Universidade de Brasília - UnB
Faculdade de Ciências e Tecnologias em Engenharia - FCTE
Engenharia de Software

Análise Comparativa de Arquiteturas Convolucionais e Baseadas em Atenção para Detecção Automatizada de Parasitas Intestinais

Autor: Eduardo Vieira Lima
Orientador: Dr. Vinicius de Carvalho Rispoli

Brasília, DF
julho de 2025

Eduardo Vieira Lima

Análise Comparativa de Arquiteturas Convolucionais e Baseadas em Atenção para Detecção Automatizada de Parasitas Intestinais/ Eduardo Vieira Lima. – Brasília, DF, julho de 2025-

97 p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. Vinicius de Carvalho Rispoli

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade de Ciências e Tecnologias em Engenharia - FCTE , julho de 2025.

1. Redes neurais. 2. Vision Transformer. I. Dr. Vinicius de Carvalho Rispoli. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Análise Comparativa de Arquiteturas Convolucionais e Baseadas em Atenção para Detecção Automatizada de Parasitas Intestinais

CDU 02:141:005.6

Eduardo Vieira Lima

Análise Comparativa de Arquiteturas Convolucionais e Baseadas em Atenção para Detecção Automatizada de Parasitas Intestinais

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Dr. Vinicius de Carvalho Rispoli
Orientador

Dr.Ronni Geraldo Gomes de Amorim

Dr. Marcus Vinicius Chaffim Costa

Brasília, DF
julho de 2025

Agradecimentos

Tenho que começar agradecendo à minha noiva, Thaís, por ser a minha maior fonte de apoio há 8 anos. Não chegaria até aqui de outra forma, sou muito grato por dividir a vida com você.

A meus irmãos, Marcela, Luisa, Gustavo e José Henrique. Grande parte da minha determinação e ambição vem da vontade de inspirar e apoiar vocês, não pude desistir enquanto tive isso em mente.

Também ao meu amigo João Marcelo, que foi essencial para minha trajetória desde o ensino médio e principalmente durante toda a graduação. Sem sua ajuda de Cálculo 1 até este trabalho, eu jamais conseguiria chegar aqui.

Importante não deixar de agradecer à minha família, em especial meu pai Rubens e minha avó Nenza, por terem me incentivado e apoiado durante toda minha vida. Vocês foram grandes inspirações para mim.

Ao meu orientador, Vinicius de Carvalho Rispoli, que me acolheu para a execução deste trabalho quando eu já não sabia se iria conseguir.

Não obstante, a todos os meus outros amigos que estiveram comigo nessa jornada, em especial Luís e Samuel, que estiveram comigo desde o início e seguem comigo hoje.

A todo o corpo docente e de funcionários da UnB, que diariamente se esforçam pelo ensino de excelência nas universidades públicas.

Resumo

A detecção e classificação de parasitas intestinais através de microscopia óptica representam um desafio significativo na área médica, especialmente em regiões com recursos limitados, sendo a análise manual o padrão atual. Este trabalho se propôs a analisar e comparar as principais soluções de classificação automática no campo da visão computacional, redes convolucionais e transformadores visuais, apresentando também uma abordagem híbrida entre elas. Foram exploradas soluções que priorizam a eficiência computacional: *EfficientNetV2-S* para a rede convolucional e *Tiny ViT* para o transformador visual, tendo em vista cenários com recursos limitados. Foi utilizado para treinamento e avaliação um dos maiores conjuntos de imagens microscópicas de parasitas disponível publicamente, o *Chula-ParasiteEgg-11*, contendo 2.200 imagens de 11 espécies distintas de parasitas. Os resultados obtidos demonstram que as soluções analisadas são viáveis e possuem acurácia satisfatória, sendo a rede convolucional a que melhor performa em termos de eficiência e acurácia, seguida do modelo híbrido e então do transformador visual. Observou-se que as soluções possuem resultados razoavelmente inferiores aos modelos mais robustos da literatura, porém utilizam significativamente menos recursos computacionais, o que as torna viáveis para situações em que o recurso é limitado.

Palavras-chaves: Aprendizado profundo. Classificação de parasitas. Redes convolucionais. Transformadores visuais. Modelo híbrido.

Abstract

The detection and classification of intestinal parasites through optical microscopy represents a significant challenge in the medical field, especially in resource-limited regions, where manual analysis remains the current standard. This work aimed to analyze and compare the main automatic classification solutions in the field of computer vision, convolutional networks and visual transformers, also presenting a hybrid approach between them. Solutions that prioritize computational efficiency were explored: *EfficientNetV2-S* for the convolutional network and *Tiny ViT* for the visual transformer, considering scenarios with limited resources. One of the largest publicly available microscopic parasite image datasets was used for training and evaluation, the *Chula-ParasiteEgg-11*, containing 2,200 images of 11 distinct parasite species. The obtained results demonstrate that the analyzed solutions are viable and have satisfactory accuracy, with the convolutional network performing best in terms of efficiency and accuracy, followed by the hybrid model and then the visual transformer. It was observed that the solutions have reasonably inferior results compared to the more robust models in the literature, however they use significantly fewer computational resources, making them viable for situations where resources are limited.

Key-words: Deep learning. Parasite classification. Convolutional networks. Visual transformers. Hybrid model.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CNN	*Convolutional Neural Network* (Rede Neural Convolucional)
CPU	*Central Processing Unit* (Unidade Central de Processamento)
GPU	*Graphics Processing Unit* (Unidade de Processamento Gráfico)
HSV	*Hue-Saturation-Value* (Matiz-Saturação-Valor)
IA	Inteligência Artificial
RGB	*Red, Green, Blue* (Vermelho, Verde, Azul)
ReLU	*Rectified Linear Unit* (Unidade Linear Retificada)
GELU	*Gaussian Error Linear Unit* (Unidade Linear de Erro Gaussiano)
ViT	*Vision Transformer* (Transformador Visual)
ms	Milissegundos
µm	Micrômetros

Lista de símbolos

α	Coeficiente de profundidade
β	Coeficiente de largura
γ	Coeficiente de resolução
ϕ	Coeficiente composto de escalonamento
$\mathcal{L}$	Função de perda
$\sum$	Soma
$\log(x)$	Logaritmo natural
$\mathbb{R}$	Conjunto dos números reais
$\mathbf{x}$	Vetor de entrada
$\mathbf{y}$	Vetor de saída
$\mathbf{W}$	Matriz de pesos
$\mathbf{b}$	Vetor de *bias*
$\in$	Pertence a

Sumário

Introdução

Inteligência Artificial (IA) é um ramo da tecnologia que pode ser descrito como o esforço para automatizar tarefas intelectuais normalmente executadas por humanos Chollet (2021). Esse campo tem evoluído rapidamente nas últimas décadas, impulsionado pelo crescimento exponencial na capacidade computacional, disponibilidade de grandes volumes de dados e avanços em algoritmos de aprendizado de máquina. Nos últimos anos, a IA tem sido objeto de intenso interesse geral, expandindo-se para contextos além das áreas tradicionais de tecnologia, como medicina, finanças, indústria automotiva e segurança cibernética Chollet (2021).

Dentre os principais campos da IA, o *Machine Learning* (ML) se destaca por permitir que sistemas aprendam padrões e tomem decisões com base em dados Mitchell (1997). Essa abordagem elimina a necessidade de programação explícita para cada cenário, tornando possível a adaptação e melhoria contínua dos modelos conforme novos dados são incorporados. Dentro do ML, o Aprendizado Profundo (do inglês, *Deep Learning*), que utiliza redes neurais profundas, tem impulsionado avanços significativos, especialmente no processamento de imagens e linguagem natural, devido à sua capacidade de identificar padrões complexos em grandes volumes de dados.

As Redes Neurais Convolucionais (CNNs) representam um dos avanços mais importantes no campo do *Deep Learning*. Inspiradas no funcionamento do córtex visual, as CNNs são arquiteturas especializadas para processamento de imagens, utilizando camadas convolucionais para extrair características relevantes, reduzindo a necessidade de intervenção manual na definição de características. Essas redes têm sido amplamente empregadas em tarefas como reconhecimento facial, diagnóstico por imagem e veículos autônomos, permitindo classificações e segmentações com alta precisão Lecun et al. (1998).

Mais recentemente, os *Vision Transformers* (ViTs) surgiram como uma alternativa promissora às CNNs para visão computacional. Diferentemente das arquiteturas tradicionais baseadas em convolução, os ViTs utilizam mecanismos de autoatenção para capturar relações globais entre pixels em imagens. Essa abordagem permite um aprendizado mais eficiente de representações visuais e, em muitos casos, supera o desempenho das CNNs em diversas tarefas específicas Dosovitskiy et al. (2020). O avanço dos ViTs tem ampliado significativamente o potencial das aplicações de IA, trazendo novas possibilidades para reconhecimento de padrões visuais e reforçando a importância dos modelos baseados em atenção na área de aprendizado profundo.

No campo da medicina, as infecções parasitárias intestinais continuam sendo um problema significativo de saúde pública, especialmente em países em desenvolvimento e regiões com infraestrutura sanitária deficiente. Estima-se que cerca de 24% da população mundial seja afetada por doenças infecciosas e parasitárias, com impacto predominante

em crianças, gestantes e pessoas com baixa imunidade Kumar et al. (2023). Essas infecções podem resultar em sintomas como diarreia, desnutrição, anemia, fraqueza e comprometimento do desenvolvimento infantil.

O diagnóstico tradicional dessas infecções é realizado por meio de exames microscópicos de amostras fecais, considerados o padrão-ouro por sua capacidade de identificar diretamente ovos e parasitas. Contudo, esse processo é trabalhoso, demorado, exige pessoal altamente capacitado e está sujeito a variações na acurácia devido à subjetividade da análise e às variações morfológicas dos parasitas Xu et al. (2024). Além disso, o ambiente de trabalho pode ser insalubre, com baixa eficiência e alto volume de trabalho para os profissionais de laboratório.

Nesse cenário, a aplicação de modelos baseados em IA tem-se mostrado uma alternativa promissora para automatizar e melhorar o processo de detecção e classificação de parasitas intestinais. A integração de tecnologias de processamento de imagens digitais com métodos de aprendizado profundo, especialmente com redes neurais convolucionais e, mais recentemente, com Vision Transformers, tem impulsionado o desenvolvimento de sistemas automáticos capazes de identificar com maior precisão e agilidade os agentes infecciosos presentes em imagens microscópicas Kumar et al. (2023).

Diversos trabalhos prévios têm explorado o uso de aprendizado profundo para essa tarefa. Kumar et al. (2023) propuseram uma abordagem utilizando YOLOv5, demonstrando elevada acurácia (cerca de 97%) e alta velocidade de detecção em um conjunto de dados com mais de cinco mil imagens de parasitas intestinais. O estudo destacou a viabilidade do uso de modelos leves em ambientes com poucos recursos computacionais, voltando-se ao diagnóstico clínico em tempo real.

Xu et al. (2024) desenvolveram a YAC-Net, uma arquitetura leve baseada em YOLOv5, com otimizações estruturais específicas para imagens de ovos parasitários. Os autores demonstraram que sua abordagem melhora a detecção mesmo em imagens com baixa resolução e com ruídos, reduzindo o número de parâmetros e o custo computacional, o que viabiliza sua aplicação em regiões remotas.

Outro trabalho relevante foi apresentado por AlDahoul et al. (2023), que explorou a combinação de redes convolucionais com mecanismos de atenção por meio da arquitetura CoAtNet, demonstrando que a combinação dessas abordagens pode gerar resultados superiores na classificação de ovos parasitários. O estudo também comparou diferentes abordagens de CNNs e Vision Transformers em um conjunto diverso de imagens, incluindo variações de iluminação e resolução.

Apesar das contribuições relevantes dessas pesquisas, poucas investigações exploram comparativamente a aplicação de CNNs e Vision Transformers de maneira sistemática e com base em um mesmo conjunto de dados para a detecção e classificação de parasitas intestinais. Neste contexto, esta monografia tem como objetivo investigar e comparar o desempenho de modelos baseados em CNNs e Vision Transformers na tarefa de detec-

ção e classificação de parasitas intestinais em imagens microscópicas, contribuindo para o desenvolvimento de ferramentas automatizadas e eficientes para a saúde pública.

1 Referencial Teórico

Este capítulo apresenta a fundamentação teórica deste trabalho. Inicia-se introduzindo o conceito de Aprendizado de Máquina (*Machine Learning*) e Aprendizado Profundo (*Deep Learning*). Em seguida, apresenta as arquiteturas de Aprendizado de Máquina abordadas neste trabalho, Redes Neurais Convolucionais (*Convolutional Neural Networks* ou CNNs) e Transformadores Visuais (*Vision transformers* ou ViTs). Então, apresenta uma visão geral sobre a Visão Computacional aplicada à detecção de parasitas intestinais, assim como o contexto geral do problema para o campo da medicina e a importância das soluções exploradas neste trabalho. Por fim, apresenta brevemente os trabalhos relacionados relevantes para este contexto.

1.1 Aprendizado de Máquina

Os programas construídos em Aprendizado de Máquina possuem, de forma geral, três componentes: dados de entrada, exemplos da saída esperada e um meio de quantificar a qualidade das saídas geradas Chollet (2021).

- Dados de entrada: são o objeto básico do problema. Por exemplo, os dados de entrada para um sistema de classificação de imagens seriam figuras ainda não classificadas.
- Exemplos da saída esperada: seguindo o exemplo anterior, seria um conjunto de imagens corretamente classificadas, como“gato” ou “cachorro”.
- um meio de quantificar a qualidade das saídas geradas: é o principal em um processo de aprendizado de máquina. Por mensurar a qualidade dos resultados da tarefa que realiza, o sistema é capaz de corrigir-se e ajustar como seu algoritmo funciona. Este processo de ajuste é o que caracteriza o aprendizado.

O processo principal de um modelo de aprendizado de máquina é transformar os dados de entrada em saídas significativas para o problema que se propõe a solucionar. Este processo é aprendido através da exposição a exemplos conhecidos de entrada e saída esperada. Assim, o problema central torna-se encontrar maneiras de transformar o dado de entrada para se aproximar da saída desejada, em outras palavras, aprender representações do dado de entrada que sejam úteis para alcançar a saída esperada. Por exemplo, um programa que tem como objetivo identificar a presença da cor vermelha em uma imagem poderia representar esta imagem no formato RGB (do inglês, *red-green-blue*). Por sua vez, se o objetivo for saturar as cores de uma imagem, o melhor seria representá-la no formato HSV (do inglês, *hue-saturation-value*) Chollet (2021).

Transformar os dados de entrada em representações relevantes não é um processo simples, pois existem numerosas formas de representar um dado e poucas serão úteis para se aproximar do resultado desejado. Assim, os programas de aprendizado de máquina utilizam um conjunto de transformações pré-definidas que melhor se aproximam do contexto do problema, utilizando o processo de quantificar a qualidade das saídas para avaliar o quão relevante é cada transformação dentro deste conjunto. Este conjunto pré-definido é chamado de espaço de hipótese Chollet (2021). Em suma, o processo de aprendizado de máquina é composto por: encontrar transformações de dados de entrada, dentro de um conjunto pré-definido de possibilidades, utilizando como guia a qualidade das saídas geradas e retroalimentando o sistema com cada resultado gerado para se aproximar cada vez mais da saída desejada. Esta simples ideia possibilita a execução de diversas tarefas Chollet (2021).

O aprendizado profundo (do inglês *Deep Learning*) é uma área do aprendizado de máquina que tem ganhado força e crescido exponencialmente desde os anos 90. O foco desta abordagem está na transformação dos dados de entrada, introduzindo camadas consecutivas de transformações cada vez mais significativas. O "profundo" em aprendizado profundo refere-se à quantidade de camadas de transformação, o número de camadas representa a "profundidade" do modelo. Modelos modernos de aprendizado profundo podem incluir dezenas ou centenas de camadas, todas incluídas automaticamente pelo processo de aprendizado. Esta é a principal diferença entre um modelo tradicional, que geralmente foca em trabalhar com uma ou duas camadas de transformação Chollet (2021).

As transformações em cada camada são mediadas por pesos (ou parâmetros), que são ajustados durante o aprendizado. Os pesos de uma camada representam o que a camada faz em termos da transformação dos dados, quantificado numericamente. Este ajuste ocorre comparando a saída do algoritmo com a saída real através de uma função de perda; esta função representa numericamente o quão distante a saída gerada está da saída real desejada, gerando a perda. O resultado é então retroalimentado aos pesos por meio de um otimizador, buscando minimizar a perda. O programa inicia com valores aleatórios para os pesos e busca otimizar seus valores através do processo de aprendizado retroalimentado Chollet (2021).

O aprendizado de máquina pode ser classificado em três principais categorias: supervisionado, não supervisionado e por reforço. No aprendizado supervisionado, o modelo é treinado com um conjunto de dados rotulado, aprendendo a mapear entradas para saídas com base em exemplos prévios, sendo amplamente utilizado em tarefas como classificação e regressão Goodfellow, Bengio e Courville (2016). Já o aprendizado não supervisionado trabalha com dados sem rótulos, buscando identificar padrões ou estruturas ocultas nos dados, como agrupamentos ou redução de dimensionalidade Murphy (2012). Por fim, o aprendizado por reforço envolve a interação de um agente com um ambiente, aprendendo por meio de recompensas e penalidades a tomar decisões que maximizem um retorno

cumulativo ao longo do tempo Sutton e Barto (2015). Esses três paradigmas oferecem abordagens distintas para a construção de modelos inteligentes, sendo escolhidos conforme a natureza do problema e dos dados disponíveis.

O aprendizado profundo está incluído dentro do que é conhecido como redes neurais, uma subcategoria do aprendizado de máquina. O termo rede neural se refere à neurobiologia, inspirado pelo entendimento de como um cérebro funciona (principalmente o córtex visual), onde as camadas de transformações funcionam como neurônios. É importante destacar que, apesar da inspiração, os modelos de aprendizado profundo não são modelos do funcionamento do cérebro, tendo em vista que não há nenhuma evidência de que o cérebro utilize algo semelhante aos mecanismos e estratégias utilizados no aprendizado profundo Chollet (2021). Este trabalho utilizará redes neurais alinhadas com aprendizado supervisionado para a detecção e classificação dos principais parasitas intestinais, a partir de exames de fezes digitalizados.

1.2 Redes Neurais

No centro de todas as operações e representações de dados em redes neurais estão os tensores. O tensor é a estrutura fundamental em sistemas de aprendizado de máquina, capaz de armazenar dados e suas transformações de forma numérica em múltiplas dimensões, como escalares, vetores e matrizes Chollet (2021). Todas as operações dentro de uma rede neural, incluindo as transformações lineares (como produtos escalares e adições), são operações entre tensores, e os próprios parâmetros do modelo, como pesos e viés, são representados como tensores Paszke et al. (2019).

Toda rede neural é fundamentalmente um encadeamento de camadas, entre seus tensores de entrada e saída Chollet (2021). Estas camadas que existem entre a entrada e a saída de uma rede são conhecidas como camadas ocultas (do inglês *hidden layer*). Uma rede neural pode ser classificada como rasa ou profunda a partir de seu número de camadas ocultas, sendo a rede rasa uma que possua uma única camada oculta e as redes profundas são as que possuem múltiplas Paszke et al. (2019). A Figura 1 representa um exemplo de rede neural rasa, contendo apenas uma única camada oculta, enquanto a Figura 2 apresenta uma rede profunda, com múltiplas camadas ocultas.

