FUNDAÇÃO GETULIO VARGAS
EMAp - ESCOLA DE MATEMÁTICA APLICADA
ESPECIALIZAÇÃO EM IA APLICADA A SISTEMAS MILITARES

HELBER SOARES MOTA
ROBERT VICENTE LIBOTTI

RECONHECIMENTO DE IMAGENS: USO DE RECURSOS DE VISÃO COMPUTACIONAL NA DETECÇÃO DE ALVOS E VIGILÂNCIA NO ESPECTRO MILITAR

Rio de Janeiro - RJ
2024

HELBER SOARES MOTA
ROBERT VICENTE LIBOTTI

RECONHECIMENTO DE IMAGENS:
USO DE RECURSOS DE VISÃO COMPUTACIONAL
NA DETECÇÃO DE ALVOS E VIGILÂNCIA NO ESPECTRO MILITAR

Trabalho de conclusão de curso apresentado à EMAp - Escola de Matemática Aplicada da Fundação Getúlio Vargas como parte dos requisitos necessários para a conclusão do Curso de Especialização em IA Aplicada a Sistemas Militares.

Orientador: Prof. Rafael de Pinho André

Rio de Janeiro
2024

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Reconhecimento de Imagens: Uso de Recursos de Visão Computacional na Detecção de Alvos e Vigilância no Espectro Militar

EMAp - Escola de Matemática Aplicada da Fundação Getúlio Vargas
Rio de Janeiro - Brasil
robert.libotti@gmail.com, helberjf@gmail.com

Resumo

A crescente complexidade dos conflitos modernos exige soluções tecnológicas que possam identificar e classificar veículos militares de forma eficiente. Essa identificação é fundamental em aplicações de reconhecimento aéreo e terrestre, monitoramento de fronteiras, ações de sabotagem, tarefas de inteligência, contra-ataque e no uso de drones em operações táticas. O emprego de inteligência artificial (IA) nesse processo, mais especificamente através de modelos de redes neurais convolucionais (CNNs), se mostra uma abordagem robusta e viável, oferecendo alta acurácia e velocidade na detecção e classificação de objetos em imagens complexas. O objetivo deste trabalho consiste em propor uma abordagem para detectar e classificar algumas classes de veículos militares (a saber: ASTROS, CLANF, JLTV, M113, PIRANHA e SK105) através de imagens que podem ser capturadas a partir de drones, câmeras fixas ou móveis combinando o modelo YOLOv10 e uma CNN. Os resultados experimentais mostram que o modelo alcançou um desempenho considerável (com precisão acima de 90% em muitos casos) e mesmo que a quantidade de imagens utilizadas no treino tenha sido pequena para uma aplicação real, demonstra a viabilidade para utilização em dispositivos embarcadas ou em uma arquitetura em nuvem com processamento remoto, por exemplo.

Palavras-chave

YOLO; PyTORCH, CNN; Visão Computacional; IA; Veículos Militares.

1. Introdução

A inteligência artificial (IA) é definida como “a capacidade de um sistema interpretar corretamente dados externos, aprender a partir desses dados e utilizar esse conhecimento para atingir objetivos e tarefas específicas por meio de adaptação flexível” (Kaplan; Haenlein, 2019). É um campo da ciência da computação voltado para a criação de sistemas capazes de realizar tarefas que normalmente exigiriam inteligência humana, como reconhecimento de padrões, aprendizado e tomada de decisão. Baseada em técnicas como aprendizado de máquina e aprendizado profundo, a IA é amplamente aplicada em soluções específicas de Visão Computacional para detectar e classificar objetos em imagens. No contexto deste trabalho, a IA é essencial para o desenvolvimento de tecnologias que permitam a identificação rápida e precisa de veículos militares em cenários complexos diversos.

2. Objetivos

2.1. Objetivo Geral

A introdução da IA em sistemas de defesa representa uma evolução significativa na forma como os dados são processados e utilizados em operações críticas, reforçando a importância do tema em um mundo cada vez mais digitalizado e interconectado.

2.2. Objetivos Específicos

Neste trabalho, propomos explorar o uso do YOLO para o reconhecimento de veículos militares através de imagens para uso em sistemas embarcados em drones (ou qualquer dispositivo que possa desempenhar processamento local) ou remotamente através de uma arquitetura em nuvem, com foco na aplicação prática e análise do desempenho do modelo. Para isso, serão abordadas etapas como a coleta e anotação de dados específicos, incluindo a criação de caixas delimitadoras (*bounding boxes*) para cada veículo presente nas imagens do banco de dados, o treinamento do modelo em um *dataset* customizado e a validação de seu desempenho em diferentes condições. Adicionalmente, serão verificados os desafios enfrentados, como a presença de ruídos nas imagens, variações de iluminação, oclusões e a

diversidade visual dos veículos militares, que incluem tanques, carros blindados, veículos anfíbios de assalto e outros meios militares de transporte especializados.

A relevância deste estudo reside no potencial de aprimorar sistemas automatizados de detecção, contribuindo para o avanço tecnológico na área de defesa e segurança. Além disso, os resultados podem servir como base para a implementação de soluções práticas em diferentes contextos, como vigilância autônoma por drones e câmeras, análise estratégica em tempo real, uso de sistemas de inteligência artificial de reconhecimento de imagens em planejamentos por cartas topográficas, como por exemplo a carta de trafegabilidade sendo feita de forma automática e apoio à tomada de decisões táticas.

Assim, este trabalho está estruturado em capítulos que abordam, inicialmente, os fundamentos teóricos do reconhecimento de imagens, das redes neurais convolucionais e do YOLO, seguidos da metodologia adotada para o treinamento e validação do modelo. Posteriormente, serão apresentados os resultados obtidos e suas implicações práticas, culminando nas considerações finais e sugestões para trabalhos futuros. O objetivo principal é demonstrar como o uso do YOLO pode ser eficaz no reconhecimento de veículos militares, oferecendo uma contribuição significativa para a área de visão computacional e suas aplicações no setor militar.

3. Fundamentação Teórica

3.1. Conceitos Relacionados

A fundamentação teórica deste trabalho aborda os conceitos-chave relacionados ao reconhecimento de imagens, redes neurais convolucionais (CNNs) e ao modelo YOLO (*You Only Look Once*). Esses tópicos são fundamentais para compreender a aplicação de técnicas de visão computacional no reconhecimento de veículos militares em imagens.

3.2. Reconhecimento de Imagens

O reconhecimento de imagens é uma subárea da visão computacional que visa identificar e classificar objetos em imagens digitais. Essa tarefa envolve a análise de

características específicas, como formas, cores, texturas e padrões, que permitem diferenciar objetos e atribuí-los a categorias definidas.

Com o avanço da tecnologia, os métodos tradicionais baseados em extração manual de características deram lugar a abordagens baseadas em aprendizado de máquina, especialmente aprendizado profundo (*deep learning*). Esses métodos utilizam redes neurais artificiais capazes de aprender representações complexas diretamente a partir dos dados, eliminando a necessidade de intervenção manual para a extração de características.

A importância do reconhecimento de imagens está em sua ampla gama de aplicações, que incluem sistemas de vigilância, diagnóstico médico, direção autônoma e, no contexto militar, a identificação de veículos, tropas e equipamentos em ambientes operacionais.

3.3. Redes Neurais Convolucionais

Redes Neurais Convolucionais (Convolutional Neural Networks, CovNets ou CNNs) são um tipo específico de rede neural projetada para processar dados que possuem uma estrutura em forma de grade, como imagens. Elas são amplamente utilizadas em tarefas de visão computacional, como reconhecimento de objetos, segmentação de imagens e detecção de padrões.

A estrutura básica de uma CNN é composta pelos seguintes elementos:

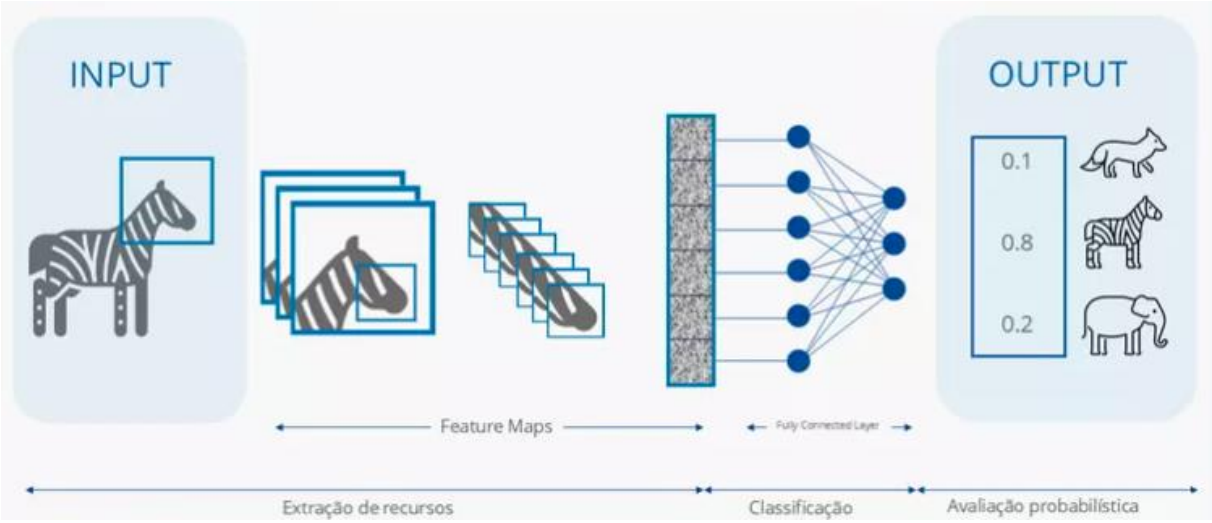
Tabela 1. Estrutura Básica de uma CNN

Camadas Convolucionais	Aplicam filtros (ou kernels) à imagem de entrada para extrair características relevantes, como bordas, texturas e formas. Cada filtro aprende uma característica específica durante o treinamento. O resultado dessa operação é um mapa de características (feature map), que destaca as regiões onde os padrões foram detectados.
Camadas de Pooling (ou Subamostragem)	Reduzem a dimensionalidade das características extraídas, mantendo as informações mais importantes e aumentando a eficiência computacional. Exemplos incluem o max pooling (seleção do valor máximo em uma região) e o average pooling (média dos valores).
Camadas Fully Connected	Geralmente realizam a classificação final com base nas características extraídas.

Funções de Ativação	Introduzem não linearidade ao modelo, permitindo a aprendizagem de padrões complexos. Funções como ReLU (Rectified Linear Unit) e softmax são amplamente utilizadas.

O uso de CNNs em tarefas de reconhecimento de imagens revolucionou a área, proporcionando ganhos significativos em precisão e eficiência. CovNets são capazes de extrair características relevantes de dados automaticamente e com alta precisão. No entanto, para treinar redes neurais convolucionais com eficiência é necessário que se tenha acesso a recursos computacionais significativos. O Treinamento necessita de grande quantidade de volume de dados rotulados e GPUs poderosas para obtenção de bons resultados.

Figura 1. Convolutional Neural Network (CNN)



Fonte: https://www.ionos.com/pt-br/digitalguide/sites-de-internet/desenvolvimento-web/convolutional-neural-network/

3.4. Algoritmos de Detecção de Objetos

3.4.1 R-CNN (O Pioneiro)

A R-CNN, ou Regiões com recursos da CNN, entrou em cena em 2014, marcando uma mudança de paradigma na detecção de objetos. Como funciona:

- Gera propostas de região (~2000) usando pesquisa seletiva
- Extrai recursos da CNN de cada região

- Classifica regiões usando classificadores SVM

3.4.2 Fast R-CNN

A Fast R-CNN abordou as limitações de velocidade de seu antecessor, mantendo alta precisão. Como funciona:

- Processa a imagem inteira através da CNN uma vez
- Usa o agrupamento de ROI para extrair recursos para cada proposta de região
- Usa camada softmax para classificação e regressão de caixa delimitadora

3.4.3 Faster R-CNN

A Faster R-CNN introduziu a Region Proposal Network (RPN), tornando todo o pipeline de detecção de objetos treinável de ponta a ponta. Como funciona:

- Usa uma rede totalmente convolucional para gerar propostas de região
- Compartilha recursos convolucionais de imagem completa com a rede de detecção
- Treina RPN e Fast R-CNN juntos

Exemplo de uso: Na condução autônoma, o Faster R-CNN pode detectar e classificar veículos, pedestres e sinais de trânsito quase que em tempo real, o que é crucial para tomada rápida de decisões.

3.4.4 YOLO

O YOLO (*You Only Look Once*) revolucionou a detecção de objetos ao enquadrá-la como um único problema de regressão, direto dos pixels da imagem para as coordenadas de caixas delimitadoras e probabilidades de classe. Como funciona:

- Divide a imagem em uma grade
- Para cada célula da grade, prevê caixas delimitadoras e probabilidades de classe
- Aplica uma única passagem para frente em toda a imagem

3.5. Comparando Prós e Contras

De acordo com a literatura pesquisada, comparando os algoritmos de detecção e reconhecimento de imagens R-CNN, Fast R-CNN, Faster R-CNN e YOLO, temos o seguinte:

Tabela 2. Prós e Contras: R-CNN, Fast R-CNN, Faster R-CNN e YOLO

	PRÓS	CONTRAS
R-CNN (Regions with Convolutional Neural Networks)	**Alta precisão**: R-CNN é conhecido por sua alta precisão na detecção de objetos **Flexibilidade**: Pode ser usado em uma variedade de aplicações devido à sua precisão.	**Lento**: O processo é relativamente lento porque envolve várias etapas, incluindo a geração de propostas de região e a extração de características. **Requer muitos recursos**: Devido ao seu processo complexo, consome muitos recursos computacionais.
Fast R-CNN	**Mais rápido:** Melhora significativamente o tempo de processamento em comparação com o R-CNN. **Maior eficiência**: Processa a imagem inteira uma vez e usa RoI pooling para extrair características de cada região proposta.	**Ainda depende de propostas externas:** A geração de propostas de região ainda é um gargalo. **Requer ajustes manuais**: Precisa de ajustes manuais para otimizar o desempenho.
Faster R-CNN	**Propostas rápidas:** Introduz uma rede de proposta de região (RPN) que gera propostas de região rapidamente. **Mais rápido e eficiente:** Melhora ainda mais o tempo de processamento e a eficiência em comparação com Fast R-CNN.	**Complexidade:** A introdução da RPN aumenta a complexidade do modelo. **Requer ajustes manuais:** Assim como Fast R-CNN, precisa de ajustes manuais para otimizar o desempenho.
YOLO (You Only Look Once)	**Rápido**: YOLO é muito mais rápido do que R-CNN porque processa a imagem inteira em uma única passagem. **Eficiente**: Usa menos recursos computacionais em comparação com R-CNN.	**Menor precisão**: Embora seja rápido, YOLO pode ser menos preciso em comparação com R-CNN, especialmente em imagens complexas. **Dificuldade em detectar objetos pequenos**: YOLO pode ter dificuldade em detectar objetos pequenos ou que estão próximos uns dos outros.

fonte: https://datadance.ai/machine-learning/r-cnn-vs-r-cnn-fast-vs-r-cnn-faster-vs-yolo/

Figura 2. Comparação da Evolução na Detecção de Objetos (R-CNN, Fast R-CNN, Faster R-CNN e YOLO)

Algorithm	Speed	Accuracy	Real-time?	End-to-end Trainable	Year
R-CNN	47s/image	High	No	No	2014
Fast R-CNN	2s/image	Higher	No	Partial	2015
Faster R-CNN	0.2s/image	Highest	Near	Yes	2015
YOLO	0.02s/image	Good	Yes	Yes	2016

fonte: https://datadance.ai/machine-learning/r-cnn-vs-r-cnn-fast-vs-r-cnn-faster-vs-yolo/

4. Metodologia

Este capítulo apresenta a metodologia adotada para o desenvolvimento do modelo de reconhecimento de veículos militares utilizando o YOLO. São descritas as etapas práticas de coleta e anotação de dados, configuração do modelo, treinamento e validação, detalhando os procedimentos técnicos e as ferramentas utilizadas. O objetivo é garantir a reprodutibilidade e a clareza do processo.

4.1. Coleta e Anotação de Dados`

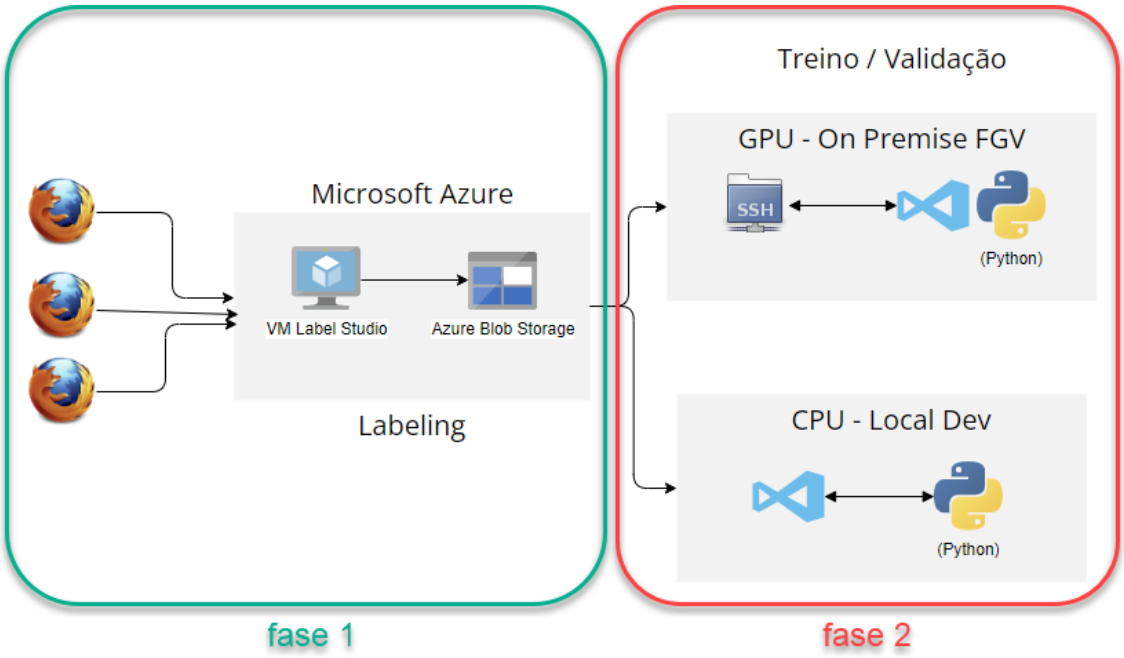
A primeira etapa da metodologia consiste na obtenção de um conjunto de dados adequado para o treinamento do modelo. Para este trabalho, foram utilizadas imagens de veículos militares provenientes de fontes públicas, como bancos de imagens, vídeos de desfiles militares e registros de operações militares.

Critérios de seleção das imagens: diversidade visual, incluindo diferentes tipos de veículos militares, como tanques, carros blindados de combate e veículos anfíbios de assalto. Variações de cenário: ambientes urbanos, florestais, desérticos e outros. Diferentes condições climáticas e de iluminação: imagens em alta e baixa iluminação, presença de ruído e oclusões.

Após a coleta de 4361 imagens, foi realizada a classificação dos arquivos de treino/validação que foram armazenados em um *Data Lake* privado em nuvem no *Microsoft Azure* (*Microsoft Azure Blob Storage*) e que foi utilizado posteriormente como repositório de dados para o *Label Studio* que é uma ferramenta de código aberto para rotulagem de dados que permite a integração com diversos sistemas de armazenamento em nuvem. Para rodar o *Label Studio* em estrutura própria, utilizamos

um servidor virtual Linux Ubuntu 24.04 em nuvem com 4vcpus, 16GiB RAM e HD Premium SSD LRS de 30GiB de acordo com a arquitetura abaixo:

Figura 3. Arquitetura Utilizada Azure/OnPremise/Local



Na primeira fase, alimentamos um ***datalake*** com as imagens em nuvem e fizemos o trabalho de *labeling* com auxílio do *Label Studio*. Na segunda fase, para processamento e fins comparativos, lemos as anotações e as imagens a partir do hardware local (processador Intel i7 de 10ª geração com 8 núcleos, 32Gb RAM e HD SSD de 1TB mas sem GPU) e hardware *on premise* do ambiente do pesquisador da FGV (GPU: Tesla V100-PCIE-32GB com 32768.0 MB Memória), treinamos nosso modelo customizado e escrevemos os resultados para análise e inferência no Wandb (wandb.io).

4.1.1 Labeling

No processo inicial de *labeling*, foram criadas caixas delimitadoras (*bounding boxes*) nos dados de treino e validação a fim de identificar cada classe de veículo desejado. Cada *bounding box* foi rotulada com a classe correspondente aos veículos tratados no escopo deste trabalho.

Figura 4. Exemplos de Labeling Gerados Pelo Label Studio


1: ASTROS
ASTROS
CLANF
JLTV
M113
PIRANHA
SK105
MILITAR-TRUCK
1: CLANF
2: CLANF
3: CLANF
1: SK105

4.1.2 Detalhes do processo de Anotação e Divisão do DataSet

As caixas delimitadoras foram ajustadas para englobar totalmente o veículo, evitando incluir áreas irrelevantes. Cada veículo foi rotulado com uma classe específica: “SK105” (tanque leve), ”ASTROS” (sistema de lançadores múltiplos de foguetes - MLRS), “CLAnf” (carro lagarta anfíbio), “JLTV” (veículo tático multifuncional leve), ”M113” (veículo blindado de transporte de pessoal0 e “PIRANHA" (veículo blindado de combate com rodas).

