Capítulo 4

Aprendizado Federado na Prática: Da Teoria à Implementação na Triagem do Câncer do Colo do Útero

Leonardo Augusto Ferreira, Marcelo A. P. Xavier, Andrea G. Campos, Walmir M. Caminhas, Frederico Gadelha Guimarães

Abstract

This work presents a comprehensive approach to the development of federated learning systems applied to cervical cancer screening using cytopathological images. Initially, the theoretical foundations of federated learning are discussed, with emphasis on the Federated Averaging (FedAvg) algorithm, as well as the principles of convolutional neural networks (CNNs) for medical image analysis. Subsequently, a structured pipeline is introduced, encompassing whole slide image (WSI) preprocessing, patch extraction and selection, and distributed model training. The practical implementation integrates technologies such as Docker, Flower, and PyTorch, enabling the simulation of a multicenter environment with multiple clients and a central server. The dataset is partitioned across different clients, preserving data privacy and reflecting real-world heterogeneous distributions. Finally, the implementation details of both clients and the federated server are presented, demonstrating how collaborative learning can be achieved without sharing sensitive data. The results indicate that the combination of deep learning and federated learning provides a promising solution for building robust, scalable, and privacy-preserving systems in healthcare applications.

Resumo

Este trabalho apresenta uma abordagem completa para o desenvolvimento de sistemas de aprendizado federado aplicados à triagem do câncer do colo do útero a partir de imagens citopatológicas. Inicialmente, são discutidos os fundamentos teóricos do aprendizado federado, com destaque para o algoritmo Federated Averaging (FedAvg), bem como os princípios das redes neurais convolucionais (CNNs) aplicadas à análise de

imagens médicas. Em seguida, é descrito um pipeline estruturado que engloba o pré-processamento de lâminas digitais (WSI), a extração e seleção de patches relevantes e o treinamento distribuído dos modelos. A implementação prática integra tecnologias como Docker, Flower e PyTorch, permitindo a simulação de um ambiente multicêntrico com múltiplos clientes e um servidor central. Os dados são particionados entre diferentes clientes, preservando a privacidade e refletindo cenários reais com distribuições heterogêneas. Por fim, são apresentados os detalhes da implementação dos clientes e do servidor federado, evidenciando como o aprendizado colaborativo pode ser realizado sem o compartilhamento de dados sensíveis. Os resultados demonstram que a combinação entre aprendizado profundo e aprendizado federado constitui uma solução promissora para o desenvolvimento de sistemas robustos, escaláveis e alinhados às exigências regulatórias da área da saúde.

4.1. Introdução

O Aprendizado Federado (Federated Learning - FL) [McMahan et al. 2017] tem emergido como um paradigma promissor para o desenvolvimento de modelos de inteligência artificial em cenários que envolvem dados sensíveis e distribuídos, como na área da saúde. Diferentemente das abordagens centralizadas, o FL permite que múltiplas instituições colaborem no treinamento de modelos sem compartilhar dados brutos, preservando a privacidade e atendendo a regulamentações como a Lei Geral de Proteção de Dados (LGPD - Lei nº 13.709/2018). Além disso, essa abordagem possibilita o aprendizado a partir de dados heterogêneos provenientes de diferentes fontes, contribuindo para a construção de modelos mais robustos e com maior capacidade de generalização em cenários reais.

Nesse paradigma distribuído, cada cliente (por exemplo, hospitais ou dispositivos móveis) realiza o treinamento local do modelo em seus próprios dados e compartilha apenas atualizações de parâmetros, como pesos ou gradientes, com um servidor central. Esse servidor agrega as contribuições - tipicamente por meio do algoritmo FedAvg (Federated Averaging) - ponderando os modelos locais de acordo com o volume de dados de cada cliente, e gera um modelo global atualizado. Esse processo ocorre de forma iterativa em múltiplas rodadas de comunicação, permitindo o aprendizado de padrões globais a partir de dados descentralizados e reduzindo a necessidade de transferência de grandes volumes de dados. Apesar de suas vantagens, o FL ainda enfrenta desafios como a heterogeneidade dos dados (non-IID), limitações de comunicação e variações na capacidade computacional entre os participantes.

