *et al.* (2015) desenvolveram o artigo que apresenta as redes neurais residuais (*ResNets*), as quais se utilizam de conexões de salto para armazenar os valores resultantes das camadas e para, também, evitar a diminuição do valor total da alteração na função de custo resultante.

A partir desses conceitos, o *YOLO*, trabalho de Redmon *et al.* (2015), utiliza-se dessa tecnologia dentro de sua *pipeline*. *YOLO* pode ser considerado uma *pipeline* ao invés de apenas uma aplicação de rede neural, visto que se utiliza de várias tecnologias para melhorar seu desempenho na identificação de caixas delimitadoras de classes em imagens. Possui, dessa forma, uma estrutura com cabeça (*head*), coluna (*backbone*), pescoço (*neck*) e camadas de predição densa e esparsa. Em cada uma dessas camadas, utiliza-se do conceito de *Bag of Freebies* (*BoF*) e *Bag of Specials* (*BoS*), usados respectivamente como adições para melhoria no treinamento e na inferência. No artigo de Bochkovskiy *et al.* (2020), *BoF* compõem métodos que melhoram o desempenho da inferência sem aumentar o tempo de processamento de inferência, ou seja, apenas às custas da mudança ou aumento do tempo de treinamento. Já as *BoS* compõem técnicas que aumentam a precisão do processo de inferência às custas de aumento de tempo de processamento.

O arcabouço de identificação de caixas delimitadoras *YOLO* é um dos mais popularmente utilizados e encontra-se atualmente na sua sétima versão. Ele é uma alternativa rápida e boa para identificação de objetos em tempo real dado que se utiliza de apenas uma iteração da rede neural para identificação de objetos por quadro de vídeo graças ao fato de que sua inferência e seu treinamento levam em conta o contexto das imagens sendo processadas. A figura 1.3 apresenta a utilização do *YOLO* para identificação de ciclistas.

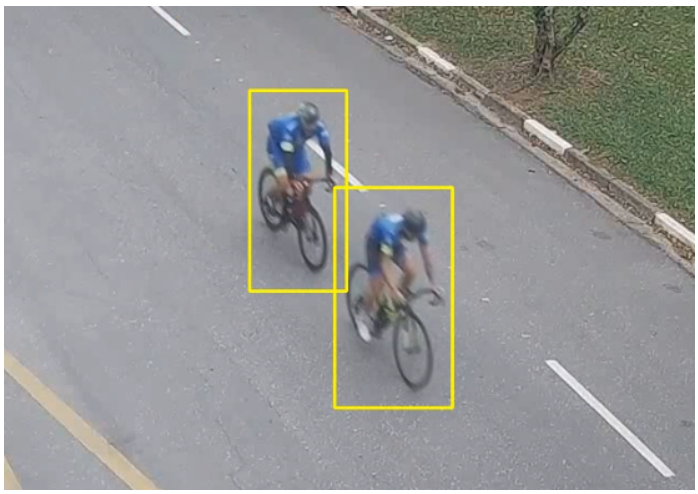

Figura 1.3: *Ciclistas identificados pelo YOLO*

O *YOLO* possui duas partes: a funcionalidade de treinamento dos dados, com a alternativa de *fine-tuning*, ou seja, utilização de parâmetros do *dataset* pré-treinado *COCO*, desenvolvido por Lin *et al.* (2014); ou, então, realizar o treinamento de classes customizadas, como é o caso do trabalho de Nardi *et al.* (2022), o *OpenImages Cyclists*. Após a etapa de treinamento, resta a execução da etapa de inferência para o sistema, que se utiliza dos pesos e configuração do treinamento para alcançar os resultados sob as classes a serem identificadas. A inferência pode ser realizada tanto na GPU quanto na CPU - caso o usuário do *YOLO* opte pela utilização daquela, é possível aumentar drasticamente a velocidade de processamento, o que torna o reconhecimento em tempo real do *YOLO* possível.

1.5 Multiprocessamento

Multiprocessamento é uma forma de paralelização em que se utiliza mais de um processo para a execução de um programa. Isso pode ser feito a partir de CPUs disponíveis no computador, tanto as virtuais quanto as físicas.