Seguindo o Princípio de Pareto, o *dataset* foi dividido em treinamento (80%) e validação (20%) para garantir a avaliação objetiva do modelo.

4.2. Configuração do Modelo YOLO

A configuração do YOLO foi realizada com base em sua versão YOLOv10, devido à sua eficiência e facilidade de uso em frameworks modernos como PyTorch. As etapas de configuração envolveram ajustes na arquitetura do modelo, parâmetros de treinamento e preparação do ambiente computacional.

Para implementação e treinamento utilizamos a linguagem de programação Python na versão 3.10 pela compatibilidade com bibliotecas de aprendizado profundo.

Figura 5. GPU Utilizada no Servidor de Pesquisa da FGV

Tabela 3. Principais Parâmetros Utilizados no PreTreino GPUxCPU

mode: train	Modo de treino
model: yolov10n.yaml	Utilizamos o modelo n (nano) do YOLO v10 por ser a opção mais leve e fácil de implementar posteriormente em um dispositivo embarcado
epochs: 300	Após alguns pre-testes foi possível verificar convergência por volta da época 200. Posteriormente, para os dados apresentados neste trabalho, limitamos em 600 épocas e configuramos para parar automaticamente caso passasse por 10 epocas sem efetiva evolução do modelo.
workers: 8	Quantidade de processos paralelos
verbose: true	Gerar informações detalhadas sobre o processo de treinamento
show_labels: true	Exibir os rótulos dos objetos detectados
show_conf: true	Exibir a pontuação de confiança
show_boxes: true	Exibir as caixas ao redor dos objetos detectados

4.2.1 Treinamento do Modelo

O treinamento do modelo foi realizado utilizando o conjunto de dados anotado na etapa de ***labeling*** (dados de treino e validação). Durante esta etapa, o YOLO foi exposto às imagens de treinamento, ajustando seus pesos para minimizar a perda e aumentar a precisão na detecção dos tipos de veículos que definimos no escopo.

A função de perda do YOLO considera três componentes principais:

- **Erro de localização:** Avalia a precisão das coordenadas das bounding boxes previstas.
- **Erro de classificação:** Mede a correspondência entre a classe prevista e a classe real do objeto.
- **Erro de confiança**: Avalia a certeza do modelo em relação à presença de um objeto.

Treinamento em múltiplas escalas: O YOLOv10 foi configurado para treinar em imagens de tamanhos e iluminação variados em ângulos e ambientes diversos, e pudemos verificar sua robustez em detectar objetos de diferentes dimensões em cenários diversos.

Validação durante o treinamento: A cada época, o modelo foi validado utilizando o conjunto de validação, gerando métricas como precisão (mAP) e taxa de erro.

4.2.2 Métricas de Validação, Testes e Avaliação do Modelo

Após o treinamento, o modelo foi avaliado utilizando o conjunto de teste, composto por imagens inéditas que não foram vistas pelo modelo durante o treinamento. Essa etapa garantiu uma avaliação objetiva do desempenho do YOLO.

Essas métricas são usadas para monitorar e ajustar o desempenho do modelo durante o treinamento e a validação, garantindo que ele seja capaz de detectar e classificar objetos com alta precisão.

1. **Recall-Confidence(B)**: refere-se à capacidade de um modelo de identificar corretamente todas as instâncias relevantes de uma classe. É a proporção de verdadeiros positivos (TP) sobre a soma de verdadeiros positivos e falsos negativos (FN). Em outras palavras, é a capacidade do modelo de encontrar todos os exemplos positivos.

$$Recall = \frac{TP}{TP + FN}$$

2. **Precision-Recall(B)**: combina a precisão e o recall para fornecer uma única medida de desempenho. A precisão é a proporção de verdadeiros positivos sobre a soma de verdadeiros positivos e falsos positivos (FP). O Precision-Recall é frequentemente usado em gráficos para avaliar o desempenho de classificadores.

$$Precision = \frac{TP}{TP + FP}$$

3. **Precision-Confidence(B)**: refere-se à confiança que temos na precisão do modelo. A confiança pode ser interpretada como a probabilidade de que uma previsão positiva seja correta. Isso é importante em contextos em que a precisão é crítica, como diagnósticos médicos.

4. **F1-Confidence(B)**: o F1-Score é a média harmônica da precisão e do recall, proporcionando um equilíbrio entre os dois. É útil quando precisamos de um equilíbrio entre precisão e recall. A confiança no F1-Score indica a robustez do modelo em termos de equilíbrio entre precisão e recall.

$$F1 - Score = 2 \times \frac{Precision\ x\ Recall}{Precision + Recall}$$

5. **mAP50 (Mean Average Precision at IoU 0.50)**: a precisão média (AP) é calculada como a área sob a curva de precisão-recall para uma classe específica. O mAP50 é a média dessas precisões médias para todas as classes, considerando um limiar de IoU (*Intersection over Union*) de 0.50. Isso significa que uma detecção é considerada correta se a sobreposição entre a caixa predita e a caixa real for de pelo menos 50%.

6. **mAP50-95**: é uma média da precisão média calculada em diferentes limiares de IoU, variando de 0.50 a 0.95 (em incrementos de 0.05). Ela fornece uma visão mais abrangente do desempenho do modelo em diferentes níveis de dificuldade de detecção

7. **val/dfl_loss**: refere-se à perda de distribuição de deslocamento (*Distribution Focal Loss*) durante a validação. A DFL é uma técnica usada para melhorar a precisão da localização dos objetos, ajustando as previsões das caixas delimitadoras para que correspondam mais precisamente aos objetos reais nas imagens.

8. **val/cls_loss**: refere-se à perda de classificação (*Classification Loss*) durante a validação. A perda de classificação mede a precisão com que o modelo está atribuindo a classe correta aos objetos detectados. É uma métrica crucial para avaliar a capacidade do modelo de distinguir entre diferentes tipos de objetos.

9. **val/box_loss**: refere-se à perda de caixa delimitadora (*Bounding Box Loss*) durante a validação. A perda de caixa delimitadora avalia a precisão das coordenadas das caixas delimitadoras previstas pelo modelo em comparação com as caixas delimitadoras reais dos objetos nas imagens.

10. **GFLOPs** (*Giga Floating Point Operations per Second*): é uma métrica que mede o desempenho de um modelo de aprendizado de máquina em termos de operações de ponto flutuante por segundo. Quando falamos em GFLOPs, estamos nos referindo a bilhões de operações de ponto flutuante por segundo.

11. **Model parameters:** são variáveis internas de um modelo de aprendizado de máquina que são ajustadas durante o processo de treinamento. Eles são essenciais para definir como o modelo transforma os dados de entrada em previsões. São valores que o modelo aprende diretamente dos dados durante o treinamento. Eles são ajustados para minimizar a função de perda e melhorar a precisão do modelo.

4.2.3 Otimizadores

Os otimizadores ajudam a minimizar a função de perda durante o treinamento da rede neural. A seguir uma visão geral de como os diferentes otimizadores podem ser usados com YOLO.

4.2.3.1 SGD (Stochastic Gradient Descent)

É uma versão do gradiente descendente que atualiza os pesos da rede neural usando apenas um único exemplo de treinamento por vez. Isso ajuda a encontrar o mínimo global mais rápido, mas pode ser instável devido à alta variabilidade nas atualizações. Características principais:

- Simplicidade e eficiência em termos de memória.
- Utilizado em muitas implementações básicas do YOLO devido à sua simplicidade.
- Pode ser combinado com técnicas como momentum para melhorar a convergência.

4.2.3.2 Adam (Adaptive Moment Estimation)

Combina as vantagens de dois outros algoritmos, o RMSProp e o Momentum. Ele calcula médias móveis dos gradientes passados e das suas magnitudes quadráticas, permitindo adaptações mais rápidas e precisas. Adam é amplamente utilizado por sua eficiência e desempenho. Principais características:

- Adaptativo e eficiente, ajustando a taxa de aprendizado com base em momentos passados.
- Amplamente utilizado em redes YOLO modernas por sua capacidade de rápida convergência e estabilidade.

4.2.3.3 AdamW (Adam with Weight Decay)

É uma variação do Adam que incorpora a regularização por decaimento dos pesos diretamente no algoritmo. Principais características:

- Variação do Adam com regularização por decaimento de peso.
- Ajuda a evitar overfitting, promovendo melhor generalização, o que é útil em cenários com conjuntos de dados limitados.

4.2.3.4 RAdam (Rectified Adam)

É uma melhoria sobre o Adam que retifica o problema de variância nas etapas iniciais do treinamento. Ao fazer isso, RAdam ajuda a estabilizar o processo de otimização e melhora a precisão do modelo. Principais características:

- Estabiliza a variância no início do treinamento, melhorando a precisão.
- Pode ser benéfico em treinamentos de redes YOLO para garantir convergência estável e precisa.

4.2.3.5 RMSProp (Root Mean Square Propagation)

Algoritmo adaptativo que ajusta a taxa de aprendizado para cada parâmetro. Ele divide a taxa de aprendizado pelo valor médio dos gradientes recentes, permitindo que o algoritmo se adapte dinamicamente às variações no dado. Principais características:

- Ajusta a taxa de aprendizado com base na magnitude recente dos gradientes.
- Pode ser usado para melhorar a adaptação em diferentes fases do treinamento, especialmente em redes YOLO que lidam com dados variados.

4.2.3.6 Nadam (Nesterov-accelerated Adaptive Moment Estimation)

É uma variante do Adam que combina Nesterov Momentum com Adam. Isso acelera a convergência, especialmente em problemas com mínimos locais complexos. Nadam é conhecido por ser mais rápido e eficiente em certos cenários. Principais características:

- Combina as vantagens do Adam com Nesterov Momentum para aceleração adicional.
- Ajuda a alcançar uma convergência mais rápida, o que é crucial para treinar redes YOLO em grandes conjuntos de dados.

4.2.3.7 Resumo das Características dos Algoritmos Otimizadores

Cada um dos otimizadores apresentado tem suas próprias vantagens e é escolhido com base nos requisitos específicos do problema e nas características do conjunto de dados.

Tabela 4. Comparativo dos Otimizadores

Algoritmo	Características Principais	Vantagens	Desvantagens
SGD	Atualização por exemplo único	Simples e rápido	Instável, pode ser lento
Adam	Combina RMSProp e Momentum	Eficiente, bom desempenho	Pode superestimar gradientes
AdamW	Adam com regularização por decaimento de pesos	Melhor generalização	Similar ao Adam
RAdam	Retificação de variância no início	Mais estável, melhora a precisão	Pode ser mais complexo
RMSProp	Ajuste dinâmico da taxa de aprendizado	Adaptação rápida às variações	Pode não convergir sempre
NAdam	Combina Nesterov Momentum com Adam	Convergência mais rápida	Complexidade adicional

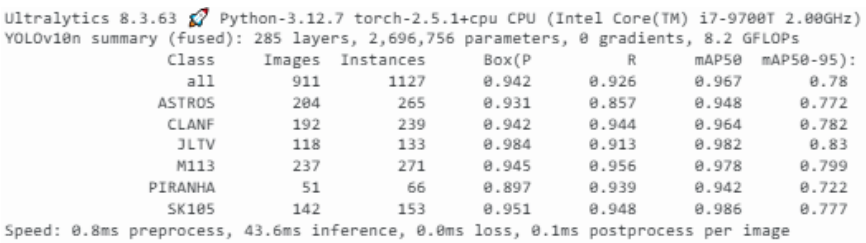
4.3. Treinos

4.3.1 Treino de Teste Comparativo GPU x CPU

Inicialmente, fizemos um treino comparativo de 120 épocas com os mesmos dados de treino/validação usando uma GPU Tesla V100-PCIE-32GB com 32768.0 MB Memória (no Ambiente de Pesquisa *On Premise* da FGV) e uma CPU Intel i7 de 10ª geração com 8 núcleos, 32Gb RAM e HD SSD de 1TB.

Apesar de resultados finais semelhantes, usar uma GPU diminuiu consideravelmente o tempo necessário de processamento comparado a uma CPU (de 39h para apenas 49m) conforme imagens abaixo:

Figura 6. Teste Pré-Treino com 120 Épocas - Sem GPU



```
Ultralytics 8.3.63 🚀 Python-3.12.7 torch-2.5.1+cpu CPU (Intel Core(TM) i7-9700T 2.00GHz)
YOLOv10n summary (fused): 285 layers, 2,696,756 parameters, 0 gradients, 8.2 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95):
                   all        911       1127      0.942      0.926      0.967       0.78
                ASTROS        204        265      0.931      0.857      0.948      0.772
                 CLANF        192        239      0.942      0.944      0.964      0.782
                  JLTV        118        133      0.984      0.913      0.982       0.83
                  M113        237        271      0.945      0.956      0.978      0.799
               PIRANHA         51         66      0.897      0.939      0.942      0.722
                 SK105        142        153      0.951      0.948      0.986      0.777
Speed: 0.8ms preprocess, 43.6ms inference, 0.0ms loss, 0.1ms postprocess per image
```

Figura 7 - Teste Pré-Treino com 120 Épocas - Com GPU

```
Ultralytics 8.3.66 🚀 Python-3.10.12 torch-2.5.1+cu124 CUDA:0 (Tesla V100-PCIE-32GB, 32494MiB)
YOLOv10n summary (fused): 285 layers, 2,696,756 parameters, 0 gradients, 8.2 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95):
                   all        911       1127      0.943      0.927      0.969      0.802
                ASTROS        204        265      0.916      0.857      0.954      0.794
                 CLANF        192        239      0.946      0.946      0.967      0.791
                  JLTV        118        133      0.968      0.907      0.974      0.826
                  M113        237        271      0.948      0.967      0.983      0.816
              PIRANHA         51         66      0.913      0.939      0.955      0.766
                 SK105        142        153       0.97      0.948      0.983      0.819
Speed: 0.1ms preprocess, 0.6ms inference, 0.0ms loss, 0.1ms postprocess per image
```

Figura 8. Labels

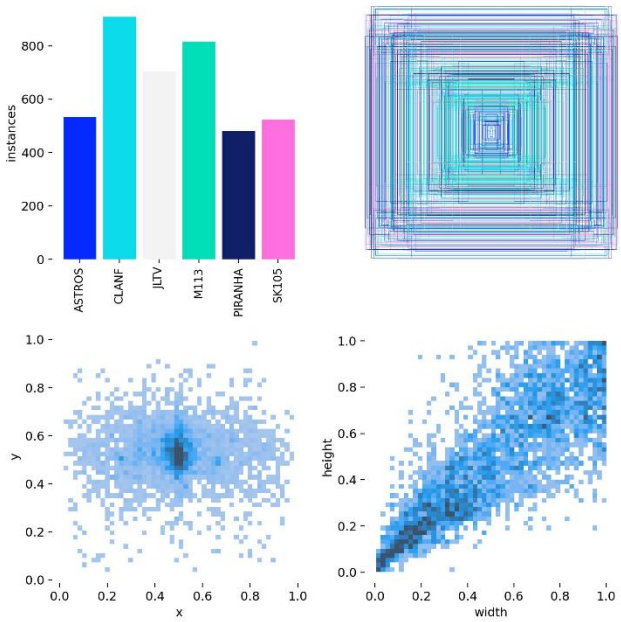


Figura 9. Matriz de Confusão Normalizada

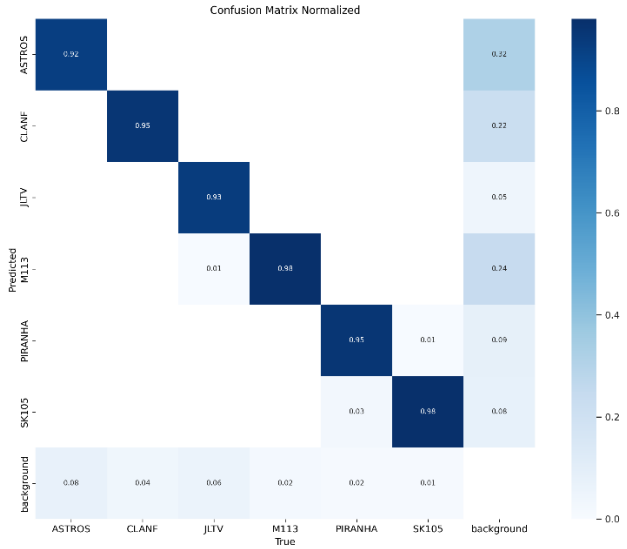


Figura 10. Comparativo (GPUxCPU) – Curvas

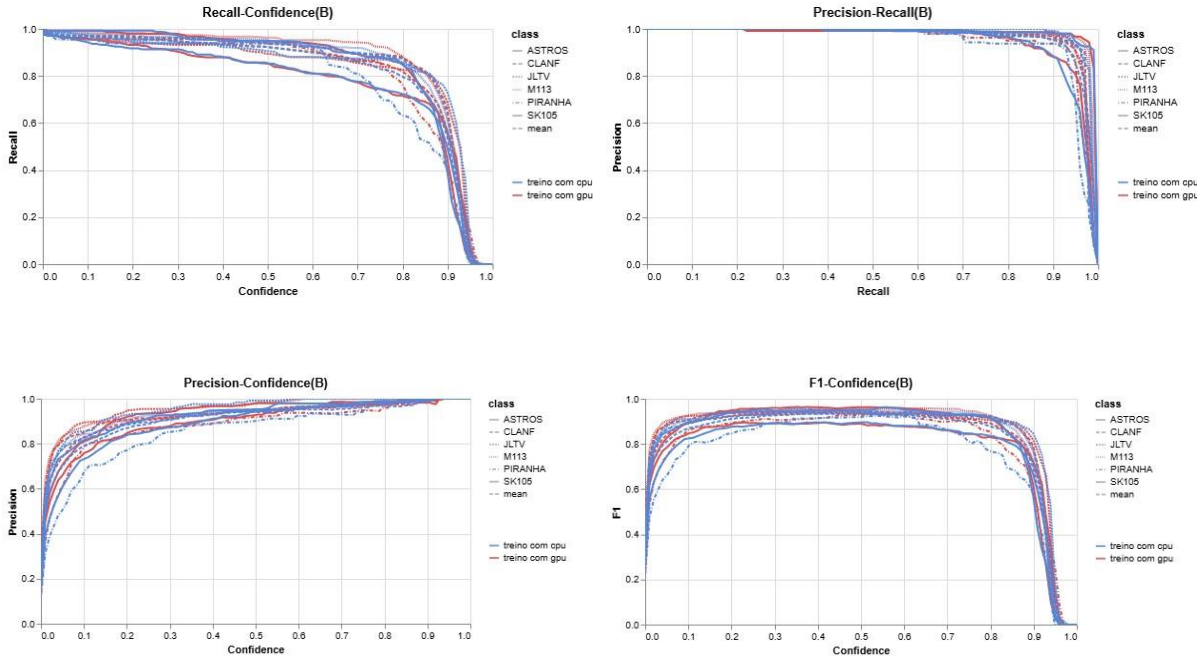


Figura 11. Comparativo - Metricas (GPUxCPU)

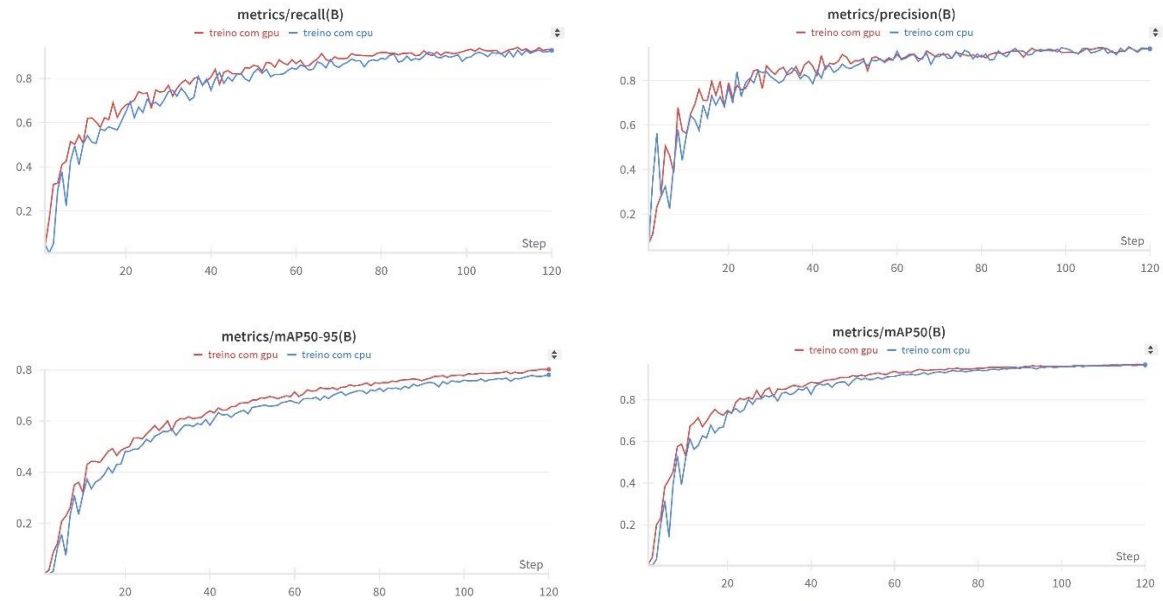


Figura 12. Comparativo Modelo (GFLOPs, parameters e speed)

