CAPÍTULO 3

CONFIGURAÇÃO DO AMBIENTE DE DESENVOLVIMENTO, PADRÃO DE DATASET E TREINAMENTO PARA YOLO

1. INTRODUÇÃO E PREPARAÇÃO DO AMBIENTE INTERATIVO

O treinamento de detectores da família YOLO (You Only Look Once) requer um ambiente de software compatível com os aceleradores gráficos disponíveis. Em fluxos com Jupyter Notebook ou scripts locais, utilizam-se o pacote Ultralytics, a biblioteca Pandas e o PyTorch compilado com suporte a CUDA e cuDNN. As seções seguintes cobrem a instalação dessas dependências, a padronização do conjunto de dados e os parâmetros de treinamento para o YOLOv12.

2. INSTALAÇÃO DAS DEPENDÊNCIAS PRINCIPAIS E FRAMEWORKS

A manipulação de tabelas e o treinamento do detector utilizam os pacotes *pandas* e *ultralytics*. Em células do Jupyter Notebook, a instalação pode ser feita com o prefixo *!*:

In [1]:

```
!pip install pandas ultralytics
```

Out [1]:

```
Collecting pandas Using cached pandas-2.2.2-cp310-cp310-manylinux2014_x86_64.whl (13.0 MB)
Collecting ultralytics Using cached ultralytics-8.3.80-py3-none-any.whl (915 kB)
Successfully installed pandas-2.2.2 ultralytics-8.3.80
```

A biblioteca Pandas processa metadados de anotação, contagens de classes e tabelas estatísticas de inferência, enquanto a Ultralytics fornece os motores de treinamento, validação e exportação para os modelos YOLO.

3. CONFIGURAÇÃO DA ACELERAÇÃO DE HARDWARE NVIDIA CUDA E CUDNN

O cálculo de gradientes e o processamento de tensores durante o treinamento exigem aceleração por GPU. Para compatibilidade com os drivers CUDA 12, instalam-se os pacotes de execução da NVIDIA, incluindo cuDNN (*CUDA Deep Neural Network library*) e cuBLAS (*CUDA Basic Linear Algebra Subprograms*):

In [2]:

```
!pip install --upgrade nvidia-cudnn-cu12 nvidia-cublas-cu12 nvidia-cuda-runtime-cu12
```

Out [2]:

```
Requirement already satisfied: nvidia-cudnn-cu12 in ./.venv/lib/python3.10/site-packages
Requirement already satisfied: nvidia-cublas-cu12 in ./.venv/lib/python3.10/site-packages
Requirement already satisfied: nvidia-cuda-runtime-cu12 in ./.venv/lib/python3.10/site-packages
Successfully upgraded nvidia-cudnn-cu12-9.1.0 nvidia-cublas-cu12-12.1.3 nvidia-cuda-runtime-cu12-12.1.105
```

Essas bibliotecas nativas otimizam a execução de operações matriciais de convolução, funções de ativação e normalização por lote diretamente na GPU, reduzindo drasticamente o tempo necessário por época durante o ajuste dos parâmetros do modelo.

4. INSTALAÇÃO E REINSTALAÇÃO FORÇADA DO ECOSSISTEMA PYTORCH

O PyTorch constitui a fundação matemática e o *backend* de processamento de tensores utilizado pelas implementações da família YOLO. Em cenários nos quais há conflitos de versão entre bibliotecas de processamento paralelo ou degradação na comunicação com a placa de vídeo, garante-se a estabilidade do sistema realizando a reinstalação forçada das bibliotecas centrais do PyTorch (*torch*, *torchvision* e *torchaudio*) diretamente do repositório oficial compilado para CUDA 12.1 (*cu121*).