Em geral é importante notar a diferença entre o processamento de *threads* e de processos visto que o primeiro utiliza-se de um mesmo contexto para todas as *threads* - dessa forma, os valores de acesso às variáveis são compartilhados, por exemplo. Isso pode submetê-las a condições de corrida, que podem ser contornadas a partir de bloqueios. Já o multiprocessamento possui isolamento das variáveis. Um conjunto de processos pode rodar uma mesma função, mas possuirá contextos diferentes. O *Python*, como já contextualizado anteriormente, utiliza-se de uma GIL, o que faz com que a paralelização *multithread* de alguns programas seja limitada.

Capítulo 2

Resultados e Discussões

A utilização das tecnologias apresentadas no capítulo anterior resume-se à utilização do arcabouço *Django* para desenvolvimento da interface para o sistema; a utilização do *YOLO* para realização do processo de inferência na identificação de caixas delimitadoras de classes nas imagens dos vídeos; uso do *middleware Celery* para apresentação da barra de carregamento do processamento do vídeo; e multiprocessamento como método de paralelização da pipeline.

2.1 Desenvolvimento

O desenvolvimento do projeto de paralelização por multiprocessamento foi realizado inicialmente no ambiente de desenvolvimento disponibilizado pelo *Jupyter Project* (2022), o *Jupyter Notebook*. Após essa fase, o projeto foi migrado para o ambiente local e mais tarde transferido para a rede *data.ime.usp.br* que possuía maiores capacidades computacionais com GPU de 11264MiB de memória e 16 cores disponíveis para processamento.

Como abordagem aplicada para se estudar os diferentes *pipelines* de processamento, os grupos de processos foram separados em leitores de vídeos, ou seja, o pré-processamento; a inferência, ou seja, o processamento; e os escritores de vídeo, ou seja, o pós-processamento. Inicialmente, a estrutura de processamento baseava-se em uma quantidade arbitrária de processos de inferência disponíveis com apenas um leitor e um escritor de quadros de vídeo, como observado na figura 2.1. Mais tarde, contudo, outra implementação foi desenvolvida, a qual obteve melhores resultados, apresentada na figura 2.2. Embora exigisse uma maior quantidade de processos, essa *pipeline* mostrou maior velocidade de processamento: a partir de um *pipeline* próprio para cada um dos processos de inferência do *YOLO*, que não poderiam compartilhar o contexto de cada processo das GPUs, cada um dos processos com um contexto receberia seu próprio leitor e escritor de quadros de vídeo.

Além dessas duas implementações, foram desenvolvidas outras duas *pipelines* diferentes para estudo sobre qual seria a alternativa de processamento que alcançasse maior velocidade. A hipótese levantada para essas abordagens provém da possibilidade da existência de um gargalo encontrado na alimentação dos quadros de vídeo para o processamento do *YOLO* - assim, como a inferência realizada pelo *YOLO* seria muito rápida, haveria uma

necessidade de um maior número de leitores de quadros de vídeo para que a exigência de alimentação do processo de inferência fosse suprido.

As outras duas *pipelines* desenvolvidas para estudo dessa hipótese ofereceram maior quantidade de leitores para 1 processo de inferência do *YOLO*, embora houvessem diferença na quantidade de escritores de vídeo: uma *pipeline* realizava a escrita dos quadros de vídeo com apenas um processo, vista na figura 2.3 enquanto a outra utilizava-se de vários escritores, a figura 2.4. Como já mencionado, esses dois *pipelines* estudados não obtiveram resultados tão bons quanto a segunda implementação.