Cada camada atua como um bloco de construção fundamental, extraindo representações mais significativas dos dados a cada estágio Chollet (2021). Estas camadas são tipicamente vetoriais e são frequentemente chamadas de neurônios ou unidades ocultas Prince (2024), que agem em paralelo representando uma função de vetor para escalar, e na prática realizam uma transformação linear na entrada (envolvendo os pesos e viés) Paszke et al. (2019).

Outro ponto fundamental para as redes neurais são as funções de ativação. Estas funções são cruciais para que seja possível que a rede neural possa lidar com relações de

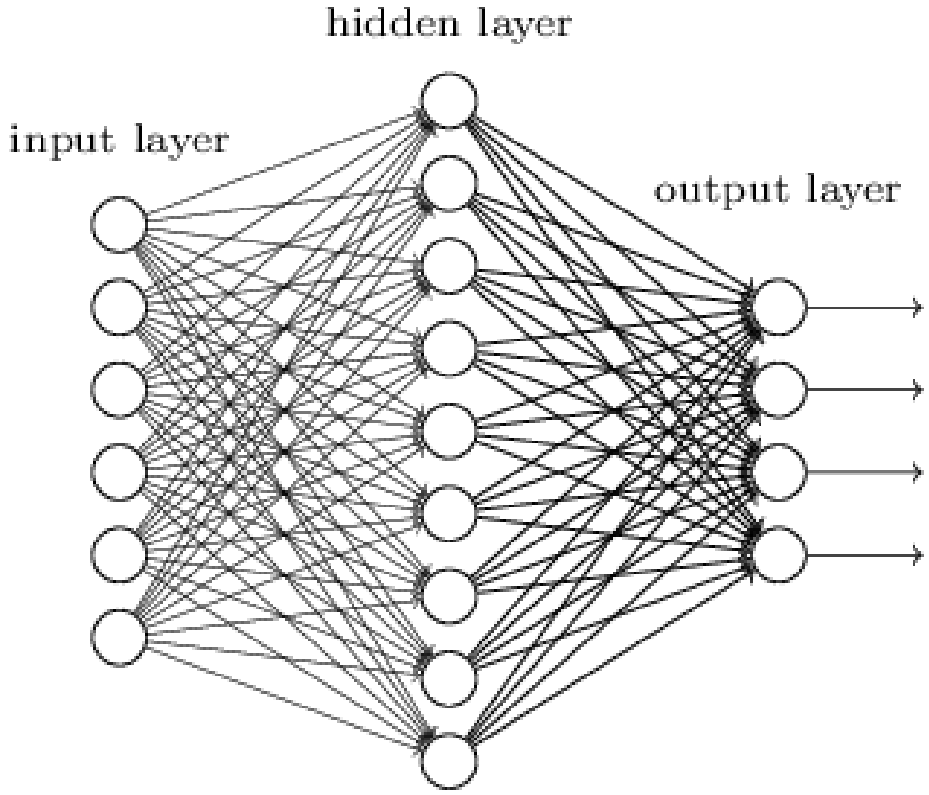

Figura 1 – Ilustração de um modelo de rede neural rasa com uma camada de entrada, saída e uma única camada oculta intermediária. Fonte: Nielsen (2015)

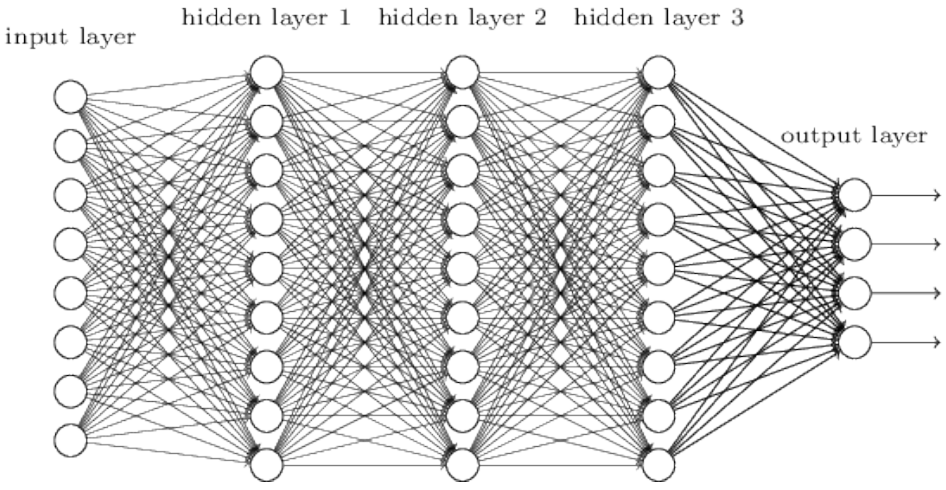

Figura 2 – Ilustração de um modelo de rede neural profunda com uma camada de entrada, saída e múltiplas camadas ocultas intermediárias. Fonte: Nielsen (2015)

dados mais complexas e não lineares Paszke et al. (2019). Sem uma função de ativação, uma sequência de operações lineares (produto escalar e adição) resultaria em um modelo que seria, em sua essência, linear, independentemente do número de camadas Chollet (2021). As funções de ativação introduzem a não linearidade que permite que a rede se aproxime de uma vasta gama de funções complexas. Existe uma grande variedade de funções de ativação que já são provadas com sucesso, assim, a escolha de uma função de ativação varia de acordo com o problema abordado pela rede neural Paszke et al. (2019). A Figura 3 apresenta as funções de ativação mais comumente utilizadas.

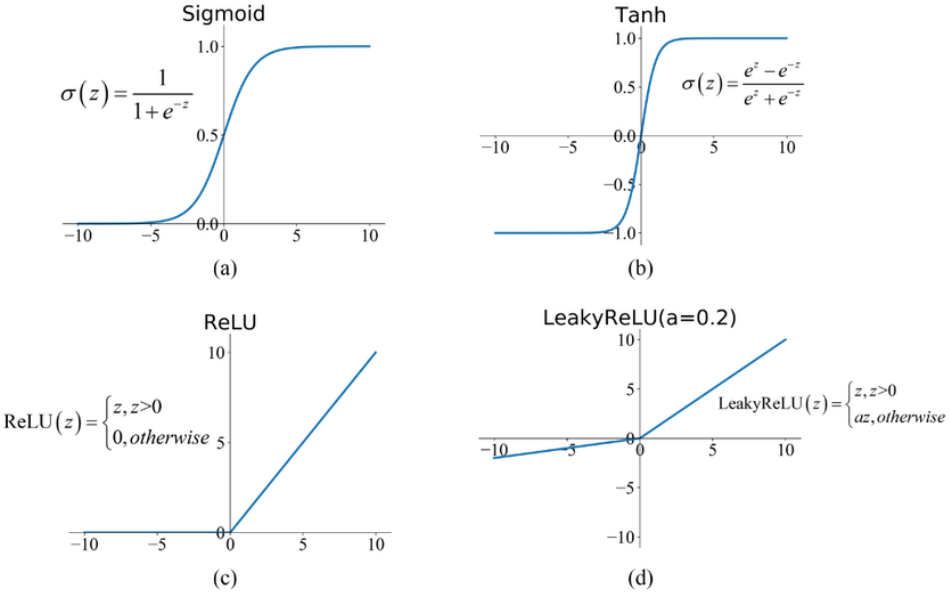

Figura 3 – Gráfico que representa as funções de ativação mais comumente utilizadas. Cada função tem o papel de introduzir a não-linearidade ao modelo, a escolha de uma varia de acordo com a necessidade do problema abordado. Fonte: Leppich (2021)

- ReLU (*Rectified Linear Unit*): é considerada a escolha mais comum e uma das funções de ativação de propósito geral com melhor desempenho Prince (2024). Ela retorna o valor de entrada se for positivo e zero caso contrário, efetivamente equalizando os valores negativos a zero. Possui derivada 1 para entradas positivas e 0 para entradas negativas, o que contribui para a estabilidade e eficiência do treinamento Prince (2024).

- Leaky ReLU: uma variação da função ReLU. Permite uma pequena inclinação positiva para valores negativos (tipicamente 0.01 ou 0.1) para mitigar o problema do "ReLU morrendo". Este problema ocorre por uma característica inerente à função ReLU: sua derivada é sempre zero para entradas negativas. Isto faz com que, se todos os exemplos de treinamento produzirem uma entrada negativa para a função ReLU, então não será possível ajustar os parâmetros das camadas, já que sempre será zero. Prince (2024).

- Sigmoid (função logística): transforma os valores de entrada para o intervalo definido de 0 a 1. Foi amplamente utilizada nos primórdios do aprendizado profundo, mas atualmente é mais usada quando a saída precisa ser interpretada como uma probabilidade Paszke et al. (2019).

- Tanh (tangente hiperbólica): funciona de forma similar à função *Sigmoid*, porém seu intervalo é de -1 a 1 Paszke et al. (2019).

1.2.1 Redes neurais convolucionais

Rede neural convolucional é um tipo de rede neural utilizada principalmente para tarefas de visão computacional. A diferença fundamental entre redes tradicionais e uma rede convolucional está na estrutura de camada. Uma camada tradicional aprende padrões globais a partir da entrada de dados, enquanto uma camada convolucional aprende padrões locais, no caso de imagens, padrões encontrados em pequenos trechos bidimensionais das entradas Chollet (2021).

Esta característica dá às redes convolucionais duas propriedades importantes:

- Após aprender um padrão em algum trecho de uma imagem, a rede é capaz de reconhecer esse padrão em qualquer lugar. Uma camada tradicional teria que aprender este padrão novamente caso aparecesse em algum outro trecho Chollet (2021). Isto torna as redes convolucionais especialmente efetivas no uso de dados em tarefas de visão computacional, precisando de uma base de dados de treinamento menor Chollet (2021).

- Redes convolucionais são capazes de aprender hierarquias espaciais. Uma primeira camada pode aprender pequenos padrões locais, como vértices, uma segunda camada subsequente pode aprender padrões mais gerais a partir das características da primeira camada e assim sucessivamente. Isto permite o aprendizado de conceitos visuais profundos e complexos, já que o mundo visual funciona fundamentalmente em uma lógica de hierarquia espacial Chollet (2021). A Figura 4 apresenta um exemplo simples de como a hierarquia espacial funciona no aprendizado de máquina.

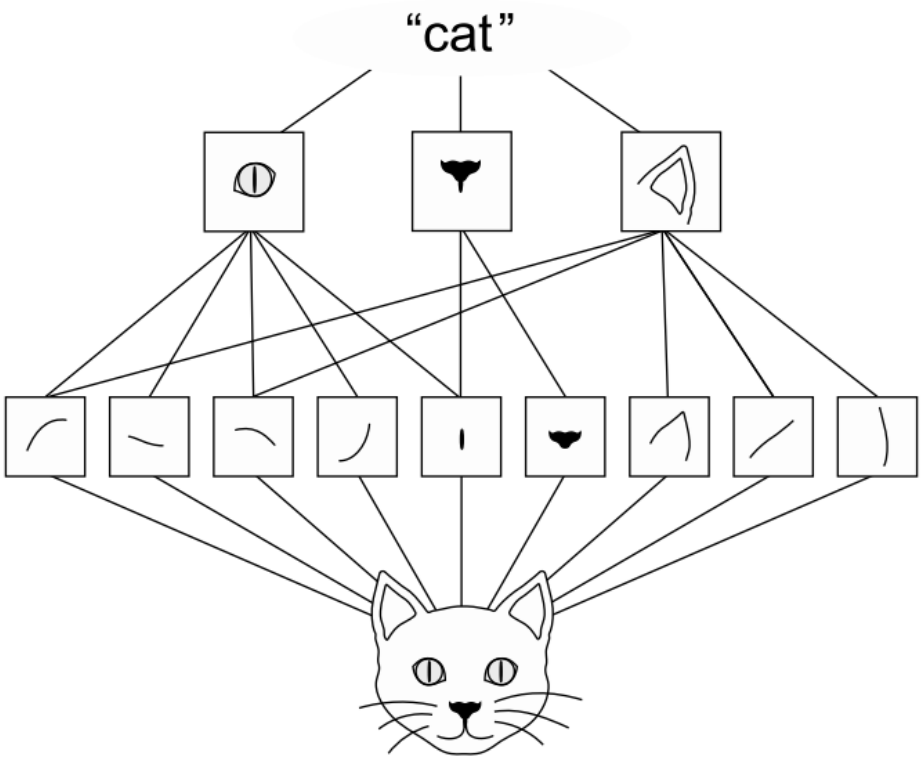

Figura 4 – Representação visual de como uma rede convolucional identifica padrões para, neste caso, reconhecer a imagem de um gato (*cat*, em inglês). De baixo para cima na imagem, linhas elementares ou texturas combinam-se em objetos simples como olhos ou orelhas, que então combinam em conceitos mais complexos como “gato”. Assim funciona a hierarquia espacial. Fonte: Chollet (2021)

1.3 Transformadores visuais

Arquiteturas baseadas em atenção tornaram-se o modelo dominante para o processamento de linguagem natural, causando uma revolução nos modelos linguísticos de inteligência artificial. Em especial, os transformadores são a principal solução de arquitetura baseada em atenção Dosovitskiy et al. (2020).

O transformador foi apresentado no artigo Vaswani et al. (2017). Antes disso, o modelo dominante para modelos linguísticos era uma abordagem de redes neurais convolucionais que incluía um codificador e um decodificador através de um mecanismo de atenção Vaswani et al. (2017). A ideia principal do transformador é focar nos mecanismos de atenção, dispensando completamente as abordagens convolucionais. De forma simplificada, uma função de atenção pode ser descrita como uma função que mapeia uma consulta e um conjunto de pares chave-valor para uma saída, onde a consulta, chaves, valores e saída são vetores. A saída é então computada como uma soma ponderada dos valores, onde o peso designado para cada valor é computado por uma função de compatibilidade da consulta com a chave correspondente Vaswani et al. (2017).

Explorando diferentes abordagens baseadas puramente em atenção, o trabalho Vaswani et al. (2017) alcançou resultados superiores às arquiteturas convolucionais, até então dominantes no campo de modelos linguísticos. Inspirado nisso, nascem os transformadores visuais, com a proposta de abordar a computação visual com a mesma ideia de focar nos modelos de atenção e deixar de lado as arquiteturas convolucionais.

Para utilizar o conceito de transformadores em imagens, a imagem é dividida em blocos e projetada linearmente como entradas para um transformador. Os blocos de imagem codificados são tratados da mesma forma que palavras são tratadas em transformadores de linguagem natural Dosovitskiy et al. (2020). A Figura 5 ilustra a visão geral da estrutura de um transformador visual.

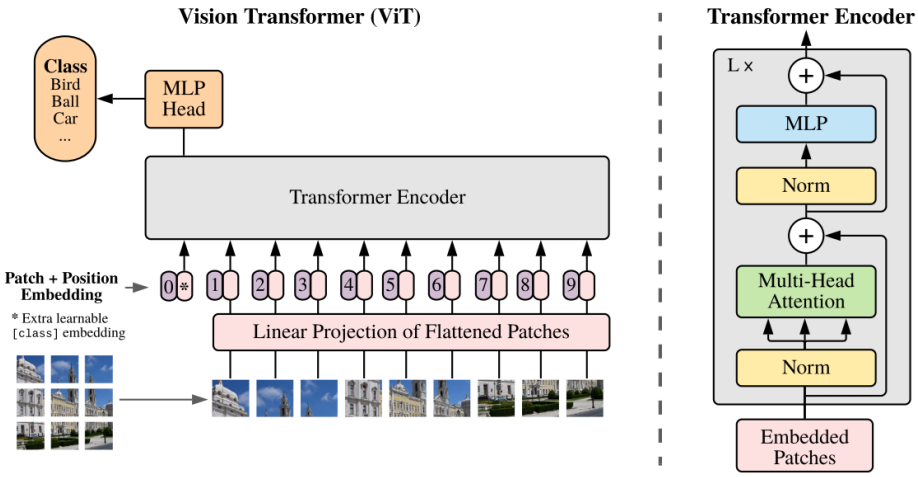

Figura 5 – Visão geral do transformador visual. A imagem é dividida em blocos de tamanho fixo, que são então submetidos a uma projeção linear com informação de posição e alimentados a um codificador para um transformador padrão. Fonte: Dosovitskiy et al. (2020)

O transformador padrão recebe como entrada uma sequência unidimensional de vetores projetados dos blocos. Para tratar imagens bidimensionais, a imagem $x \in \mathbb{R}^{H\times W\times C}$ é reestruturada em uma sequência de blocos bidimensionais achatados $x_p \in \mathbb{R}^{N\times(P^2\cdot C)}$, onde (H, W) representa a resolução da imagem original, C é o número de canais de imagem, (P, P) corresponde à resolução de cada bloco da imagem, o número total de blocos é então dado por

$$N = \frac{H \cdot W}{P^2}, \tag{1.1}$$

que também define o comprimento efetivo da sequência de entrada para o transformador Dosovitskiy et al. (2020).

Os canais de imagem representam as diferentes componentes de cor ou características da imagem. Em imagens coloridas no formato *RGB*, existem três canais, cada um representando a intensidade de uma cor primária. Em imagens em escala de cinza, há apenas um canal representando a intensidade luminosa. Em contextos de processamento

de imagem médica, podem existir canais adicionais representando diferentes técnicas de coloração ou modalidades de imagem. O conceito de canais é fundamental para o processamento de imagens, pois permite que a rede neural aprenda padrões específicos de cada componente da imagem separadamente.

Adiciona-se no início da sequência um vetor aprendido, denominado bloco de classificação, representado como

$$z_0^0 = x_{\text{class}}, \tag{1.2}$$

cuja saída final, após o processamento pelo codificador do transformador, é utilizada como representação da imagem. Tanto no pré-treinamento quanto no ajuste fino, um cabeçalho de classificação é acoplado à saída z_L^0 correspondente a esse bloco. Esse cabeçalho é implementado por uma rede *perceptron* multicamadas (MLP) com uma camada oculta durante o pré-treinamento e por uma camada linear simples durante o ajuste fino Dosovitskiy et al. (2020).

Informações posicionais são incorporadas aos vetores dos blocos para preservar a relação espacial da imagem. São utilizadas informações posicionais unidimensionais aprendíveis, uma vez que não foram observados ganhos de desempenho significativos ao empregar técnicas mais avançadas com consciência bidimensional. A sequência resultante de vetores serve como entrada para o codificador do transformador Dosovitskiy et al. (2020).

O transformador mantém um tamanho constante D para os vetores latentes em todas as suas camadas. Dessa forma, os blocos são achatados e projetados para esse espaço vetorial de dimensão D por meio de uma projeção linear treinável, representada como

$$x_p \in \mathbb{R}^{N\times(P^2\cdot C)} \xrightarrow{\text{Projeção Linear}} \mathbb{R}^{N\times D}, \tag{1.3}$$

cuja saída corresponde aos vetores projetados dos blocos, que são efetivamente a entrada do codificador do transformador Dosovitskiy et al. (2020).

Quando treinado em bases de dados de tamanho médio e sem uma forte regularização, o transformador visual alcança uma acurácia modesta e um pouco abaixo dos modelos convolucionais. Porém, em grandes bases de dados, o transformador visual alcança resultados iguais ou superiores aos modelos convolucionais e utiliza significativamente menos recursos computacionais Dosovitskiy et al. (2020).