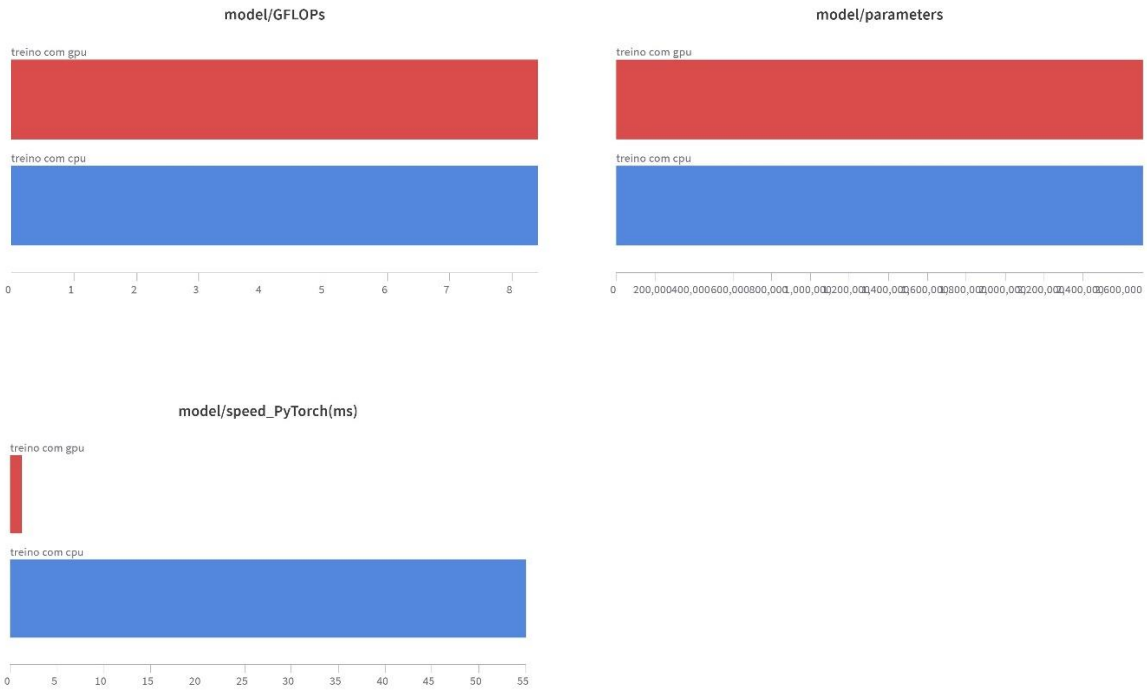


Figura 13. Comparativo Treino (dfl_loss, cls_loss e box_loss)

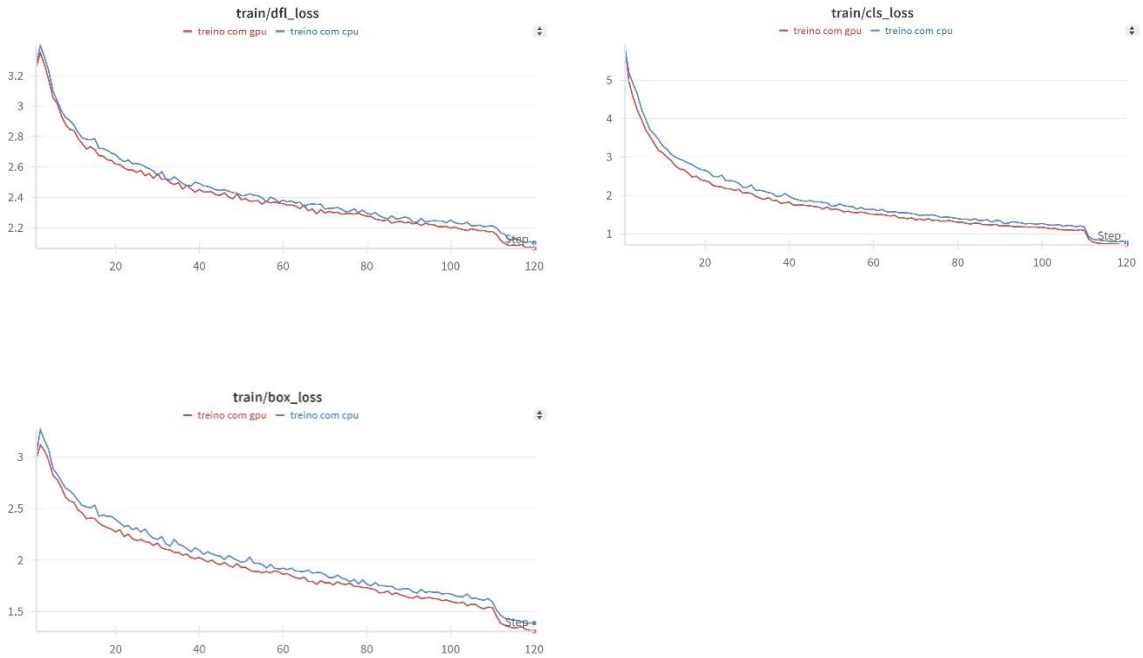


Figura 14. Comparativo Validação (dfl_loss, cls_loss e box_loss)

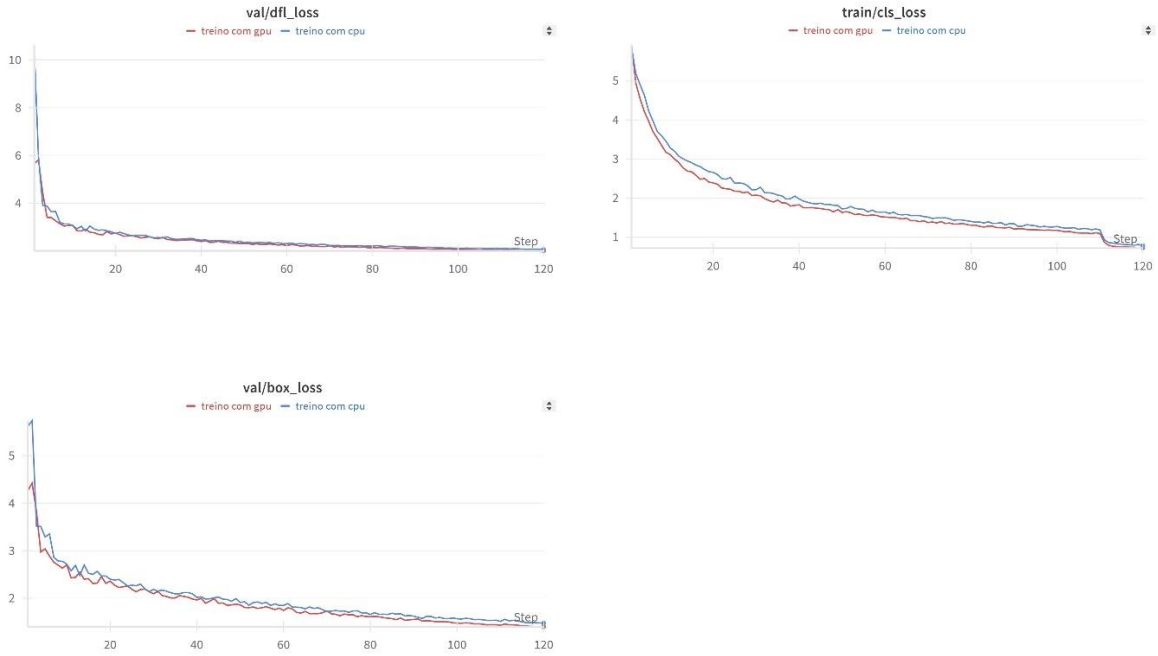
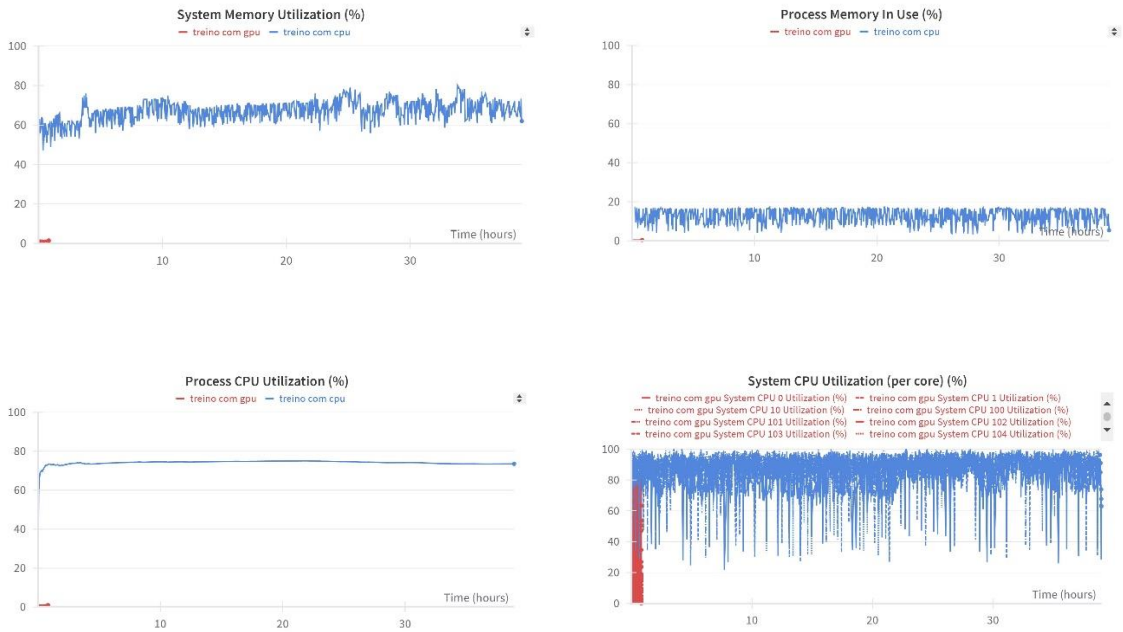


Figura 15. Uso do Sistema Durante Treino (memória, cpu, disco e gpu)



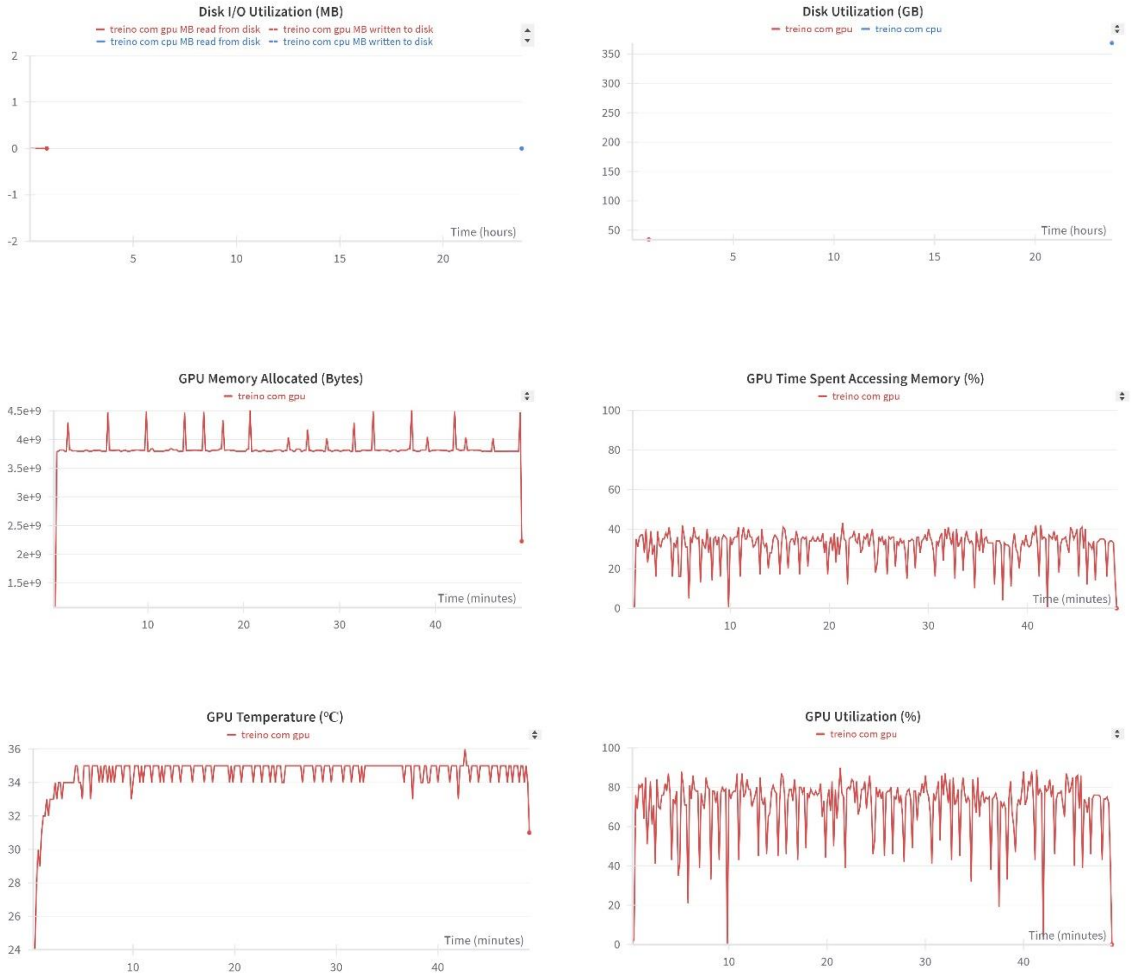


Figura 16. Exemplos de Predições de Teste Com o Modelo Treinado


JLTV 0.9
ASTROS 0.9
ASTROS 0.4
CLANF 0.9
CLANF 0.9
CLANF 0.9
M113 0.9
ASTROS 1.0
CLANF 0.9
M113 0.7
ASTROS 0.9
PIRANHA 0.4
PIRANHA 0.8
JLTV 0.9
SK105 0.9
CLANF 0.6
ASTROS 0.9
M113 0.8


JLTV 0.9
CLANF 0.9
CLANF 0.9
CLANF 0.9
M113 0.8
SK105 0.9
M113 0.9
JLTV 0.9
M113 0.9
ASTROS 0.9
ASTROS 0.3
SK105 0.7
CLANF 0.9
CLANF 0.9
SK105 0.4
ASTROS 0.9
SK105 0.9

4.3.2 Otimizadores em Treinos Isolados

Para o comparativo, fizemos treinos individuais substituindo o otimizador e mantendo as demais configurações (limitando para o máximo de 600 épocas e interrompendo o treino caso passe 10 épocas sem evolução do modelo).

Tabela 5. Resumo do treino (entre otimizadores)

Otimizador	Convergiu?	Tempo (horas)	Epocas processadas	Melhor Resultado (época)
Adam	SIM	1,473	125	115
AdamW	SIM	2,861	286	276
RAdam	SIM	0,826	83	73
RMSProp	SIM	0,205	20	10
NAdam	SIM	3,060	290	280
SGD	SIM	1,732	179	169

4.3.2.1 Adam

Treino: 1,473h/125 Épocas

Convergência: Época 115

Figura 17. Resumo (Adam)

```
Ultralytics 8.3.66 🚀 Python-3.10.12 torch-2.5.1+cu124 CUDA:0 (Tesla V100-PCIE-32GB, 32494MiB)
YOLOv10n summary (fused): 285 layers, 2,696,756 parameters, 0 gradients, 8.2 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%
                   all        911       1127       0.88       0.89      0.927      0.689
               ASTROS         204        265      0.844      0.868      0.903       0.68
                 CLANF        192        239       0.88      0.921      0.941      0.724
                  JLTV        118        133      0.972      0.857      0.948      0.715
                  M113        237        271      0.916      0.923       0.96      0.713
               PIRANHA         51         66      0.787      0.864      0.885      0.591
                 SK105        142        153      0.881      0.908      0.926      0.709
Speed: 0.1ms preprocess, 0.6ms inference, 0.0ms loss, 0.1ms postprocess per image
```

Figura 18. Curvas (Adam)

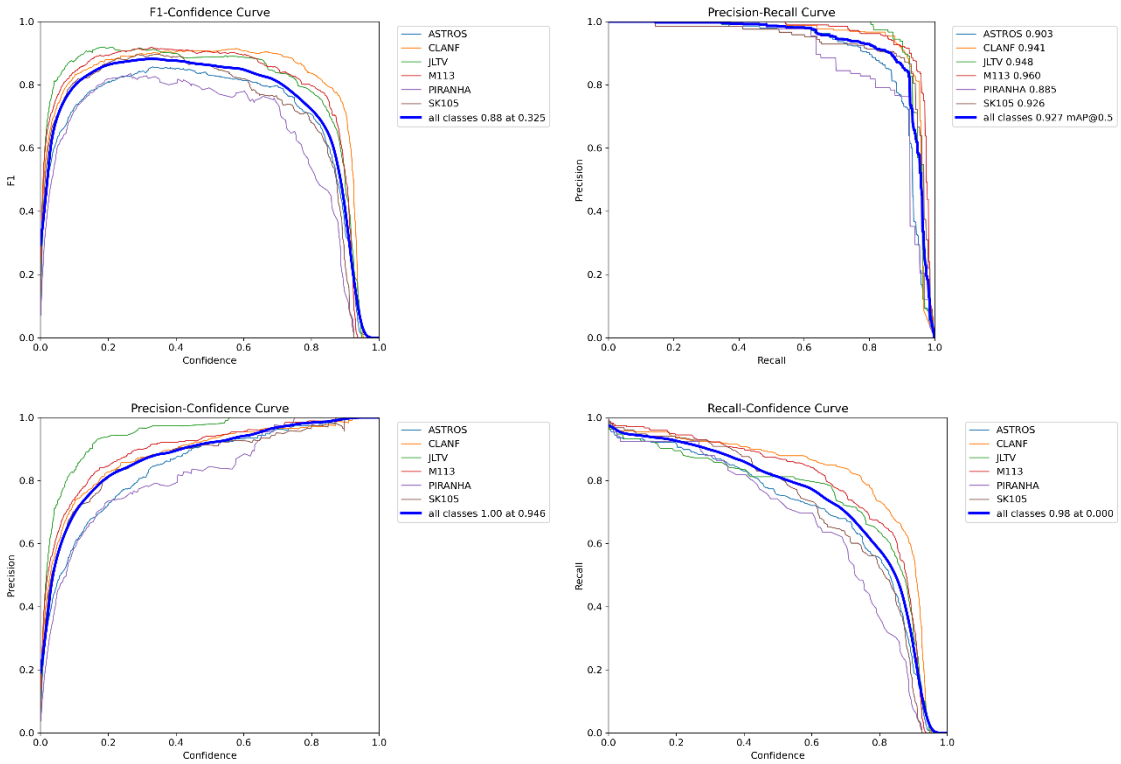


Figura 19. Treino/Validação (Adam)

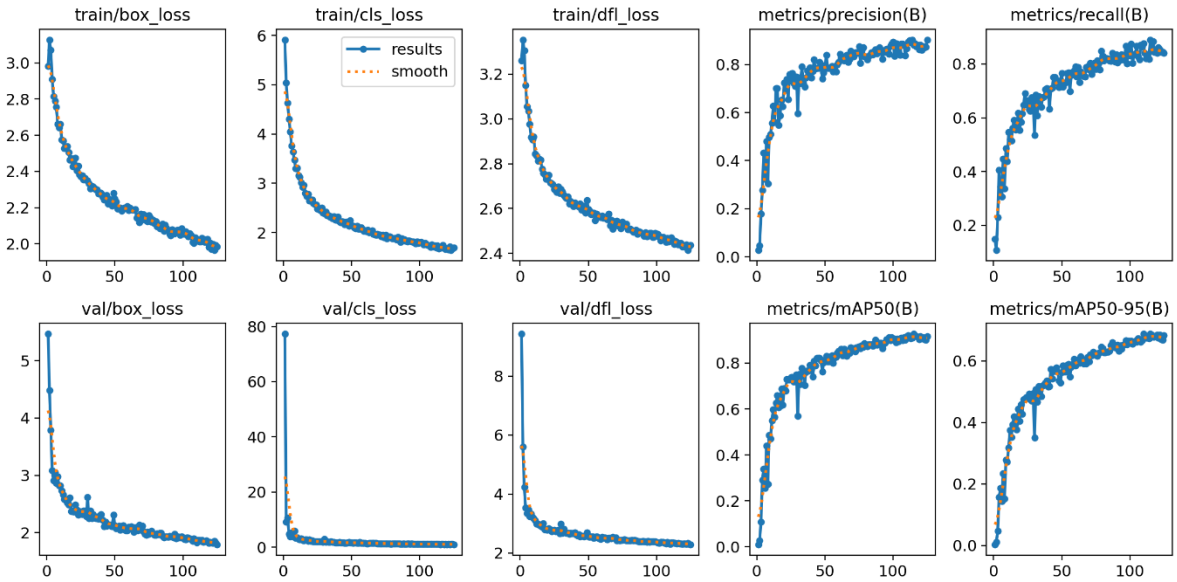


Figura 20. Predições (Adam)

4.3.2.2 AdamW

Treino: 2,861h/286 Épocas

Convergência: Época 276

Figura 21. Resumo (AdamW)

```
Ultralytics 8.3.66 🚀 Python-3.10.12 torch-2.5.1+cu124 CUDA:0 (Tesla V100-PCIE-32GB, 32494MiB)
YOLOv10n summary (fused): 285 layers, 2,696,756 parameters, 0 gradients, 8.2 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%
                   all        911       1127      0.956      0.959      0.983      0.869
                ASTROS        204        265      0.945      0.912      0.973      0.862
                 CLANF        192        239       0.95      0.953      0.979      0.835
                  JLTV        118        133      0.969      0.962      0.989      0.898
                  M113        237        271      0.955       0.97      0.987      0.878
               PIRANHA         51         66      0.938       0.97      0.979      0.864
                 SK105        142        153      0.981      0.988      0.993      0.875
Speed: 0.1ms preprocess, 0.7ms inference, 0.0ms loss, 0.1ms postprocess per image
```