In [3]:

```
!pip install --force-reinstall torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121
```

Out [3]:

```
Looking in indexes: https://download.pytorch.org/whl/cu121
Collecting torch Using cached torch-2.5.1
Collecting torchvision Using cached torchvision-0.20.1
Collecting torchaudio Using cached torchaudio-2.5.1
Successfully installed torch-2.5.1+cu121 torchaudio-2.5.1+cu121 torchvision-0.20.1+cu121
```

5. VERIFICAÇÃO DA DISPONIBILIDADE DE GPU E AMBIENTE FINAL

Após o término da instalação dos componentes de software, valida-se o funcionamento correto da aceleração por hardware através do módulo PyTorch no Python:

In [4]:

```
import torch

print(f"Versao do PyTorch: {torch.__version__}")
print(f"CUDA Disponivel: {torch.cuda.is_available()}")
if torch.cuda.is_available():
    print(f"Dispositivo GPU: {torch.cuda.get_device_name(0)}")
```

Out [4]:

```
Versao do PyTorch: 2.5.1+cu121
CUDA Disponivel: True
Dispositivo GPU: NVIDIA GeForce RTX 4090
```

Com a confirmação da disponibilidade da GPU e a validação do ambiente interativo estilo Jupyter Notebook, a infraestrutura de software está pronta para a preparação dos dados.

6. ESTRUTURAÇÃO DO DATASET NO FORMATO YOLO

Para que as redes de detecção de objetos aprendam a identificar e localizar alvos em imagens, o conjunto de dados precisa estar organizado rigorosamente segundo a especificação padrão do formato YOLO. Este padrão é universal e mantido de forma consistente em todas as gerações modernas da arquitetura — incluindo **YOLOv8, YOLOv9, YOLOv10, YOLOv11 e YOLOv12**. O que evolui entre as versões são as estruturas de rede e os algoritmos de inferência, enquanto a organização dos arquivos de imagens e rótulos permanece uniforme.

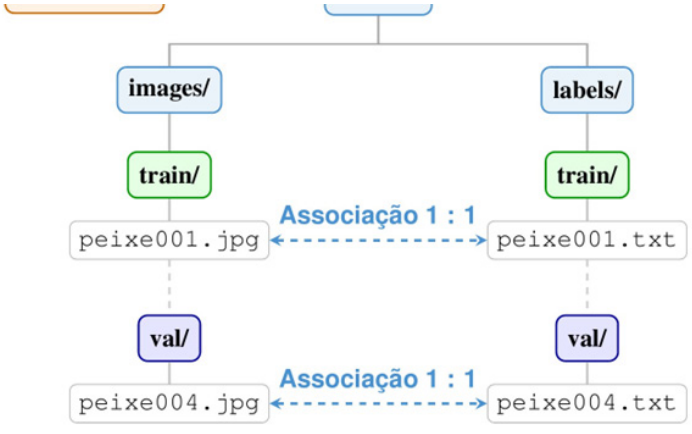
6.1 Organização dos Diretórios de Imagens e Rótulos

O formato YOLO exige a separação clara entre as matrizes de pixels (imagens) e seus respectivos arquivos textuais de marcação espacial (rótulos/*labels*). A raiz do conjunto de dados é composta por dois diretórios principais — *images/* e *labels/* — e pelo arquivo de configuração global *data.yaml*:

```
dataset/
+── images/
|+── train/
||+── img001.jpg
||+── img002.jpg
||+── ...
|+── val/
||+── img101.jpg
||+── ...
|+── test/ (opcional)
|+── img201.jpg
|+── ...
+── labels/
|+── train/
||+── img001.txt
||+── img002.txt
||+── ...
|+── val/
||+── img101.txt
||+── ...
|+── test/ (opcional)
|+── img201.txt
|+── ...
+── data.yaml
```

A Figura 13 ilustra esquematicamente a topologia do conjunto de dados e a associação de correspondência biunívoca (1 : 1) entre cada imagem e seu arquivo de anotação correspondente.