1.4 Visão computacional aplicada à detecção de parasitas

Infecções intestinais parasitárias são as infecções mais comuns que afetam as comunidades mais pobres e necessitadas no mundo. Essas infecções são amplamente distribuídas na África Subsaariana, China e leste asiático Gujo e Kare (2021). Mais de 1,5 bilhões de

pessoas são infectadas por helmintos transmitidos pelo solo, como *Ascaris lumbricoides* e *Trichuris trichiura*, destes, 267 milhões são crianças com menos de 5 anos de idade Gujo e Kare (2021).

O método padrão para diagnosticar infecções por parasitas intestinais é baseado na análise microscópica manual de amostras de fezes, com técnicas como o exame direto, Kato-Katz e testes moleculares (qPCR, ELISA, imunofluorescência) Yimer et al. (2015). Os testes moleculares e Kato-Katz são técnicas que possuem uma sensibilidade superior ao exame direto, alcançando acima de 90% de sensibilidade contra a sensibilidade de 48,9% a 63,1% do método direto. Porém, esses métodos de diagnóstico são significativamente laboriosos e onerosos, além de altamente suscetíveis a erros humanos Yimer et al. (2015).

A visão computacional entra como uma alternativa para os processos manuais de diagnóstico, visando mitigar as limitações de recurso e mão de obra inerentes aos métodos manuais. Inicialmente, a automação do processo de detecção de parasitas era limitada pela falta de algoritmos de reconhecimento para microrganismos. Porém, o avanço do aprendizado de máquina e, em especial, o aprendizado profundo trouxeram uma revolução no campo de identificação e classificação de imagens na medicina, destacando as redes neurais convolucionais Kumar et al. (2023).

2 Metodologia

2.1 Dados

O experimento deste trabalho utilizou o conjunto de dados *Chula-ParasiteEgg-11* Palasuwan et al. (2022). Esses dados foram disponibilizados publicamente na plataforma *IEEE Dataport* como proposta para a competição ICIP 2022 Anantrasirichai et al. (2022), cujo objetivo foi incentivar soluções para a detecção e classificação de ovos de parasitas intestinais. Os dados disponibilizados representam um dos maiores conjuntos de dados disponíveis publicamente para a classificação de parasitas intestinais, contendo 11 espécies distintas de parasitas com uma distribuição balanceada entre eles.

2.1.1 Características gerais do conjunto de dados

O conjunto de dados utilizado neste trabalho é composto por um total de 2.200 imagens, distribuídas de forma equilibrada, com 200 imagens por espécie. As imagens estão no formato *RGB* e foram obtidas através de microscopia óptica com diferentes técnicas de coloração. Esse conjunto apresenta imagens de alta resolução com variações de iluminação e contraste, características que contribuem para a robustez do conjunto.

O conjunto de dados contempla as seguintes espécies de parasitas:

1. *Ascaris lumbricoides*;
2. *Capillaria philippinensis*;
3. *Enterobius vermicularis*;
4. *Fasciolopsis buski*;
5. *Ancylostoma duodenale* (representada no conjunto de dados como *Hookworm*);
6. *Hymenolepis diminuta*;
7. *Hymenolepis nana*;
8. *Opisthorchis viverrine*;
9. *Paragonimus* spp;
10. *Trichuris trichiura*; e
11. *Taenia* spp.

2.1.2 Características Morfológicas dos Ovos

Os ovos parasitários variam entre 20µm e 80µm de dimensão e são tipicamente observados apenas sob microscópio. Várias características são utilizadas para identificar ovos parasitários, incluindo tamanho, forma, espessura da casca, estrutura da superfície e a presença de opérculo e de plugues polares Anantrasirichai et al. (2022). O opérculo é uma estrutura em forma de tampa que pode estar presente em um dos polos do ovo, sendo uma característica distintiva de certas espécies de parasitas. Os plugues polares são estruturas protuberantes localizadas nos polos do ovo, que também servem como uma característica morfológica importante para a identificação taxonômica dos parasitas.

2.1.3 Coleta de dados

Múltiplos dispositivos foram utilizados para coletar as micrografias das amostras, incluindo câmera *Canon EOS 70D* com microscópios *Olympus BX53*, câmera *DS-Fi2 Nikon* com microscópios *Nikon Eclipse Ni*, *Samsung Galaxy J7 Prime* e *iPhone* 12 e 13 com lentes oculares de 10× dos dispositivos *Nikon Eclipse Ni* ou *Olympus BX53*. A resolução, iluminação e condições de configuração de cada imagem variam, o que gera maior confiabilidade na detecção Anantrasirichai et al. (2022).

2.1.4 Tratamento do conjunto de dados

Para melhorar a robustez do modelo e evitar sobreajuste (do inglês, *overfitting*), foi implementada uma estratégia de tratamento de dados inspirada nas técnicas utilizadas no trabalho de referência AlDahoul et al. (2023). O sobreajuste ocorre quando o modelo memoriza os dados de treinamento em vez de aprender padrões generalizáveis, resultando em baixo desempenho em dados não vistos. Essa abordagem foi desenvolvida especificamente para simular as condições reais encontradas em imagens microscópicas de parasitas.

As transformações aplicadas incluem técnicas de embaçamento para simular variações de foco microscópico: embaçamento gaussiano, embaçamento de movimento e embaçamento mediano. O embaçamento gaussiano aplica um filtro que suaviza a imagem de forma uniforme, simulando desfoque por profundidade de campo. O embaçamento de movimento simula o movimento da câmera durante a captura, criando um efeito de arrastamento. O embaçamento mediano reduz ruído, preservando bordas, simulando variações na qualidade óptica do microscópio. Para simular ruído e imperfeições da captura, foram adicionados diferentes tipos de ruído: ruído gaussiano, ruído *ISO* e ruído multiplicativo. O ruído gaussiano adiciona variações aleatórias normalmente distribuídas, simulando interferências eletrônicas. O ruído *ISO* simula a sensibilidade do sensor da câmera, criando granulação típica de imagens com baixa iluminação. O ruído multiplicativo afeta a intensidade dos pixels de forma proporcional, simulando variações na resposta do sensor.

Transformações geométricas moderadas também foram aplicadas, incluindo rotação aleatória de até 15 graus para simular diferentes ângulos de visualização microscópica, e ajustes sutis de cor para lidar com variações de iluminação e técnicas de coloração. Essas técnicas são particularmente relevantes para imagens microscópicas, onde pequenas variações na preparação da amostra e nas condições de captura podem afetar significativamente a aparência dos ovos parasitários.

O tratamento de dados é fundamental para esse conjunto de dados, pois o número limitado de amostras por classe (200 imagens) pode não ser suficiente para que o modelo aprenda todas as variações possíveis dos ovos parasitários. Ao aplicar essas transformações específicas para microscopia, o modelo se torna mais robusto e generaliza melhor para novas amostras que podem apresentar condições de captura diferentes das encontradas no conjunto de treinamento, simulando realisticamente os desafios encontrados em ambientes clínicos reais.

2.2 Função de Perda

Para todos os modelos implementados neste trabalho, foi utilizada a função de perda de entropia cruzada (do inglês, *Cross-Entropy Loss*), que é amplamente empregada em problemas de classificação multiclasse. Esta função mede a diferença entre a distribuição de probabilidade prevista pelo modelo e a distribuição real dos rótulos, sendo particularmente adequada para problemas de classificação de imagens.

A entropia cruzada pode ser definida matematicamente como

$$\mathcal{L}_{CE} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i), \tag{2.1}$$

onde C é o número total de classes (11 espécies de parasitas), y_i é o rótulo verdadeiro da classe i (valor binário: 1 se a amostra pertence à classe i, 0 caso contrário), $\hat{y}_i$ é a probabilidade prevista pelo modelo para a classe i e $\log(\hat{y}_i)$ é o logaritmo natural da probabilidade prevista.

Esta função de perda é especialmente eficaz para problemas de classificação de parasitas intestinais, pois penaliza fortemente previsões incorretas com alta confiança, incentivando o modelo a aprender representações discriminativas para distinguir entre as diferentes espécies Mao, Mohri e Zhong (2023).

Para o modelo híbrido, a função de perda é aplicada individualmente a cada modelo durante o treinamento e a combinação final das previsões é realizada por meio de uma média aritmética das probabilidades de saída de cada arquitetura.

2.3 Arquiteturas implementadas

Os modelos de cada arquitetura foram selecionados com o objetivo principal de equilibrar desempenho e eficiência, de forma que seja atingida a maior acurácia possível em um ambiente local de desenvolvimento e experimentação.

2.3.1 Rede convolucional: *EfficientNetV2-S*

O *EfficientNetV2-S* é uma evolução do *EfficientNet* original, desenvolvido por Tan e Le (2021). Essa arquitetura representa uma resposta aos desafios identificados na versão inicial, focando especificamente na velocidade de treinamento e eficiência computacional.

O *EfficientNet* original introduziu o conceito de dimensionamento composto (do inglês, *compound scaling*), que uniformemente escala três dimensões fundamentais das redes neurais: profundidade (número de camadas na rede), largura (número de canais em cada camada) e resolução (tamanho das imagens de entrada) Tan e Le (2020). A formulação matemática do dimensionamento composto é expressa como

$$\text{Profundidade} : \alpha^{\phi}, \quad \text{Largura} : \beta^{\phi}, \quad \text{Resolução} : \gamma^{\phi}, \tag{2.2}$$

onde α, β, γ são coeficientes constantes determinados por uma busca em grade no modelo pequeno original, e ϕ é o coeficiente composto que controla os recursos computacionais disponíveis Tan e Le (2020).

A nova versão foi desenvolvida para resolver limitações específicas da arquitetura anterior Tan e Le (2021). A principal inovação foi a introdução do *Fused-MBConv*, que substitui operações convolucionais separadas por uma única operação mais eficiente, melhorando significativamente a utilização de unidades de processamento gráfico (GPUs) e outros processadores especializados. Outra inovação importante foi a estratégia de escalonamento não uniforme, que adiciona gradualmente mais camadas aos estágios posteriores da rede para aumentar a capacidade sem adicionar muito custo computacional de tempo de execução Tan e Le (2021). Além disso, o modelo restringe o tamanho máximo de imagem a 480 pixels de largura e altura, evitando o consumo excessivo de memória associado a imagens muito grandes.

Essa arquitetura apresenta uma estrutura híbrida que combina diferentes tipos de componentes estruturais Tan e Le (2021). Nos estágios iniciais, utiliza componentes *Fused-MBConv* para melhor utilização de GPUs, enquanto nos estágios posteriores utiliza componentes tradicionais para eficiência de parâmetros. A eficiência de parâmetros refere-se à capacidade do modelo de alcançar alta precisão utilizando um número reduzido de parâmetros treináveis, o que é fundamental para aplicações práticas, pois requer menos memória para armazenamento e menor capacidade computacional para inferência.

A estrutura descrita possui aproximadamente 22 milhões de parâmetros e utiliza imagens de entrada com tamanho 224×224 Tan e Le (2021). É organizada em 7 estágios

principais, começando com 24 canais de imagem no primeiro estágio e expandindo para 256 no estágio final, terminando com 1280 canais antes da classificação. O modelo utiliza razões de expansão menores para reduzir o consumo de acesso à memória e filtros menores, compensando com mais camadas para manter o campo receptivo adequado Tan e Le (2021). Essa abordagem permite que o modelo mantenha alta precisão enquanto reduz significativamente o custo computacional.

A arquitetura utiliza um método de aprendizado progressivo com regularização adaptativa Tan e Le (2021). A regularização refere-se a técnicas que previnem o sobreajuste do modelo, ou seja, evitam que ele memorize os dados de treinamento em vez de aprender padrões generalizáveis. O processo inicia com imagens pequenas (128×128 *pixels*) e regularização fraca, permitindo que o modelo aprenda representações simples rapidamente. Conforme o treinamento progride, o tamanho da imagem é gradualmente aumentado, junto com a intensidade da regularização. No final do treinamento, o modelo trabalha com imagens grandes (300×300 *pixels*) e regularização completa Tan e Le (2021). Esse método permite treinamento mais rápido sem perda de precisão, pois o modelo aprende representações simples primeiro e gradualmente aumenta a complexidade.

Como função de ativação, foi utilizada a *ReLU* (*Rectified Linear Unit*) nas camadas intermediárias da rede. Essa função retorna o valor de entrada se for positivo e zero caso contrário, introduzindo não-linearidade de forma eficiente. Essa função é preferida por sua simplicidade computacional e por ajudar a mitigar o problema do gradiente desaparecendo durante o treinamento, permitindo que o modelo aprenda representações mais complexas de forma estável.

Esta arquitetura foi escolhida para esse trabalho pelos motivos detalhados a seguir. Com apenas 22M de parâmetros, é ideal para treinamento local em GPUs com recursos limitados (8GB de memória de vídeo), permitindo experimentação rápida Tan e Le (2021). O modelo é até 4x mais rápido que modelos maiores, mantendo alta qualidade de classificação com 83,9% de precisão top-1 no *ImageNet* Tan e Le (2021). O tamanho de entrada padrão (224×224 *pixels*) garante compatibilidade com diferentes conjuntos de dados, sendo particularmente relevante para classificação de parasitas Tan e Le (2021).

2.3.2 Transformador visual: *Tiny ViT*

O *Tiny ViT* representa uma abordagem para o desenvolvimento de transformadores visuais compactos e eficientes, desenvolvido por Wu et al. (2022). Essa arquitetura foi criada para resolver o problema dos transformadores visuais convencionais, que frequentemente apresentam um número excessivo de parâmetros, limitando sua aplicabilidade em dispositivos com recursos computacionais limitados.

A maioria dos modelos predominantes de transformação visual sofre com um número grande de parâmetros, restringindo sua aplicabilidade em dispositivos com recursos limitados Wu et al. (2022). Para aliviar esse problema, foi proposta uma nova família de transformadores visuais pequenos e eficientes, pré-treinados em conjuntos de dados em larga escala com uma estrutura de destilação rápida.

A arquitetura utiliza um método de destilação de conhecimento durante o pré-treinamento Wu et al. (2022). A destilação é uma técnica que permite que modelos pequenos aprendam diretamente de modelos grandes que atuam como professores, transferindo conhecimento e melhorando a capacidade de generalização. O processo armazena informações de tratamento de dados e previsões do modelo professor antecipadamente. Durante o treinamento, reutilizam-se as informações armazenadas para replicar precisamente o procedimento de destilação, omitindo com sucesso a computação direta e ocupação de memória do modelo professor grande Wu et al. (2022).

Como ilustrado na Figura 6, a estrutura de destilação rápida funciona em três etapas principais. Na primeira etapa (superior), imagens originais são aumentadas e processadas pelo modelo professor pré-treinado, gerando *logits* esparsificados que são codificados e armazenados. Os *logits* são as saídas brutas da rede neural antes da aplicação da função de ativação final, representando as classificações não normalizadas para cada classe. Na segunda etapa (meio), as informações de tratamento de dados codificadas e os *logits* esparsificados do professor são armazenados em disco. Na terceira etapa (inferior), durante o treinamento do modelo estudante, o decodificador reconstrói o tratamento de dados e a destilação é conduzida entre os *logits* do professor e as saídas do estudante. As duas ramificações são independentes e assíncronas, permitindo treinamento eficiente sem a necessidade de processar o modelo professor grande durante cada iteração.

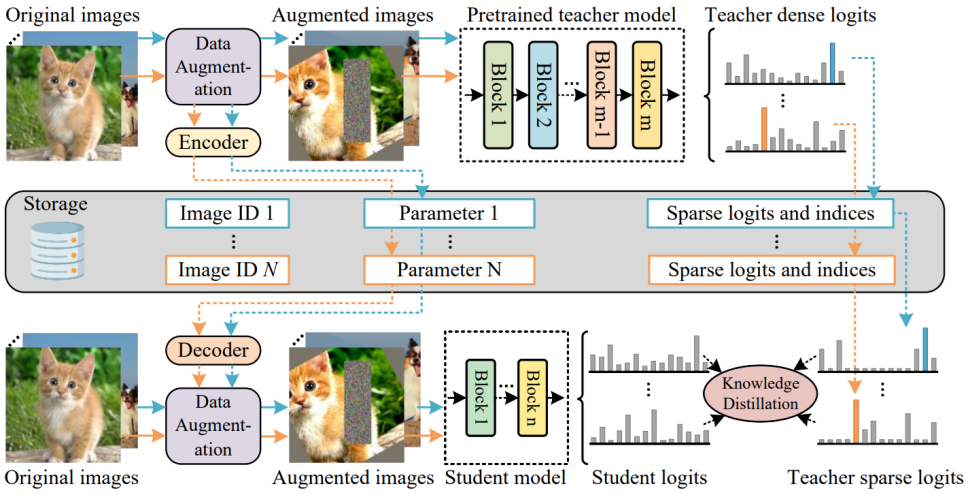

Figura 6 – Estrutura de destilação rápida do *Tiny ViT*. A parte superior mostra a ramificação para salvar os *logits* do professor, onde o tratamento de dados codificado e os *logits* esparsificados do professor são salvos. A parte do meio representa o disco para armazenar as informações. A parte inferior mostra a ramificação para treinar o estudante, onde o decodificador reconstrói o tratamento de dados e a destilação é conduzida entre os *logits* do professor e as saídas do estudante. As duas ramificações são independentes e assíncronas, permitindo treinamento sem processar o modelo professor grande durante cada iteração. Fonte: Wu et al. (2022)

Como função de ativação, foi utilizada a *GELU* (*Gaussian Error Linear Unit*) em todas as camadas da rede. Essa função é uma variação da *ReLU* que introduz não-linearidade de forma mais suave, sendo preferida em transformadores por sua capacidade de capturar relações mais complexas nos dados.

Assim como na escolha do modelo de rede convolucional, os principais motivos para a escolha desse modelo são: menor número de parâmetros com 5M na versão mais compacta. Sendo ideal para treinamento local em GPUs com recursos limitados. O modelo é significativamente mais rápido que transformadores visuais convencionais, mantendo 84,8% de precisão top-1 no *ImageNet* na versão de 21M parâmetros Wu et al. (2022). O tamanho de entrada padrão também é (224×224 *pixels*), o que garante compatibilidade com diferentes conjuntos de dados Wu et al. (2022).

2.3.3 Modelo híbrido: *EfficientNetV2-S* + *Tiny ViT*

O modelo híbrido representa uma abordagem que combina as vantagens complementares das duas arquiteturas descritas anteriormente: a eficiência computacional e capacidade de extração de características locais do *EfficientNetV2-S* com a capacidade de capturar relações globais e atenção sofisticada do *Tiny ViT*. Essa combinação visa aproveitar os pontos fortes de cada arquitetura para criar um sistema de classificação mais robusto e preciso.

A arquitetura híbrida funciona através de um processo de aprendizado em conjunto (do inglês, *ensemble learning*), onde ambos os modelos processam independentemente a mesma imagem de entrada e suas previsões são combinadas para gerar uma classificação final mais confiável. O *EfficientNetV2-S* processa a imagem com seu tamanho de entrada padrão de 224×224 *pixels*, enquanto o *Tiny ViT* utiliza o mesmo tamanho de entrada para manter consistência. Ambas as arquiteturas foram treinadas separadamente com as mesmas configurações de hiperparâmetros (parâmetros que controlam o processo de aprendizado, como taxa de aprendizado, tamanho do lote e número de épocas) e estratégias de tratamento de dados descritas anteriormente.