Figura 22. Curvas (AdamW)

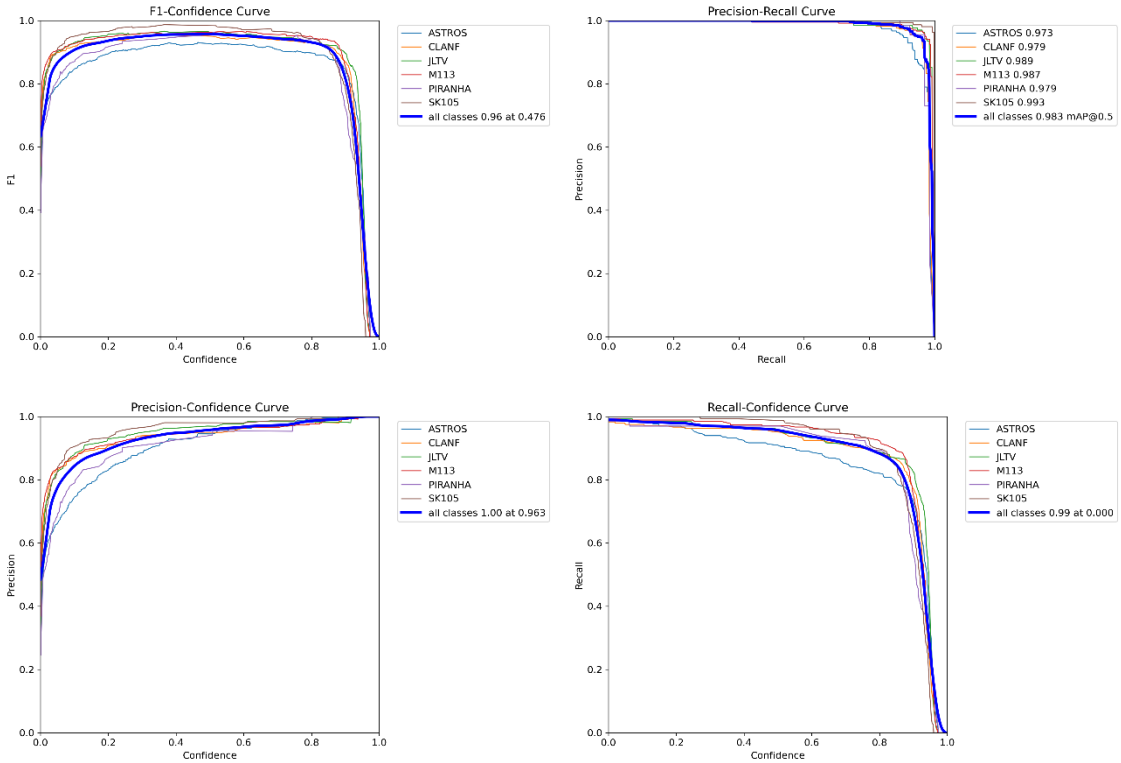


Figura 23. Treino/Validação (AdamW)

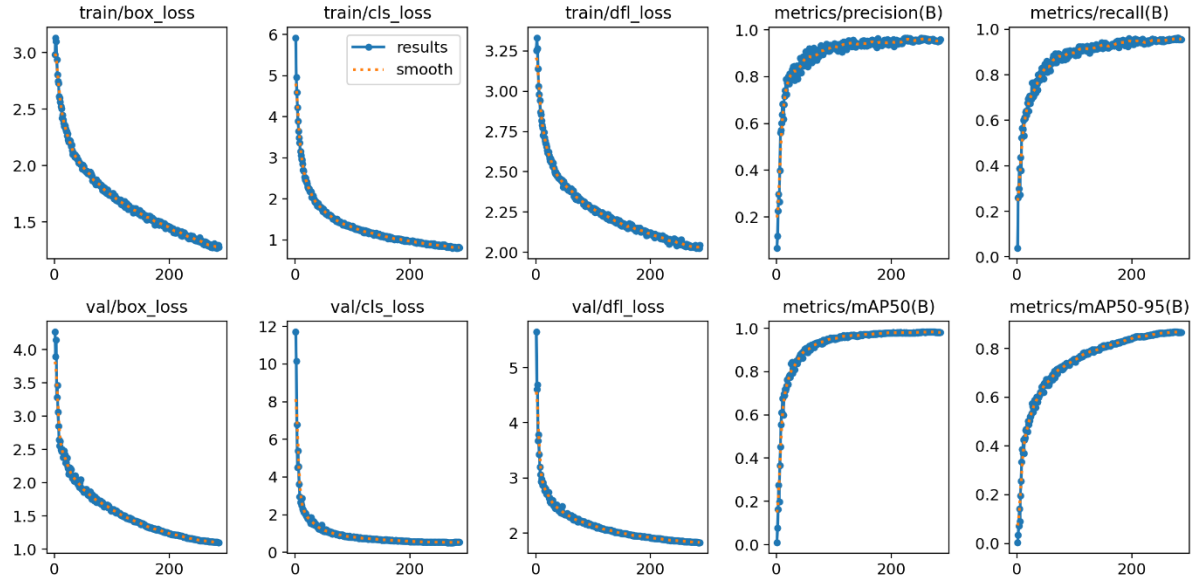


Figura 24. Predições (AdamW)

4.3.2.3 RAdam

Treino: 0,826h/83 Épocas

Convergência: Época 73

Figura 25. Resumo (Radam)

```
Ultralytics 8.3.66 🚀 Python-3.10.12 torch-2.5.1+cu124 CUDA:0 (Tesla V100-PCIE-32GB, 32494MiB)
YOLOv10n summary (fused): 285 layers, 2,696,756 parameters, 0 gradients, 8.2 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%
                   all        911       1127      0.868      0.818      0.894      0.652
                ASTROS        204        265      0.815      0.797      0.863      0.612
                 CLANF        192        239      0.909      0.908      0.931      0.711
                  JLTV        118        133       0.93      0.812      0.906      0.666
                  M113        237        271      0.901      0.877      0.934      0.692
               PIRANHA         51         66      0.812       0.59      0.787      0.533
                 SK105        142        153      0.844      0.922      0.944      0.697
Speed: 0.1ms preprocess, 0.7ms inference, 0.0ms loss, 0.2ms postprocess per image
```

Figura 26. Curvas (RAdam)

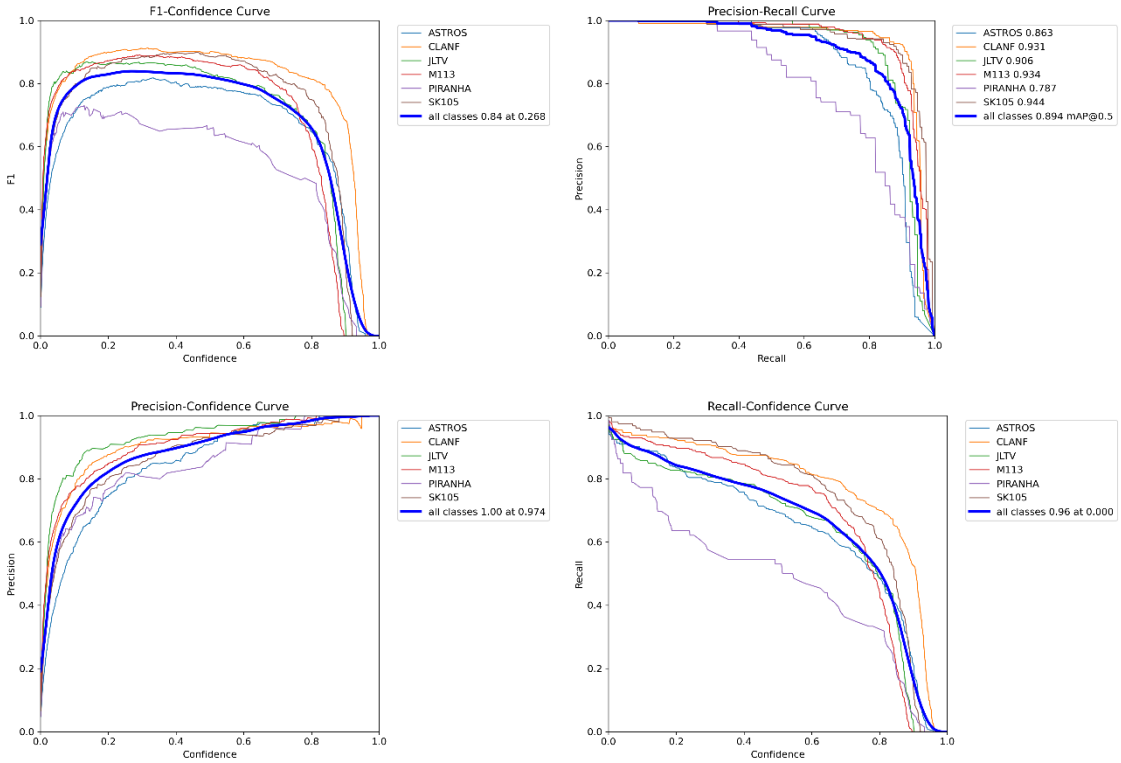


Figura 27. Treino/Validação (RAdam)

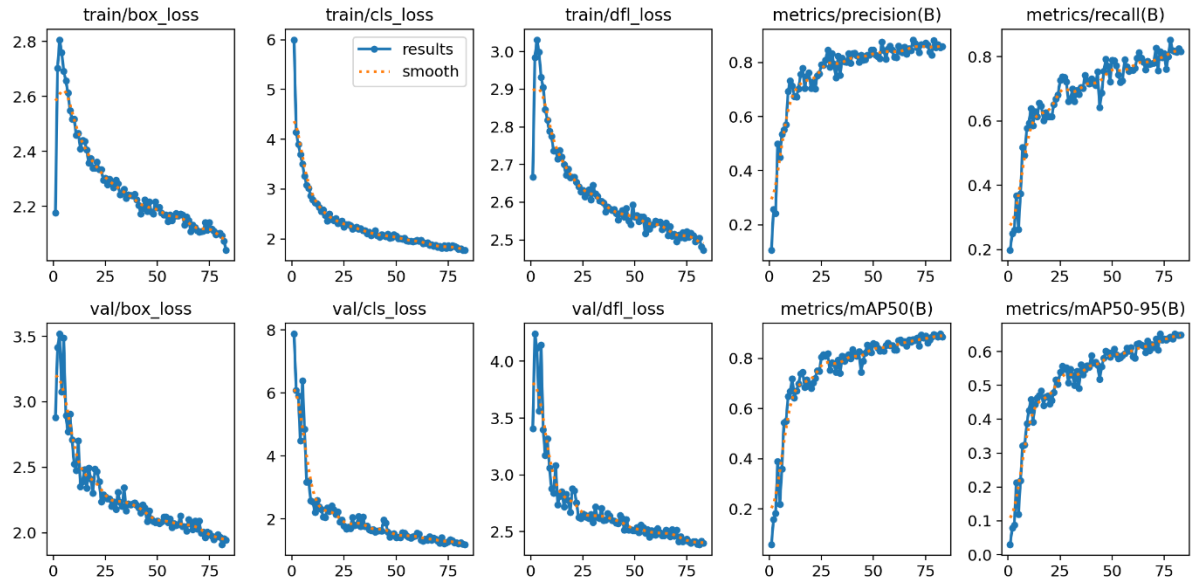


Figura 28. Predições (RAdam)

4.3.2.4 RMSProp

Treino: 0,697h/72 Épocas

Convergência: Época 42

Figura 29. Resumo (RMSProp)

```
Ultralytics 8.3.66 🚀 Python-3.10.12 torch-2.5.1+cu124 CUDA:0 (Tesla V100-PCIE-32GB, 32494MiB)
YOLOv10n summary (fused): 285 layers, 2,696,756 parameters, 0 gradients, 8.2 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
                   all        911       1127     0.0437      0.133     0.0429     0.0123
                ASTROS        204        265     0.0669     0.0113     0.0415     0.0121
                 CLANF        192        239   0.000639      0.339    0.00297   0.000611
                  JLTV        118        133     0.0154     0.0226    0.00787    0.00138
                  M113        237        271     0.0587     0.0923     0.0355     0.0107
               PIRANHA         51         66     0.0101     0.0152    0.00879   0.000879
                 SK105        142        153      0.111       0.32      0.161     0.0483
Speed: 0.1ms preprocess, 0.6ms inference, 0.0ms loss, 0.1ms postprocess per image
```

Os números e as imagens a seguir confirmam que este foi o treino mais destoante. Para essa situação especificamente, talvez essa não seja a melhor opção de otimizador.

Figura 30. Curvas (RMSProp)

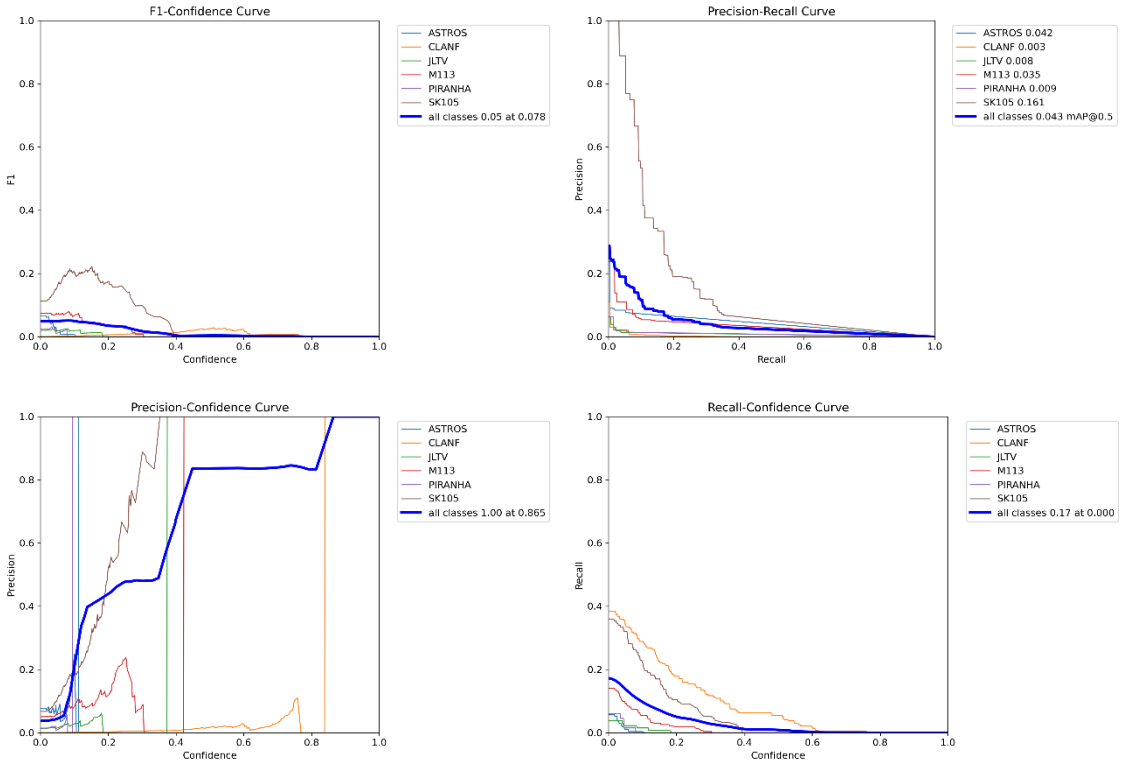


Figura 31. Treino/Validação (RMSProp)

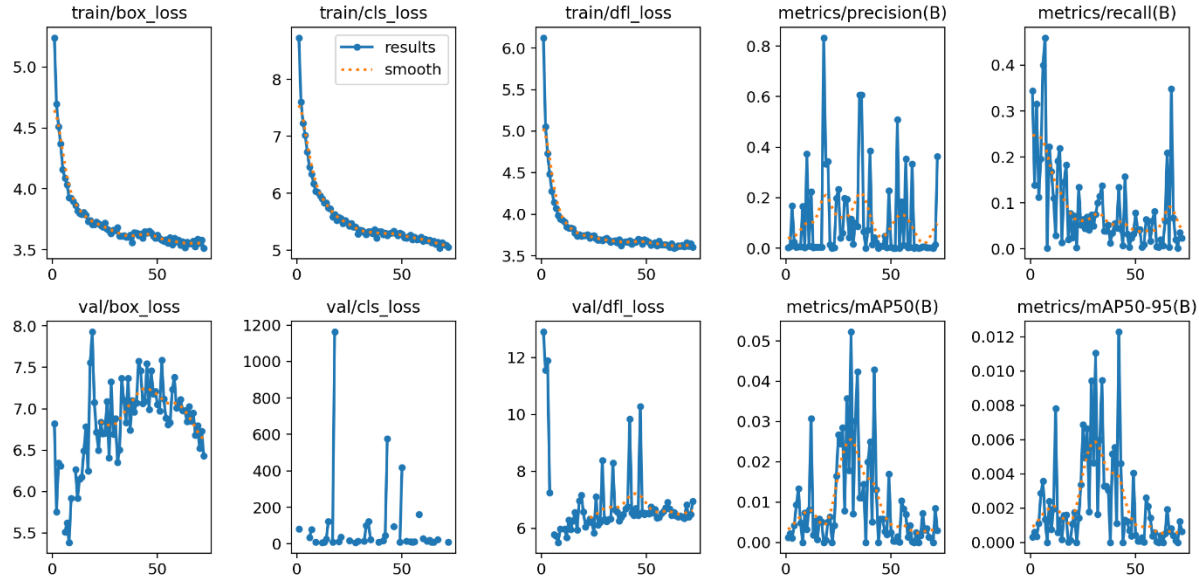


Figura 32. Predição (RMSProp)

4.3.2.5 NAdam

Treino: 3,060h/290 Épocas

Convergência: Época 280

Figura 33 - Resumo Treino Otimizadores - NAdam

```
Ultralytics 8.3.66 🚀 Python-3.10.12 torch-2.5.1+cu124 CUDA:0 (Tesla V100-PCIE-32GB, 32494MiB)
YOLOv10n summary (fused): 285 layers, 2,696,756 parameters, 0 gradients, 8.2 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%
                   all        911       1127      0.957      0.922      0.974      0.819
               ASTROS         204        265      0.963      0.875      0.965      0.803
                CLANF         192        239      0.949      0.929      0.969      0.804
                 JLTV         118        133      0.989      0.925      0.985      0.846
                 M113         237        271      0.956      0.962      0.986      0.838
              PIRANHA          51         66      0.913      0.894      0.954      0.782
                SK105         142        153      0.973      0.947      0.985      0.839
```

Figura 34. Curvas (NAdam)

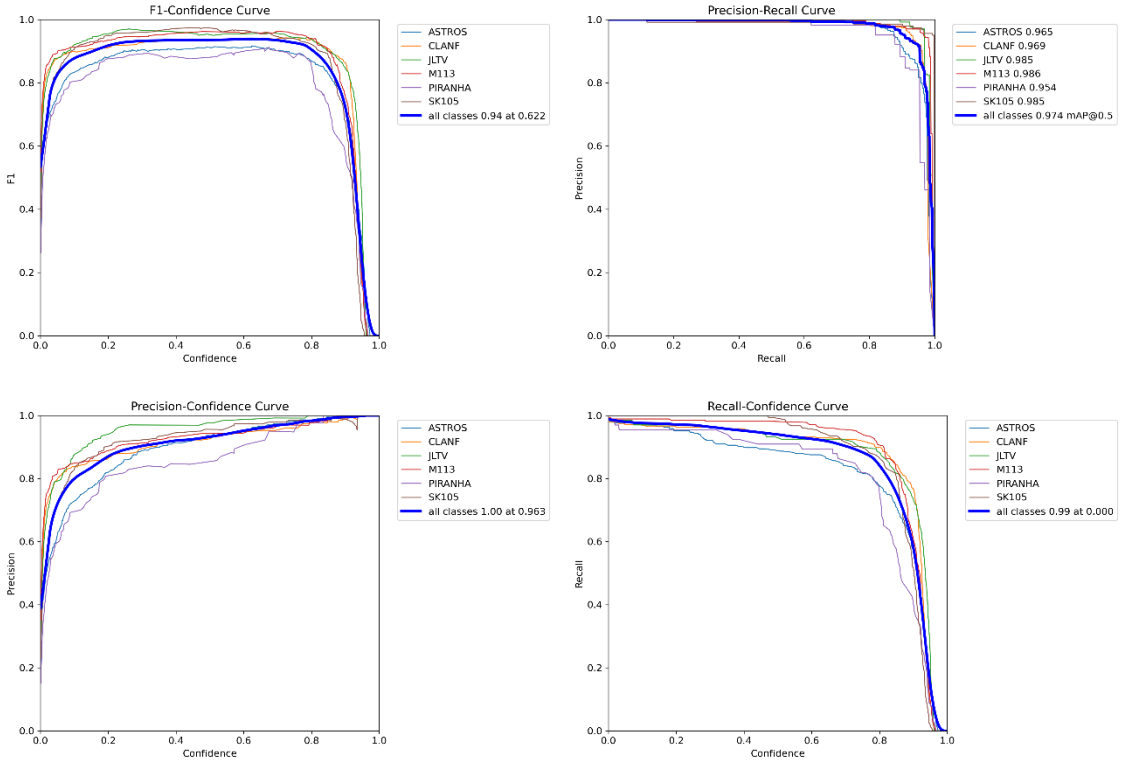


Figura 35. Treino/Validação (NAdam)