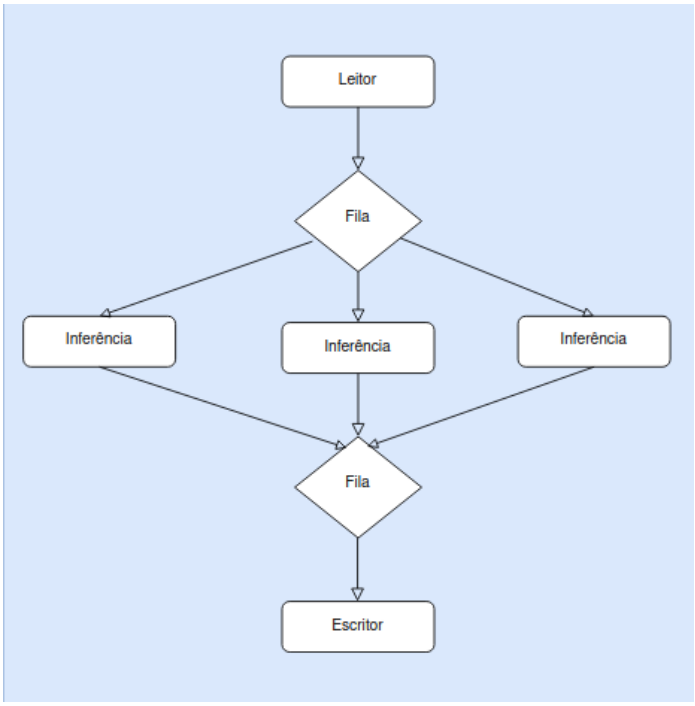

Figura 2.1: *Primeira pipeline*

Após realizado o processamento das *pipelines*, foi necessário realizar a junção das seções de vídeo processadas. Isso foi feito a partir da utilização do comando *ffmpeg* para a combinação dos intervalos. A utilização do comando foi escolhida a fim de que não se realizasse nova renderização integral do vídeo, com o intuito de se eliminar tempo de processamento. Inicialmente os quadros de vídeo processados eram salvos como imagens, mas a alimentação de um *buffer* de vídeo e subsequente junção das seções apresentou-se como uma alternativa mais rápida.

As implementações das *pipelines* diferem na quantidade de processos utilizados para cada seção de processamento, ou seja, pré-processamento, processamento e pós-processamento. Contudo, a arquitetura do código segue uma mesma lógica - há um programa principal que inicia a execução dos subprocessos. A fim de se evitar vazamentos de memória causados pela quantidade de quadros de vídeo em processamento ou em transferência nas filas, o programa regula a quantidade de quadros sendo processados a todo momento. Isso é feito a partir de mensagens enviadas por filas ou sinais da biblioteca *multiprocessing*. Assim, a regulação é realizada por meio da comunicação entre os processos. Os processos escritores de quadro de vídeo constituem a última fase a lidar com