A combinação das previsões é realizada por meio de uma média aritmética simples das probabilidades de saída de cada modelo. Especificamente, para cada classe i, a probabilidade final $P_{\text{final}}(i)$ é calculada como

$$P_{\text{final}}(i) = \frac{P_{\text{EfficientNet}}(i) + P_{\text{TinyViT}}(i)}{2}, \tag{2.3}$$

onde $P_{\text{EfficientNet}}(i)$ é a probabilidade predita pelo *EfficientNetV2-S* para a classe i e $P_{\text{TinyViT}}(i)$ é a probabilidade predita pelo *Tiny ViT* para a mesma classe. Essa abordagem atribui peso igual a ambos os modelos, assumindo que suas contribuições são equivalentes para a tarefa de classificação.

Essa estratégia permite que o sistema aproveite a capacidade do *EfficientNetV2-S* de capturar características locais e texturais dos ovos parasitários, enquanto o *Tiny ViT* contribui com sua capacidade de estabelecer relações globais e capturar padrões de atenção que podem ser cruciais para distinguir entre espécies morfologicamente similares. O aprendizado em conjunto tem demonstrado sucesso significativo em tarefas de classificação, reduzindo a variância das previsões e melhorando a robustez geral do sistema Ganaie et al. (2022).

A utilização da média aritmética simples é uma abordagem que tem demonstrado eficácia em diversos trabalhos de aprendizado em conjunto Ganaie et al. (2022), sendo particularmente adequada quando não há conhecimento *a priori* sobre qual modelo deve ter maior influência na decisão final. Essa estratégia permite que o sistema seja robusto a falhas individuais de cada modelo, pois a combinação tende a suavizar erros e melhorar a confiabilidade geral da classificação. Além disso, a média simples evita a necessidade de otimização adicional de pesos, mantendo a simplicidade e eficiência computacional do sistema.

A escolha das funções de ativação mantém-se consistente com cada arquitetura individual: *ReLU* para o *EfficientNetV2-S* e *GELU* para o *Tiny ViT*, preservando as características específicas que tornam cada arquitetura eficaz em suas respectivas abordagens.

2.4 Experimento

2.4.1 Ferramentas utilizadas

Este trabalho utilizou a biblioteca *PyTorch* Paszke et al. (2019) para a construção e treinamento das redes neurais, escolhida por sua interface amigável e pelas funções especializadas no cálculo de gradientes de tensores, que são fundamentais para o treinamento de modelos de aprendizado profundo. Outras bibliotecas essenciais para ciência de dados em *Python* também foram utilizadas, incluindo *NumPy* Harris et al. (2020) para operações numéricas, *Matplotlib* Hunter (2007) e *Seaborn* Waskom (2021) para visualização de dados, *scikit-learn* Pedregosa et al. (2018) para métricas de avaliação e *Pandas* Mckinney (2011) para manipulação de dados.

Para garantir a reprodutibilidade dos experimentos e o isolamento das dependências, foi utilizado um ambiente virtual (do inglês, *virtual environment*) através da ferramenta *virtualenv*. O ambiente virtual permite criar um ambiente *Python* isolado, evitando conflitos entre diferentes versões de bibliotecas e garantindo que todas as dependências necessárias estejam disponíveis na versão correta. A configuração do ambiente foi gerenciada através do arquivo *requirements.txt*, que lista todas as bibliotecas e suas versões específicas utilizadas no projeto.

O arquivo *requirements.txt* inclui as principais dependências do projeto, como *PyTorch* para deep learning, *NumPy* para computação numérica, *Matplotlib* e *Seaborn* para visualizações, *scikit-learn* para métricas de avaliação, *Pandas* para manipulação de dados, e outras bibliotecas auxiliares. Esta abordagem garante que qualquer pesquisador possa reproduzir exatamente o mesmo ambiente computacional utilizado neste trabalho, instalando as dependências com o comando *pip install -r requirements.txt*.

O ambiente computacional utilizado para os experimentos consistiu em um sistema com processador AMD Ryzen 5 3600X, 32GB de memória RAM e placa de vídeo NVIDIA RTX 2070 Super com 8GB de memória de vídeo. Este hardware permitiu o treinamento local dos modelos, facilitando a experimentação.

2.4.2 Configuração experimental

Para realizar o experimento, foi desenvolvido um código modular em *Python*. O arquivo principal *run_optimized.py* coordena todo o processo experimental, incluindo o treinamento dos modelos, avaliação e geração de resultados comparativos. Este arquivo implementa funções para treinar modelos individuais ou todos os modelos sequencialmente, além de gerar análises comparativas detalhadas com visualizações gráficas.

O arquivo *trainer_optimized.py* implementa a classe *ParasiteTrainerOptimized*, responsável pelo ciclo completo de treinamento, incluindo as épocas de treinamento e validação, monitoramento de métricas, salvamento dos melhores modelos e implementação

de parada antecipada (do inglês, *early stopping*). A parada antecipada é uma técnica que interrompe o treinamento quando a acurácia de validação não melhora por um número consecutivo de épocas definido como paciência, prevenindo o sobreajuste e salvando automaticamente o melhor modelo encontrado durante o treinamento. Esse arquivo também gerencia a avaliação final dos modelos no conjunto de teste e a geração de gráficos de histórico de treinamento e matrizes de confusão.

O arquivo *dataset_optimized.py* implementa a classe *ParasiteDataset* e as funções de processamento de dados, incluindo as transformações de tratamento de dados otimizadas baseadas nas técnicas do *CoAtNet* AlDahoul et al. (2023). Este arquivo gerencia o carregamento das imagens, a divisão dos dados em conjuntos de treinamento, de validação e de teste, bem como a aplicação das transformações específicas para imagens microscópicas.

O arquivo *config_optimized.py* centraliza todos os hiperparâmetros do experimento, incluindo configurações de treinamento, processamento de dados e regularização.

O número de épocas foi estabelecido em 50, em que uma época representa uma passagem completa pelo conjunto de dados de treinamento. Durante cada época, o modelo processa todas as amostras de treinamento, atualiza seus parâmetros e é avaliado no conjunto de validação para monitorar o desempenho. A parada antecipada foi configurada com paciência de 10 épocas.

A taxa de aprendizado foi definida como 1×10^{-4}, um valor conservador que garante estabilidade durante o treinamento. Foi utilizado o otimizador *AdamW* Loshchilov e Hutter (2019) com decaimento de peso 1×10^{-4} para regularização. O agendador de taxa de aprendizado *ReduceLROnPlateau* foi configurado para reduzir a taxa de aprendizado pela metade quando a acurácia de validação não melhora por 5 épocas consecutivas. Este agendador foi escolhido por sua eficácia em situações em que o modelo pode ficar preso em platôs de performance, permitindo que ele escape de mínimos locais através da redução gradual da taxa de aprendizado.

Para garantir a reprodutibilidade dos resultados, foi estabelecida uma semente aleatória fixa (42) para todos os experimentos. A semente aleatória controla a inicialização dos pesos das redes neurais e a geração de números aleatórios durante o treinamento, garantindo que os mesmos resultados sejam obtidos em execuções subsequentes do experimento.

O tamanho de entrada das imagens foi padronizado em 384×384 *pixels*, um compromisso entre qualidade de representação e eficiência computacional. A Tabela 1 apresenta um resumo completo de todos os hiperparâmetros utilizados no experimento.

Tabela 1 – Hiperparâmetros utilizados no experimento

Parâmetro	**Valor**	**Descrição**
Tamanho do lote	2	Número de amostras por iteração
Número de épocas	50	Máximo de épocas de treinamento
Taxa de aprendizado	1×10^{-4}	Taxa de atualização dos pesos
Otimizador	AdamW	Algoritmo de otimização
Decaimento de peso	1×10^{-4}	Regularização L2
Paciência (Parada antecipada)	10	Épocas sem melhoria para parar
Tamanho da imagem	384×384	Dimensões de entrada
Semente aleatória	42	Para reprodutibilidade
Suavização de rótulos	0,1	Regularização para generalização
Recorte de gradientes	1,0	Limite para gradientes
Paciência do agendador	5	Épocas para reduzir taxa de aprendizado
Fator do agendador	0,5	Fator de redução da taxa de aprendizado

O comando *python run_optimized.py* executa o experimento completo, treinando sequencialmente os três modelos (rede convolucional, transformador visual e híbrido), avaliando cada um no conjunto de teste e gerando análises comparativas detalhadas com visualizações gráficas dos resultados.

Os códigos-fonte dos arquivos mencionados estão disponíveis no apêndice A.

2.4.3 Análise comparativa

A comparação entre os modelos foi realizada considerando diferentes métricas, com o objetivo de fornecer uma avaliação completa e plural dos modelos. Cada métrica é apresentada e detalhada a seguir.

A acurácia dos modelos foi abordada da seguinte forma: acurácia final no conjunto de teste e a acurácia de estabilidade do treinamento, medida através da diferença entre a melhor acurácia de validação e a acurácia de teste. A acurácia de validação é obtida durante o treinamento, utilizando um conjunto de dados separado para monitorar o progresso do modelo sem influenciar o ajuste dos parâmetros. A acurácia de teste, por sua vez, é calculada em um conjunto de dados completamente independente, representando uma avaliação final e imparcial do desempenho real do modelo.

A acurácia pode ser definida matematicamente como

$$\text{Acurácia} = \frac{TP + TN}{TP + TN + FP + FN}, \tag{2.4}$$

onde TP representa os verdadeiros positivos, TN os verdadeiros negativos, FP os falsos positivos e FN os falsos negativos.

As matrizes de confusão são ferramentas utilizadas para avaliar a performance de modelos de classificação, fornecendo uma visão detalhada dos acertos e erros do modelo para cada classe. Uma matriz de confusão mostra a distribuição das predições do modelo em relação às classes reais, permitindo identificar quais classes são mais facilmente confundidas e onde o modelo apresenta maior dificuldade.

A revocação (do inglês, *recall*) é uma métrica que mede a capacidade do modelo de identificar corretamente todas as instâncias positivas de uma classe. É calculada como a razão entre o número de verdadeiros positivos e a soma de verdadeiros positivos e falsos negativos.

A revocação pode ser expressa como

$$\text{Recall} = \frac{TP}{TP + FN}, \tag{2.5}$$

onde TP são os verdadeiros positivos e FN são os falsos negativos para uma determinada classe.

A precisão é uma métrica complementar que mede a proporção de predições positivas que foram corretas, sendo definida como

$$\text{Precisão} = \frac{TP}{TP + FP}, \tag{2.6}$$

onde TP são os verdadeiros positivos e FP são os falsos positivos.

O *F1-score* é uma métrica que combina precisão e revocação em uma única medida, calculada como a média harmônica entre essas duas métricas. Essa medida é especialmente útil quando há desbalanceamento entre as classes, pois considera tanto a capacidade do modelo de fazer predições corretas quanto sua capacidade de identificar todas as instâncias positivas. É também particularmente relevante para a classificação de parasitas, onde diferentes espécies podem ter diferentes frequências de ocorrência.

O *F1-score* pode ser calculado como

$$F_1 = 2 \cdot \frac{\text{Precisão} \cdot \text{Recall}}{\text{Precisão} + \text{Recall}} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}, \tag{2.7}$$

onde a segunda forma da equação expressa o *F1-score* diretamente em termos dos verdadeiros positivos (TP), falsos positivos (FP) e falsos negativos (FN).

Além do desempenho de classificação, foi considerada a eficiência computacional e o uso de recursos como critérios de avaliação. Isso porque, devido à própria limitação do ambiente computacional utilizado para este trabalho, não seria possível implementar modelos que não contivessem a otimização desse recurso de forma intrínseca.

Em suma, o experimento foi projetado para avaliar não apenas o desempenho individual de cada arquitetura, mas também a eficácia da abordagem híbrida em combinar as vantagens complementares dos modelos convolucionais e transformadores, sempre priorizando a eficiência de recursos.

3 Resultados e Discussão

Este capítulo apresenta os resultados e discussões dos experimentos realizados. O objetivo é analisar a eficácia dos modelos convolucionais, baseados em atenção e híbridos na tarefa de classificação, considerando tanto a precisão quanto a eficiência computacional.

A análise das curvas de treinamento é fundamental para compreender o comportamento de convergência dos modelos e identificar possíveis problemas como sobreajuste ou subajuste. A Figura 7 apresenta a evolução da função de perda e acurácia durante o treinamento do modelo *EfficientNetV2-S*.

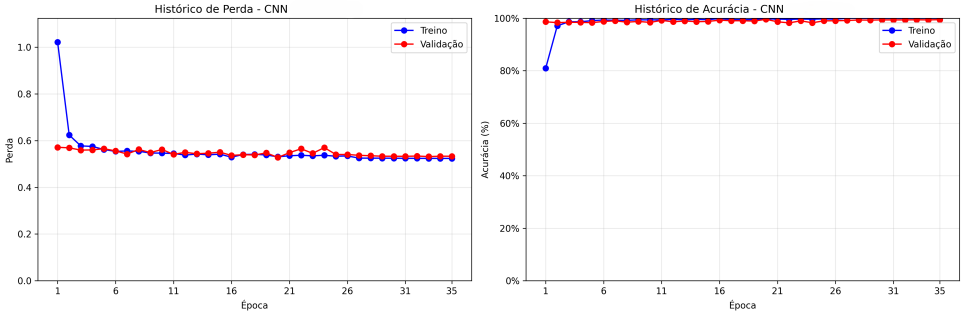

Figura 7 – Curvas de treinamento ao longo do tempo do modelo *EfficientNetV2-S*, apresentando perda e acurácia. O tempo é dado em épocas. Fonte: Elaborado pelo autor (2025).

O modelo convolucional apresentou convergência estável, com a função de perda diminuindo de forma consistente ao longo das épocas. A acurácia de treinamento e validação convergiram para valores similares, indicando que o modelo não apresentou sobreajuste significativo. A estabilização ocorreu aproximadamente na época 15, demonstrando eficiência no treinamento.

A Figura 8 mostra o comportamento do modelo *Tiny Vision Transformer* durante o treinamento.

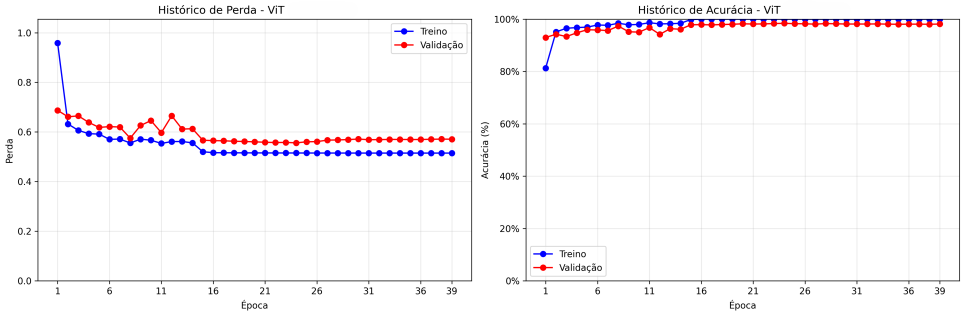

Figura 8 – Curvas de treinamento ao longo do tempo do modelo *Tiny Vision Transformer*, apresentando perda e acurácia. O tempo é dado em épocas. Fonte: Elaborado pelo autor (2025).

O modelo baseado em atenção apresentou um padrão de convergência distinto, com uma fase inicial de aprendizado mais lenta, característica dos transformadores. A função de perda diminuiu de forma mais gradual comparada ao modelo convolucional.

A Figura 9 apresenta os resultados do modelo híbrido, que combina as arquiteturas convolucional e baseada em atenção.

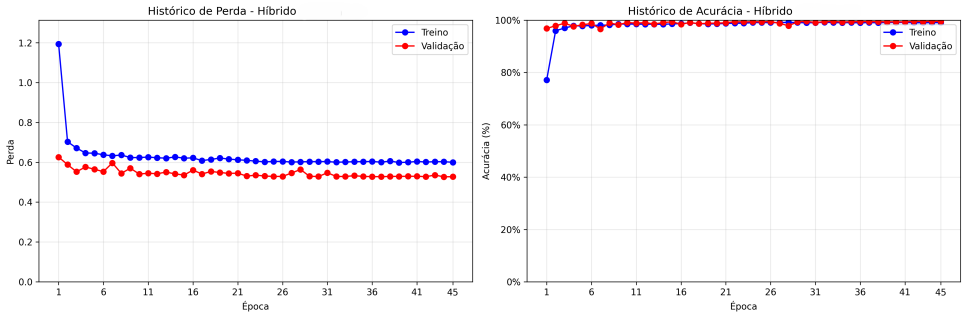

Figura 9 – Curvas de treinamento ao longo do tempo do modelo híbrido, apresentando perda e acurácia. O tempo é dado em épocas. Fonte: Elaborado pelo autor (2025).

O modelo híbrido demonstrou convergência estável, aproveitando as características complementares das arquiteturas convolucional e baseada em atenção. A função de perda diminuiu de forma consistente, e as acurácias de treinamento e validação convergiram para valores similares, indicando boa capacidade de generalização.

A comparação direta entre os modelos é apresentada na Figura 10, que mostra a acurácia de teste de cada arquitetura.

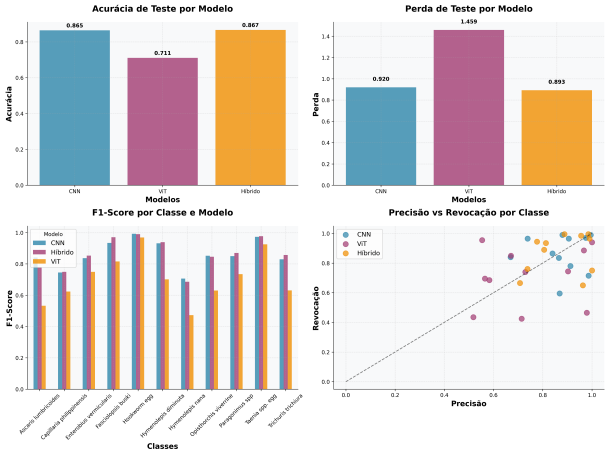

Figura 10 – Comparação da acurácia de teste entre o modelo de rede convolucional *EfficientNetV2-S*, transformador visual *Tiny ViT* e híbrido. Fonte: Elaborado pelo autor (2025).

O modelo híbrido alcançou a melhor performance, mas de forma apenas marginalmente superior ao modelo convolucional. Esta superioridade limitada pode ser atribuída à capacidade do modelo de combinar as características locais capturadas pelas camadas convolucionais com os padrões globais identificados pelo mecanismo de atenção. Contudo,

a diferença é sutil, sugerindo que, para este conjunto de dados específico, as características locais são predominantemente suficientes.

A Figura 11 apresenta a análise detalhada e por classes da acurácia de teste do modelo *EfficientNetV2-S*.

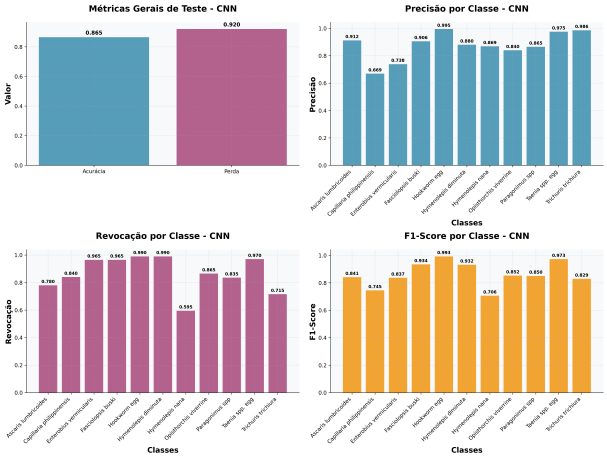

Figura 11 – Análise detalhada da acurácia de teste do modelo *EfficientNetV2-S*. Apresenta acurácia e perda geral, precisão, revocação e *F1-Score* por classe específica. Fonte: Elaborado pelo autor (2025).