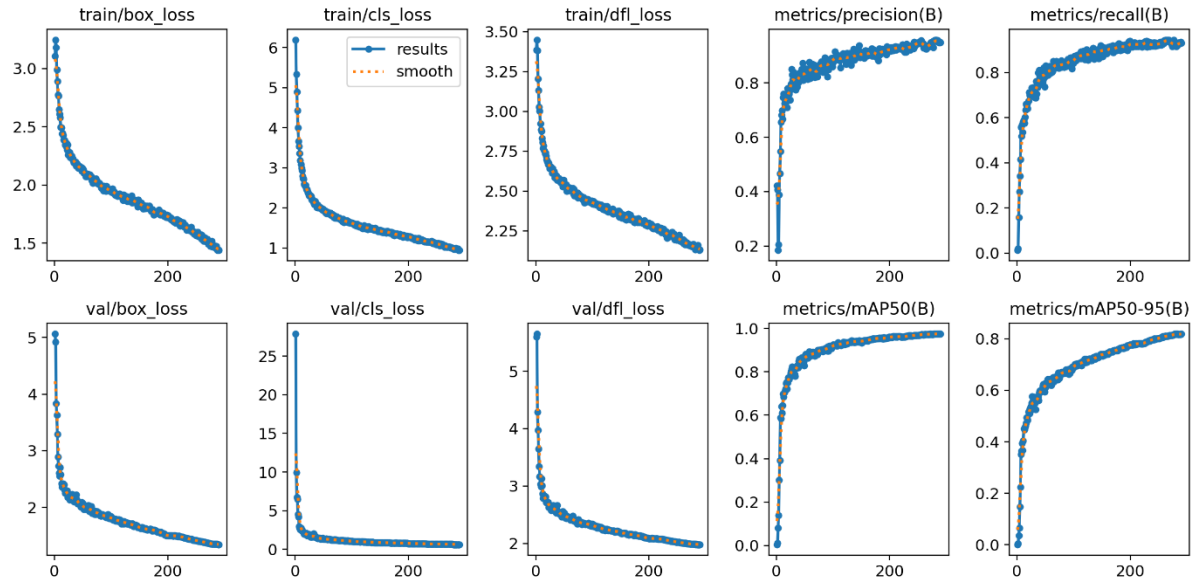


Figura 36. Predições (NAdam)

4.3.2.6 SGD

Treino: 1,732h/179 Épocas

Convergência: Época 169

Figura 37 - Resumo Treino Otimizadores - SGD

```
Ultralytics 8.3.66 🚀 Python-3.10.12 torch-2.5.1+cu124 CUDA:0 (Tesla V100-PCIE-32GB, 32494MiB)
YOLOv10n summary (fused): 285 layers, 2,696,756 parameters, 0 gradients, 8.2 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%
                   all        911       1127      0.959      0.956      0.982      0.852
               ASTROS         204        265      0.956      0.906      0.971      0.838
                CLANF         192        239      0.943      0.962      0.972      0.819
                 JLTV         118        133      0.992      0.973      0.994      0.896
                 M113         237        271      0.949      0.967      0.989       0.86
              PIRANHA          51         66      0.937      0.939      0.976       0.83
                SK105         142        153      0.975      0.987       0.99      0.867
Speed: 0.1ms preprocess, 0.7ms inference, 0.0ms loss, 0.4ms postprocess per image
```

Figura 38. Curvas (SGD)

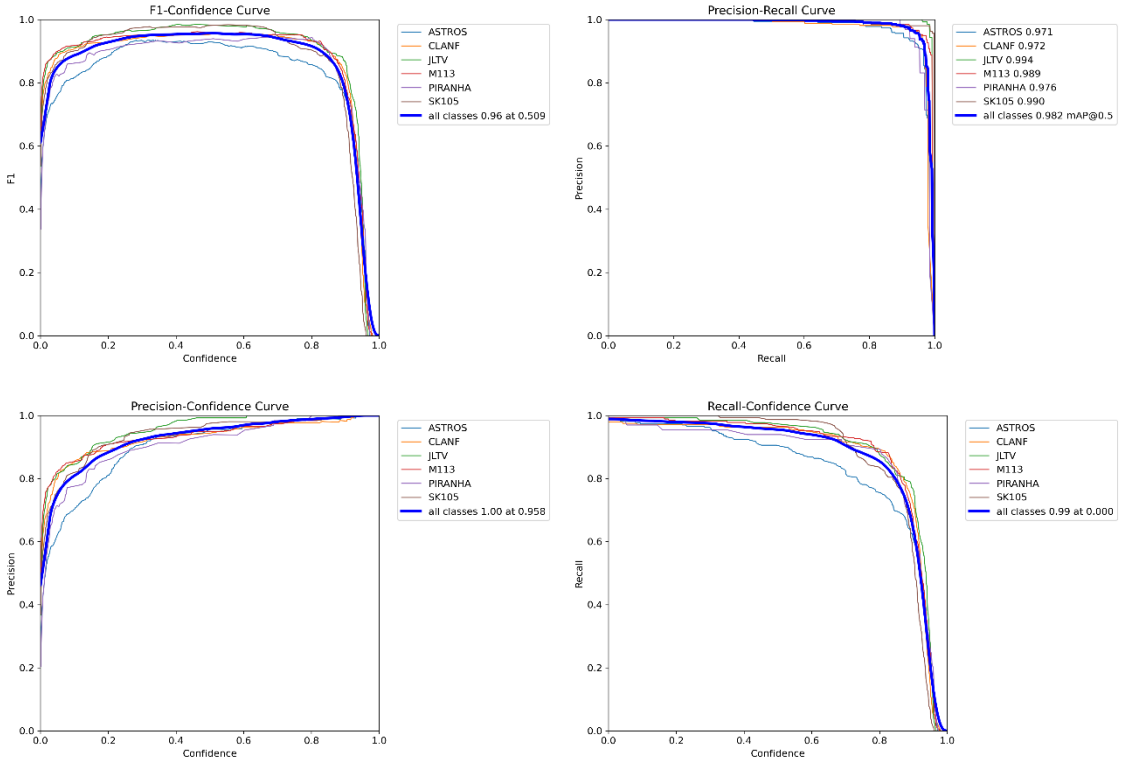


Figura 39. Treino/Validação (SGD)

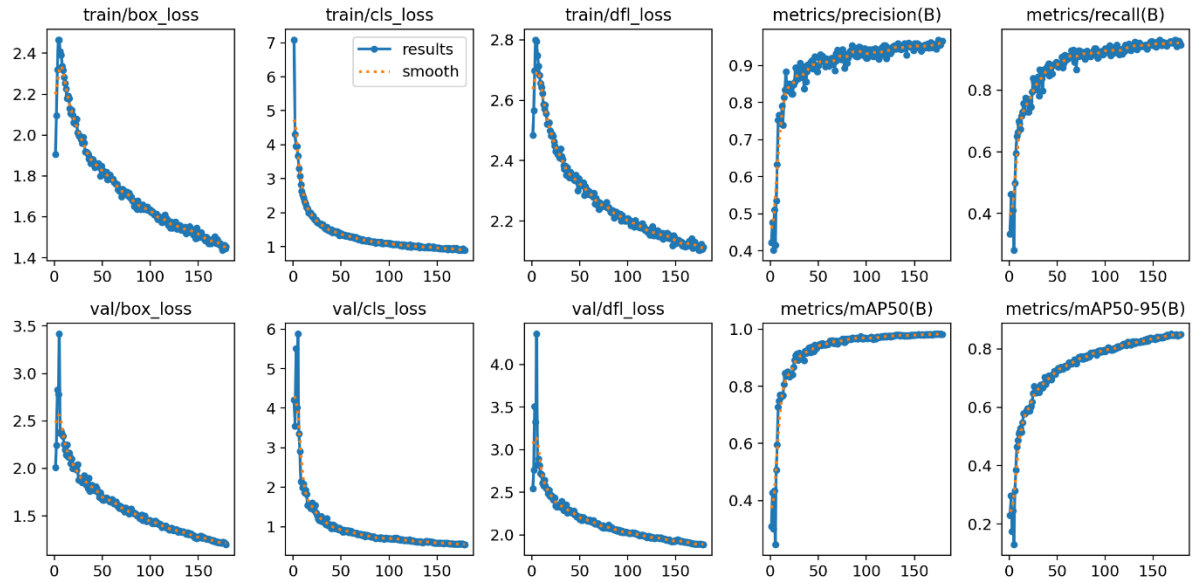


Figura 40. Predições (SGD)

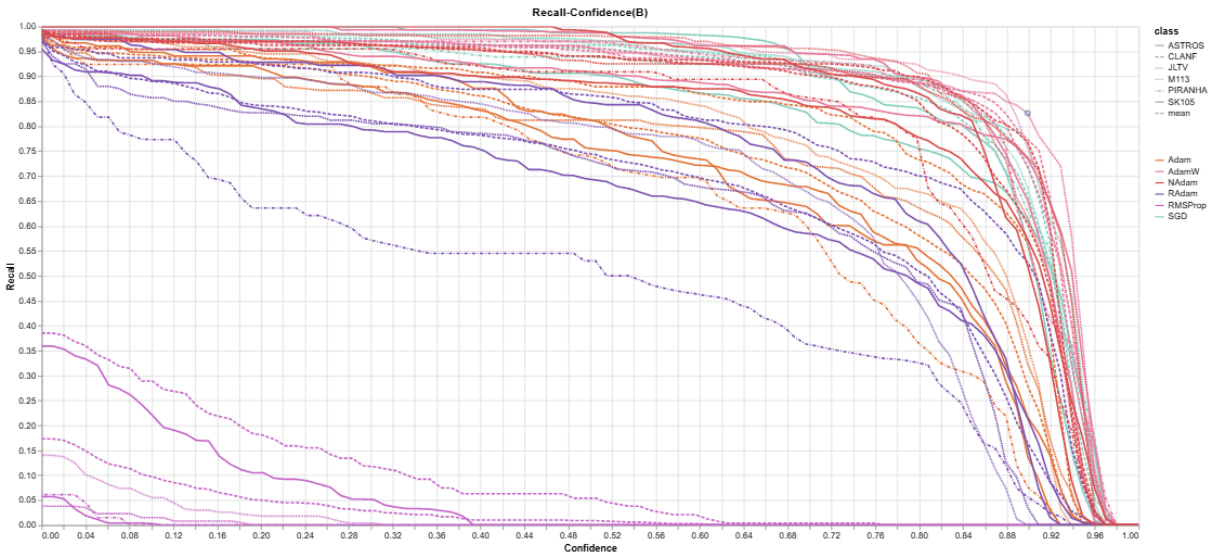
4.4. Resultado Comparativo

Comparamos os resultados obtidos usando os otimizadores Adam, AdamW, NAdam, RAdam, SGD e RMSProp (cada qual com suas particularidades). Os dados dos treinos com RMSProp foram considerados os mais discrepantes e, para o cenário proposto, mostrou-se insatisfatório conforme os dados a seguir e confirmados com os dados de predição.

4.4.1 Recall Confidence

Combina o recall com a confiança associada às previsões. Isso pode ser útil para avaliar não apenas a capacidade do modelo de identificar corretamente as instâncias positivas, mas também a confiança com que faz essas previsões. Por exemplo, um modelo pode ter um recall alto, mas se a confiança nas previsões for baixa, pode não ser tão útil em aplicações práticas.

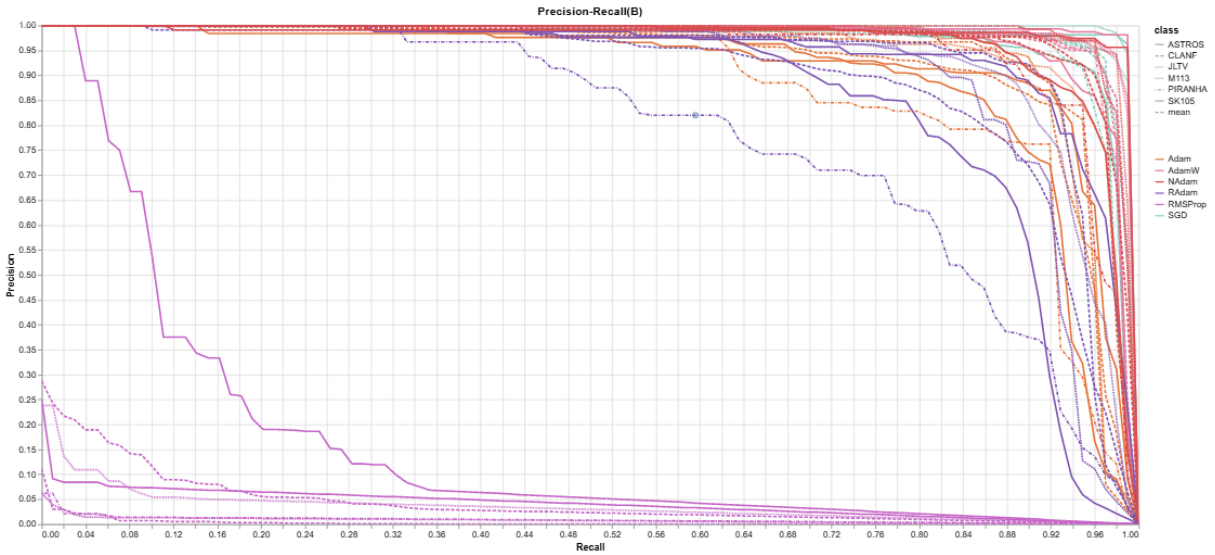
Figura 41. Comparativo - Recall Confidence



4.4.2 Precision Recall

É a proporção de verdadeiros positivos (TP) sobre a soma de verdadeiros positivos e falsos negativos (FN). Ele mede a capacidade do modelo de encontrar todos os exemplos positivos.

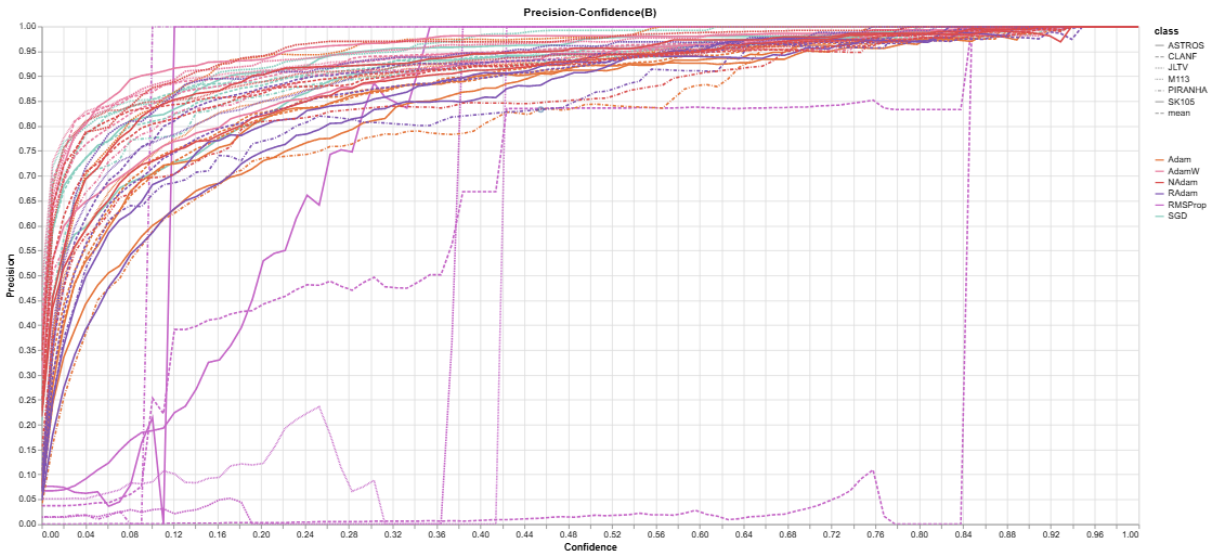
Figura 42. Comparativo - Precision Recall



4.4.3 Precision Confidence

Em muitos modelos de aprendizado de máquina, especialmente em detecção de objetos, cada previsão vem com uma pontuação de confiança que indica a probabilidade de a previsão estar correta.

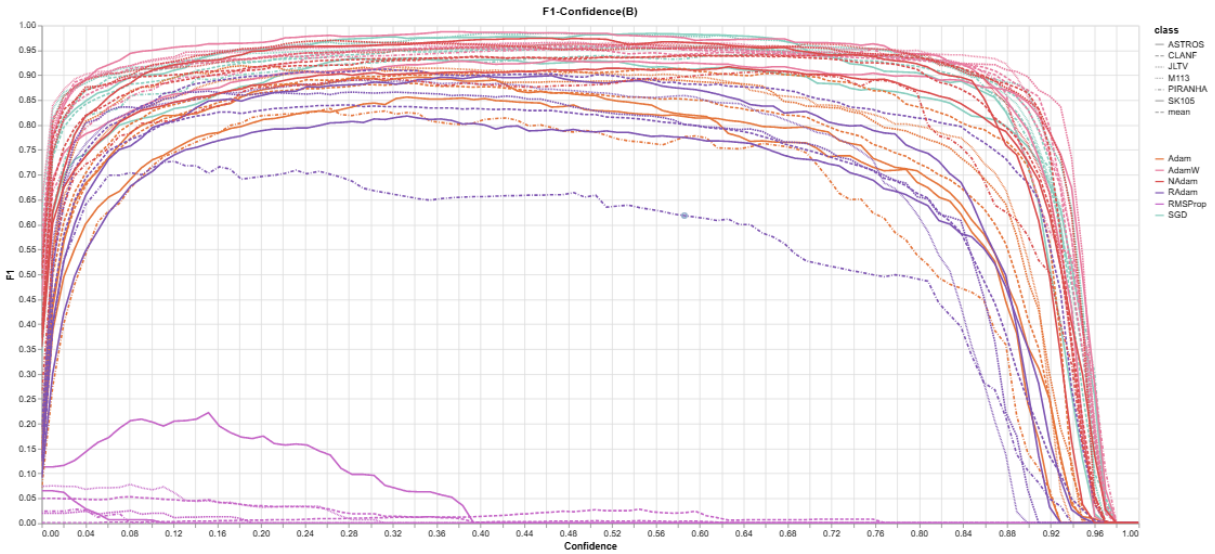
Figura 43. Comparativo - Precision Confidence



4.4.4 F1-Confidence

Combina o F1-Score com a confiança associada às previsões. Isso pode ser útil para avaliar não apenas o equilíbrio entre precisão e recall, mas também a confiança com que o modelo faz essas previsões. Um alto F1-Score com alta confiança indica que o modelo é não apenas equilibrado em termos de precisão e recall, mas também confiante em suas previsões.

Figura 44. Comparativo - F1 Confidence



4.4.5 Lr (Learning Rate)

A taxa de aprendizado determina o tamanho dos passos que o algoritmo de otimização dá ao mover-se em direção ao mínimo da função de perda.

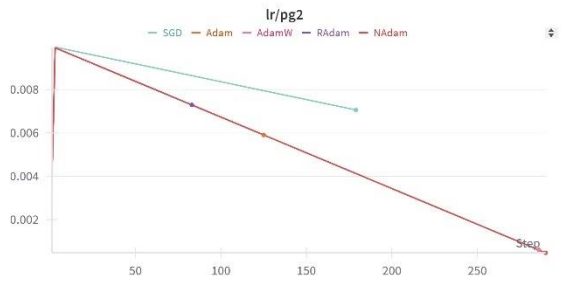
- **Taxa de Aprendizado Alta**: Pode fazer com que o modelo converja rapidamente, mas corre o risco de pular o mínimo global e não convergir adequadamente.
- **Taxa de Aprendizado Baixa**: Pode levar a uma convergência mais estável e precisa, mas o treinamento pode ser muito lento e pode ficar preso em mínimos locais.

Os algoritmos usados ajustam a taxa de aprendizado automaticamente com base no histórico de gradientes.

4.4.5.1 lr/pg2

Este grupo geralmente inclui os vieses (biases) das camadas. Os vieses são ajustados separadamente dos pesos e podem ter uma taxa de aprendizado diferente.

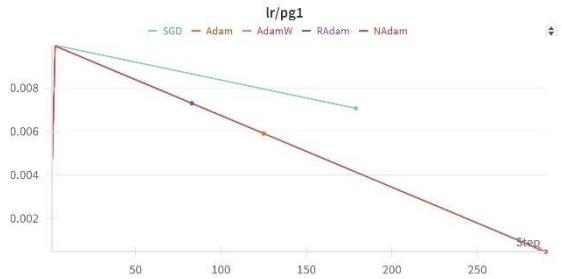
Figura 45. Comparativo - lr/pg2



4.4.5.2 lr/pg1

Este grupo inclui parâmetros que têm decaimento de peso, como os pesos das camadas convolucionais. O decaimento de peso é uma técnica usada para evitar overfitting, penalizando grandes pesos durante o treinamento.

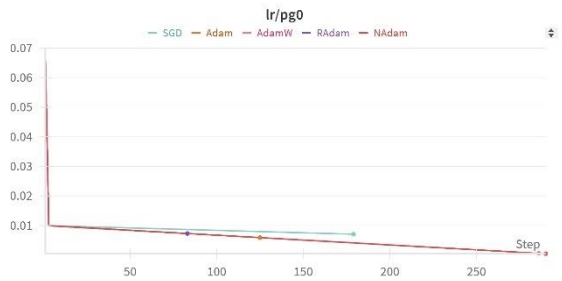
Figura 46. Comparativo - lr/pg1



4.4.5.3 lr/pg0

Este grupo inclui parâmetros que não têm decaimento de peso, como os pesos das camadas de normalização em lote (BatchNorm). A taxa de aprendizado aplicada a este grupo é específica para esses parâmetros.