Figura 13: Estrutura de diretórios e associação de correspondência biunívoca (1 : 1) entre arquivos de imagens e rótulos no padrão YOLO.



6.2 Estrutura do Arquivo de Configuração data.yaml

O arquivo *data.yaml* atua como o mapa de navegação para a biblioteca Ultralytics, definindo os caminhos relativos ou absolutos dos conjuntos de dados e o dicionário de classes mapeadas. Para um dataset aplicado à aquicultura — como o monitoramento automatizado de espécimes de tilápia (*fish*) —, a configuração completa incluindo o conjunto opcional de teste é declarada da seguinte forma:

```
path: /content/dataset
train: images/train
val: images/val
test: images/test

names:
  0: fish
```

Uma vez que a partição de teste (*test*) é **opcional**, o arquivo *data.yaml* pode ser simplificado mantendo apenas as divisões essenciais de treinamento e validação:

```
path: /content/dataset
train: images/train
val: images/val

names:
  0: fish
```

6.3 Formato dos Arquivos de Anotação (.txt)

Para cada imagem com objetos de interesse no dataset (a exemplo de *images/train/img001.jpg*), deve existir um arquivo textual homônimo em *labels/train/img001.txt*. Caso uma imagem seja de fundo (*background image*) sem nenhum objeto da classe alvo, o arquivo *.txt* correspondente pode permanecer vazio ou ser omitido, auxiliando o modelo a reduzir a taxa de Falsos Positivos.

O conteúdo do arquivo *.txt* descreve a posição e a dimensão espacial das caixas delimitadoras (*bounding boxes*) de todos os objetos presentes na imagem. Um exemplo típico contendo dois objetos identificados é apresentado a seguir:

```
0 0.512 0.438 0.215 0.321
0 0.723 0.612 0.184 0.276
```

Cada linha do arquivo representa a anotação de um único objeto, estruturada rigorosamente em cinco valores separados por espaço:

<classe> <xcentro> <ycentro> <largura> <altura>

- **Classe (*classe*):** Índice numérico inteiro (iniciando em 0) correspondente à categoria mapeada no arquivo *data.yaml* (por exemplo, 0 para a classe *fish*).
- **Centro Horizontal (*xcentro*):** Coordenada X do ponto central da caixa delimitadora, normalizada entre 0 e 1 pela largura da imagem (W).
- **Centro Vertical (*ycentro*):** Coordenada Y do ponto central da caixa delimitadora, normalizada entre 0 e 1 pela altura da imagem (H).
- **Largura (*largura*):** Largura total da caixa delimitadora, normalizada entre 0 e 1 (w/W).
- **Altura (*altura*):** Altura total da caixa delimitadora, normalizada entre 0 e 1 (h/H).

Matematicamente, se uma imagem possui dimensão de 1920 × 1080 pixels e um objeto está centrado no pixel ($x = 983$, $y = 473$) com largura de 412,8 pixels e altura de 346,68 pixels, os valores normalizados calculados são descritos pelas equações a seguir:

$$x_{\text{centro}} = \frac{983}{1920} \approx 0{,}512, \quad (3.1)$$

$$y_{\text{centro}} = \frac{473}{1080} \approx 0{,}438, \quad (3.2)$$

$$\text{largura} = \frac{412{,}8}{1920} \approx 0{,}215, \quad (3.3)$$

$$\text{altura} = \frac{346{,}68}{1080} \approx 0{,}321. \quad (3.4)$$

Essa normalização torna os rótulos de coordenadas independentes da dimensão absoluta em pixels da imagem de entrada.