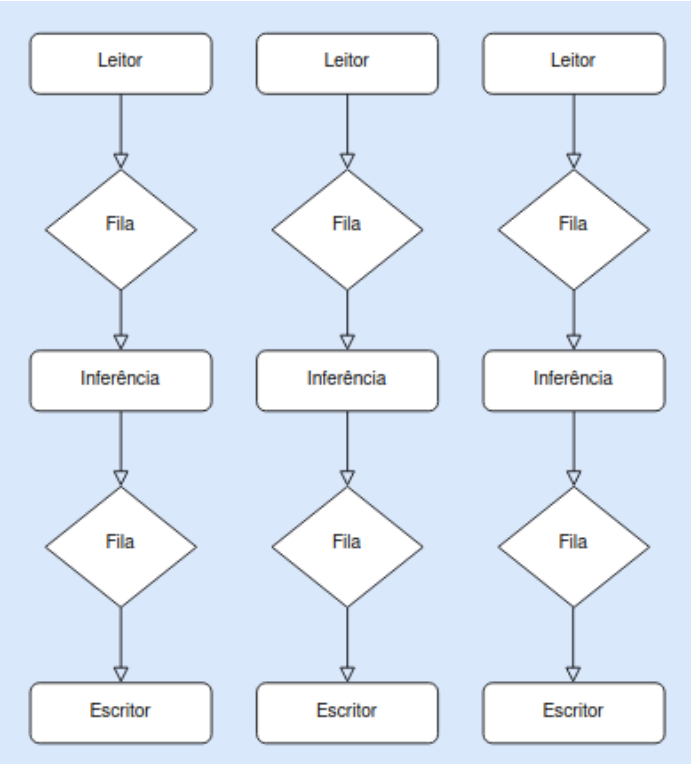

Figura 2.2: *Segunda pipeline*

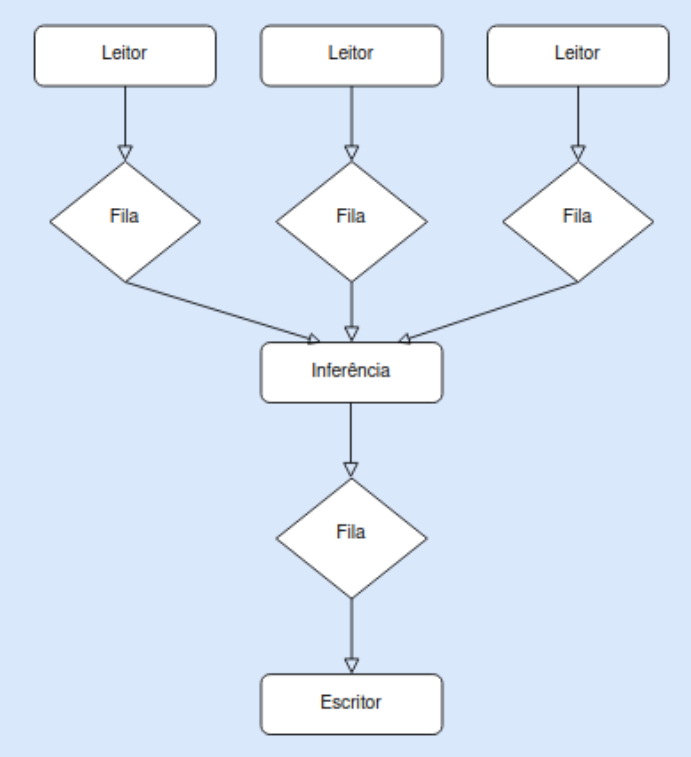

Figura 2.3: *Terceira pipeline*

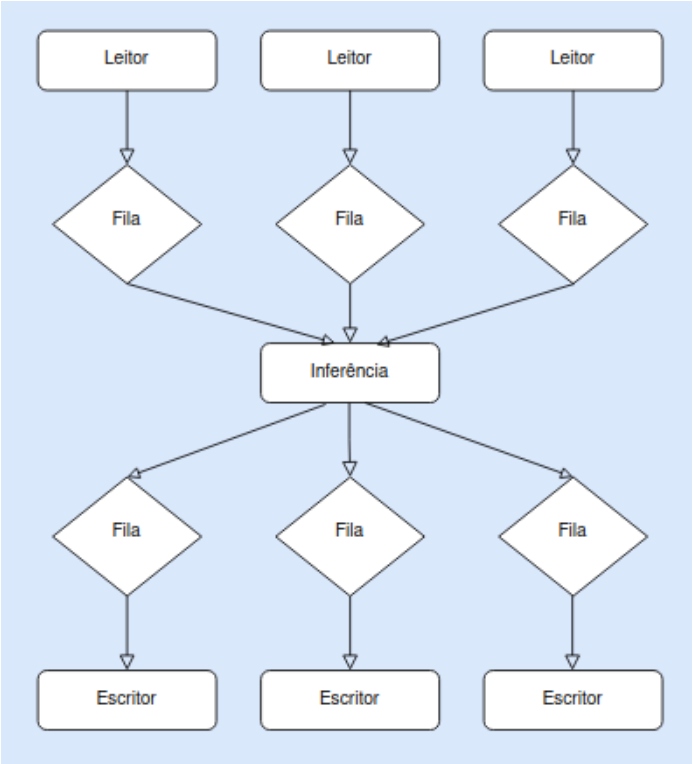

Figura 2.4: *Quarta pipeline*

o processamento dos quadros de vídeo, logo, são eles que realizam a sinalização para os outros processos ”acima” de que os quadros de vídeo já foram processados e que se pode liberar o consumo de mais quadros de vídeo para o programa.