Nesse cenário, uma aplicação particularmente relevante do Aprendizado Federado é a automatização da análise de exames citopatológicos destinados ao rastreamento do câncer do colo do útero. Essa enfermidade configura-se como um relevante problema de saúde pública, tanto no Brasil quanto em nível global. De acordo com o Instituto Nacional de Câncer, foram estimados aproximadamente 17.010 novos casos para o triênio de 2023 a 2025, correspondendo a uma incidência de 15,38 casos por 100 mil habitantes. Além disso, em 2022, a mortalidade associada à doença foi de 4,51 óbitos por 100 mil mulheres, evidenciando a necessidade de estratégias mais eficazes para o diagnóstico precoce [INCA 2022].

As elevadas taxas de mortalidade são agravadas por limitações estruturais no sistema de saúde brasileiro, especialmente no que se refere à escassez e à distribuição desigual de médicos patologistas no território nacional. O Brasil conta com menos da metade da quantidade de patologistas recomendada pela Organização Mundial da Saúde, segundo dados da Sociedade Brasileira de Patologia [Scheffer 2025]. Ademais, observa-se uma menor presença desses especialistas nas regiões Norte e Nordeste, em contraste com uma maior concentração nas regiões Sul e Sudeste. Essa disparidade na distribuição de profissionais repercute diretamente nos indicadores de saúde, resultando em taxas de mortalidade mais elevadas nas localidades com menor acesso à assistência especializada.

Nesse contexto, técnicas de inteligência artificial, particularmente aquelas fundamentadas em deep learning, têm sido amplamente investigadas como ferramentas de apoio à análise de imagens citológicas[Diniz et al. 2021, Teixeira et al. 2023, Terra et al. 2023]. Esses métodos vêm apresentando desempenho promissor na identificação de padrões clínicos relevantes e na automatização de processos diagnósticos. Contudo, grande parte dessas abordagens é desenvolvida sob um paradigma centralizado, utilizando dados oriundos de uma única instituição ou de bases consolidadas em repositórios unificados. Tal estratégia impõe limitações significativas, sobretudo no que se refere à capacidade de generalização dos modelos, uma vez que não contempla adequadamente a variabilidade decorrente de diferentes laboratórios, equipamentos de aquisição de imagens e protocolos de preparo das lâminas.

Essa perspectiva evidencia dois desafios centrais na aplicação de modelos de aprendizado de máquina em citopatologia digital. O primeiro refere-se à mudança de domínio (domain shift), caracterizada pela discrepância entre as distribuições dos dados de treinamento e de aplicação, o que compromete a generalização dos modelos em ambientes externos. Essa variabilidade pode ser atribuída a fatores não biológicos, como diferenças de scanners, protocolos de coloração, métodos de fixação e condições de armazenamento. O segundo desafio está relacionado à centralização dos dados médicos, que levanta questões críticas de privacidade, segurança e conformidade regulatória. Nesse contexto, legislações como a LGPD impõem restrições rigorosas ao compartilhamento de dados sensíveis, limitando a viabilidade de abordagens tradicionais baseadas em agregação de dados.

Assim, o Aprendizado Federado surge como uma solução estratégica para mitigar simultaneamente os desafios de privacidade e de generalização. Ao permitir o treinamento colaborativo entre múltiplas instituições sem a necessidade de transferência de dados brutos, o FL possibilita a incorporação de padrões provenientes de diferentes distribuições de dados, reduzindo os efeitos do domain shift e promovendo maior robustez dos modelos. Dessa forma, essa abordagem se apresenta como uma alternativa viável para o desenvolvimento de sistemas de apoio à decisão clínica mais eficazes, especialmente em cenários complexos e heterogêneos como o rastreamento do câncer do colo do útero. Para ilustrar o funcionamento do aprendizado federado no contexto da citopatologia digital, a Figura 4.1 apresenta um fluxo simplificado do treinamento colaborativo entre múltiplas instituições de saúde. Nesse cenário, cada centro realiza o processamento local de suas lâminas citológicas e contribui para a construção de um modelo global sem a necessidade de compartilhamento de dados sensíveis, evidenciando os principais componentes e etapas desse paradigma distribuído.