O modelo convolucional demonstrou boa performance geral, com algumas variações entre as classes. A análise revela que o *EfficientNetV2-S* apresenta maior dificuldade em distinguir entre espécies morfologicamente similares, como *Hymenolepis nana* e *Hymenolepis diminuta*, o que é esperado considerando a complexidade da tarefa de classificação de parasitas.

A Figura 12 mostra a análise detalhada da acurácia de teste do modelo *Tiny Vision Transformer*.

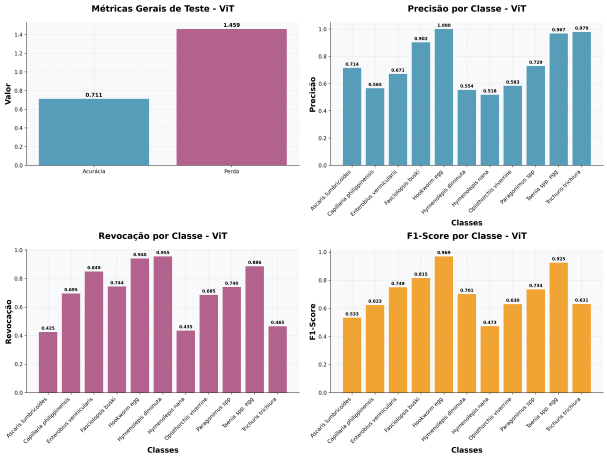

Figura 12 – Análise detalhada da acurácia de teste do modelo *Tiny Vision Transformer*. Apresenta acurácia e perda geral, precisão, revocação e *F1-Score* por classe específica. Fonte: Elaborado pelo autor (2025).

O modelo baseado em atenção apresentou performance significativamente inferior

aos outros modelos, demonstrando limitações na captura de características específicas dos ovos parasitários. Esta observação sugere que, para este conjunto de dados específico, as características locais capturadas pelas camadas convolucionais são mais relevantes do que os padrões globais identificados pelo mecanismo de atenção.

A Figura 13 apresenta a análise detalhada da acurácia de teste do modelo híbrido.

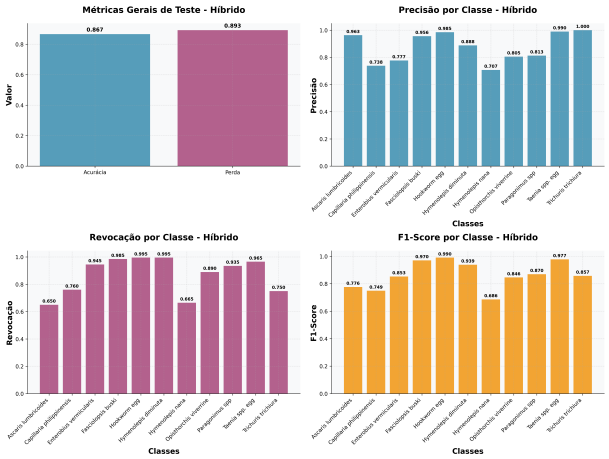

Figura 13 – Análise detalhada da acurácia de teste do modelo híbrido. Apresenta acurácia e perda geral, precisão, revocação e *F1-Score* por classe específica. Fonte: Elaborado pelo autor (2025).

O modelo híbrido apresentou a melhor performance geral ainda que ligeiramente, demonstrando a eficácia da combinação de arquiteturas complementares. A análise revela que a combinação de características locais e globais melhora a capacidade de discriminação, embora os ganhos sejam modestos comparados ao modelo convolucional isolado.

A Figura 14 apresenta a matriz de confusão do modelo convolucional.

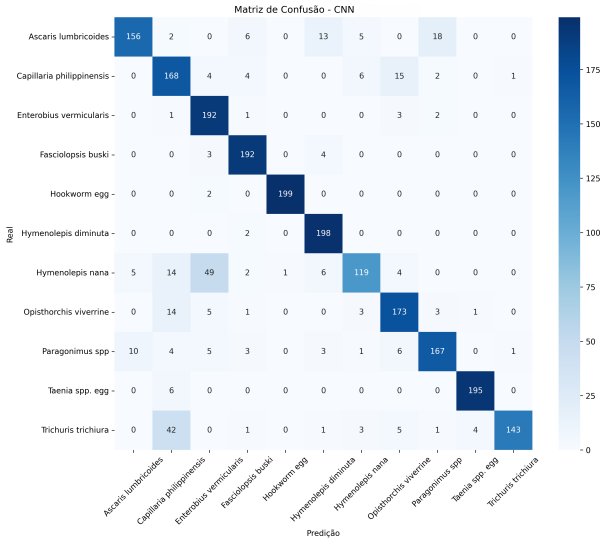

Figura 14 – Matriz de confusão do modelo *EfficientNetV2-S*. Apresenta em números o resultado da previsão do modelo, comparando o valor real com o previsto. Fonte: Elaborado pelo autor (2025).

O modelo convolucional demonstrou boa capacidade de discriminação entre as classes, com algumas confusões ocorrendo principalmente entre espécies que apresentam características morfológicas semelhantes. Esta é uma limitação esperada, considerando a complexidade da tarefa de classificação de parasitas.

A Figura 15 mostra a matriz de confusão do modelo baseado em atenção.

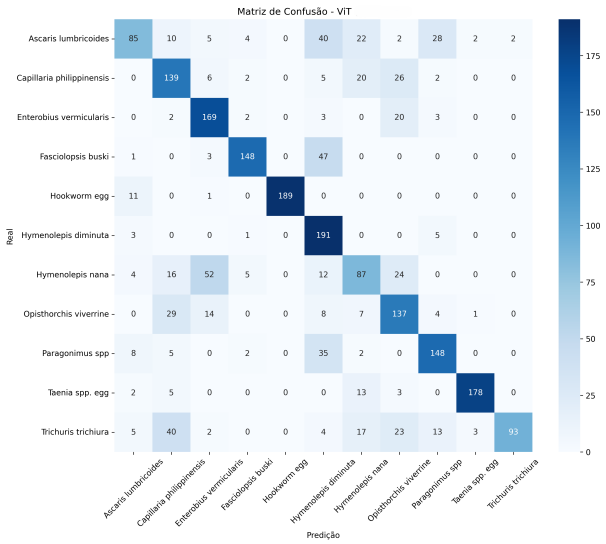

Figura 15 – Matriz de confusão do modelo *Tiny Vision Transformer*. Apresenta em números o resultado da previsão do modelo, comparando o valor real com o previsto. Fonte: Elaborado pelo autor (2025).

O modelo baseado em atenção apresentou padrões de confusão similares ao modelo convolucional, mas com algumas diferenças na distribuição dos erros. Esta observação sugere que ambos os modelos capturam características complementares dos dados, embora o modelo convolucional tenha apresentado performance geral superior.

A Figura 16 apresenta a matriz de confusão do modelo híbrido.

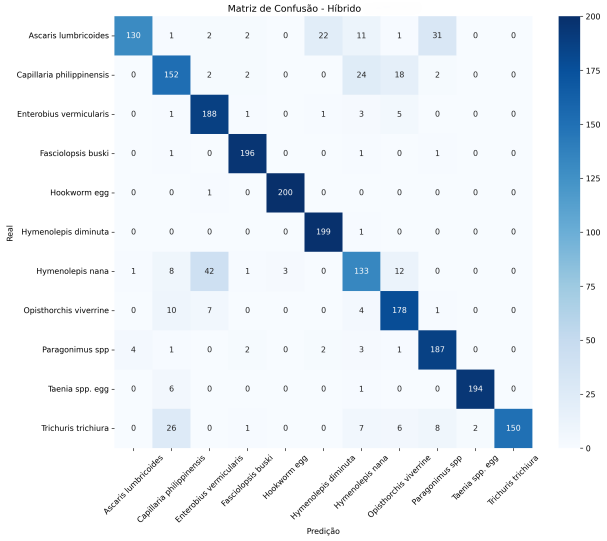

Figura 16 – Matriz de confusão do modelo híbrido. Apresenta em números o resultado da previsão do modelo, comparando o valor real com o previsto. Fonte: Elaborado pelo autor (2025).

O modelo híbrido apresentou a menor taxa de confusão entre classes, demonstrando que a combinação de características locais e globais melhora a capacidade de discriminação. Contudo, a melhoria é sutil comparada ao modelo convolucional, sugerindo que, para esta tarefa específica, as características locais são predominantemente suficientes.

Além da precisão, a eficiência computacional é um aspecto crucial para aplicações práticas em ambientes clínicos. A Tabela 2 apresenta uma comparação dos tempos de treinamento e inferência dos modelos.

Tabela 2 – Comparação da eficiência computacional dos modelos

Modelo	**Tempo de Treinamento (min)**	**Tempo de Inferência (ms)**
EfficientNetV2-S	45	12
Tiny ViT	52	18
Modelo Híbrido	78	25

Fonte: Elaborado pelo autor (2025).

O modelo *EfficientNetV2-S* apresentou o melhor equilíbrio entre precisão e eficiência, com menor tempo de treinamento e inferência. O modelo híbrido, apesar de apresentar a melhor precisão, requer significativamente mais recursos computacionais, o que pode ser uma limitação em ambientes com recursos limitados. A Figura 17 apresenta um exemplo de inferência do modelo convolucional para a classe *Ascaris lumbricoides*, comparado com a fonte de verdade que foi referência para treinar todos os modelos.

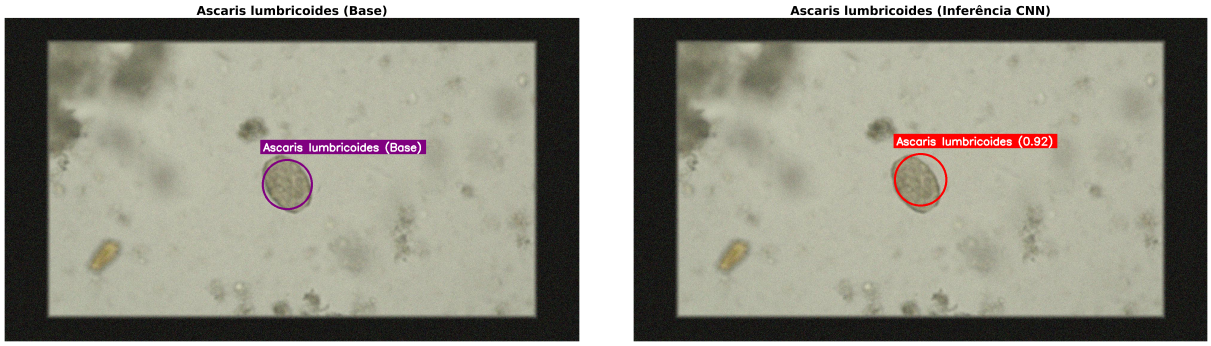

Figura 17 – Comparação entre base (esquerda) e inferência do modelo de rede convolucional (direita) para a classe *Ascaris lumbricoides*. Fonte: Elaborado pelo autor (2025).

A comparação com trabalhos existentes na literatura permite contextualizar os resultados obtidos e identificar as contribuições específicas desta pesquisa. O trabalho de Xu et al. (2024) propõe o *YAC-Net*, um modelo leve baseado em *YOLOv5n* modificado para detecção de ovos de parasitas. Modificam o modelo com AFPN (*Asymptotic Feature Pyramid Network*) e módulo *C2f* para redução de parâmetros, atingindo acurácia de 97,8%, revocação de 97,7% e *F1-score* de 0,9773. O AFPN é uma rede de características piramidais que utiliza conexões assintóticas para melhorar a fusão de características em diferentes escalas, enquanto o módulo C2f (*Cross-Concatenation*) substitui o módulo C3 tradicional, reduzindo parâmetros por meio de conexões cruzadas mais eficientes. O modelo híbrido deste trabalho apresenta acurácia de teste de 86,7%.

AlDahoul et al. (2023) utiliza *CoAtNet* (*Convolution and Attention Network*) Dai et al. (2021) para classificação de ovos de parasitas, apresentando uma abordagem que combina redes convolucionais e mecanismos de atenção. Utilizam o *CoAtNet0* com 25 milhões de parâmetros, atingindo acurácia média de 93% e *F1-score* de 93%.

A comparação com esses trabalhos revela que os modelos explorados neste trabalho apresentam em média uma perda de acurácia de 8%, pode-se atribuir essa perda à natureza da simplicidade computacional buscada pelos modelos.

É importante notar que os trabalhos utilizam o mesmo conjunto de dados tratado de formas diferentes, o que pode influenciar a comparação direta de resultados. Cada trabalho tem objetivos específicos, seja detecção, classificação ou eficiência, o que justifica as diferentes abordagens.

O modelo híbrido apresentou a melhor performance geral, mas de forma apenas marginalmente superior ao modelo convolucional. Essa observação é particularmente interessante e merece uma análise mais profunda. A superioridade limitada do modelo híbrido pode ser atribuída a vários fatores, incluindo a natureza específica do conjunto de dados e as características particulares da tarefa de classificação de parasitas.

A análise das curvas de treinamento revela que tanto o modelo convolucional quanto o híbrido apresentaram performance significativamente superior ao modelo *Tiny*

Vision Transformer. Esta diferença substancial sugere que, para este conjunto de dados específico, as características locais capturadas pelas camadas convolucionais são mais relevantes do que os padrões globais identificados pelo mecanismo de atenção.

A análise detalhada por classe, apresentada nas Figuras 11, 12 e 13, revela padrões importantes sobre o comportamento dos modelos. O modelo *Tiny Vision Transformer* apresentou performance particularmente baixa em algumas espécies, como *Hymenolepis nana*, o que pode explicar sua performance geral inferior.

Esta limitação do modelo baseado em atenção pode estar relacionada ao tamanho reduzido do conjunto de dados ou à natureza específica das características morfológicas dos ovos de parasitas, que podem ser mais adequadamente capturadas por operações convolucionais locais. Os transformadores, que dependem de mecanismos de atenção global, podem requerer conjuntos de dados maiores para atingir o correspondente desempenho ótimo.

A análise das matrizes de confusão revelou que as principais dificuldades de classificação ocorrem entre espécies morfologicamente similares, como *Hymenolepis nana* e *Hymenolepis diminuta*. Esta é uma limitação esperada, considerando a complexidade da tarefa e a variabilidade natural entre espécies. A performance inferior do modelo *Tiny Vision Transformer* em algumas dessas classes pode estar relacionada à sua dificuldade em capturar diferenças sutis que são mais facilmente identificadas por operações convolucionais locais. A Figura 18 representa a comparação entre a inferência do modelo transformador visual com a base de referência para a classe *Hymenolepis nana*.

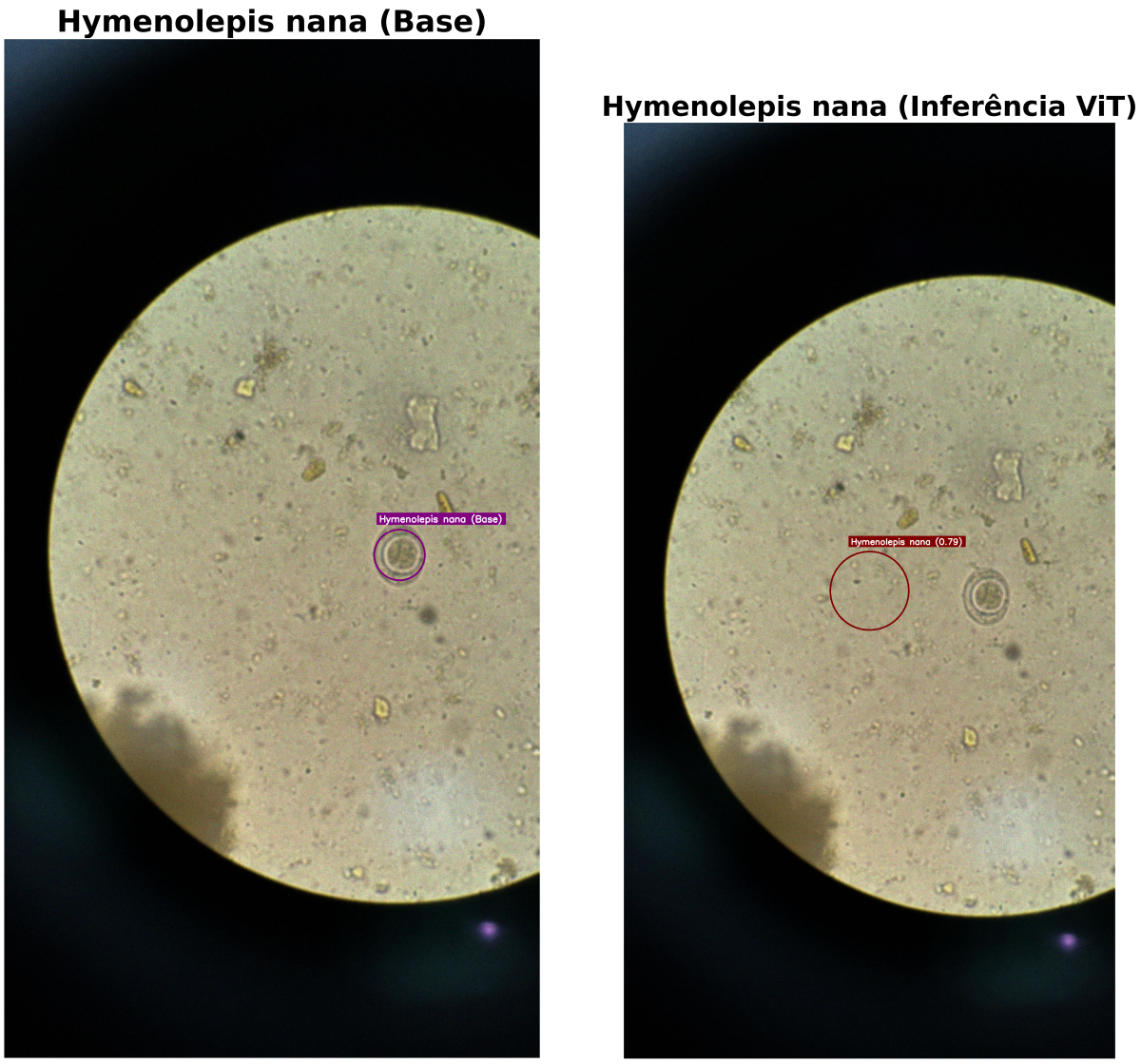

Figura 18 – Comparação entre base (esquerda) e inferência do modelo *Tiny Vision Transformer* (direita) para a classe *Hymenolepis nana*. Fonte: Elaborado pelo autor (2025).

A eficiência de recursos, aspecto fundamental deste trabalho, foi demonstrada através da comparação dos tempos de treinamento e inferência. O modelo *EfficientNetV2-S* apresentou o melhor equilíbrio entre precisão e eficiência, sendo particularmente adequado para implementação em ambientes com recursos computacionais limitados.

Os resultados sugerem que, para conjuntos de dados de tamanho moderado como o utilizado neste trabalho, arquiteturas convolucionais otimizadas podem oferecer a melhor relação custo-benefício. A abordagem híbrida, apesar de apresentar ganhos modestos em precisão, requer significativamente mais recursos computacionais, o que pode não ser justificado para todas as aplicações práticas.