6.4 Exemplo Prático Completo de um Dataset

Considere um projeto de detecção de espécimes aquáticos (*10_peixes*). A estrutura completa de arquivos organizada para treinamento e validação apresenta-se da seguinte maneira:

```
dataset/
+--- images/
| +--- train/
| | +--- peixe001.jpg
| | +--- peixe002.jpg
| | +--- peixe003.jpg
| +--- val/
| +--- peixe004.jpg
| +--- peixe005.jpg
+--- labels/
| +--- train/
| | +--- peixe001.txt
| | +--- peixe002.txt
| | +--- peixe003.txt
| +--- val/
| +--- peixe004.txt
| +--- peixe005.txt
+--- data.yaml
```

Reforça-se que este formato de dataset permanece rigorosamente o mesmo para as versões **YOLOv8, YOLOv9, YOLOv10, YOLOv11 e YOLOv12**. Dessa forma, um mesmo repositório de dados anotados pode ser reutilizado diretamente para comparar o desempenho estatístico entre diferentes gerações da família YOLO.

7. SCRIPT DE TREINAMENTO DO MODELO YOLOV12

A seguir, é apresentado o código em Python para instanciação dos pesos pré-treinados da arquitetura YOLOv12s (Small) e inicialização do treinamento no conjunto de dados de detecção de peixes:

In [5]:

```
from ultralytics import YOLO

model= YOLO("modelos/yolo12s.pt")

model.train(
    data="datasets_extraidos/10_peixes/data.yaml",
    epochs=200,
    patience=50,
    imgsz=640,
    batch=16,
    device=0,
    workers=8,
    plots=True,
    project="treinamentos",
    name="yolo12", exist_ok=True
)
```

Out [5]:

```
Ultralytics 8.3.80 – Python–3.10.12 torch–2.5.1+cu121 CUDA:0 (NVIDIA GeForce RTX 4090, 24564MB)
Model summary: 225 layers, 9,271,854 parameters, 0 gradients, 21.6 GFLOPs ...
(Treinamento em progresso: Epoca 200/200 concluida com sucesso)
Results saved to treinamentos/yolo12
```

8. ANÁLISE DETALHADA E EXPLICAÇÃO DO CÓDIGO

O script de treinamento apresentado executa o processo de aprendizado de forma direta e estruturada. A seguir, detalham-se os componentes e a especificação dos hiperparâmetros configurados:

8.1 Importação e Carregamento do Modelo

- *from ultralytics import YOLO*: Importa a classe principal da biblioteca Ultralytics responsável pela gerência da arquitetura e rotinas da rede.
- *model = YOLO("modelos/yolo12s.pt")*: Carrega o arquivo de pesos pré-treinados da variante YOLOv12s (Small), localizada na pasta *modelos/*. A utilização de pesos pré-treinados no conjunto de dados MS COCO (COCO) ativa o paradigma de Transfer Learning (Aprendizado por Transferência), permitindo acelerar a convergência do modelo.

8.2 Configuração dos Hiperparâmetros de Treinamento

O método *model.train()* recebe os parâmetros operacionais e hiperparâmetros que definem a dinâmica de ajuste dos pesos da rede neural. A Tabela 1 apresenta a descrição e o impacto técnico de cada hiperparâmetro utilizado.

A configuração proposta na tabela acima é tecnicamente coerente como ponto de partida para treinamento de um detector YOLO em GPU única, especialmente quando há Tabela 1: Descrição detalhada dos hiperparâmetros de treinamento configurados para o modelo YOLOv12.

Hiperparâmetro	Parâmetro	Valor	Descrição e Impacto Técnico
Épocas	*epochs*	200	Número máximo de iterações completas sobre todo o conjunto de dados.
Paciência	*patience*	50	Épocas limite sem evolução no mAP para parada antecipada (Early Stopping).
Resolução	*imgsz*	640	Dimensão de entrada (640 × 640 pixels) para redimensionamento das imagens.
Tamanho do Lote	*batch*	16	Quantidade de imagens processadas simultaneamente por iteração na GPU.
Dispositivo	*device*	0	Identificador da GPU física (ex: CUDA:0) dedicada ao processamento.
Subprocessos	*workers*	8	Quantidade de threads de CPU para carregamento em paralelo e Data Augmentation.

limitação moderada de VRAM. Contudo, *epochs=200* e *patience=200*, juntos, tornam o early stopping praticamente inativo: o treinamento tende a executar todas as 200 épocas, a menos que a implementação adote outra condição de parada. Para que a paciência funcione como regularização e economia computacional, recomenda-se defini-la abaixo do total de épocas, por exemplo entre 20 e 60, ajustando-a ao tamanho e à variabilidade da base.