A implementação do projeto foi inicialmente testada com uma aplicação desenvolvida no ambiente *Jupyter Notebook* com paralelização *multithread*. Contudo, essa aplicação não obteve sucesso devido ao compartilhamento de contexto entre as *threads* - dessa forma, o código foi alterado para se realizar a paralelização com muitos processos. Foi utilizado um ambiente virtual do *Python* para se isolar as instalações necessárias da máquina em relação ao servidor principal.

Após o desenvolvimento desse sistema, foi implementada a interface do projeto a partir do arcabouço *Django*. Os modelos do *Django* não foram totalmente utilizados visto que o uso do *OpenCv*, uma biblioteca multiplataforma para desenvolvimento em visão computacional, fazia com que os arquivos fossem salvos diretamente nos repositórios do servidor. O uso do *middleware Celery* foi feito para se realizar a requisição assíncrona da barra de progresso disponível na interface, a fim de se acompanhar o processamento do vídeo a partir da utilização da função principal de administração de processos. Isso foi necessário visto que se apresentava a necessidade de se identificar as iterações em tempo real do programa, o que poderia ser feito de forma mais fácil a partir de uma fila direta até que o processamento do vídeo chegasse ao fim.

Para executar o programa, é necessário iniciar um servidor *Redis* para inicializar os trabalhadores do *Celery*. Após a inicialização, pode-se rodar o servidor do *Django*. Abaixo encontra-se a explicação da interface para utilização do programa:

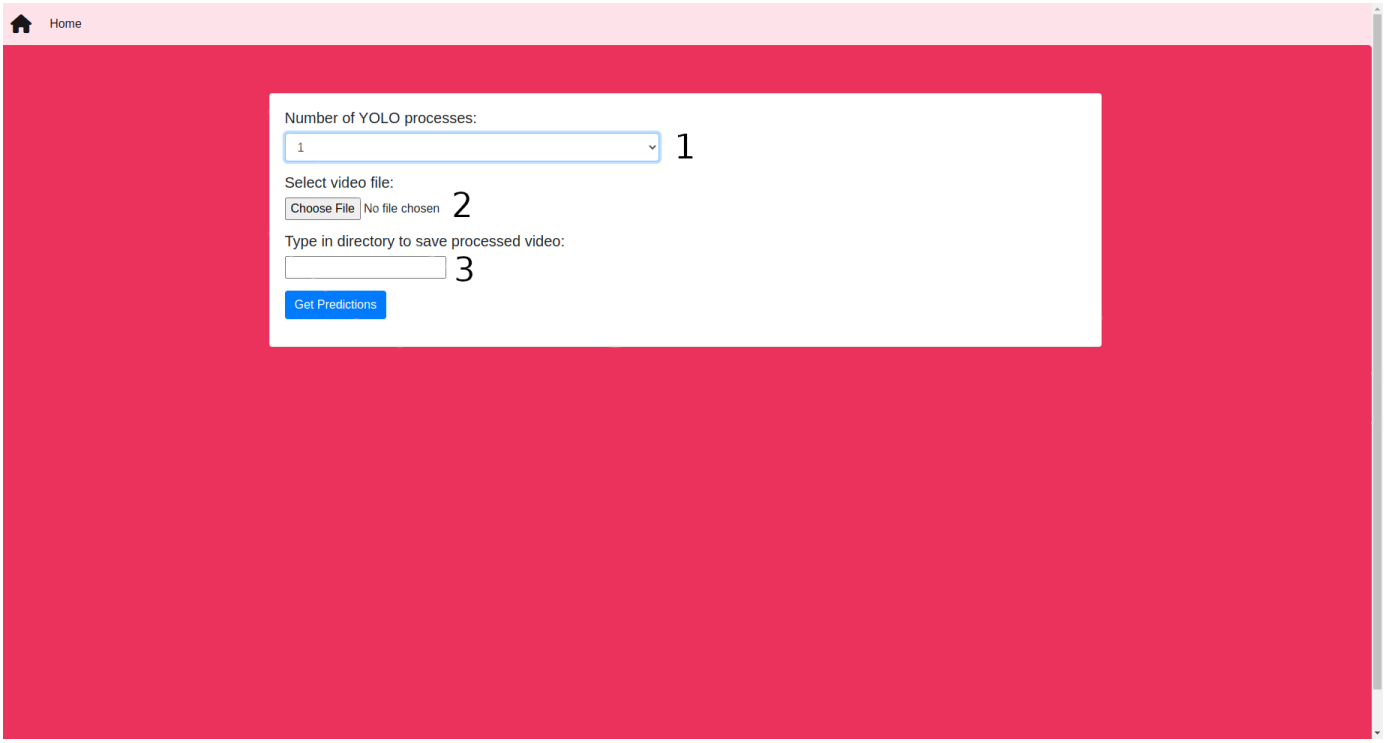

Figura 2.5: *Primeira página*

A Figura 2.5 apresenta as opções da interface. Nela, pode-se observar os botões de opção: 1) definição da quantidade de contextos do *YOLO*, ou seja, para o número escolhido, serão instanciados 3*n (n o número de processos do *YOLO*) para paralelização do processo. 2) escolha do vídeo a ser processado; 3) escolha da pasta a ser salvo.

Já a Figura 2.6 apresenta a página de processamento do vídeo. Os comandos disponibilizados são: 1) na segunda tela, pode-se observar a barra de progresso do processamento do vídeo, 2) um botão para voltar a página, 3) um botão para abertura da pasta salva, 4) um botão para combinação dos vídeos em seções,

O desenvolvimento de uma aplicação com processamento dos vídeos em lote foi iniciado. Contudo, não foi completamente integrado ao projeto.

2.2 Resultados

Os vídeos processados puderam ser testados a partir de intervalos diferentes de tempo. Foram utilizados vídeos de 10 segundos, 3 minutos e 1 hora de duração.

Os vídeos de 1 hora de duração possuíam cerca de 90 mil quadros de vídeo. Na implementação mais rudimentar de 1 processo escritor, 1 processo leitor e 1 processo de inferência, exigiam em média, 1 hora (3600 segundos) para serem inteiramente processados. Essa proporção direta de consumo de tempo manteve-se constante para os vídeos de 3 minutos e 10 segundos. Contudo, com a utilização do melhor *pipeline* com 3 processos de inferência, leitura e escrita; o processamento do vídeo de 1 hora pôde ser agilizado para uma média de 34 minutos (2040 segundos) de tempo de processamento. Dessa forma,

Figura 2.6: *Segunda página*

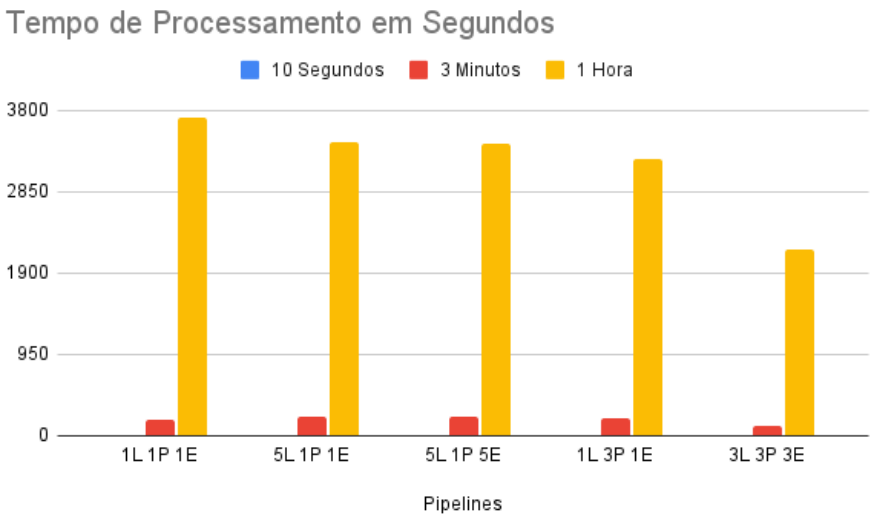

Figura 2.7: *Gráfico com medições das médias de processamento*

a necessidade advinda da abundante quantidade de vídeos gerados pelas câmeras de segurança da USP pode ser suprida, visto que o consumo de tempo de processamento cai para metade.