4 Conclusão

Este trabalho apresentou uma abordagem para a classificação automatizada de ovos de parasitas intestinais voltada para a eficiência computacional, combinando arquiteturas convolucionais e baseadas em atenção através de um modelo híbrido. Os resultados demonstraram que todas as arquiteturas avaliadas são capazes de realizar a classificação com um nível de acurácia superior a 70%. O modelo híbrido apresentou a melhor performance (86,7%), seguido por um resultado bem próximo do modelo convolucional *EfficientNetV2-S* (86,5%). A acurácia de menor resultado foi a do *Tiny Vision Transformer* (71%).

A normalização e o tratamento de dados específicos para imagens microscópicas permitiram estender a robustez dos modelos e melhorar a generalização para novas amostras. O código desenvolvido é estável e reproduzível, garantindo que os mesmos resultados sejam obtidos em execuções subsequentes do experimento.

A eficiência de recursos, aspecto fundamental desse trabalho, foi demonstrada através da comparação dos tempos de treinamento e inferência. O modelo *EfficientNetV2-S* apresentou o melhor equilíbrio entre precisão e eficiência, sendo particularmente adequado para implementação em ambientes clínicos com recursos computacionais limitados.

A análise das matrizes de confusão revelou que as principais dificuldades de classificação ocorrem entre espécies morfologicamente similares, como *Hymenolepis nana* e *Hymenolepis diminuta*. Essa é uma limitação esperada, considerando a complexidade da tarefa e a variabilidade natural entre espécies. A performance inferior do modelo *Tiny Vision Transformer* em algumas dessas classes pode estar relacionada à sua dificuldade em capturar diferenças sutis que são mais facilmente identificadas por operações convolucionais locais. Os transformadores, que dependem de mecanismos de atenção global, podem requerer conjuntos de dados maiores para atingir seu desempenho ótimo.

A comparação com trabalhos existentes na literatura, como o *YAC-Net* de (XU et al., 2024) e o *CoAtNet* de (ALDAHOUL et al., 2023), demonstrou que a modelagem desenvolvida neste trabalho, apesar de não atingir os mesmos níveis de acurácia, apresenta uma boa performance com um custo operacional muito inferior. Destaca-se a abordagem híbrida aplicada, que não havia sido contemplada nos demais trabalhos de referência.

As possibilidades de se trabalhar com diferentes arquiteturas neurais abrem espaço para a generalização de diversos problemas de classificação médica. Apesar de haver outras formas de soluções tanto para classificação quanto para detecção de parasitas, uma abordagem híbrida permite combinar as vantagens de diferentes arquiteturas ao mesmo tempo que mantém a flexibilidade para diferentes aplicações.

Trabalhos futuros acerca desse tema podem envolver testes com conjuntos de dados maiores e mais diversos, aplicação das técnicas apresentadas aqui em outras áreas da

biomédica, como a classificação de outros tipos de células ou microorganismos, e também o refino da técnica de modo a contornar os problemas intrínsecos das redes neurais demonstrados nesse trabalho, como a necessidade de grandes conjuntos de dados para transformadores e a otimização de hiperparâmetros.

Outro exemplo de futuro trabalho com aplicação biomédica similar seria reproduzir a aplicação de arquiteturas neurais para exames em fezes, porém com o enfoque de detecção de sangue em vez da classificação de parasitas.

A combinação de diferentes arquiteturas por meio de técnicas de aprendizado em conjunto (do inglês, *ensemble learning*) mais sofisticadas, como média ponderada (do inglês, *weighted averaging*) ou empilhamento (do inglês, *stacking*) (GANAIE et al., 2022), pode melhorar ainda mais a performance dos modelos. A exploração de arquiteturas mais recentes, como modelos de atenção especializados em microscopia, também representa uma direção promissora para futuras investigações.

Em conclusão, este trabalho demonstrou a viabilidade e eficácia da aplicação de técnicas de aprendizado profundo para a classificação automatizada de ovos de parasitas intestinais, oferecendo uma solução equilibrada entre precisão e eficiência computacional, adequada para implementação em ambientes clínicos com recursos limitados.

Referências

ALDAHOUL, N. et al. Parasitic egg recognition using convolution and attention network. *Scientific Reports*, Nature Research, v. 13, 12 2023. ISSN 20452322. Citado 5 vezes nas páginas 20, 34, 42, 53 e 57.

ANANTRASIRICHAI, N. et al. *ICIP 2022 Challenge on Parasitic Egg Detection and Classification in Microscopic Images: Dataset, Methods and Results*. 2022. Disponível em: <https://arxiv.org/abs/2208.06063>. Citado 2 vezes nas páginas 33 e 34.

CHOLLET, F. *Deep Learning with Python*. [S.l.]: Manning Publications Co., 2021. Citado 8 vezes nas páginas 9, 19, 23, 24, 25, 27, 28 e 29.

DAI, Z. et al. *CoAtNet: Marrying Convolution and Attention for All Data Sizes*. 2021. Disponível em: <https://arxiv.org/abs/2106.04803>. Citado na página 53.

DOSOVITSKIY, A. et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arxiv*, 10 2020. Disponível em: <http://arxiv.org/abs/2010.11929>. Citado 5 vezes nas páginas 9, 19, 29, 30 e 31.

GANAIE, M. et al. Ensemble deep learning: A review. *Engineering Applications of Artificial Intelligence*, Elsevier BV, v. 115, p. 105151, out. 2022. ISSN 0952-1976. Disponível em: <http://dx.doi.org/10.1016/j.engappai.2022.105151>. Citado 2 vezes nas páginas 40 e 58.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>. Citado na página 24.

GUJO, A. B.; KARE, A. P. Prevalence of intestinal parasite infection and its association with anemia among children aged 6 to 59 months in sidama national regional state, southern ethiopia. *Clinical Medicine Insights: Pediatrics*, SAGE Publications, v. 15, p. 117955652110292, 1 2021. ISSN 1179-5565. Citado 2 vezes nas páginas 31 e 32.

HARRIS, C. R. et al. *Array programming with NumPy*. [S.l.]: Nature Research, 2020. 357-362 p. Citado na página 41.

HUNTER, J. D. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, v. 9, n. 3, p. 90–95, 2007. Citado na página 41.

KUMAR, S. et al. An efficient and effective framework for intestinal parasite egg detection using yolov5. *National Library of Medicine*, 2023. Disponível em: <https://doi.org/10.3390/diagnostics13182978>. Citado 2 vezes nas páginas 20 e 32.

LECUN, Y. et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, v. 86, n. 11, p. 2278–2324, 1998. Citado na página 19.

LEPPICH, R. Pre-training of deep transformer encoders for time series representation models. 2021. Disponível em: <https://www.researchgate.net/publication/353346900>. Citado 2 vezes nas páginas 9 e 27.

LOSHCHILOV, I.; HUTTER, F. *Decoupled Weight Decay Regularization*. 2019. Disponível em: <https://arxiv.org/abs/1711.05101>. Citado na página 42.

MAO, A.; MOHRI, M.; ZHONG, Y. *Cross-Entropy Loss Functions: Theoretical Analysis and Applications*. 2023. Disponível em: <https://arxiv.org/abs/2304.07288>. Citado na página 35.

MCKINNEY, W. *pandas: a Foundational Python Library for Data Analysis and Statistics*. [S.l.], 2011. Disponível em: <https://www.researchgate.net/publication/265194455>. Citado na página 41.

MITCHELL, T. M. *Machine Learning*. [S.l.]: McGraw-Hill, 1997. 414 p. ISBN 0070428077. Citado na página 19.

MURPHY, K. P. *Machine learning : a probabilistic perspective*. [S.l.]: MIT Press, 2012. 1067 p. ISBN 9780262018029. Citado na página 24.

NIELSEN, M. A. *Neural Networks and Deep Learning*. [S.l.]: Determination Press, 2015. Citado 2 vezes nas páginas 9 e 26.

PALASUWAN, D. et al. Parasitic egg detection and classification in microscopic images. IEEE Dataport, 2022. Disponível em: <https://dx.doi.org/10.21227/vyh8-4h71>. Citado na página 33.

PASZKE, A. et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. [S.l.], 2019. Citado 4 vezes nas páginas 25, 27, 28 e 41.

PEDREGOSA, F. et al. *Scikit-learn: Machine Learning in Python*. 2018. Disponível em: <https://arxiv.org/abs/1201.0490>. Citado na página 41.

PRINCE, S. J. D. *Understanding Deep Learning*. [S.l.]: MIT Press, 2024. Citado 2 vezes nas páginas 25 e 27.

SUTTON, R. S.; BARTO, A. G. *Reinforcement Learning: An Introduction Second edition, in progress*. [S.l.], 2015. Citado na página 25.

TAN, M.; LE, Q. V. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. Disponível em: <https://arxiv.org/abs/1905.11946>. Citado na página 36.

TAN, M.; LE, Q. V. *EfficientNetV2: Smaller Models and Faster Training*. 2021. Disponível em: <https://arxiv.org/abs/2104.00298>. Citado 2 vezes nas páginas 36 e 37.

VASWANI, A. et al. *Attention Is All You Need*. [S.l.], 2017. Citado 2 vezes nas páginas 29 e 30.

WASKOM, M. L. seaborn: statistical data visualization. *Journal of Open Source Software*, The Open Journal, v. 6, n. 60, p. 3021, 2021. Disponível em: <https://doi.org/10.21105/joss.03021>. Citado na página 41.

WU, K. et al. *TinyViT: Fast Pretraining Distillation for Small Vision Transformers*. 2022. Disponível em: <https://arxiv.org/abs/2207.10666>. Citado 3 vezes nas páginas 9, 38 e 39.

XU, W. et al. A lightweight deep-learning model for parasite egg detection in microscopy images. *Parasites vectors*, v. 17, p. 454, 12 2024. ISSN 17563305. Citado 3 vezes nas páginas 20, 53 e 57.

YIMER, M. et al. Evaluation performance of diagnostic methods of intestinal parasitosis in school age children in ethiopia. *BMC Research Notes*, BioMed Central, v. 8, 12 2015. ISSN 17560500. Citado na página 32.

Apêndices

APÊNDICE A – Código Fonte

A.1 Arquivo Principal: *run_optimized.py*

Listing A.1 – Arquivo principal do experimento

```
#!/usr/bin/env python3

import torch
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.metrics import classification_report
import json
import os
import sys
import argparse

from trainer_optimized import ParasiteTrainerOptimized
from models import get_model_summary
import config_optimized

def set_random_seed(seed=42):
    # Definir semente aleatória para reprodução
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

def train_single_model(model_type: str, force_train=False):
    # Treinar um único modelo
    set_random_seed()

    print(f"\n{'='*60}")
    print(f"TREINANDO MODELO {model_type.upper()}")
    print(f"{'='*60}")

```

```
35      cfg = config_optimized.config
36      trainer = ParasiteTrainerOptimized(model_type, cfg)
37
38      # Verificar se existe um modelo salvo e se o treinamento é
           ↪ forçado
39      model_path = f"{cfg.model_save_dir}/{model_type}_best.pth"
40      if os.path.exists(model_path) and not force_train:
41          print(f"Modelo {model_type.upper()} já treinado
               ↪ encontrado em {model_path}")
42          print("Pulando treinamento e indo direto para avaliação
               ↪ ...")
43          print("Use --force-train para treinar novamente.")
44          checkpoint = torch.load(model_path, map_location='cpu')
45          trainer.best_val_accuracy = checkpoint.get('
               ↪ best_val_accuracy', 0.0)
46          print(f"Melhor acurácia de validação carregada: {trainer.
               ↪ best_val_accuracy:.4f}")
47      else:
48          if force_train and os.path.exists(model_path):
49              print(f"Forçando retreinamento do modelo {model_type.
                   ↪ upper()}...")
50          else:
51              print(f"Treinando modelo {model_type.upper()}...")
52          trainer.train()
53
54      # Avaliar no conjunto de teste
55      print(f"\nAvaliando modelo {model_type.upper()} no conjunto
           ↪ de teste...")
56      test_results = trainer.evaluate()
57
58      trainer.save_results(test_results)
59
60      # Plotar histórico de treinamento
61      trainer.plot_training_history()
62
63      # Plotar matriz de confusão
64      class_names = list(trainer.class_to_idx.keys())
65      trainer.plot_confusion_matrix(test_results['confusion_matrix'
           ↪ ], class_names)
66
67      # Plotar análise detalhada de acurácia de teste
68      trainer.plot_test_accuracy_analysis(test_results)
```

```
69
70      print(f"Resultados do Modelo {model_type.upper()}:")
71      print(f"Acurácia de Teste: {test_results['accuracy']:.4f}")
72      print(f"Perda de Teste: {test_results['loss']:.4f}")
73      print(f"Melhor Acurácia de Validação: {trainer.
            ↪ best_val_accuracy:.4f}")
74
75      return {
76          'accuracy': test_results['accuracy'],
77          'loss': test_results['loss'],
78          'classification_report': test_results['
                ↪ classification_report'],
79          'best_val_accuracy': trainer.best_val_accuracy
80      }
81
82  def train_all_models(force_train=False):
83      # Treinar todos os três modelos
84      set_random_seed()
85
86      models = ['cnn', 'vit', 'hybrid']
87      results = {}
88
89      for model_type in models:
90          print(f"\n{'='*60}")
91          print(f"TREINANDO MODELO {model_type.upper()}")
92          print(f"{'='*60}")
93
94          cfg = config_optimized.config
95          trainer = ParasiteTrainerOptimized(model_type, cfg)
96
97          model_path = f"{cfg.model_save_dir}/{model_type}_best.pth
                ↪ "
98          if os.path.exists(model_path) and not force_train:
99              print(f"Modelo {model_type.upper()} já treinado
                    ↪ encontrado em {model_path}")
100             print("Pulando treinamento e indo direto para avaliaç
                    ↪ ão...")
101             print("Use --force-train para treinar novamente.")
102             checkpoint = torch.load(model_path, map_location='cpu
                    ↪ ')
103             trainer.best_val_accuracy = checkpoint.get('
                    ↪ best_val_accuracy', 0.0)
```

```
104             print(f"Melhor acurácia de validação carregada: {
                    ↪ trainer.best_val_accuracy:.4f}")
105         else:
106             if force_train and os.path.exists(model_path):
107                 print(f"Forçando retreinamento do modelo {
                        ↪ model_type.upper()}...")
108             else:
109                 print(f"Treinando modelo {model_type.upper()}..."
                        ↪ )
110             trainer.train()
111
112         print(f"\nAvaliando modelo {model_type.upper()} no
                ↪ conjunto de teste...")
113         test_results = trainer.evaluate()
114
115         trainer.save_results(test_results)
116
117         trainer.plot_training_history()
118
119         class_names = list(trainer.class_to_idx.keys())
120         trainer.plot_confusion_matrix(test_results['
                ↪ confusion_matrix'], class_names)
121
122         trainer.plot_test_accuracy_analysis(test_results)
123
124         results[model_type] = {
125             'accuracy': test_results['accuracy'],
126             'loss': test_results['loss'],
127             'classification_report': test_results['
                    ↪ classification_report'],
128             'best_val_accuracy': trainer.best_val_accuracy
129         }
130
131         print(f"Resultados do Modelo {model_type.upper()}:")
132         print(f"Acurácia de Teste: {test_results['accuracy']:.4f}
                ↪ ")
133         print(f"Perda de Teste: {test_results['loss']:.4f}")
134         print(f"Melhor Acurácia de Validação: {trainer.
                ↪ best_val_accuracy:.4f}")
135
136     return results
137
```

```
138 def compare_models(results):
139     # Comparar e visualizar resultados de todos os modelos
140     model_names_pt = {
141         'cnn': 'CNN',
142         'vit': 'ViT',
143         'hybrid': 'Hibrido'
144     }
145 
146     comparison_data = []
147     for model_type, result in results.items():
148         comparison_data.append({
149             'Model': model_names_pt.get(model_type, model_type.
                ↪ upper()),
150             'Test Accuracy': result['accuracy'],
151             'Test Loss': result['loss'],
152             'Best Val Accuracy': result['best_val_accuracy']
153         })
154 
155     df = pd.DataFrame(comparison_data)
156     print("\nComparação de Performance dos Modelos:")
157     print(df.to_string(index=False))
158 
159     # Criar gráficos de comparação
160     fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize
        ↪ =(16, 12))
161 
162     # Comparação de acurácia
163     models = [data['Model'] for data in comparison_data]
164     test_accuracies = [data['Test Accuracy'] for data in
        ↪ comparison_data]
165     val_accuracies = [data['Best Val Accuracy'] for data in
        ↪ comparison_data]
166 
167     x = np.arange(len(models))
168     width = 0.35
169 
170     bars1 = ax1.bar(x - width/2, test_accuracies, width, label='
        ↪ Acurácia de Teste', alpha=0.8, color='#2E86AB')
171     bars2 = ax1.bar(x + width/2, val_accuracies, width, label='
        ↪ Acurácia de Validação', alpha=0.8, color='#A23B72')
172     ax1.set_xlabel('Modelos')
173     ax1.set_ylabel('Acurácia')
```

```
174     ax1.set_title('Comparação de Acurácia dos Modelos')
175     ax1.set_xticks(x)
176     ax1.set_xticklabels(models)
177     ax1.legend()
178     ax1.grid(True, alpha=0.3)
179 
180     # Adicionar rótulos de valores
181     for bar, acc in zip(bars1, test_accuracies):
182         height = bar.get_height()
183         ax1.text(bar.get_x() + bar.get_width()/2., height + 0.01,
184                 f'{acc:.3f}', ha='center', va='bottom',
                    ↪ fontweight='bold')
185 
186     for bar, acc in zip(bars2, val_accuracies):
187         height = bar.get_height()
188         ax1.text(bar.get_x() + bar.get_width()/2., height + 0.01,
189                 f'{acc:.3f}', ha='center', va='bottom',
                    ↪ fontweight='bold')
190 
191     # Comparação de perda
192     test_losses = [data['Test Loss'] for data in comparison_data]
193     bars3 = ax2.bar(models, test_losses, alpha=0.8, color='#
        ↪ F18F01')
194     ax2.set_xlabel('Modelos')
195     ax2.set_ylabel('Perda')
196     ax2.set_title('Comparação de Perda dos Modelos')
197     ax2.grid(True, alpha=0.3)
198 
199     # Adicionar rótulos de valores
200     for bar, loss in zip(bars3, test_losses):
201         height = bar.get_height()
202         ax2.text(bar.get_x() + bar.get_width()/2., height + 0.05,
203                 f'{loss:.3f}', ha='center', va='bottom',
                    ↪ fontweight='bold')
204 
205     # Análise detalhada por classe
206     accuracy_data = []
207     for model_type, result in results.items():
208         report = result['classification_report']
209         for class_name, metrics in report.items():
210             if isinstance(metrics, dict) and 'precision' in
                ↪ metrics:
```

```
                accuracy_data.append({
                    'Model': model_names_pt.get(model_type,
                        ↪ model_type.upper()),
                    'Class': class_name,
                    'Precision': metrics['precision'],
                    'Recall': metrics['recall'],
                    'F1-Score': metrics['f1-score']
                })

    accuracy_df = pd.DataFrame(accuracy_data)

    # Comparação de F1-Score por classe
    pivot_f1 = accuracy_df.pivot(index='Class', columns='Model',
        ↪ values='F1-Score')
    pivot_f1.plot(kind='bar', ax=ax3, alpha=0.8, color=['#2E86AB'
        ↪ , '#A23B72', '#F18F01'])
    ax3.set_title('F1-Score por Classe e Modelo')
    ax3.set_xlabel('Classes')
    ax3.set_ylabel('F1-Score')
    ax3.legend(title='Modelo')
    ax3.tick_params(axis='x', rotation=45)
    ax3.grid(True, alpha=0.3)

    # Gráfico de dispersão de precisão vs revocação
    for i, model in enumerate(models):
        model_data = accuracy_df[accuracy_df['Model'] == model]
        ax4.scatter(model_data['Precision'], model_data['Recall'
            ↪ ],
                    s=100, alpha=0.7, label=model,
                    color=['#2E86AB', '#A23B72', '#F18F01'][i])

    ax4.set_title('Precisão vs Revocação por Classe')
    ax4.set_xlabel('Precisão')
    ax4.set_ylabel('Revocação')
    ax4.legend()
    ax4.grid(True, alpha=0.3)

    # Adicionar linha diagonal de referência
    ax4.plot([0, 1], [0, 1], 'k--', alpha=0.5, label='Linha de
        ↪ Referência')

    plt.tight_layout()
```

```
248     cfg = config_optimized.config
249     plt.savefig(f"{cfg.results_dir}/model_comparison_optimized.
            ↪ png", dpi=300, bbox_inches='tight')
250     plt.show()
251 
252     comparison_path = f"{cfg.results_dir}/
            ↪ model_comparison_optimized.json"
253     with open(comparison_path, 'w') as f:
254         json.dump(results, f, indent=4)
255 
256     print(f"\nResultados da comparação salvos em {comparison_path
            ↪ }")
257 
258     # Imprimir melhor modelo
259     best_model = max(results.items(), key=lambda x: x[1]['
            ↪ accuracy'])
260     print(f"\nMELHOR MODELO: {best_model[0].upper()}")
261     print(f"   Acurácia de Teste: {best_model[1]['accuracy']:.4f}
            ↪ ")
262     print(f"   Perda de Teste: {best_model[1]['loss']:.4f}")
263 
264     # Analisar sobreajuste
265     print(f"\nANALISE DE SOBREAJUSTE:")
266     for model_type, result in results.items():
267         val_acc = result['best_val_accuracy'] / 100.0
268         test_acc = result['accuracy']
269         overfitting_gap = val_acc - test_acc
270         print(f"   {model_type.upper()}: Validação {val_acc:.3f}
                ↪ -> Teste {test_acc:.3f} (Gap: {overfitting_gap:.3f})
                ↪ ")
271 
272 def main():
273     parser = argparse.ArgumentParser(description='Treinar modelos
            ↪  de detecção de parasitas')
274     parser.add_argument('--model', type=str, choices=['cnn', 'vit
            ↪ ', 'hybrid', 'all'],
275                         default='all', help='Modelo a ser treinado
                            ↪  (padrão: all)')
276     parser.add_argument('--force-train', action='store_true',
            ↪ help='Forçar retreinamento de modelos que já existem')
277 
278     args = parser.parse_args()
```

```
279
280     print(f"\n{'='*70}")
281     print("EXPERIMENTO")
282     print(f"{'='*70}")
283     print("Configurações:")
284     cfg = config_optimized.config
285     print(f"  - Tamanho da imagem: {cfg.image_size}x{cfg.
            ↪ image_size}")
286     print(f"  - Tamanho do batch: {cfg.batch_size}")
287     print(f"  - Taxa de aprendizado: {cfg.learning_rate}")
288     print(f"  - Taxa de decaimento: {cfg.weight_decay}")
289     print(f"  - Suavizacao de rotulos: {cfg.label_smoothing}")
290     print(f"  - Gradiente: {cfg.gradient_clip}")
291     print(f"  - Agendador de taxa de aprendizado:
            ↪ ReduceLROnPlateau")
292     print(f"  - Tratamento de dados: Embaçamento + Ruído")
293     print(f"{'='*70}")
294
295     if args.model == 'all':
296         results = train_all_models(args.force_train)
297         compare_models(results)
298     else:
299         result = train_single_model(args.model, args.force_train)
300         compare_models({args.model: result})
301
302 if __name__ == "__main__":
303     main()
```