A seguir, apresenta-se uma explicação técnica detalhada sobre cada um desses hiperparâmetros, que fundamenta as decisões metodológicas do treinamento: **Interpretação da Configuração:** A chamada *model.train()* controla não apenas o tempo de treinamento, mas a relação entre capacidade de aprendizagem, custo computacional, risco de sobreajuste (*overfitting*) e estabilidade operacional do pipeline. Em detectores de objetos da família YOLO, esses efeitos devem ser avaliados principalmente por métricas de validação, como precisão (*precision*), revocação (*recall*), mAP@0.5 e mAP@0.5:0.95, e não exclusivamente pela perda de treinamento (*training loss*).

Épocas (***epochs=200*):** O parâmetro *epochs* estabelece o máximo de vezes que a rede verá integralmente o conjunto de treinamento. Em cada época, todas as imagens anotadas são processadas, as perdas de localização, classificação e objectness são calculadas, e os pesos da rede são atualizados por retropropagação. A escolha de 200 épocas é justificável porque detectores YOLO normalmente precisam de dezenas ou centenas de épocas para estabilizar suas curvas de aprendizado, sobretudo em bases próprias, com poucos exemplos, elevada heterogeneidade de iluminação, oclusão, variação de escala,

fundo complexo ou objetos agrícolas naturais. Em aplicações de visão computacional agrícola, por exemplo, o modelo precisa aprender variações de frutos, folhas, solo, sombra, estádio fenológico, posição da câmera e condições ambientais. Entretanto, 200 não deve ser entendido como um valor universalmente ótimo. Trata-se de um limite máximo de treinamento, cuja adequação deve ser confirmada pelas curvas de validação e pelo desempenho no conjunto de teste independente.

- **10–50 épocas:** Útil para testes rápidos, depuração do pipeline e verificação de rótulos. Pode ocorrer subtreinamento (*underfitting*); o modelo não converge adequadamente.
- **50–150 épocas:** Indicado para bases de tamanho moderado, transferência de aprendizado com pesos pré-treinados e classes visualmente bem separadas. Pode ser suficiente se o mAP estabilizar cedo.
- **150–300** épocas: Ideal para bases próprias, objetos pequenos, maior variabilidade visual, imagens agrícolas ou aquáticas complexas. Favorece convergência mais completa, mas exige monitoramento contra sobreajuste.
- **Acima de 300** épocas: Aplicável a bases amplas de alta diversidade ou arquiteturas maiores. Aumenta o tempo computacional e pode elevar o sobreajuste se não houver ganho em validação.
- **Se o valor for menor:** É frequente observar perdas em queda contínua, redução da revocação (o detector deixa de localizar objetos), mAP reduzido para alvos pequenos, caixas delimitadoras imprecisas e instabilidade entre diferentes execuções (subtreinamento).
- **Se o valor for maior:** Se houver dados suficientes e regularização, o desempenho pode melhorar. Contudo, se a base for pequena ou possuir rótulos imperfeitos, o modelo pode memorizar padrões específicos (sobreajuste), reduzindo a capacidade de generalização. Os sinais de sobreajuste incluem a estagnação ou piora do mAP de validação enquanto a perda de treinamento continua caindo. A parada antecipada pode detectar o início do sobreajuste, conforme demonstram estudos clássicos na área (PRECHELT, 1998; CARUANA; LAWRENCE; GILES, 2001).
- **Paciência (*patience=200* ou *patience=50*):** O parâmetro *patience* define quantas épocas consecutivas podem transcorrer sem melhoria em uma métrica de validação antes que o processo seja interrompido (*early stopping*). O melhor modelo não é necessariamente o obtido na última época, mas aquele com melhor desempenho na validação (YAO; ROSASCO; CAPONNETTO, 2007; MAHSERECI et al., 2017). Nesta configuração, ao usar *epochs=200* e *patience=200*, o treinamento dispõe de, no máximo, 200 épocas. Para a parada antecipada ser acionada, seriam necessárias 200 épocas sem avanço, o que significa que o processo alcançará a época 200 antes da paciência surtir efeito. Isso funciona como se o *early stopping* estivesse desativado, o que é útil se o objetivo for fixar exatamente 200 épocas para comparações, mas não atua como regularização ativa.