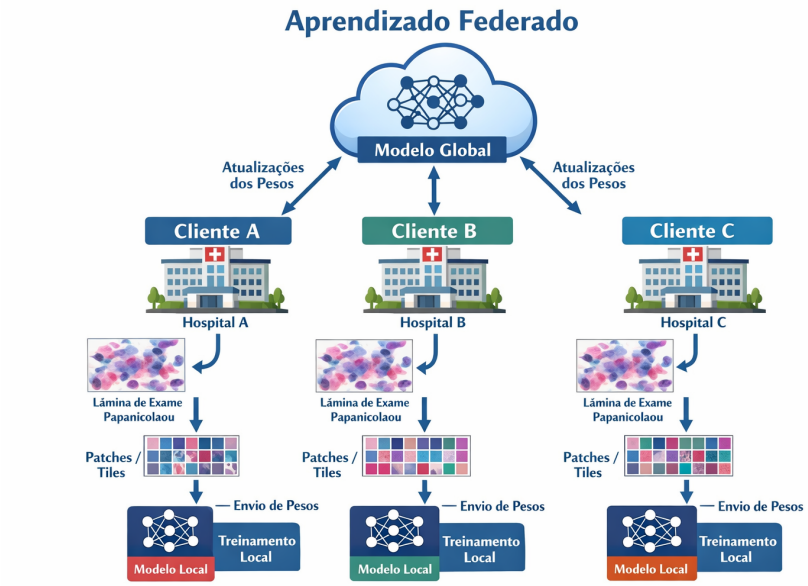

Figura 4.1. Esquema ilustrativo do Aprendizado Federado aplicado à citopatologia cervical. Cada instituição (clientes A, B e C), representada por hospitais distintos, realiza o processamento local de lâminas de exame de Papanicolaou, incluindo a extração de patches/tiles e o treinamento de modelos de aprendizado profundo. Em vez de compartilhar dados sensíveis, apenas os pesos atualizados dos modelos locais são enviados ao servidor central, onde são agregados para compor um modelo global. Esse modelo é então redistribuído aos clientes, permitindo um treinamento colaborativo, preservando a privacidade dos dados e aumentando a robustez do sistema frente à variabilidade entre instituições

4.2. Fundamentos de Redes Neurais Convolucionais (CNN) para Imagens Médicas

As Redes Neurais Convolucionais (CNNs) [LeCun 1989] constituem uma classe de modelos de aprendizado profundo especialmente projetados para lidar com dados que apresentam estrutura espacial ou sequencial, como imagens e séries temporais. No contexto deste minicurso, o foco recai sobre imagens citopatológicas digitalizadas, conhecidas como Whole Slide Images (WSI), que podem ser interpretadas como grades bidimensionais de pixels com forte organização espacial.

O principal diferencial das CNNs está na sua capacidade de explorar relações locais entre os dados por meio da operação de convolução. Diferentemente de modelos clássicos de aprendizado de máquina, que tratam os dados como vetores sem estrutura explícita, as CNNs incorporam, em sua própria arquitetura, o conhecimento de que os dados possuem organização espacial [Bishop and Bishop 2024]. Essa característica é fundamental em aplicações médicas, nas quais padrões morfológicos, texturas e relações entre estruturas celulares são determinantes para o diagnóstico.

Em problemas como a análise de exames citológicos, informações relevantes estão associadas à forma e à organização de estruturas como núcleos celulares, citoplasma e padrões de agrupamento. As CNNs são capazes de aprender automaticamente essas características, eliminando a necessidade de engenharia manual de atributos e permitindo

a construção de modelos mais robustos e generalizáveis.

Outro aspecto importante é a natureza hierárquica das representações aprendidas. Camadas iniciais da rede tendem a capturar padrões simples, como bordas e contrastes, enquanto camadas mais profundas identificam estruturas mais complexas, como formas celulares e padrões patológicos. Essa capacidade de abstração progressiva é essencial para lidar com a complexidade das imagens médicas.

No caso específico das WSI, que possuem resolução extremamente alta (ordem de gigapixels), o processamento direto é inviável. Por isso, uma estratégia amplamente utilizada, e adotada neste trabalho, consiste em dividir a lâmina em pequenas regiões denominadas *patches*. Cada patch é processado individualmente pela CNN, permitindo a extração de características locais que posteriormente podem ser agregadas para decisões em nível global.