Na figura 2.7, pode-se observar a diferença das médias de tempo de processamento das diferentes pipelines implementadas. Os dados apresentados encontram-se em tempo de total de processamento em segundos para a quantidade de processos utilizados nas diferentes *pipelines*. Na imagem, ”L” indica leitor de quadros de vídeo (ou seja, pré-processamento); ”P” indica processamento (ou seja, o processo de inferência realizado pelo *YOLO*) e ”E” indica o processo escritor de quadros de vídeos. Em cores diferentes são apresentadas as durações dos vídeos processados: 1 hora, 3 minutos e 10 segundos de vídeo. Como é possível observar, o melhor desempenho foi a implementação da *pipeline* com 3 processos leitores para 3 processos de inferência para 3 processos escritores.

2.3 Dificuldades e Limitações

Inicialmente foram encontrados alguns problemas no desenvolvimento do arcabouço, visto que foi implementado a partir de threads. Em alguns casos, a utilização de *threads* pode ser realizada, mas isso era impossível devido à utilização da GPU: a comunicação entre GPU e CPU demanda processos distintos, visto que as *threads* criadas no contexto da CPU não possuem acesso à memória compartilhada das *threads* sob domínio da GPU. Além disso, não foi desenvolvida uma implementação específica para *Pytorch* ou *Tensorflow*.

Capítulo 3

Conclusão

Como exposto anteriormente, a *pipeline* direta possuía um consumo de tempo similar à duração do tempo dos vídeos analisados. Contudo, a *pipeline* de 3 leitores, 3 processos de inferência e 3 escritores de vídeo adquiriu resultados muito superiores em comparação à *pipeline* de controle. É possível concluir que o objetivo deste trabalho de conclusão foi, então, atingido, pois foi possível agilizar o processamento de vídeos em comparação com as versões de apenas 1 leitor, 1 escritor e 1 instância do *YOLO*.

Como próximos passos a serem realizados, levanta-se a possibilidade de se realizar a paralelização de forma distribuída, ou seja, com várias máquinas diferentes para isolamento das GPUs. Seria interessante isolar os contextos que rodam o *YOLO* em máquinas diferentes, ou seja, com GPUs isoladas - dessa forma, a vazão de dados poderia se tornar maior para apenas um processo sendo executado em comparação com a execução de muitos processos ao mesmo tempo. A partir da implementação já realizada, seria interessante reutilizar o código da melhor pipeline desenvolvida com o auxílio do *Celery*, já que o *middleware* dispõe de opções de aplicação de multiprocessamento distribuído para desenvolvimento.

Além disso, seria interessante expandir o projeto para a utilização com outros arcabouços de aprendizado profundo, como o *PyTorch* ou o *Tensorflow*, a fim de se expandir a funcionalidade deste projeto. Dessa forma, o sistema poderia se tornar reutilizável para a paralelização com vários outras implementações de arcabouços de classificação de imagens em aprendizado profundo.

Referências

[Bernal 2017] Volnys Borges Bernal. *Exclusão Mútua*. 2017. url: https://edisciplinas.usp.br/pluginfile.php/3061851/mod_resource/content/2/36-Mutex-v27.pdf (acesso em 10/01/2023) (citado na pg. 2).

[*Billiard* 2022] *Billiard*. url: https://github.com/celery/billiard (acesso em 05/12/2022) (citado na pg. 6).

[Bochkovskiy *et al.* 2020] Alexey Bochkovskiy, Chien-Yao Wang e Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. doi: 10.48550/ARXIV.2004.10934. url: https://arxiv.org/abs/2004.10934 (citado na pg. 8).

[*Celery* 2022] *Celery*. url: https://docs.celeryq.dev/en/stable/index.html (acesso em 05/12/2022) (citado na pg. 6).

[Cidade Universitária Armando Salles Oliveira 2016] Prefeitura da Cidade Universitária Armando Salles Oliveira. *CUASO em Números*. 2016. url: https://puspc.usp.br/cuaso-em-numeros/ (acesso em 22/12/2022) (citado na pg. 1).