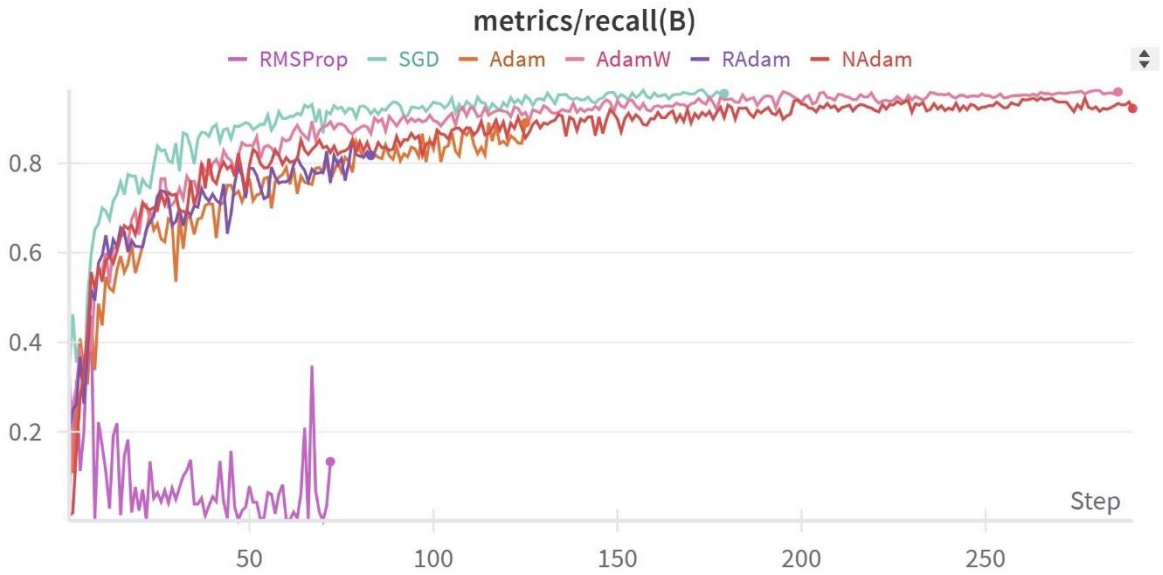
Figura 47. Comparativo - lr/pg0



4.4.6 Metrics/Recall

Recall, também conhecido como sensibilidade ou taxa de detecção, mede a capacidade do modelo de identificar corretamente todas as instâncias relevantes de uma classe. É a proporção de verdadeiros positivos (TP) sobre a soma de verdadeiros positivos e falsos negativos (FN).

Figura 48. Comparativo - metrics/recall

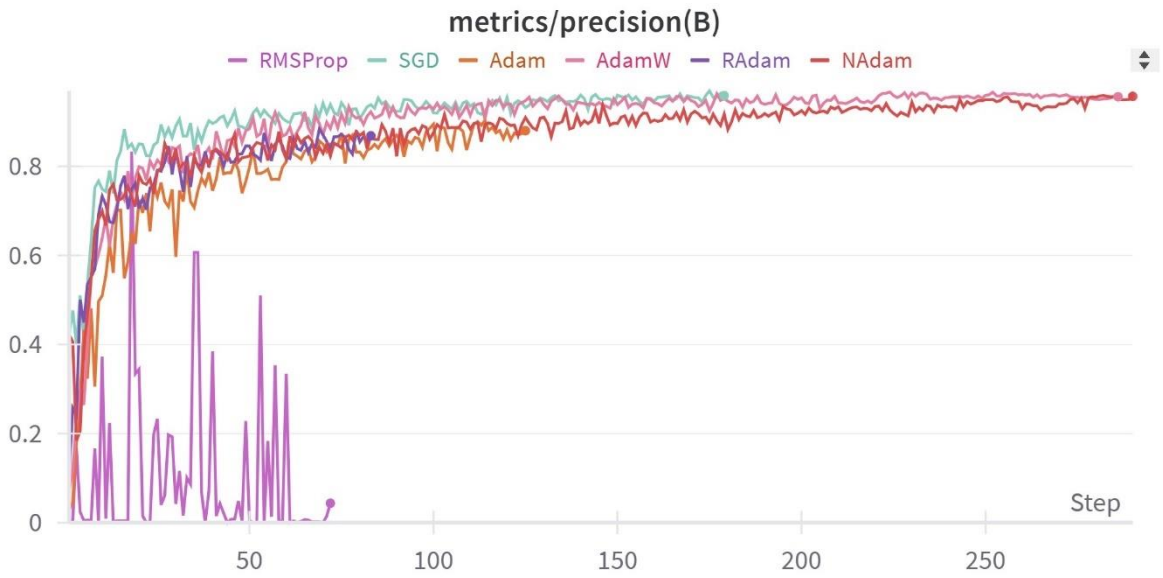


4.4.7 Metrics/precision

Precision, ou precisão, mede a proporção de verdadeiros positivos (TP) sobre o total de previsões positivas (verdadeiros positivos + falsos positivos, FP). Em outras palavras, é a capacidade do modelo de prever corretamente as instâncias positivas.

Há um *trade-off* entre recall e precisão: aumentar o recall pode diminuir a precisão e vice-versa. Por isso, métricas como o F1-Score são usadas para encontrar um equilíbrio entre as duas.

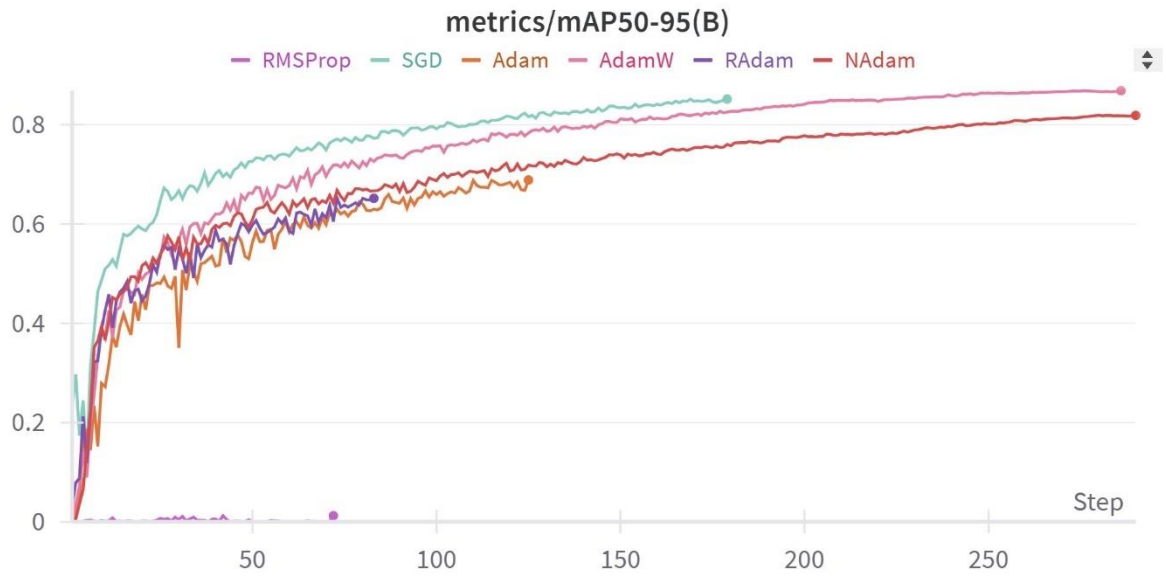
Figura 49. Comparativo - metrics/precision



4.4.8 Metrics/mAP50-95

mAP50-95 é uma métrica mais robusta e informativa do que mAP50 (que considera apenas um limiar de IoU de 0.50). Ela avalia o desempenho do modelo em uma gama de cenários, desde detecções mais fáceis (IoU de 0.50) até detecções mais difíceis (IoU de 0.95).

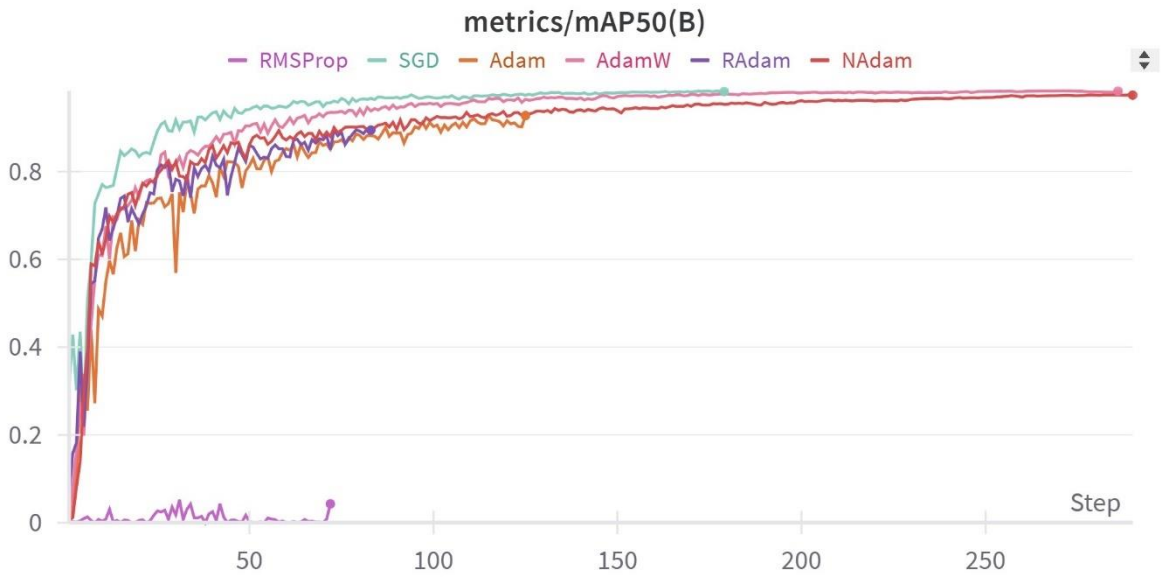
Figura 50. Comparativo - metrics/mAP50-95



4.4.9 Metrics/mAP50

é uma métrica comum usada para avaliar o desempenho de modelos de detecção de objetos, fornecendo uma visão clara de como o modelo se comporta em termos de precisão e recall com um limiar de IoU de 0.50.

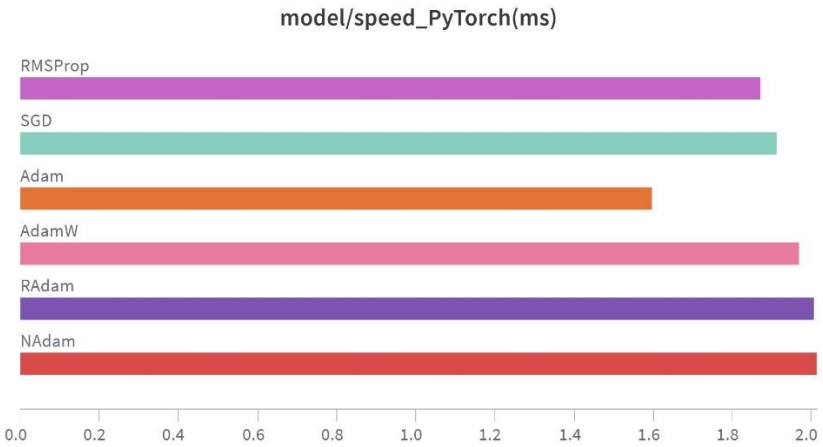
Figura 51. Comparativo - metrics/mAP50



4.4.10 Model/speed(ms)

Tempo que um modelo de aprendizado de máquina leva para processar uma única amostra de entrada, medido em milissegundos (ms). Crucial para avaliar a eficiência e a adequação de um modelo para aplicações em tempo real.

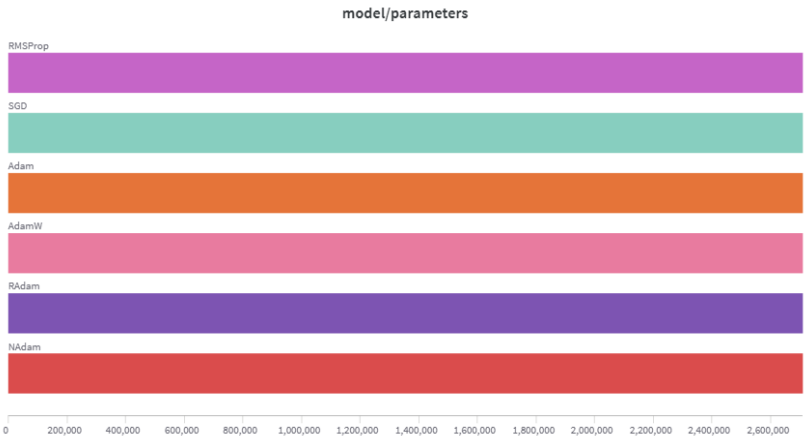
Figura 52. Comparativo - model/speed(ms)



4.4.11 Model/parameters

São variáveis internas de um modelo de aprendizado de máquina que são ajustadas durante o processo de treinamento. Eles são essenciais para definir como o modelo transforma os dados de entrada em previsões. São valores que o modelo aprende diretamente dos dados durante o treinamento. Eles são ajustados para minimizar a função de perda e melhorar a precisão do modelo.

Figura 53. Comparativo - model/parameters

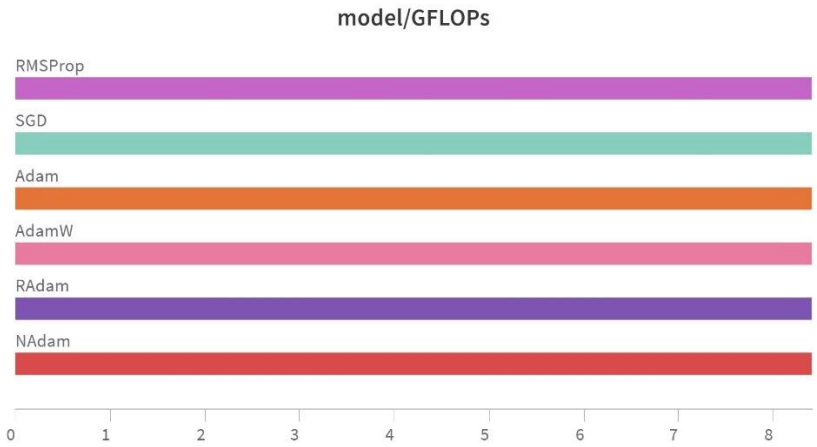


4.4.12 Model/GFLOPs

GFLOPs representa bilhões de operações de ponto flutuante por segundo. É uma medida de quantas operações matemáticas envolvendo números decimais um modelo pode realizar em um segundo. Especialmente em redes neurais profundas, o número de GFLOPs é uma medida importante de eficiência e desempenho. Modelos com maior GFLOPs podem processar dados mais rapidamente, o que é crucial para tarefas que exigem alta capacidade computacional, como visão computacional e processamento de linguagem natural.

Avaliar os GFLOPs de um modelo ajuda a entender sua complexidade e a necessidade de recursos computacionais. Isso é especialmente útil ao comparar diferentes modelos ou ao otimizar modelos para dispositivos com recursos limitados, como smartphones.

Figura 54. Comparativo - model/GFLOPs



4.4.13 Train/dfl_loss

A DFL refere-se a *Distribution Focal Loss* (uma variante da Focal Loss), projetada para focar mais em exemplos difíceis de classificar. Durante o treinamento, a DFL ajuda o modelo a diferenciar melhor entre objetos muito semelhantes ou amostras difíceis, melhorando a capacidade do modelo de lidar com casos complexos.

A DFL é usada em modelos de detecção de objetos, como YOLO, para melhorar a precisão e a robustez do modelo ao lidar com dados desafiadores.

É aplicada na regressão de caixas delimitadoras (bounding boxes), tratando as bordas como distribuições em vez de valores fixos. Isso ajuda a corrigir erros de previsão, melhorando a precisão em imagens com bordas desfocadas ou objetos parcialmente visíveis.

Figura 55. Comparativo - train/dfl_loss

4.4.14 Train/cls_loss

Refere-se à (Perda de Classificação) durante o treinamento de um modelo. A Classification Loss mede a diferença entre as previsões de classe do modelo e as classes reais dos dados de treinamento. É usada para ajustar os pesos do modelo de forma a melhorar a precisão da classificação.

A perda de classificação é calculada usando a ***Cross-Entropy Loss*** (Perda de Entropia Cruzada), cuja fórmula é:

$$H(y,p) = -\sum_{i=1}^{C} Yi \cdot \log\,(p_i)$$

Figura 56. Comparativo - train/cls_loss

4.4.15 Train/box_loss

Refere-se à ***Box Regression Loss*** (Perda de Regressão de Caixa) durante o treinamento de um modelo de detecção de objetos. A Box Regression Loss mede a diferença entre as caixas delimitadoras preditas pelo modelo e as caixas delimitadoras reais dos objetos nos dados de treinamento. Essa perda é usada para ajustar os parâmetros do modelo de forma a melhorar a precisão das previsões das caixas delimitadoras.

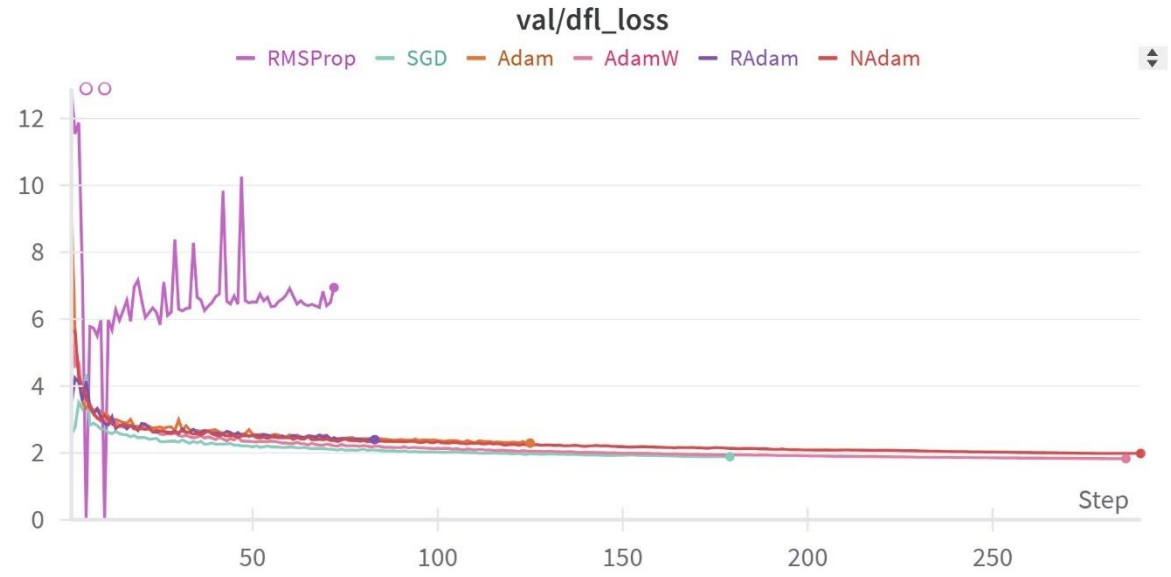
Em modelos como YOLO (You Only Look Once), a Box Regression Loss é crucial para garantir que as caixas delimitadoras preditas estejam corretamente alinhadas com os objetos detectados. Isso é essencial para a precisão geral do modelo.

Figura 57. Comparativo - train/box_loss

4.4.16 Val/dfl_loss

Idem train/dfl_loss mas durante a fase de validação de um modelo. DFL é usada para melhorar a precisão e a robustez do modelo ao lidar com dados desafiadores, especialmente em tarefas de detecção de objetos.