Além disso, CNNs têm se mostrado altamente eficazes em tarefas fundamentais para o rastreamento do câncer do colo do útero [Liang et al. 2021, Liang et al. 2023], como classificação de células, detecção de regiões suspeitas e segmentação de estruturas celulares, frequentemente superando abordagens baseadas em características manuais.

Dessa forma, no escopo deste trabalho, as CNNs desempenham o papel central na extração automática de características relevantes a partir das imagens citológicas. Quando integradas ao aprendizado federado, permitem o treinamento colaborativo de modelos robustos, capazes de generalizar para diferentes instituições, sem a necessidade de compartilhamento de dados sensíveis.

4.2.1. Convolução: Definição e Interpretação

A convolução é uma operação matemática entre duas funções que permite combinar informações ao longo do tempo ou do espaço [Goodfellow et al. 2016]. Um exemplo clássico é a suavização de sinais ruidosos por meio de uma média ponderada, em que valores são combinados utilizando uma função de pesos. Essa operação pode ser expressa como:

$$s(t) = \int x(a)\, w(t-a)\, da = (x * w)(t) \tag{1}$$

Em aplicações computacionais, utiliza-se a forma discreta:

$$s(t) = \sum_{a} x(a)\, w(t-a) \tag{2}$$

Para imagens bidimensionais, a convolução é definida por:

$$S(i,j) = \sum_{m}\sum_{n} I(m,n)\, K(i-m, j-n) \tag{3}$$

Na prática, utiliza-se frequentemente a correlação cruzada:

$$S(i,j) = \sum_m \sum_n I(i+m, j+n)\, K(m,n) \tag{4}$$

Do ponto de vista intuitivo, o filtro (ou *kernel*) pode ser interpretado como um detector de padrões. Cada filtro aprende a identificar características específicas da imagem, como bordas, texturas ou estruturas celulares. Durante o treinamento, esses filtros são ajustados automaticamente para capturar padrões relevantes para a tarefa.

A operação de convolução é controlada por parâmetros importantes, como o *stride*, que define o deslocamento do filtro sobre a imagem, e o *padding*, que consiste na adição de bordas artificiais à entrada, permitindo preservar dimensões espaciais e evitar perda de informação nas extremidades.

4.2.2. Função de Ativação: ReLU

Após a aplicação da convolução, é necessário introduzir não linearidade no modelo. Isso é feito por meio de funções de ativação, sendo a mais utilizada a ReLU (Rectified Linear Unit), definida como:

$$\mathrm{ReLU}(x) = \max(0, x) \tag{5}$$

Essa função mantém apenas os valores positivos das ativações e zera os valores negativos. A introdução dessa não linearidade permite que a rede modele relações complexas nos dados, tornando possível a composição de múltiplas camadas e a aprendizagem de representações mais sofisticadas.

Além disso, a ReLU contribui para um treinamento mais eficiente, reduzindo problemas como o desaparecimento do gradiente e acelerando a convergência do modelo.

4.2.3. Pooling

Em redes neurais convolucionais, cada bloco típico é composto por convolução, função de ativação e, por fim, uma operação de pooling.

As camadas convolucionais preservam a correspondência espacial dos padrões detectados, propriedade conhecida como *equivariância*. No entanto, em tarefas de classificação, deseja-se que pequenas variações espaciais não alterem significativamente a decisão do modelo.

Nesse contexto, o pooling atua como um mecanismo de redução que diminui a sensibilidade a variações espaciais. Ele substitui regiões locais por valores representativos, como o máximo (*max pooling*) ou a média (*average pooling*), promovendo invariância a pequenas translações.

Além disso, o pooling reduz a dimensionalidade dos dados, tornando o modelo mais eficiente e contribuindo para a redução do overfitting.

4.2.4. Representações Latentes e Embeddings

Ao longo das camadas da CNN, as transformações sucessivas produzem representações cada vez mais abstratas dos dados. O resultado final desse processo pode ser

interpretado como um vetor de características em um espaço latente, conhecido como *embedding vetorial*.

Formalmente, dado um patch de imagem P_k, a CNN pode ser vista como uma função ϕ que mapeia esse patch em um vetor:

$$\mathbf{z}_k = \phi(P_k) \tag{6}$$

Esse vetor $\mathbf{z}_k$ codifica informações relevantes da imagem, como padrões celulares e características morfológicas. No caso de imagens WSI, múltiplos patches são processados, gerando um conjunto de embeddings que podem ser utilizados em tarefas de classificação ou agregação global.