Valores recomendados para paciência:

- **0:** Desativa a parada antecipada.
- **10–20:** Bases grandes e convergência rápida. Economiza tempo, mas pode interromper o treino prematuramente.
- **20–50:** Escolha inicial equilibrada para bases moderadas e transferência de aprendizado.
- **50–100:** Evita a parada causada por flutuações temporárias em bases ruidosas ou com mAP oscilante.

Duas alternativas metodologicamente consistentes são o uso de um teto com early stopping funcional (ex: *epochs=200*, *patience=50*) ou um treinamento rigidamente fixado (*epochs=200*, *patience=0*).

Resolução (*imgsz=640*): O parâmetro *imgsz=640* define que as imagens são preparadas para entrada na rede em dimensão aproximada de 640 × 640 pixels. Em YOLO, essa dimensão afeta diretamente a sensibilidade a objetos pequenos, o consumo de memória e a velocidade de inferência (REDMON et al., 2016; BOCHKOVSKIY; WANG; LIAO, 2020; WANG; BOCHKOVSKIY; LIAO, 2023).

O número de pixels cresce com o quadrado do lado da imagem. Tomando 640 como referência, uma entrada de 320×320 contém apenas 25% dos pixels, o que acelera o treinamento mas perde detalhes vitais. Já uma entrada de 1024 × 1024 contém 2,56 vezes mais pixels, o que pode inviabilizar o treinamento em GPUs mais limitadas. Estudos com YOLO demonstram o balanço crítico entre resolução, precisão e velocidade (ZHANG et al., 2020; RAJ; VORA; KOTECHA et al., 2021; WANG; BOCHKOVSKIY; LIAO, 2020). O valor 640 é indicado quando os objetos apresentam dimensões intermediárias, a GPU possui VRAM suficiente para um bom lote, e deseja-se comparar os resultados com os estudos padrão (como as métricas COCO (LIN; MAIRE; BELONGIE et al., 2014)). **Tamanho do lote (*batch=16*):** Define que 16 imagens são agrupadas em cada atualização dos pesos, valor usual para treinamento em GPU única. Com lotes pequenos (1–4), os gradientes estimados apresentam maior variabilidade. Com lotes grandes (32–64), o processamento é mais eficiente, mas eleva o consumo de VRAM e requer cuidado na generalização (KESKAR et al., 2017; HOFFER; HUBARA; SOUDRY, 2017; GOYAL; DOLLÁR; GIRSHICK et al., 2017; SMITH et al., 2018; MASTERS; LUSCHI, 2018). O valor 16 é defensável tecnicamente para imagens de 640px, pois evita gargalos de *Out of Memory* (OOM).

Dispositivo (*device=0*): Seleciona a primeira GPU CUDA disponível. O treinamento em CPU (*device="cpu"*) é ineficiente e desencorajado para fluxos de deep learning (JOCHER; CHAURASIA; QIU, 2023). É fundamental garantir que a GPU selecionada possua os requisitos térmicos e de memória mencionados no Capítulo 5.