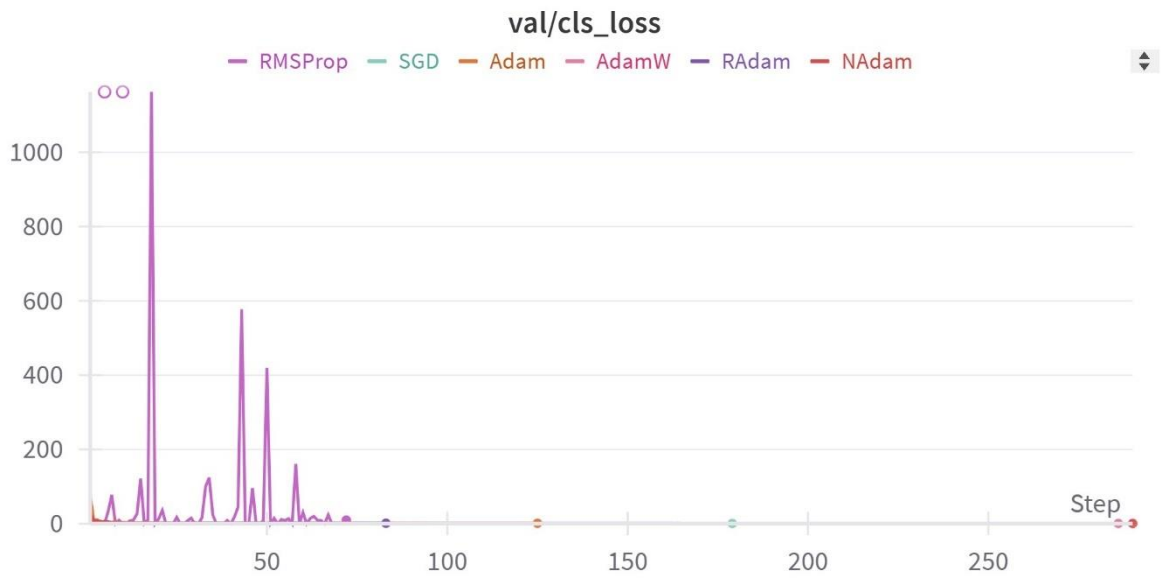
Figura 58. Comparativo - val/dfl_loss



4.4.17 Val/cls_loss

Idem train/cls_loss mas durante a fase de validação de um modelo. Usada para ajustar as previsões de classe das caixas delimitadoras detectadas

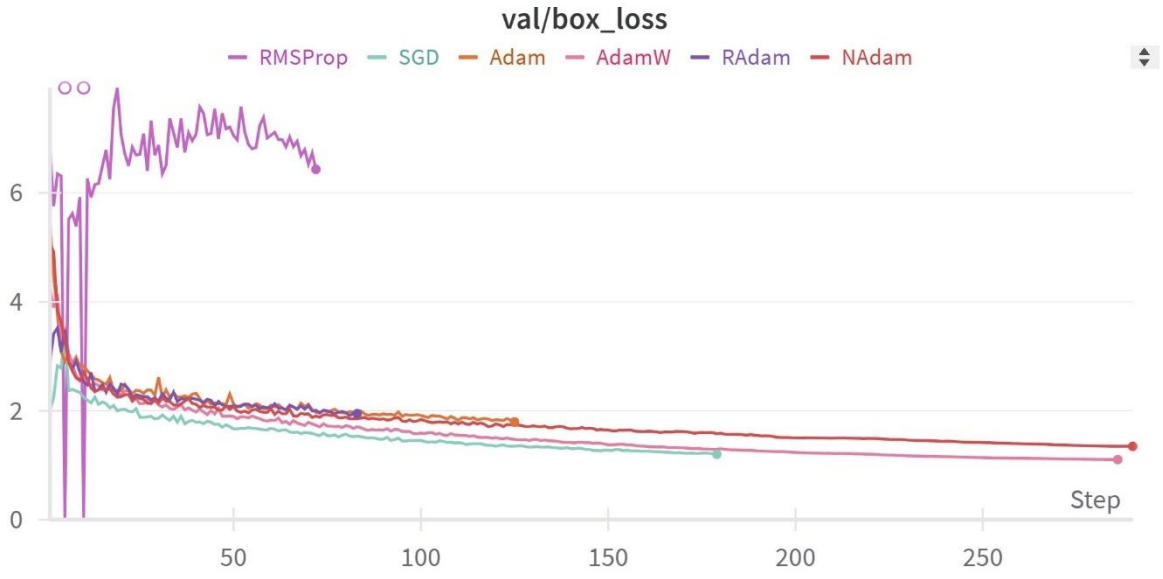
Figura 59. Comparativo - val/cls_loss



4.4.18 Val/box_loss

Idem train/box_loss (Perda de Regressão de Caixa) mas durante a fase de validação. É usada para ajustar os parâmetros do modelo de forma a melhorar a precisão das previsões das caixas delimitadoras.

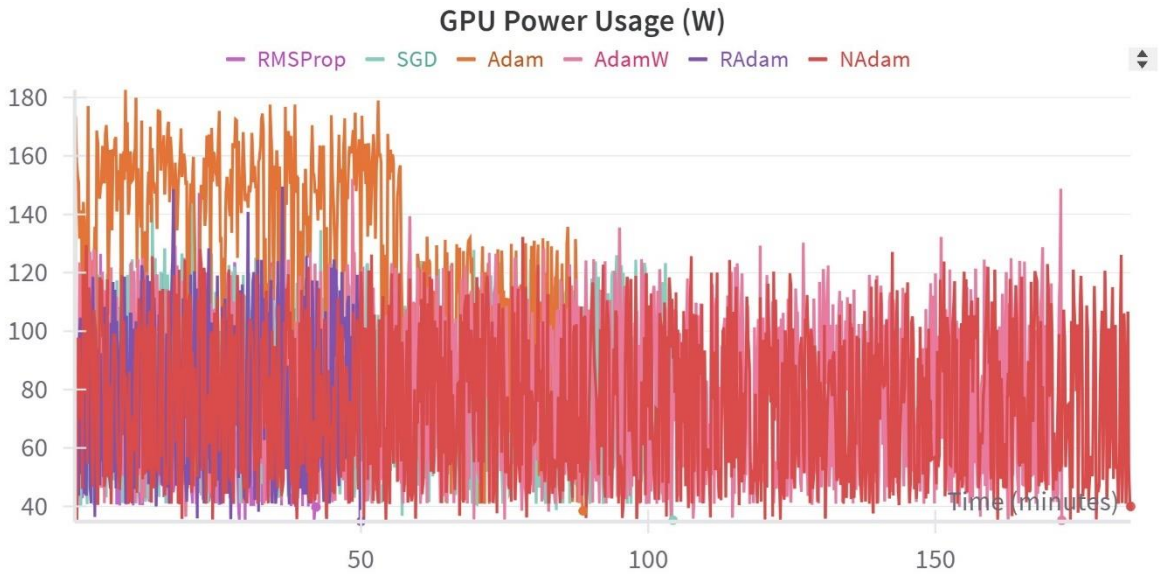
Figura 60. Comparativo - val/box_loss



4.4.19 System/Gpu Power Usage

O consumo de energia das GPUs (Unidades de Processamento Gráfico) é uma métrica importante para avaliar a eficiência e o desempenho de um sistema.

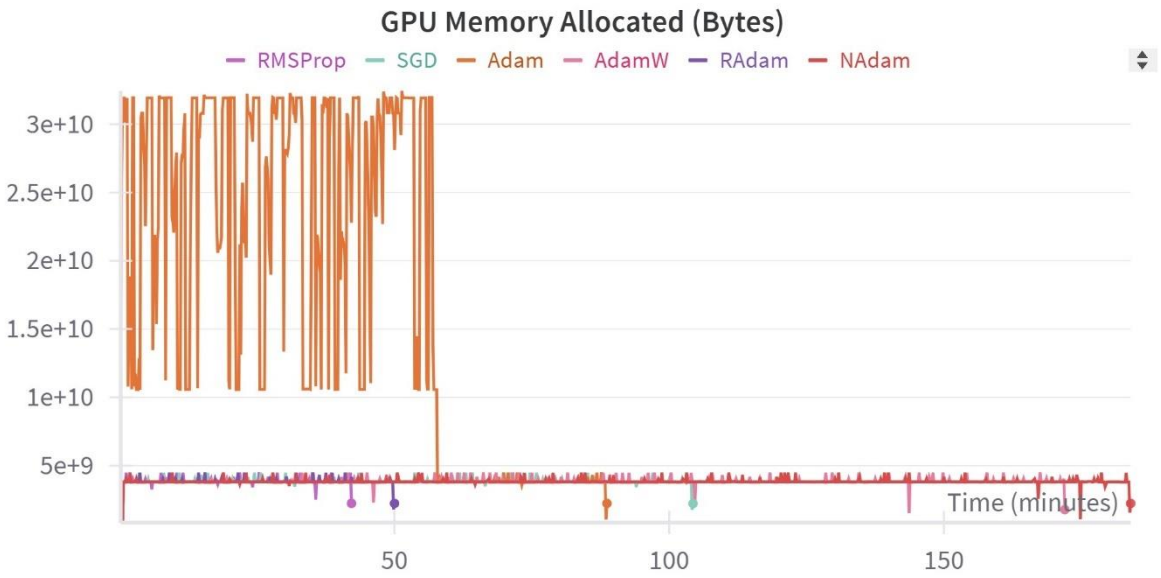
Figura 61. Comparativo - GPU Power Usage (W)



4.4.20 System/GPU Memory Allocated

Refere-se à quantidade de memória da GPU que está sendo usada por um modelo ou aplicação em um dado momento.

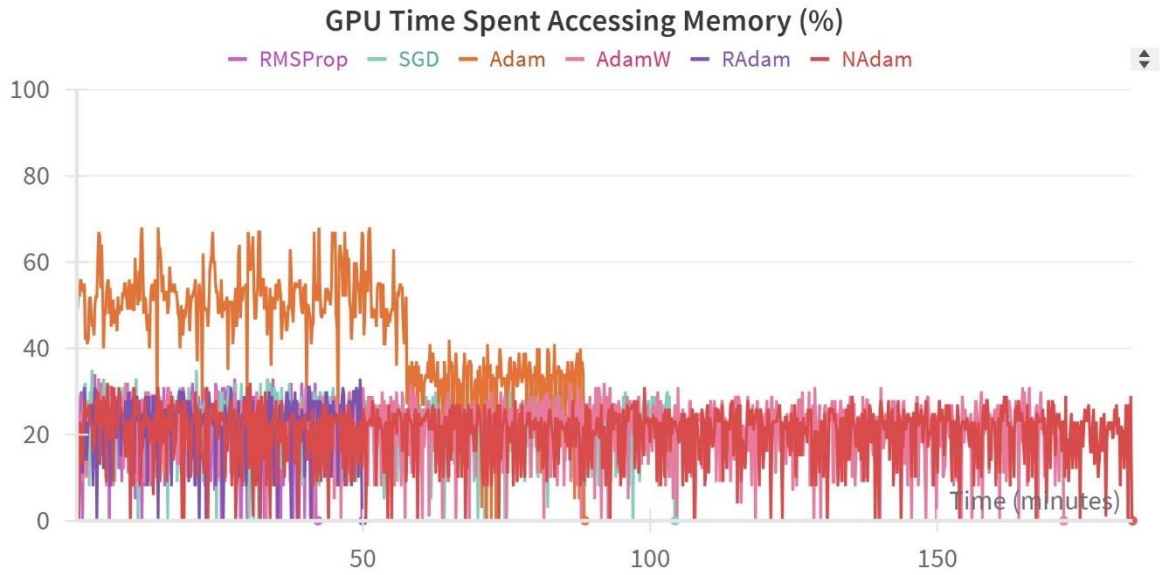
Figura 62. Comparativo - GPU Memory Allocated (Bytes)



4.4.21 System/GPU Time Spent Accessing Memory

Refere-se ao tempo que a GPU gasta acessando a memória durante a execução de operações. É uma métrica crucial para entender a eficiência do uso da memória pela GPU.

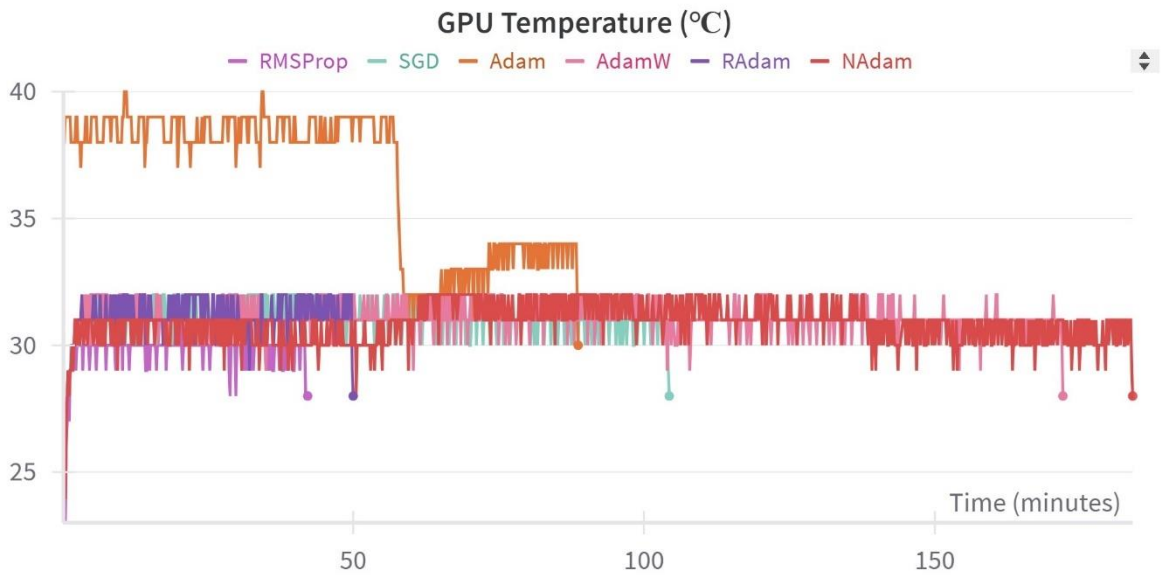
Figura 63. Comparativo - GPU Time Spent Accessing Memory(%)



4.4.22 System/GPU Temperature

Temperatura da GPU durante sua utilização.

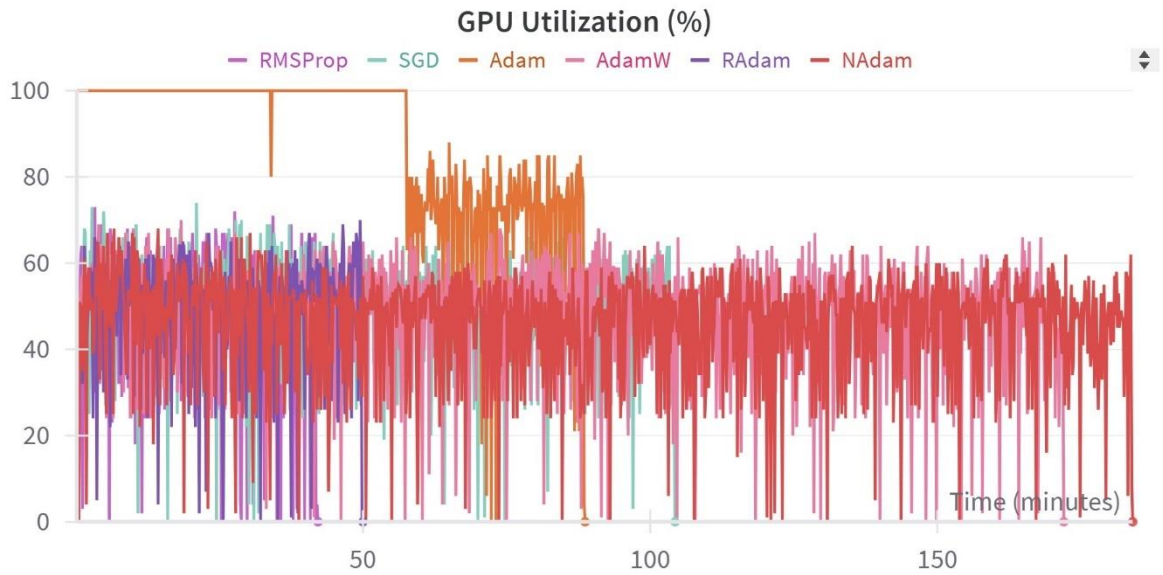
Figura 64. Comparativo - GPU Temperature (°C)



4.4.23 System/GPU Utilization %

Utilização da GPU (em %).

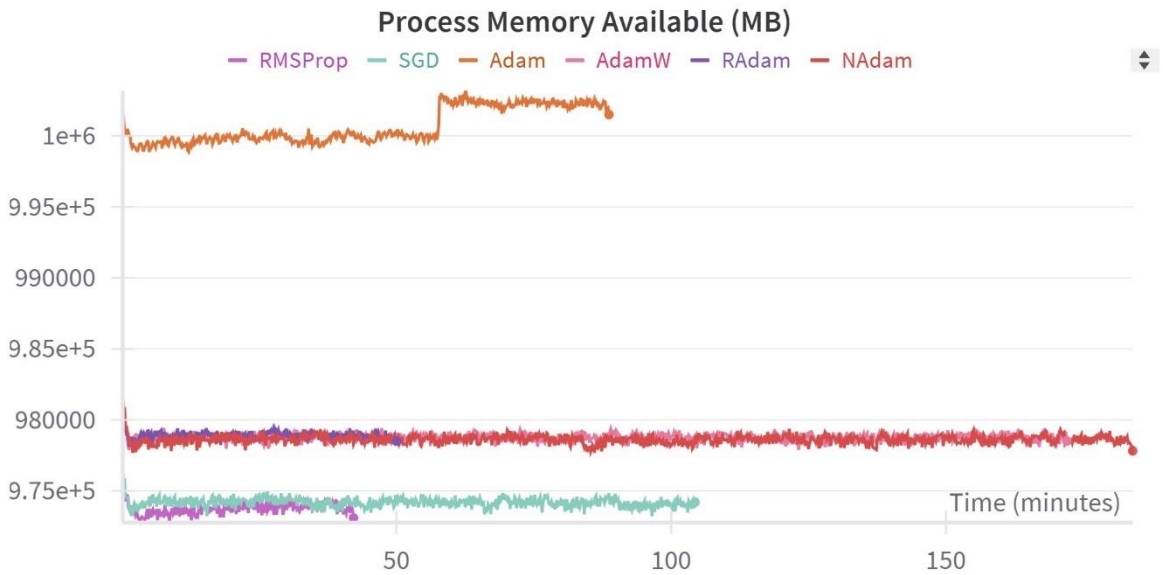
Figura 65. Comparativo - GPU Utilization (%)



4.4.24 System/Process Memory Available (MB)

Durante o treinamento de modelos YOLO (You Only Look Once), é importante monitorar a memória disponível do processo para garantir que o treinamento ocorra sem problemas.

Figura 66. Comparativo - Process Memory Available (MB)



4.5. Resultados de Predições e Discussões

Os resultados são discutidos em termos de precisão, robustez e aplicabilidade, destacando tanto os acertos quanto as limitações do modelo no reconhecimento de veículos militares em imagens. De um modo geral, o único algoritmo experimentado que se mostrou destoante dos demais em seus resultados e não conseguiu fazer predições satisfatórias foi o RMSProp.

4.5.1 Exemplos de Detecções (Individualmente)

A seguir utilizamos mais de um modelo a fim de demonstrar que é possível combinar o resultado preditivo de vários modelos (nossos modelos especializados foram treinados para detectar as 5 classes de veículos militares inicialmente planejadas) mas não identificariam veículos civis, por exemplo. Por outro lado, o modelo básico identificaria os veículos civis mas não entenderia os veículos militares. Então, adicionamos essa funcionalidade combinando os resultados de nosso treino com o modelo básico yolov10n (preparada para distinguir 80 classes diferentes dos nossos veículos) conforme a tabela a seguir:

Tabela 6. Lista de Classes Pré-Treinadas no Modelo YoloV10n

0: person	20: elephant	40: wine glass	60: dining table
1: bicycle	21: bear	41: cup	61: toilet
2: car	22: zebra	42: fork	62: tv
3: motorcycle	23: giraffe	43: knife	63: laptop
4: airplane	24: backpack	44: spoon	64: mouse
5: bus	25: umbrella	45: bowl	65: remote
6: train	26: handbag	46: banana	66: keyboard
7: truck	27: tie	47: apple	67: cell phone
8: boat	28: suitcase	48: sandwich	68: microwave
9: traffic light	29: frisbee	49: orange	69: oven
10: fire hydrant	30: skis	50: broccoli	70: toaster
11: stop sign	31: snowboard	51: carrot	71: sink
12: parking meter	32: sports ball	52: hot dog	72: refrigerator
13: bench	33: kite	53: pizza	73: book
14: bird	34: baseball bat	54: donut	74: clock
15: cat	35: baseball glove	55: cake	75: vase
16: dog	36: skateboard	56: chair	76: scissors
17: horse	37: surfboard	57: couch	77: teddy bear
18: sheep	38: tennis racket	58: potted plant	78: hair drier
19: cow	39: bottle	59: bed	79: toothbrush

Eventualmente, por falta de conhecimento, esse modelo “básico” classificaria, nossos veículos militares originalmente como “Trens”, “Barcos” ou “Caminhões” (com uma confiança não muito alta na maioria das vezes).

Tabela 7. Predição de Veiculos Civis e Militares por Modelo Não Especializado

Por outro lado, nossos modelos especializados se mostraram muito bons em classificar os veículos militares mas ruins em classificar os veículos civis.

Tabela 8. Lista de Classes Exclusivamente Treinadas nos Modelos CFN

0: ASTROS	1: CLANF	2: JLTV	3: M113	4: PIRANHA	5: SK105

Tabela 9. Predição de Veiculos Civis e Militares Por Modelo Militar Especializado

4.5.2 Classificações Combinadas

Ao combinar o resultado de vários modelos, conseguimos extrair o melhor que cada um tem a oferecer sem necessariamente precisar retreinar todos os dados.

Detecções precisas de veículos militares e civis em ambientes diversos.

Identificação de múltiplos objetos em uma única imagem, com bounding boxes corretamente posicionadas e classificações exatas.