4.2.5. Classificação e Fluxo Completo da CNN

Após a extração dos embeddings, esses vetores são utilizados por camadas densamente conectadas (*fully connected*) responsáveis pela classificação final. Essas camadas combinam as características extraídas ao longo da rede para produzir probabilidades associadas a cada classe.

De forma geral, o fluxo de uma CNN pode ser descrito como uma sequência de blocos compostos por convolução, função de ativação e pooling, seguidos por camadas densas responsáveis pela decisão final. Esse encadeamento permite transformar progressivamente os dados de entrada em representações cada vez mais abstratas e semanticamente relevantes.

No contexto do aprendizado federado, essas representações são aprendidas de forma distribuída entre diferentes clientes, permitindo que o modelo capture variações presentes em múltiplas instituições sem a necessidade de compartilhamento direto dos dados.

A Figura 4.2 ilustra esse processo no contexto da análise citológica: as imagens WSI são divididas em patches, processadas por CNNs para extração de características e treinamento local em cada instituição. Apenas os parâmetros dos modelos são compartilhados com o servidor central, que realiza a agregação federada por meio do algoritmo FedAvg.

4.2.6. Transfer Learning e Fine-Tuning

Em aplicações práticas de aprendizado profundo, especialmente no domínio de imagens médicas, é comum utilizar modelos previamente treinados em grandes bases de dados, como o ImageNet [Liang et al. 2021]. Essa abordagem, conhecida como *transfer learning*, consiste em reutilizar representações aprendidas em tarefas genéricas de visão computacional, reduzindo a necessidade de grandes volumes de dados anotados e acelerando o processo de treinamento.

A motivação para o uso de *transfer learning* está diretamente relacionada à natureza dos dados médicos. Em muitos cenários, como na citopatologia, a obtenção de dados rotulados é um processo custoso e dependente de especialistas. Nesse contexto, o *transfer learning* permite utilizar redes neurais já treinadas como extratores de características,

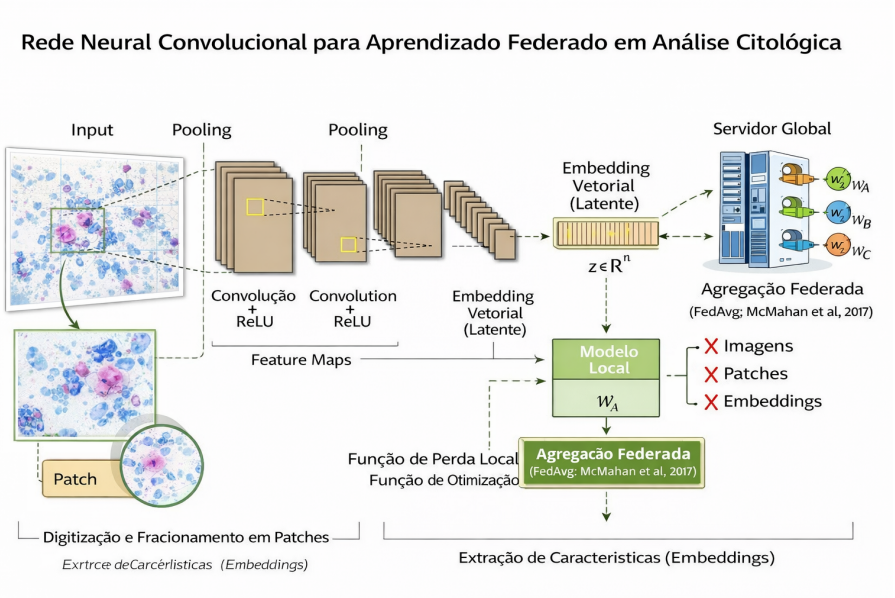

Figura 4.2. Arquitetura de uma rede neural convolucional aplicada ao aprendizado federado para análise citológica. A lâmina digital (WSI) é fragmentada em patches, que são processados por camadas convolucionais com ReLU e pooling para extração de características e geração de embeddings latentes. O treinamento ocorre localmente em cada instituição, enquanto apenas os pesos dos modelos são compartilhados com um servidor central para agregação federada (FedAvg).

aproveitando padrões previamente aprendidos, como bordas, texturas e estruturas visuais básicas, sem a necessidade de aprender essas representações do zero.