Subprocessos (*workers=8*): O parâmetro *workers* determina o número de subprocessos (na CPU) responsáveis pelo carregamento, *data augmentation* e enfileiramento das imagens

para a GPU. Aumentar o valor de *workers* sobrepõe a leitura dos dados ao processamento tensorial na GPU, atenuando gargalos de E/S (*I/O bottleneck*), embora valores excessivos possam sobrecarregar a memória RAM e a CPU (PASZKE; GROSS; MASSA et al., 2019; PYTORCH CONTRIBUTORS, 2026; ULTRALYTICS, 2026). O valor 8 (ou 2 em máquinas com menos recursos) acelera o carregamento paralelo de tensores na memória sem travamentos.

As orientações operacionais da Ultralytics para *patience*, *imgsz*, *batch* e *workers* são consistentes com a interpretação apresentada: a paciência controla a tolerância sem melhora de validação, a resolução altera a precisão e complexidade, o tamanho do lote pode ser fixo ou automático conforme a VRAM, e os subprocessos aumentam a capacidade de carregamento paralelo de tensores na memória.

8.3 Configurações de Entrada, Saída e Salvamento

Além dos hiperparâmetros de treinamento, a chamada do método especifica os seguintes argumentos de controle:

Diretório de Configuração (*data*): especifica o caminho relativo para o arquivo YAML (*datasets_extraidos/10_peixes/data.yaml*), contendo o mapeamento das classes e diretórios.

- **Geração de Gráficos (*plots=True*):** habilita a geração automática de gráficos analíticos (curvas de perda, matriz de confusão e métricas F1/PR) salvos ao longo do treinamento.
- **Diretório de Destino (*project* e *name*):** especifica a pasta final (*treinamentos/yolo12/*) onde as métricas e os pesos finais (*best.pt* e *last.pt*) serão gravados.
- **Sobrescrita (*exist_ok=True*):** assegura que o diretório especificado possa ser atualizado sem a interrupção do código por erro de pasta já existente.

REFERÊNCIAS

BOCHKOVSKIY, Alexey; WANG, Chien-Yao; LIAO, Hong-Yuan Mark. YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv preprint arXiv:2004.10934, 2020. Disponível em: https://arxiv.org/abs/2004.10934.

CARUANA, Rich; LAWRENCE, Steve; GILES, C. Lee. Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping. Advances in Neural Information Processing Systems, v. 13, 2001. Disponível em: https://proceedings.neurip s.cc/paper/2000/hash/059fdcd96baeb751218b8e8b1d5b7123-Abstract.html.

GOYAL, Priya; DOLLÁR, Piotr; GIRSHICK, Ross et al. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. In: ARXIV preprint arXiv:1706.02677. [S.l.: s.n.], 2017. Disponível em: https://arxiv.org/abs/1706.02677.

HOFFER, Elad; HUBARA, Itay; SOUDRY, Daniel. Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks. In: ADVANCES in