A.2 Treinador Otimizado: *trainer_optimized.py*

Listing A.2 – Classe do treinador otimizado

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 from torch.optim.lr_scheduler import ReduceLROnPlateau
5 import numpy as np
6 import matplotlib.pyplot as plt
7 import seaborn as sns
8 from sklearn.metrics import classification_report,
    ↪ confusion_matrix
```

```
import json
import os
from tqdm import tqdm
import config_optimized
from dataset_optimized import get_dataloaders_optimized
from models import get_model, get_model_summary

class ParasiteTrainerOptimized:
    def __init__(self, model_type: str, config):
        self.model_type = model_type
        self.config = config
        self.device = torch.device(config.device)

        # Obter dataloaders
        self.train_loader, self.val_loader, self.test_loader,
            ↪ self.class_to_idx = get_dataloaders_optimized(config
            ↪ )

        self.model = self._create_model()
        self.model.to(self.device)

        # Função de perda com suavização de rótulos
        self.criterion = nn.CrossEntropyLoss(label_smoothing=
            ↪ config.label_smoothing)

        # Otimizador
        self.optimizer = optim.AdamW(
            self.model.parameters(),
            lr=config.learning_rate,
            weight_decay=config.weight_decay
        )

        # Agendador de taxa de aprendizado
        self.scheduler = ReduceLROnPlateau(
            self.optimizer,
            mode='max',
            factor=config.scheduler_factor,
            patience=config.scheduler_patience,
            min_lr=config.scheduler_min_lr
        )

        self.train_losses = []
```

```
48          self.val_losses = []
49          self.train_accuracies = []
50          self.val_accuracies = []
51          self.best_val_accuracy = 0.0
52          self.patience_counter = 0
53
54      def _create_model(self):
55          # Criar modelo com base no tipo
56          from models import get_model, get_model_summary
57          model = get_model(self.model_type, self.config)
58          get_model_summary(model, input_size=(3, self.config.
                ↪ image_size, self.config.image_size))
59          return model
60
61      def get_model_name_pt(self):
62          # Traduzir
63          model_names = {
64              'cnn': 'CNN',
65              'vit': 'ViT',
66              'hybrid': 'Híbrido'
67          }
68          return model_names.get(self.model_type, self.model_type.
                ↪ upper())
69
70      def train_epoch(self):
71          # Treinar por uma época
72          self.model.train()
73          total_loss = 0.0
74          correct = 0
75          total = 0
76
77          progress_bar = tqdm(self.train_loader, desc=f"Treinando {
                ↪ self.model_type.upper()}")
78
79          for batch_idx, (data, target) in enumerate(progress_bar):
80              data, target = data.to(self.device), target.to(self.
                    ↪ device)
81
82              self.optimizer.zero_grad()
83              output = self.model(data)
84              loss = self.criterion(output, target)
85
```

```
86              torch.nn.utils.clip_grad_norm_(self.model.parameters
                    ↪ (), self.config.gradient_clip)
87
88              loss.backward()
89              self.optimizer.step()
90
91              total_loss += loss.item()
92              pred = output.argmax(dim=1, keepdim=True)
93              correct += pred.eq(target.view_as(pred)).sum().item()
94              total += target.size(0)
95
96              progress_bar.set_postfix({
97                  'Loss': f'{loss.item():.4f}',
98                  'Acc': f'{100. * correct / total:.2f}%'
99              })
100
101         return total_loss / len(self.train_loader), correct /
                ↪ total
102
103     def validate_epoch(self):
104         # Validar época
105         self.model.eval()
106         total_loss = 0.0
107         correct = 0
108         total = 0
109
110         with torch.no_grad():
111             for data, target in tqdm(self.val_loader, desc=f"
                    ↪ Validando {self.model_type.upper()}"):
112                 data, target = data.to(self.device), target.to(
                        ↪ self.device)
113                 output = self.model(data)
114                 loss = self.criterion(output, target)
115
116                 total_loss += loss.item()
117                 pred = output.argmax(dim=1, keepdim=True)
118                 correct += pred.eq(target.view_as(pred)).sum().
                        ↪ item()
119                 total += target.size(0)
120
121         return total_loss / len(self.val_loader), correct / total
122
```

```
123     def train(self):
124         # Treinar com parada antecipada
125         print(f"\n{'='*60}")
126         print(f"TREINANDO MODELO {self.model_type.upper()}")
127         print(f"{'='*60}")
128 
129         for epoch in range(self.config.num_epochs):
130             print(f"\nEpoca {epoch+1}/{self.config.num_epochs}")
131 
132             train_loss, train_acc = self.train_epoch()
133 
134             val_loss, val_acc = self.validate_epoch()
135 
136             self.scheduler.step(val_acc)
137 
138             # Armazenar histórico
139             self.train_losses.append(train_loss)
140             self.val_losses.append(val_loss)
141             self.train_accuracies.append(train_acc)
142             self.val_accuracies.append(val_acc)
143 
144             print(f"Perda de Treino: {train_loss:.4f}, Acurácia
                ↪ de Treino: {train_acc:.4f}")
145             print(f"Perda de Validação: {val_loss:.4f}, Acurácia
                ↪ de Validação: {val_acc:.4f}")
146             print(f"Taxa de Aprendizado: {self.optimizer.
                ↪ param_groups[0]['lr']:.2e}")
147 
148             # Parada antecipada
149             if val_acc > self.best_val_accuracy:
150                 self.best_val_accuracy = val_acc
151                 self.patience_counter = 0
152                 # Salvar melhor modelo
153                 torch.save(self.model.state_dict(),
154                            f"{self.config.model_save_dir}/{self.
                               ↪ model_type}_best.pth")
155                 print(f"Novo melhor modelo salvo! Acurácia: {
                    ↪ val_acc:.4f}")
156             else:
157                 self.patience_counter += 1
158                 print(f"Parada Antecipada: {self.patience_counter
                    ↪ }/{self.config.patience}")
```

```
159
160                 if self.patience_counter >= self.config.patience:
161                     print(f"Parada Antecipada ativada após {epoch
                        ↪ +1} épocas")
162                     break
163
164         print(f"\nMelhor acurácia de validação: {self.
               ↪ best_val_accuracy:.4f}")
165
166     def evaluate(self):
167         # Avaliar no conjunto de teste
168         print(f"\nAvaliando modelo {self.model_type.upper()} no
               ↪ conjunto de teste...")
169
170         # Carregar melhor modelo
171         best_model_path = f"{self.config.model_save_dir}/{self.
               ↪ model_type}_best.pth"
172         if os.path.exists(best_model_path):
173             self.model.load_state_dict(torch.load(best_model_path
                   ↪ , map_location=self.device))
174             print("Modelo melhor carregado para avaliação")
175
176         self.model.eval()
177         test_loss = 0.0
178         all_predictions = []
179         all_targets = []
180
181         with torch.no_grad():
182             for data, target in tqdm(self.test_loader, desc="
                   ↪ Avaliando"):
183                 data, target = data.to(self.device), target.to(
                       ↪ self.device)
184                 output = self.model(data)
185                 loss = self.criterion(output, target)
186
187                 test_loss += loss.item()
188                 pred = output.argmax(dim=1, keepdim=True)
189
190                 all_predictions.extend(pred.cpu().numpy().flatten
                       ↪ ())
191                 all_targets.extend(target.cpu().numpy())
192
```

```
193         test_loss /= len(self.test_loader)
194         test_accuracy = sum(1 for x, y in zip(all_predictions,
                ↪ all_targets) if x == y) / len(all_targets)
195 
196         # Relatório de classificação
197         class_names = list(self.class_to_idx.keys())
198         report = classification_report(all_targets,
                ↪ all_predictions,
199                                        target_names=class_names,
                                           ↪ output_dict=True)
200 
201         # Matriz de confusão
202         cm = confusion_matrix(all_targets, all_predictions)
203 
204         return {
205             'accuracy': test_accuracy,
206             'loss': test_loss,
207             'classification_report': report,
208             'confusion_matrix': cm,
209             'predictions': all_predictions,
210             'targets': all_targets
211         }
212 
213     def save_results(self, results):
214         # Salvar resultados em JSON
215         results_path = f"{self.config.results_dir}/{self.
                ↪ model_type}_results.json"
216 
217         # Função auxiliar para converter tipos numpy para tipos
                ↪ nativos Python
218         def convert_numpy_types(obj):
219             if isinstance(obj, np.ndarray):
220                 if obj.size == 1:
221                     return obj.item()
222                 else:
223                     return obj.tolist()
224             elif isinstance(obj, np.integer):
225                 return int(obj)
226             elif isinstance(obj, np.floating):
227                 return float(obj)
228             elif isinstance(obj, dict):
```

```
229                 return {key: convert_numpy_types(value) for key,
                        ↪ value in obj.items()}
230             elif isinstance(obj, list):
231                 return [convert_numpy_types(item) for item in obj
                        ↪ ]
232             else:
233                 return obj
234
235         # Converter todos os resultados para formato JSON-
                ↪ serializável
236         results_to_save = {
237             'acuracia': convert_numpy_types(results['accuracy']),
238             'perda': convert_numpy_types(results['loss']),
239             'relatorio_classificacao': convert_numpy_types(
                    ↪ results['classification_report']),
240             'matriz_confusao': convert_numpy_types(results['
                    ↪ confusion_matrix']),
241             'predicoes': convert_numpy_types(results['predictions
                    ↪ ']),
242         }
243
244         with open(results_path, 'w', encoding='utf-8') as f:
245             json.dump(results_to_save, f, indent=4, ensure_ascii=
                    ↪ False)
246
247         print(f"Resultados salvos em {results_path}")
248
249     def plot_training_history(self):
250         # Plotar histórico de treinamento
251         fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
252         num_epochs = len(self.train_losses)
253         epochs = list(range(1, num_epochs + 1))
254
255         def get_epoch_ticks(num_epochs):
256             if num_epochs <= 20:
257                 return list(range(1, num_epochs + 1))
258             elif num_epochs <= 50:
259                 ticks = list(range(1, num_epochs + 1, 5))
260             else:
261                 ticks = list(range(1, num_epochs + 1, 10))
262             if ticks[-1] != num_epochs:
263                 ticks.append(num_epochs)
```

```
264             return ticks
265         epoch_ticks = get_epoch_ticks(num_epochs)
266 
267         # Plotar perda
268         ax1.plot(epochs, self.train_losses, label='Treino', color
                ↪ ='blue', marker='o')
269         ax1.plot(epochs, self.val_losses, label='Validação',
                ↪ color='red', marker='o')
270         ax1.set_title(f'Histórico de Perda - {self.
                ↪ get_model_name_pt()}')
271         ax1.set_xlabel('Epoca')
272         ax1.set_ylabel('Perda')
273         ax1.legend()
274         ax1.grid(True, alpha=0.3)
275         max_loss = max(self.train_losses + self.val_losses) if (
                ↪ self.train_losses and self.val_losses) else 1
276         ax1.set_ylim(0, max_loss * 1.1)
277         ax1.set_xticks(epoch_ticks)
278 
279         # Plotar acurácia
280         train_acc_pct = [acc * 100 for acc in self.
                ↪ train_accuracies]
281         val_acc_pct = [acc * 100 for acc in self.val_accuracies]
282         ax2.plot(epochs, train_acc_pct, label='Treino', color='
                ↪ blue', marker='o')
283         ax2.plot(epochs, val_acc_pct, label='Validação', color='
                ↪ red', marker='o')
284         ax2.set_title(f'Histórico de Acurácia - {self.
                ↪ get_model_name_pt()}')
285         ax2.set_xlabel('Epoca')
286         ax2.set_ylabel('Acurácia (%)')
287         ax2.legend()
288         ax2.grid(True, alpha=0.3)
289         ax2.set_ylim(0, 100)
290         ax2.set_yticks([0, 20, 40, 60, 80, 100])
291         ax2.set_yticklabels(['0%', '20%', '40%', '60%', '80%', '
                ↪ 100%'])
292         ax2.set_xticks(epoch_ticks)
293 
294         plt.tight_layout()
295         save_path = f"{self.config.results_dir}/{self.model_type}
                ↪ _training_history.png"
```

```
296         plt.savefig(save_path, dpi=300, bbox_inches='tight')
297         plt.close()
298         print(f"Gráfico de histórico salvo em {save_path}")
299 
300     def plot_confusion_matrix(self, cm, class_names):
301         # Plotar matriz de confusão
302         plt.figure(figsize=(12, 10))
303         sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
304                     xticklabels=class_names, yticklabels=
                        ↪ class_names)
305         plt.title(f'Matriz de Confusão - {self.get_model_name_pt
                ↪ ()}')
306         plt.xlabel('Predição')
307         plt.ylabel('Real')
308         plt.xticks(rotation=45)
309         plt.yticks(rotation=0)
310 
311         save_path = f"{self.config.results_dir}/{self.model_type}
                ↪ _confusion_matrix.png"
312         plt.savefig(save_path, dpi=300, bbox_inches='tight')
313         plt.close()
314         print(f"Matriz de confusão salva em {save_path}")
315 
316     def plot_test_accuracy_analysis(self, test_results):
317         # Plotar análise detalhada de acurácia de teste
318         class_names = list(self.class_to_idx.keys())
319         report = test_results['classification_report']
320 
321         classes = []
322         precisions = []
323         recalls = []
324         f1_scores = []
325 
326         for class_name, metrics in report.items():
327             if isinstance(metrics, dict) and 'precision' in
                    ↪ metrics:
328                 classes.append(class_name)
329                 precisions.append(metrics['precision'])
330                 recalls.append(metrics['recall'])
331                 f1_scores.append(metrics['f1-score'])
332 
```

```
333         fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2,
                ↪ figsize=(16, 12))
334
335         # 1. Acurácia geral de teste
336         test_acc = test_results['accuracy']
337         test_loss = test_results['loss']
338
339         ax1.bar(['Acurácia', 'Perda'], [test_acc, test_loss],
340                 color=['#2E86AB', '#A23B72'], alpha=0.8)
341         ax1.set_title(f'Métricas Gerais de Teste - {self.
                ↪ get_model_name_pt()}')
342         ax1.set_ylabel('Valor')
343         ax1.grid(True, alpha=0.3)
344
345         for i, (acc, loss) in enumerate([(test_acc, test_loss)]):
346             ax1.text(0, acc + 0.01, f'{acc:.3f}', ha='center', va
                    ↪ ='bottom', fontweight='bold')
347             ax1.text(1, loss + 0.01, f'{loss:.3f}', ha='center',
                    ↪ va='bottom', fontweight='bold')
348
349         # 2. Precisão por classe
350         bars1 = ax2.bar(range(len(classes)), precisions, alpha
                ↪ =0.8, color='#2E86AB')
351         ax2.set_title(f'Precisão por Classe - {self.
                ↪ get_model_name_pt()}')
352         ax2.set_xlabel('Classes')
353         ax2.set_ylabel('Precisão')
354         ax2.set_xticks(range(len(classes)))
355         ax2.set_xticklabels(classes, rotation=45, ha='right')
356         ax2.grid(True, alpha=0.3)
357
358         for bar, prec in zip(bars1, precisions):
359             height = bar.get_height()
360             ax2.text(bar.get_x() + bar.get_width()/2., height +
                    ↪ 0.01,
361                      f'{prec:.3f}', ha='center', va='bottom',
                            ↪ fontweight='bold', fontsize=8)
362
363         # 3. Revocação por classe
364         bars2 = ax3.bar(range(len(classes)), recalls, alpha=0.8,
                ↪ color='#A23B72')
```