No presente trabalho, foi utilizada uma arquitetura ResNet-18 previamente treinada, sobre a qual foi aplicado o processo de *fine-tuning*. Enquanto o *transfer learning* refere-se ao reaproveitamento direto das representações aprendidas, o *fine-tuning* consiste em adaptar essas representações ao novo domínio de dados por meio do ajuste dos parâmetros do modelo.

Inicialmente, as camadas convolucionais da rede, responsáveis pela extração de características de baixo e médio nível, são mantidas com seus pesos pré-treinados, preservando o conhecimento adquirido em bases de dados genéricas. Em seguida, a camada final de classificação é substituída para se adequar ao número de classes do problema de citopatologia.

O processo de *fine-tuning* envolve o ajuste parcial ou total dos parâmetros da rede, permitindo que o modelo refine suas representações para capturar características específicas do domínio médico. Dependendo da estratégia adotada, pode-se optar por treinar apenas as camadas finais (mantendo as demais congeladas) ou liberar gradualmente mais camadas para atualização, de acordo com a quantidade de dados disponível e a complexidade da tarefa.

Essa distinção é fundamental: enquanto o *transfer learning* atua principalmente como um mecanismo de reutilização de conhecimento, o *fine-tuning* promove a especialização do modelo para o problema alvo. Em conjunto, essas abordagens permitem

equilibrar generalização e adaptação.

Essa estratégia é particularmente vantajosa em cenários com dados limitados, pois reduz o risco de overfitting e melhora a eficiência do treinamento. Além disso, possibilita que o modelo capture padrões específicos da citopatologia, refinando representações originalmente aprendidas em contextos mais amplos.

No contexto do aprendizado federado, o *fine-tuning* ocorre de forma distribuída entre os diferentes clientes. Cada instituição ajusta localmente os parâmetros do modelo com base em seus próprios dados, refletindo variações específicas de aquisição, preparo das lâminas e características populacionais. O servidor central, por sua vez, agrega essas atualizações por meio do algoritmo FedAvg, resultando em um modelo global que incorpora conhecimento proveniente de múltiplas fontes.

Dessa forma, a combinação entre *transfer learning*, *fine-tuning* e aprendizado federado permite construir modelos mais robustos e generalizáveis, ao mesmo tempo em que respeita restrições de privacidade e reduz a necessidade de compartilhamento direto de dados sensíveis.

4.3. Fundamentos Conceituais do Aprendizado Federado

O Aprendizado Federado, conforme proposto em [McMahan et al. 2017], teve sua origem no contexto de dispositivos móveis. Esses dispositivos são amplamente utilizados no cotidiano e estão equipados com diversos sensores embarcados, como câmeras e microfones, responsáveis pela geração contínua de grandes volumes de dados, incluindo imagens, áudios e vídeos. Entretanto, a heterogeneidade entre os dispositivos, decorrente de diferenças de hardware, condições de uso e perfis de usuários, resulta em distribuições de dados distintas entre os clientes.

Consequentemente, os dados utilizados nesse cenário apresentam natureza não independente e não identicamente distribuída (non-IID), além de frequentemente desbalanceada. Adicionalmente, por se tratarem de informações sensíveis geradas localmente, esses dados permanecem privados, o que inviabiliza sua centralização e motiva o uso de abordagens descentralizadas.

A aplicação de modelos de aprendizado de máquina nesses grandes volumes de dados possibilitaria a extração de padrões complexos e a construção de modelos preditivos robustos. Entretanto, em cenários nos quais os dados estão distribuídos e são sensíveis, como no contexto de dispositivos móveis, abordagens centralizadas tornam-se inviáveis. Além disso, a natureza privada dessas informações impõe restrições éticas e legais ao seu compartilhamento, especialmente em conformidade com regulamentações de proteção de dados.

Nesse contexto, surge o Aprendizado Federado (Federated Learning - **FL**) como solução para o impasse entre a necessidade de utilizar grandes volumes de dados para treinamento e as restrições relacionadas à privacidade e à distribuição heterogênea desses dados. O FL é uma técnica que permite que diferentes entidades, denominadas clientes (dispositivos móveis, hospitais ou outras instituições), realizem o treinamento local de modelos de aprendizado de máquina utilizando seus próprios dados, sem a necessidade de compartilhamento direto dessas informações.