[Columbia 2023] Columbia. *Artificial Intelligence (AI) vs. Machine Learning*. 2023. url: https://ai.engineering.columbia.edu/ai-vs-machine-learning/ (acesso em 08/02/2023) (citado na pg. 6).

[Erika Yamamoto 2019] Aline Naoe e Erika Yamamoto. *USP regulamenta o ciclismo esportivo na Cidade Universitária*. Mai. de 2019. url: https://jornal.usp.br/universidade/usp-regulamenta-o-ciclismo-esportivo-na-cidade-universitaria/ (acesso em 22/12/2022) (citado na pg. 1).

[Foundation 2022] Django Software Foundation. *Django Documentation*. url: https://docs.djangoproject.com/en/4.1/ (acesso em 05/12/2022) (citado na pg. 5).

[He *et al.* 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren e Jian Sun. "Deep residual learning for image recognition". *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. url: http://arxiv.org/abs/1512.03385 (citado na pg. 8).

[Hernandez 2021] Rafael D. Hernandez. *The Model View Controller Pattern – MVC Architecture and Frameworks Explained*. 2021. URL: https://www.freecodecamp.org/news/content/images/size/w1600/2021/04/MVC3.png (acesso em 07/02/2023) (citado nas pgs. vi, 5).

[IBM s.d.] IBM. *What is gradient descent?* URL: https://www.ibm.com/topics/gradient-descent (citado na pg. 8).

[*Jupyter Project* 2022] *Jupyter Project*. URL: https://jupyter.org/ (acesso em 05/12/2022) (citado na pg. 11).

[Lin *et al.* 2014] Tsung-Yi Lin *et al.* "Microsoft COCO: common objects in context". *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312 (citado na pg. 9).

[Nardi *et al.* 2022] Ednilza Nardi, Bruno Padilha, Leonardo Kamaura e João Ferreira. "Openimages cyclists: expandindo a generalização na detecção de ciclistas em câmeras de segurança". In: *Anais do XXXVII Simpósio Brasileiro de Bancos de Dados*. Búzios: SBC, 2022, pp. 229–240. DOI: 10.5753/sbbd.2022.224626. URL: https://sol.sbc.org.br/index.php/sbbd/article/view/21809 (citado nas pgs. 1, 2, 9).

[Norman 2014] Jeremy M. Norman. *The Inspiration for Artificial Neural Networks, Building Blocks of Deep Learning*. 2014. URL: https://historyofinformation.com/detail.php?entryid=4726 (acesso em 08/02/2023) (citado na pg. 7).

[Ramamoorthy e Li 1977] C. V. Ramamoorthy e Hon Fung Li. "Pipeline architecture". *ACM Comput. Surv.* 9.1 (1977), pp. 61–102. DOI: 10.1145/356683.356687. URL: https://doi.org/10.1145/356683.356687 (citado na pg. 2).

[Redmon *et al.* 2015] Joseph Redmon, Santosh Divvala, Ross Girshick e Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. 2015. DOI: 10.48550/ARXIV.1506.02640. URL: https://arxiv.org/abs/1506.02640 (citado na pg. 8).

[Rosenblatt 1958] Frank Rosenblatt. *The Design of an Intelligent Automaton*. 1958. URL: https://news.cornell.edu/sites/default/files/styles/full_size/public/2019-09/0925_rosenblatt4.jpg?itok=UQKthwdI (acesso em 08/02/2023) (citado nas pgs. vi, 7).

[*Universidade de São Paulo* 2022] *Universidade de São Paulo*. 2022. URL: https://www6.usp.br/institucional/a-usp/#organizacao (acesso em 22/12/2022) (citado na pg. 1).

[Yamamoto 2018] Erika Yamamoto. *USP inaugura Centro de Monitoramento Eletrônico*. 2018. URL: https://jornal.usp.br/institucional/usp-inaugura-centro-de-monitoramento-eletronico-na-cidade-universitaria/ (acesso em 30/01/2023) (citado na pg. 1).