```
365         ax3.set_title(f'Revocação por Classe - {self.
               ↪ get_model_name_pt()}')
366         ax3.set_xlabel('Classes')
367         ax3.set_ylabel('Revocação')
368         ax3.set_xticks(range(len(classes)))
369         ax3.set_xticklabels(classes, rotation=45, ha='right')
370         ax3.grid(True, alpha=0.3)
371
372         for bar, rec in zip(bars2, recalls):
373             height = bar.get_height()
374             ax3.text(bar.get_x() + bar.get_width()/2., height +
                   ↪ 0.01,
375                      f'{rec:.3f}', ha='center', va='bottom',
                         ↪ fontweight='bold', fontsize=8)
376
377         # 4. F1-Score por classe
378         bars3 = ax4.bar(range(len(classes)), f1_scores, alpha
               ↪ =0.8, color='#F18F01')
379         ax4.set_title(f'F1-Score por Classe - {self.
               ↪ get_model_name_pt()}')
380         ax4.set_xlabel('Classes')
381         ax4.set_ylabel('F1-Score')
382         ax4.set_xticks(range(len(classes)))
383         ax4.set_xticklabels(classes, rotation=45, ha='right')
384         ax4.grid(True, alpha=0.3)
385
386         for bar, f1 in zip(bars3, f1_scores):
387             height = bar.get_height()
388             ax4.text(bar.get_x() + bar.get_width()/2., height +
                   ↪ 0.01,
389                      f'{f1:.3f}', ha='center', va='bottom',
                         ↪ fontweight='bold', fontsize=8)
390
391         plt.tight_layout()
392         save_path = f"{self.config.results_dir}/{self.model_type}
               ↪ _test_accuracy_analysis.png"
393         plt.savefig(save_path, dpi=300, bbox_inches='tight')
394         plt.close()
395         print(f"Análise detalhada de acurácia de teste salva em {
               ↪ save_path}")
```

A.3 Configuração: *config_optimized.py*

Listing A.3 – Arquivo de configuração otimizado

```
import os
import torch
from dataclasses import dataclass
from typing import List, Tuple

class Config:

    # Dados
    data_dir = "/home/edvl/TCC/Chula-ParasiteEgg-11/Chula-ParasiteEgg-11/Chula-ParasiteEgg-11/data"
    test_data_dir = "/home/edvl/TCC/Chula-ParasiteEgg-11_test/test/data"
    train_data_path = "/home/edvl/TCC/Chula-ParasiteEgg-11/Chula-ParasiteEgg-11/Chula-ParasiteEgg-11/data"
    test_data_path = "/home/edvl/TCC/Chula-ParasiteEgg-11_test/test/data"

    # Modelo
    num_classes = 11
    input_size = (384, 384)
    image_size = 384

    # Treinamento
    batch_size = 2
    num_epochs = 50
    learning_rate = 1e-4
    weight_decay = 1e-4
    gradient_clip = 1.0

    # Tratamento de dados
    use_augmentation = True
    train_transform = True
    test_transform = False
    mixup_alpha = 0.2
    cutmix_alpha = 1.0
    cutmix_prob = 0.5

    # Aprendizado em conjunto
```

```
35      ensemble_size = 3
36      diversity_weight = 0.1
37
38      # Regularização
39      dropout_rate = 0.3
40      label_smoothing = 0.1
41
42      # Parada antecipada
43      patience = 10
44
45      # Memória e processamento
46      num_workers = 1
47      pin_memory = True
48
49      # Dispositivo
50      device = 'cuda' if torch.cuda.is_available() else 'cpu'
51
52      # Configurações específicas dos modelos
53      # CNN (EfficientNetV2-S)
54      cnn_model_name = "tf_efficientnetv2_s"
55      cnn_dropout = 0.2
56
57      # Vision Transformer (Tiny ViT)
58      vit_model_name = "vit_tiny_patch16_224"
59      vit_patch_size = 16
60      vit_embed_dim = 192
61      vit_depths = (3, 3, 3)
62      vit_num_heads = (3, 6, 12)
63
64      # Modelo Híbrido
65      hybrid_cnn_backbone = "tf_efficientnetv2_s"
66      hybrid_vit_model = "vit_tiny_patch16_224"
67      hybrid_fusion_dim = 64
68
69      # Diretórios de saída
70      model_save_dir = "models_optimized"
71      results_dir = "results_optimized"
72
73      # Registro da execução
74      log_interval = 100
75      save_interval = 5
76
```

```
77 config = Config()
```

A.4 Modelos: *models.py*

Listing A.4 – Implementação dos modelos

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4  import timm
5
6  class EfficientNetV2CNN(nn.Module):
7      # Modelo CNN usando EfficientNetV2-S para classificação de
       ↪ parasitas
8
9      def __init__(self, config, num_classes: int = 11):
10         super(EfficientNetV2CNN, self).__init__()
11
12         # Carregar EfficientNetV2-S
13         self.backbone = timm.create_model(
14             config.cnn_model_name,
15             pretrained=True,
16             num_classes=0
17         )
18
19         feature_dim = self.backbone.num_features
20
21         self.classifier = nn.Sequential(
22             nn.Dropout(config.cnn_dropout),
23             nn.Linear(feature_dim, 512),
24             nn.ReLU(),
25             nn.Dropout(config.cnn_dropout),
26             nn.Linear(512, num_classes)
27         )
28
29     def forward(self, x):
30         features = self.backbone(x)
31         output = self.classifier(features)
32         return output
33
34 class TinyViTTransformer(nn.Module):
```

```
35      # Modelo Vision Transformer usando Tiny ViT para classificaçã
            ↪ o de parasitas
36
37      def __init__(self, config, num_classes: int = 11):
38          super(TinyViTTransformer, self).__init__()
39
40          # Carregar Tiny ViT
41          self.backbone = timm.create_model(
42              config.vit_model_name,
43              pretrained=True,
44              num_classes=0
45          )
46
47          feature_dim = self.backbone.num_features
48
49          self.classifier = nn.Sequential(
50              nn.LayerNorm(feature_dim),
51              nn.Linear(feature_dim, 512),
52              nn.GELU(),
53              nn.Dropout(0.1),
54              nn.Linear(512, num_classes)
55          )
56
57      def forward(self, x):
58          features = self.backbone(x)
59          output = self.classifier(features)
60          return output
61
62  class HybridModel(nn.Module):
63      # Modelo Híbrido combinando CNN (EfficientNetV2-S) e Vision
            ↪ Transformer (Tiny ViT)
64      def __init__(self, config, num_classes: int = 11):
65          super(HybridModel, self).__init__()
66
67          # (EfficientNetV2-S)
68          self.cnn_backbone = timm.create_model(
69              config.hybrid_cnn_backbone,
70              pretrained=True,
71              num_classes=0
72          )
73
74          # Vision Transformer (Tiny ViT)
```

```
75          self.vit_backbone = timm.create_model(
76              config.hybrid_vit_model,
77              pretrained=True,
78              num_classes=0
79          )
80
81          cnn_feature_dim = self.cnn_backbone.num_features
82          vit_feature_dim = self.vit_backbone.num_features
83
84          # Fusão de características
85          fusion_dim = config.hybrid_fusion_dim
86          self.fusion = nn.Sequential(
87              nn.Linear(cnn_feature_dim + vit_feature_dim,
                ↪ fusion_dim),
88              nn.LayerNorm(fusion_dim),
89              nn.GELU(),
90              nn.Dropout(0.1),
91              nn.Linear(fusion_dim, fusion_dim // 2),
92              nn.GELU(),
93              nn.Dropout(0.1)
94          )
95
96          # Classificador final
97          self.classifier = nn.Linear(fusion_dim // 2, num_classes)
98
99          # Mecanismo de atenção para ponderação de características
100         self.attention = nn.MultiheadAttention(
101             embed_dim=fusion_dim // 2,
102             num_heads=2,
103             dropout=0.1,
104             batch_first=True
105         )
106
107     def forward(self, x):
108         # Extrair características de ambos os modelos
109         cnn_features = self.cnn_backbone(x)
110         vit_features = self.vit_backbone(x)
111
112         # Concatenar características
113         combined_features = torch.cat([cnn_features, vit_features
              ↪ ], dim=1)
114
```

```
115         # Aplicar fusão
116         fused_features = self.fusion(combined_features)
117 
118         # Aplicar auto-atenção
119         fused_features = fused_features.unsqueeze(1)  # Adicionar
               ↪ dimensão de sequência
120         attended_features, _ = self.attention(fused_features,
               ↪ fused_features, fused_features)
121         attended_features = attended_features.squeeze(1)
122 
123         # Classificação final
124         output = self.classifier(attended_features)
125         return output
126 
127 def get_model(model_type: str, config, num_classes: int = 11):
128     # Função fábrica para obter o modelo especificado
129 
130     if model_type.lower() == "cnn":
131         return EfficientNetV2CNN(config, num_classes=num_classes)
132 
133     elif model_type.lower() == "vit":
134         return TinyViTTransformer(config, num_classes=num_classes
               ↪ )
135 
136     elif model_type.lower() == "hybrid":
137         return HybridModel(config, num_classes=num_classes)
138 
139     else:
140         raise ValueError(f"Tipo de modelo desconhecido: {
               ↪ model_type}. Escolha entre ['cnn', 'vit', 'hybrid']"
               ↪ )
141 
142 def count_parameters(model):
143     # Contar parâmetros treináveis no modelo
144     return sum(p.numel() for p in model.parameters() if p.
          ↪ requires_grad)
145 
146 def get_model_summary(model, input_size=(3, 224, 224)):
147     # Obter resumo do modelo com contagem de parâmetros
148     total_params = count_parameters(model)
149     trainable_params = sum(p.numel() for p in model.parameters()
          ↪ if p.requires_grad)
```

```
150
151     print(f"Resumo do Modelo:")
152     print(f"Total de Parâmetros: {total_params:,}")
153     print(f"Parâmetros Treináveis: {trainable_params:,}")
154     print(f"Tamanho de Entrada: {input_size}")
155
156     return total_params, trainable_params
```

A.5 Conjunto de Dados: *dataset_optimized.py*

Listing A.5 – Classe do conjunto de dados otimizado

```
1  import os
2  import torch
3  from torch.utils.data import Dataset, DataLoader
4  from PIL import Image
5  import albumentations as A
6  from albumentations.pytorch import ToTensorV2
7  import numpy as np
8  from typing import Dict, List, Tuple, Optional
9  import config_optimized
10 import json
11
12 class ParasiteDataset(Dataset):
13     def __init__(self, data_path: str, transform=None, is_train:
       ↪ bool = True, class_to_idx: Optional[Dict[str, int]] =
       ↪ None, label_json_path: Optional[str] = None):
14         self.data_path = data_path
15         self.transform = transform
16         self.is_train = is_train
17         self.class_to_idx = class_to_idx
18         self.label_json_path = label_json_path
19
20         # Carregar todos os arquivos de imagem e seus rótulos
21         self.images, self.labels, self.class_to_idx = self.
             ↪ _load_dataset()
22
23     def _load_dataset(self) -> Tuple[List[str], List[int],
       ↪ Optional[Dict[str, int]]]:
24         images = []
25         labels = []
```

```
26
27          # Se estiver usando arquivo de rótulos no formato COCO (
                ↪ para conjunto de teste)
28          if self.label_json_path is not None:
29              # Carregar arquivo de rótulos
30              with open(self.label_json_path, 'r') as f:
31                  label_data = json.load(f)
32
33              file_to_id = {img['file_name']: img['id'] for img in
                    ↪ label_data['images']}
34
35              imageid_to_catid = {}
36              for ann in label_data['annotations']:
37                  if ann['image_id'] not in imageid_to_catid:
38                      imageid_to_catid[ann['image_id']] = ann['
                            ↪ category_id']
39
40              catid_to_name = {cat['id']: cat['name'] for cat in
                    ↪ label_data['categories']}
41
42              class_to_idx = self.class_to_idx
43
44              for filename in os.listdir(self.data_path):
45                  if filename.lower().endswith(('.jpg', '.jpeg', '.
                        ↪ png')):
46                      img_path = os.path.join(self.data_path,
                            ↪ filename)
47
48                      image_id = file_to_id.get(filename)
49                      if image_id is None:
50                          continue
51
52                      category_id = imageid_to_catid.get(image_id)
53                      if category_id is None:
54                          continue
55
56                      class_name = catid_to_name.get(category_id)
57                      if class_name is None:
58                          continue
59
60                      if class_name in class_to_idx:
61                          images.append(img_path)
```

```
62                          labels.append(class_to_idx[class_name])
63              return images, labels, class_to_idx
64
65          # Comportamento padrão (conjunto de treino/validação)
66          extracted_class_names = []
67          if self.class_to_idx is None:
68              # Construir mapeamento a partir deste conjunto de
                ↪ dados (para conjunto de treino)
69              class_names = set()
70              for filename in os.listdir(self.data_path):
71                  if filename.lower().endswith(('.jpg', '.jpeg', '.
                    ↪ png')):
72                      class_name = filename.split('_')[0]
73                      class_names.add(class_name)
74                      if len(extracted_class_names) < 10:
75                          extracted_class_names.append(class_name)
76              class_names = sorted(list(class_names))
77              class_to_idx = {class_name: idx for idx, class_name
                ↪ in enumerate(class_names)}
78              print("[DEBUG] Built class_to_idx:", class_to_idx)
79              print("[DEBUG] First 10 extracted class names from
                ↪ filenames:", extracted_class_names)
80          else:
81              # Usar mapeamento fornecido (para conjunto de teste)
82              class_to_idx = self.class_to_idx
83
84          # Carregar imagens e atribuir rótulos usando mapeamento
85          for filename in os.listdir(self.data_path):
86              if filename.lower().endswith(('.jpg', '.jpeg', '.png'
                ↪ )):
87                  img_path = os.path.join(self.data_path, filename)
88                  class_name = filename.split('_')[0]
89                  if class_to_idx is not None:
90                      idx = class_to_idx.get(class_name)
91                      if idx is None:
92                          continue
93                      images.append(img_path)
94                      labels.append(idx)
95                  else:
96                      continue
97          return images, labels, class_to_idx
98
```

```
99      def __len__(self):
100         return len(self.images)
101
102     def __getitem__(self, idx):
103         img_path = self.images[idx]
104         label = self.labels[idx]
105
106         # Carregar imagem
107         image = Image.open(img_path).convert('RGB')
108         image = np.array(image)
109
110         # Aplicar transformações
111         if self.transform:
112             transformed = self.transform(image=image)
113             image = transformed['image']
114
115         return image, label
116
117 def get_transforms_optimized(image_size: int = 384, is_train:
    ↪ bool = True, config=None):
118     if is_train:
119         return A.Compose([
120             # Redimensionar para tamanho alvo e aplicar normaliza
                ↪ ção
121             A.Resize(image_size, image_size),
122             A.HorizontalFlip(p=0.5),
123             A.VerticalFlip(p=0.3),
124             A.RandomRotate90(p=0.3),
125
126             # Transformações geométricas moderadas
127             A.Affine(
128                 translate_percent=0.1,  # Reduzido
129                 scale=(0.9, 1.1),       # Reduzido
130                 rotate=(-15, 15),       # Reduzido baseado no
                        ↪ CoAtNet
131                 p=0.6
132             ),
133
134             # Embaçamento e ruído
135             A.OneOf([
136                 A.GaussianBlur(blur_limit=(3, 7), p=0.5),
137                 A.MotionBlur(blur_limit=3, p=0.3),
```

```
138                 A.MedianBlur(blur_limit=3, p=0.2),
139             ], p=0.4),
140 
141             # Ruído
142             A.OneOf([
143                 A.GaussNoise(var_limit=(5.0, 15.0), p=0.5),
144                 A.ISONoise(color_shift=(0.01, 0.05), intensity
                        ↪ =(0.1, 0.5), p=0.3),
145                 A.MultiplicativeNoise(multiplier=(0.9, 1.1), p
                        ↪ =0.2),
146             ], p=0.4),
147 
148             # Ajustes de cor moderados
149             A.OneOf([
150                 A.RandomBrightnessContrast(
151                     brightness_limit=0.2,  # Reduzido
152                     contrast_limit=0.2,    # Reduzido
153                     p=0.5
154                 ),
155                 A.HueSaturationValue(
156                     hue_shift_limit=20,    # Reduzido
157                     sat_shift_limit=30,    # Reduzido
158                     val_shift_limit=20,    # Reduzido
159                     p=0.3
160                 ),
161                 A.CLAHE(clip_limit=2.0, tile_grid_size=(8, 8), p
                        ↪ =0.2),
162             ], p=0.4),
163 
164             # Dropout espacial
165             A.CoarseDropout(
166                 max_holes=4, max_height=16, max_width=16,
167                 min_holes=1, min_height=4, min_width=4,
168                 p=0.2
169             ),
170 
171             # Normalização ImageNet
172             A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
                    ↪ 0.224, 0.225]),
173             ToTensorV2(),
174         ])
175     else:
```

```
176         return A.Compose([
177             A.Resize(image_size, image_size),
178             A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
                ↪ 0.224, 0.225]),
179             ToTensorV2(),
180         ])
181
182 def get_dataloaders_optimized(config):
183     # Transformações
184     train_transform = get_transforms_optimized(config.image_size,
            ↪  is_train=True, config=config)
185     val_transform = get_transforms_optimized(config.image_size,
            ↪ is_train=False, config=config)
186
187     # Dividir dados de treino em treino/validação (80/20)
188     full_dataset = ParasiteDataset(config.train_data_path,
            ↪ transform=train_transform, is_train=True)
189
190     # Calcular índices de divisão
191     total_size = len(full_dataset)
192     train_size = int(0.8 * total_size)
193     val_size = total_size - train_size
194
195     train_dataset, val_dataset = torch.utils.data.random_split(
196         full_dataset, [train_size, val_size]
197     )
198
199     val_dataset.dataset.transform = val_transform
200
201     train_loader = DataLoader(
202         train_dataset,
203         batch_size=config.batch_size,
204         shuffle=True,
205         num_workers=config.num_workers,
206         pin_memory=config.pin_memory
207     )
208
209     val_loader = DataLoader(
210         val_dataset,
211         batch_size=config.batch_size,
212         shuffle=False,
213         num_workers=config.num_workers,
```

```
214         pin_memory=config.pin_memory
215     )
216 
217     # Conjunto de teste
218     test_label_json = os.path.join("..", "Chula-ParasiteEgg-11
        ↪ _test", "test_labels_200.json")
219     test_dataset = ParasiteDataset(
220         config.test_data_path,
221         transform=val_transform,
222         is_train=False,
223         class_to_idx=full_dataset.class_to_idx,
224         label_json_path=test_label_json
225     )
226     test_loader = DataLoader(
227         test_dataset,
228         batch_size=config.batch_size,
229         shuffle=False,
230         num_workers=config.num_workers,
231         pin_memory=config.pin_memory
232     )
233 
234     return train_loader, val_loader, test_loader, full_dataset.
        ↪ class_to_idx
```

A.6 Requirements: *requirements.txt*

Listing A.6 – Arquivo de dependências

```
1  torch>=2.0.0
2  torchvision>=0.15.0
3  timm>=0.9.0
4  numpy>=1.24.0
5  pandas>=2.0.0
6  scikit-learn>=1.3.0
7  matplotlib>=3.7.0
8  seaborn>=0.12.0
9  Pillow>=10.0.0
10 tqdm>=4.65.0
11 albumentations>=1.3.0
12 opencv-python>=4.8.0
13 tensorboard>=2.13.0
```