Neural Information Processing Systems. [S.l.: s.n.], 2017. v. 30.
Disponível em: https://proceedings.neurips.cc/paper/2017/hash/a5e0ff62be0b08456fc7f1e88812af3d-Abstract.html.
JOCHER, Glenn; CHAURASIA, Ayush; QIU, Jing. Ultralytics YOLO. [S.l.: s.n.], 2023. Software de detecção, segmentação, classificação e estimativa de pose baseado na família YOLO. Disponível em: https://github.com/ultralytics/ultralytics.
KESKAR, Nitish Shirish et al. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In: INTERNATIONAL Conference on Learning Representations. [S.l.: s.n.], 2017. Disponível em: https://arxiv.org/abs/1609.04836.
LIN, Tsung-Yi; MAIRE, Michael; BELONGIE, Serge et al. Microsoft COCO: Common Objects in Context. In: EUROPEAN Conference on Computer Vision. [S.l.]: Springer, 2014. P. 740–755. DOI: 10.1007/978-3-319-10602-1_48. Disponível em:
https://doi.org/10.1007/978-3-319-10602-1_48.
MAHSERECI, Maren et al. Early Stopping without a Validation Set. arXiv preprint arXiv:1703.09580, 2017. Disponível em: https://arxiv.org/abs/1703.09580.
MASTERS, Dominic; LUSCHI, Carlo. Revisiting Small Batch Training for Deep Neural Networks. arXiv preprint arXiv:1804.07612, 2018. Disponível em: https://arxiv.org/abs/1804.07612.
PASZKE, Adam; GROSS, Sam; MASSA, Francisco et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. Advances in Neural Information Processing Systems, v. 32, 2019. Disponível em: https://proceedings.neurips.cc/paper/2 019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.
PRECHELT, Lutz. Automatic Early Stopping Using Cross Validation: Quantifying the Criteria. Neural Networks, v. 11, n. 4, p. 761–767, 1998. DOI: 10.1016/S0893-6080(98)00010-0. Disponível em: https://doi.org/10.1016/S0893-6080(98)00010-0.
PYTORCH CONTRIBUTORS. Data Loading Optimization in PyTorch. [S.l.: s.n.], 2026. Disponível em: https://docs.pytorch.org/tutorials/intermediate/intermediate_data_loading_tutorial.html. Documentação técnica sobre otimização de DataLoader, subprocessos e carregamento paralelo.
RAJ, Rishabh; VORA, Sanjay; KOTECHA, Ketan et al. Malaria Parasite Detection in Thick Blood Smear Microscopic Images Using Modified YOLOv3 and YOLOv4 Models. BMC Bioinformatics, v. 22, n. 1, p. 112, 2021. DOI:
10.1186/s12859-021-04036-4. Disponível em: https://doi.org/10.1186/s12859-021-04036-4.
REDMON, Joseph 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. P. 779–788.
SMITH, Samuel L. et al. Don't Decay the Learning Rate, Increase the Batch Size. In: INTERNATIONAL Conference on Learning Representations. [S.l.: s.n.], 2018.
Disponível em: https://arxiv.org/abs/1711.00489.
ULTRALYTICS. Ultralytics YOLO Configuration and Hyperparameters. [S.l.: s.n.], 2026.

Disponível em: https://docs.ultralytics.com/usage/cfg/. Documentação técnica dos parâmetros de treinamento, incluindo epochs, patience, imgsz, batch, device e workers.
WANG, Chien-Yao; BOCHKOVSKIY, Alexey; LIAO, Hong-Yuan Mark. Scaled-YOLOv4: Scaling Cross Stage Partial Network. arXiv preprint arXiv:2011.08036, 2020. Disponível em: https://arxiv.org/abs/2011.08036.
WANG, Chien-Yao; BOCHKOVSKIY, Alexey; LIAO, Hong-Yuan Mark. YOLOv7: Trainable Bag-of-Freebies Sets New State-of-the-Art for Real-Time Object Detectors. In: PROCEEDINGS of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. [S.l.: s.n.], 2023. P. 7464–7475. DOI: 10.1109/CVPR52729.2023.00721.
Disponível em: https://doi.org/10.1109/CVPR52729.2023.00721. YAO, Yuan; ROSASCO, Lorenzo; CAPONNETTO, Andrea. On Early Stopping in Gradient Descent Learning. Constructive Approximation, v. 26, n. 2, p. 289–315, 2007. DOI: 10.1007/s00365-006-0663-2. Disponível em: https://doi.org/10.1007/s00365-006-0663-2.
ZHANG, Xiaoguo et al. Improve YOLOv3 Using Dilated Spatial Pyramid Module for Multi-Scale Object Detection. International Journal of Advanced Robotic Systems, v. 17, n. 4, 2020. DOI: 10.1177/1729881420936062. Disponível em: https://doi.org/10.1177/1729881420936062.