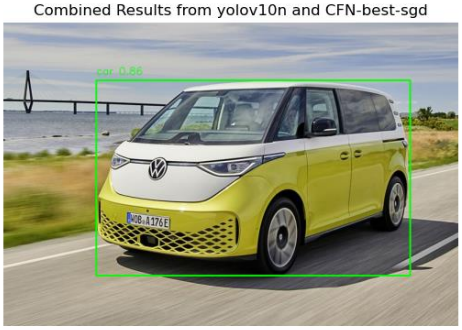
Tabela 10. Predições Combinadas de Mais de um Modelo

Combined Results from yolov10n and CFN-best-adamw 	Combined Results from yolov10n and CFN-best-adamw 
Combined Results from yolov10n and CFN-best-sgd 	Combined Results from yolov10n and CFN-best-adamw 
Combined Results from yolov10n and CFN-best-sgd 	Combined Results from yolov10n and CFN-best-sgd 


Combined Results from yolov10n and CFN-best-sgd
JLTV 0.84


Combined Results from yolov10n and CFN-best-sgd
M113 0.93


Combined Results from yolov10n and CFN-best-sgd
M113 0.93

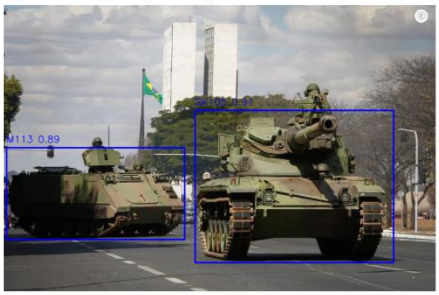

Combined Results from yolov10n and CFN-best-sgd


Combined Results from yolov10n and CFN-best-sgd
PIRANHA 0.97

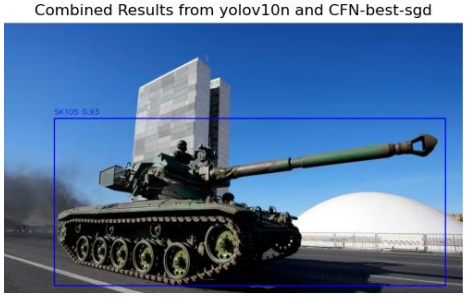

Combined Results from yolov10n and CFN-best-sgd
PIRANHA 0.92
PIRANHA 0.67

Combined Results from yolov10n and CFN-best-sgd

Combined Results from yolov10n and CFN-best-sgd



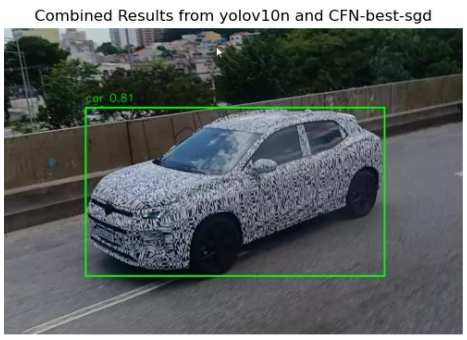
Combined Results from yolov10n and CFN-best-sgd



Combined Results from yolov10n and CFN-best-sgd

Combined Results from yolov10n and CFN-best-sgd

Combined Results from yolov10n and CFN-best-sgd


Combined Results from yolov10n and CFN-best-sgd
car 0.81


Combined Results from yolov10n and CFN-best-sgd
SK105 0.80

5. Considerações Finais

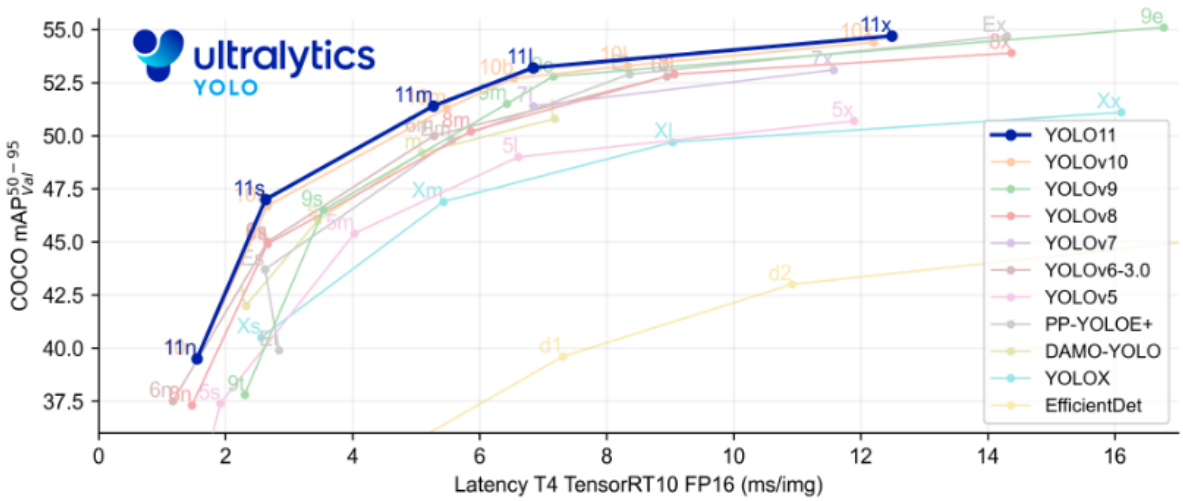
5.1. Sobre a Evolução do Yolo

Como vimos, a evolução da R-CNN até o YOLO representa uma jornada de evolução na detecção de objetos onde cada algoritmo é evoluído a partir de seu antecessor.

O YOLO, por tratar a detecção de objetos como um único problema de regressão, processando toda a imagem em uma passagem para frente, torna-o extremamente rápido e capaz de processamento em tempo real. Além disso, podemos adaptá-lo para nossa necessidade em cada cenário levando em conta capacidade de processamento, acurácia e velocidade de resposta.

Como visto no gráfico abaixo a cada versão o modelo tem evoluído apresentando novos recursos e melhorias com aumento de desempenho e flexibilidade.

Figura 67. Comparativo Latência x mAP (média das precisões médias de todas as classes no conjunto de dados)



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

A introdução da IA em sistemas de defesa representa uma evolução significativa na forma como os dados são processados e utilizados em operações críticas, reforçando a importância do tema em um mundo cada vez mais digitalizado e interconectado.

5.2. Conclusão

O presente trabalho investigou o uso do modelo YOLO para o reconhecimento de veículos militares em imagens, com foco na aplicação prática e na análise de desempenho do modelo. Foram realizadas etapas que incluíram a coleta e anotação

de um dataset customizado, o treinamento do modelo em diferentes condições e a validação de sua eficiência em cenários variados.

Os resultados obtidos demonstraram que o YOLO é uma solução viável e eficaz para a tarefa proposta, especialmente devido à sua capacidade de operar em tempo real, processando imagens a uma velocidade de 8.2 GFLOPs, o modelo alcançou seus melhores mAPs em 98.3%, 98.2% e 97.4% com IoU até 0.5 e mAPs 86.9%, 85.2% e 81.9% IoU de 0.5~0.95 (otimizadores AdamW, SGD e NAdam respectivamente).

Adicionalmente, foi avaliado o potencial de implementação deste sistema em drones. Os resultados indicam que o YOLO é particularmente adequado para sistemas embarcados, pois sua alta velocidade e baixa latência permitem o reconhecimento de veículos em tempo real durante o voo. Isso torna viável sua aplicação em missões táticas, vigilância aérea e patrulhamento de fronteiras, onde a identificação precisa e rápida é fundamental para a tomada de decisões em ambientes dinâmicos. Os arquivos obtidos referentes aos treinos dos modelos com seus pesos não ultrapassaram 5Mb).

A análise dos resultados mostrou que o modelo é robusto em cenários bem iluminados e com objetos claramente visíveis, mas obtém bons resultados também em condições adversas como baixa iluminação, oclusões e camuflagem desde que seja treinado previamente para tal. Esses aspectos destacam a importância de aprimorar o dataset e explorar técnicas que aumentem a capacidade do modelo de lidar com situações mais complexas.

5.3. Contribuições

Este estudo trouxe contribuições relevantes para o campo da visão computacional aplicada ao setor militar, como:

Desenvolvimento de um fluxo completo de trabalho: Foram definidas todas as etapas necessárias para o uso prático do YOLO, desde a coleta e anotação de dados até o treinamento, validação e análise dos resultados do modelo.

Criação de um *dataset* customizado: Incluindo imagens anotadas de diferentes tipos de veículos militares em cenários variados, o que pode ser utilizado como base para estudos futuros.

Avaliação detalhada do desempenho: Foram realizadas análises quantitativas e qualitativas do modelo, abrangendo métricas como precisão, velocidade e robustez em condições adversas.

Proposta de aplicações práticas: Os resultados obtidos reforçam o potencial do YOLO para uso em sistemas de vigilância, drones e monitoramento em tempo real, oferecendo suporte à tomada de decisões em ambientes críticos.

5.4. Limitações

Apesar das contribuições, algumas limitações foram identificadas:

Diversidade do *dataset*: Embora o dataset tenha incluído diferentes cenários, ele ainda não abrange todas as possíveis variações encontradas em operações reais, como condições climáticas extremas ou veículos de designs menos comuns.

Restrição ao YOLO: O estudo focou exclusivamente no YOLO, sem explorar combinações ou modelos híbridos que poderiam oferecer melhor desempenho em cenários específicos.

5.5. Trabalhos Futuros

Com base nos resultados obtidos e nas limitações identificadas, algumas direções para trabalhos futuros são propostas:

1. Expansão do *Dataset*:
    - Incluir mais imagens com variações climáticas extremas, ângulos inusitados e veículos menos comuns.
    - Aumentar a quantidade de exemplos anotados em condições adversas, como baixa iluminação e camuflagem intensa.
    - Priorizar imagens aéreas para utilização com drones.

2. Técnicas de Pré-processamento:

   - Aplicar técnicas como aumento de contraste, redução de ruído e balanceamento de cores para melhorar a qualidade das imagens de entrada.
   - Utilizar dados sintéticos gerados por simulações para complementar o dataset.

3. Treinamento Híbrido:

   - Combinar o YOLO com outros modelos, como Faster R-CNN, para aproveitar os pontos fortes de ambos.
   - Implementar arquiteturas mais recentes, como YOLOv11 a fim de explorar avanços tecnológicos mais recentes.

4. Uso em Drones e Sistemas Embarcados:

   - Testar o modelo diretamente em drones, avaliando sua capacidade de operar em tempo real durante o voo.
   - Desenvolver soluções de integração para adaptar o modelo ao hardware de drones, levando em conta limitações como consumo de energia e capacidade de processamento.

5. Integração com Tecnologias Avançadas:

   - Incorporar informações adicionais, como dados de sensores térmicos, infravermelhos e gps para aumentar a precisão em cenários complexos.

5.6. Considerações Finais

A detecção e o reconhecimento de veículos militares por meio de inteligência artificial representam um avanço significativo na área de defesa e segurança. Este trabalho demonstrou que o YOLO pode ser uma ferramenta poderosa para aplicações práticas, equilibrando precisão e eficiência em tempo real. Sua possível implementação em drones amplia ainda mais o alcance e a aplicabilidade da tecnologia, permitindo o monitoramento autônomo de áreas extensas e a coleta de informações estratégicas em tempo hábil.

Entretanto, os desafios encontrados reforçam a necessidade de continuidade nos estudos, com melhorias que visem ampliar a robustez do modelo em cenários reais e complexos. Por fim, este estudo não apenas contribui para o campo da visão computacional, mas também oferece uma base sólida para futuras pesquisas e aplicações práticas, destacando o potencial da inteligência artificial na transformação de sistemas de defesa modernos.

6. Referências

REDMON, J. et al. **You only look once: Unified, real-time object detection**. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). [S.l.: s.n.], 2016.

J. Redmon, A. Farhadi, **YOLO9000: better, faster, stronger**, In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 6517–6525

Lindsay, **G. Convolutional neural networks as a model of the visual system: past, present, and future**. Journal of Cognitive Neuroscience, v. 33, p. 2017-2031, 2020. Disponível em: https://doi.org/10.1162/jocn_a_01544. Acesso em: 10 set. 2024.

BRAGA, A. de P.; CARVALHO, A. de L. F.; LUDERMIR, T. **Redes neurais artificiais: teoria e aplicações**. Livros Técnicos e Científicos, 2000. ISBN 9788521612186. Disponível em: https://books.google.com.br/books?id=cUgEaAEACAAJ.

HUANG, R.; PEDOEEM, J.; CHEN, **C. Yolo-lite: a real-time object detection algorithm optimized for non-gpu computers**. In: IEEE. 2018 IEEE International Conference on Big Data (Big Data). [S.l.], 2018. p. 2503–2510.

R-CNN vs R-CNN Fast vs R-CNN Faster vs YOLO. Datadance. 2024. 20 out. 2024. Disponível em https://datadance.ai/machine-learning/r-cnn-vs-r-cnn-fast-vs-r-cnn-faster-vs-yolo/. Acesso em 20 dez. 2024.

LECUN, Yann; BOTTOU, Léon; BENGIO, Yoshua; HAFFNER, Patrick. **Gradient-based learning applied to document recognition**. Disponível em: https://ieeexplore.ieee.org/document/726791. Acesso em: 15 dez. 2024.

Ao Wang, Hui Chen, Lihao Liu, Kai Chen, Zijia Lin, Jungong Han, Guiguang Ding. **YOLOv10: Real-Time End-to-End Object Detection**. Disponível em: https://arxiv.org/abs/2405.14458. Acesso em: 30 out. 2024.

ULTRALYTICS. **Ultralytics YOLO Source Codes and Examples on Github**. Disponível em https://github.com/ultralytics/ultralytics. Acesso em: 20 set. 2024.

ULTRALYTICS. **YOLOv10: Real-Time End-to-End Object Detection**. Disponível em https://docs.ultralytics.com/models/yolov10. Acesso em: 20 set. 2024.

MICROSOFT. **Azure Blob Storage**. Disponível em https://azure.microsoft.com/en-us/products/storage/blobs/. Acesso em: 10 ago. 2024.

Label Studio**. Label Studio Quick Start**. Disponível em https://labelstud.io/guide/quick_start. Acesso em 10 out. 2024.

Apêndice A - Treino de Modelos Customizados Yolo

```
import subprocess
import sys
import os


path_inicio = '/home/robert.libotti/TCC-FGVCFN-YOLO'
path_saida = '/home/robert.libotti/OUTPUT'
path_yolo = path_saida + '/yolov10'
wandb_key = '<PRIVADO>'
project_id = 'FGVCFN_2025Nv01'
epochs = 600
patience = 10
versao_yolo = 'yolov10n'
optimizers = ['RMSProp', 'SGD','NAdam', 'RMSProp', 'RAdam', 'AdamW', 'Adam']


def install(package):
    subprocess.check_call([sys.executable, '-m', 'pip', 'install', package])


def checagem():
    print('Checando se o ambiente está correto...')
    import torch
    import torchvision
    import wandb
    import ultralytics
    import yolov10
    current_directory = os.getcwd()
    print('Ambiente OK')
    print('Versão do Pytorch:', torch.__version__)
    print('Versão do Torchvision:', torchvision.__version__)
    print('Versão do Wandb:', wandb.__version__)
    print('Versão do Ultralytics:', ultralytics.__version__)
    print(f'YOLO ({versao_yolo}): ', yolov10.__version__)
    print(f"Diretorio corrente: {current_directory}")

    if torch.cuda.is_available():
        print("GPU is available.")
    else:
        print("GPU is not available.")

def verificar_gpu():
    import GPUtil
    gpus = GPUtil.getGPUs()
    for gpu in gpus:
        print(f"ID: {gpu.id}")
        print(f"Nome: {gpu.name}")
```

```
        print(f"Memória Total: {gpu.memoryTotal} MB")
        print(f"Memória Utilizada: {gpu.memoryUsed} MB")
        print(f"Memória Livre: {gpu.memoryFree} MB")
        print(f"Utilização: {gpu.load * 100}%")
        print(f"Temperatura: {gpu.temperature} °C")
        print("-" * 30)


def passo01_instala_bibliotecas():
    install('torch') # torchvision wandb ultralytics yolov10
    install('torchvision')
    install('wandb')
    install('ultralytics')
    install('yolov10')
    install('gitpython')
    install('gputil')

def passo02_gitclone_yolo():
    os.chdir(path_saida)
    if not os.path.exists(path_yolo):
        repo_url = 'https://github.com/THU-MIG/yolov10.git'
        try:
            # Clone the repository
            git.Repo.clone_from(repo_url, path_yolo)
            print("Repository cloned successfully!")
        except Exception as e:
            print(f"An error occurred: {e}")

    checagem()

def passo03_treino(optimizer):
    from ultralytics import YOLO, settings
    import wandb
    import random
    import os

    os.chdir(path_yolo)
    project_name = f'{epochs}epochs{optimizer}'
    checagem()

    settings.update({"wandb": True})
    # Initialize W&B run
    wandb.login(key=wandb_key)  # userdata.get('WANDB_API_KEY_TCC')
    wandb.init(project=project_id, name=project_name)

    # opcao 1
    # model = YOLOv10.from_pretrained('jameslahm/yolov10n')

    # opcao 2
```

```
    # wget https://github.com/
THU-MIG/yolov10/releases/download/v1.1/yolov10{n/s/m/b/l/x}.pt
    # model = YOLOv10('yolov10{n/s/m/b/l/x}.pt')

    url_yolo = f'https://github.com/THU-MIG/yolov10/releases/download/v1.1/yolov10n.pt'
    local_path_yolo = os.path.join(path_yolo, f'{versao_yolo}.pt')

    try:
        import urllib.request
        print(f"Downloading {url_yolo} to {local_path_yolo}...")
        urllib.request.urlretrieve(url_yolo, local_path_yolo)
        print("Download complete.")
    except Exception as e:
        print(f"Error downloading {url_yolo}: {e}")

    model = YOLO(versao_yolo)
    config = f'{path_inicio}/config.yaml'

    model.train(project=project_id, data=config, epochs=epochs, patience=patience, verbose=True,
optimizer=optimizer)




# Press the green button in the gutter to run the script.
if __name__ == '__main__':

    passo01_instala_bibliotecas()
    passo02_gitclone_yolo()

    checagem()

    for optimizer in optimizers:
        passo03_treino(optimizer)
```

Apêndice B - Usando Modelos Customizados Para Classificar e Combinar Resultados de Forma Personalizada

```
import torch
from PIL import Image
import cv2
import os
import matplotlib.pyplot as plt
from ultralytics import YOLO
import json


# Função para carregar o modelo
def load_yolo_model(model_path):
    model = YOLO(model_path)
    return model

# Função para realizar a detecção de objetos
def detect_objects(model, image_path, idx_classes, min_confidence):
    results = model(image_path)

    if len(results) == 0:
        return results

    filtered_results = [result for result in results if result.boxes.cls.numel() > 0 and result.boxes.conf[0] >= min_confidence and int(result.boxes.cls[0]) in idx_classes]
    return filtered_results

# Função para desenhar as bounding boxes nas imagens
def draw_boxes(image_path, results, output_path):
    for i, r in enumerate(results):
        # Plot results image
        im_bgr = r.plot()  # BGR-order numpy array
        im_rgb = Image.fromarray(im_bgr[..., ::-1])  # RGB-order PIL image

        r.save(filename=f"{output_path}")


def draw_boxes_cv2(image, results1, results2, tickness, min_confidence_yolo, min_confidence_cfn):
    for result in results1:
        for box in result.boxes:
            x1, y1, x2, y2 = map(int, box.xyxy[0])
            indice = box.cls[0]
            label = result.names[int(indice)]
            confidence = box.conf[0]
            if label == 'car' and confidence > min_confidence_yolo:
                cv2.rectangle(image, (x1, y1), (x2, y2), (0, 255, 0), tickness)
                cv2.putText(image, f'{label} {confidence:.2f}', (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0), 1, cv2.LINE_AA)
```

```
    for result in results2:
        for box in result.boxes:
            x1, y1, x2, y2 = map(int, box.xyxy[0])
            indice = box.cls[0]
            label = result.names[int(indice)]
            confidence = box.conf[0]
            if confidence >= min_confidence_cfn:
                cv2.rectangle(image, (x1, y1), (x2, y2), (255, 0, 0), tickness)
                cv2.putText(image, f'{label} {confidence:.2f}', (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 0, 0), 1,
cv2.LINE_AA)

    return image

# Diretórios
current_dir = os.getcwd()
input_dir = f'{current_dir}\\input-files'
output_dir = f'{current_dir}\\output-files'
models = ['best-sgd', 'best-adamw', 'best-nadam'] #'best-rmsprop', 'best-radam', 'best-adam', 'yolov10n' 'best-
sgd', 'best-adam', 'best-adamw', 'best-nadam', 'best-radam', 'best-rmsprop',
models_dir = f'{current_dir}\\model-files'
model1 = load_yolo_model(f'{models_dir}\\yolov10n.pt')

# para cada model
for modelo in models:
    model2 = load_yolo_model(f'{models_dir}\\{modelo}.pt')

    # Processar cada imagem no diretório de entrada
    for img_file in os.listdir(input_dir):
        img_path = os.path.join(input_dir, img_file)

        img = cv2.imread(img_path)
        min_confidence_yolo = 0.5
        min_confidence_cfn = 0.6

        # 0: person, 1: bicycle, 2: car, 3: motorcycle, 4: airplane, 5: bus, etc.
        results1 = detect_objects(model1, img_path, [2], min_confidence_yolo)

        # ASTROS, CLANF, JLTV, M113, PIRANHA, SK105
        results2 = detect_objects(model2, img_path, [0, 1, 2, 3, 4, 5], min_confidence_cfn)

        fname, fext = os.path.splitext(img_file)

        # resultados combinados (com filtros)
        combined_result = img.copy()
        combined_result = draw_boxes_cv2(combined_result, results1, results2, 2, min_confidence_yolo,
min_confidence_cfn)
```

```
        plt.figure(figsize=(10, 10))
        plt.imshow(cv2.cvtColor(combined_result, cv2.COLOR_BGR2RGB))
        plt.title(f'Combined Results from yolov10n and CFN-{modelo}')
        plt.axis('off')
        #plt.show()

        plt.savefig(f'{output_dir}\\{fname}_combined_{min_confidence_yolo}-{min_confidence_cfn}_{modelo}.png')

print("Processamento concluído e arquivos salvos no diretório de saída.")
```