Nesse paradigma, cada cliente treina um modelo local a partir de seus dados privados e envia apenas os parâmetros atualizados (como pesos ou gradientes) para um servidor central, denominado servidor global. Esse servidor é responsável por agregar as atualizações recebidas de múltiplos clientes, geralmente por meio de estratégias como a média ponderada, gerando um modelo global mais robusto. Em seguida, o modelo global atualizado é redistribuído aos clientes, que o utilizam como ponto de partida para novas rodadas de treinamento local. Esse processo iterativo permite o aprendizado colaborativo entre diferentes fontes de dados, preservando a privacidade e lidando com a natureza distribuída e não-IID dos dados.

Uma das características mais relevantes dessa abordagem federada reside na separação entre o acesso aos dados brutos e o processo de agregação global do modelo. Em outras palavras, o aprendizado ocorre de forma distribuída nos clientes, onde os modelos são treinados localmente com acesso direto aos dados, enquanto o modelo global é atualizado sem que haja a necessidade de centralizar informações sensíveis em um único repositório.

Apesar dessa separação reduzir significativamente a necessidade de compartilhamento de dados, o servidor responsável pela coordenação do treinamento ainda desempenha um papel central e, portanto, demanda um certo nível de confiança. Ainda assim, essa arquitetura apresenta vantagens relevantes do ponto de vista de segurança e privacidade. Em cenários nos quais o objetivo de aprendizado pode ser definido localmente em cada cliente, o Aprendizado Federado permite que o treinamento ocorra sem a exposição direta dos dados.

Como consequência, há uma redução substancial dos riscos associados a vazamentos ou ataques, uma vez que a superfície de exposição é limitada aos dispositivos locais. Diferentemente das abordagens tradicionais, nas quais os dados precisam ser transferidos para a nuvem, o modelo federado restringe a vulnerabilidade ao ambiente do cliente, tornando o sistema global mais seguro e alinhado com requisitos de proteção de dados.

4.3.1. Formulação Matemática

A formulação mais clássica e amplamente utilizada no Aprendizado Federado (Federated Learning) é a do algoritmo *Federated Averaging* [McMahan et al. 2017], conhecido como FedAvg.

A ideia central do método consiste em realizar o treinamento de modelos de forma distribuída: cada cliente treina localmente utilizando seus próprios dados, enquanto um servidor central coordena o processo e agrega os modelos por meio de uma média ponderada.

Sejam:

- K : número total de clientes (hospitais, dispositivos ou instituições);
- n_k : número de amostras disponíveis no cliente k;
- $n = \sum_{k=1}^{K} n_k$: número total de amostras;

- $w^{(t)}$: modelo global na rodada federada t;
- $w_k^{(t+1)}$: modelo atualizado no cliente k após o treinamento local na rodada t;
- $\mathscr{D}_k$: conjunto de dados local do cliente k.

Uma **rodada federada** consiste no envio do modelo global aos clientes, no treinamento local e na posterior agregação dos modelos no servidor.

A cada rodada, o servidor envia o modelo global $w^{(t)}$ aos clientes selecionados. Cada cliente realiza treinamento local e obtém um modelo atualizado $w_k^{(t+1)}$. Em seguida, o servidor agrega esses modelos por meio de uma média ponderada:

$$w^{(t+1)} = \sum_{k=1}^{K} \frac{n_k}{n} w_k^{(t+1)}. \tag{7}$$

A ponderação por n_k garante que clientes com maior volume de dados tenham maior influência na atualização do modelo global.

A equação de agregação implica que:

- clientes com mais dados contribuem mais significativamente para o modelo global;
- clientes com poucos dados possuem menor influência;
- a atualização global corresponde a uma média ponderada dos modelos locais.

Essa estratégia permite combinar informações provenientes de diferentes fontes de dados, preservando suas características locais.

O problema global pode ser formulado como a minimização de uma função objetivo distribuída:

$$\min_w F(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w), \tag{8}$$

onde $F_k(w)$ representa a função de perda local no cliente k, definida como:

$$F_k(w) = \frac{1}{n_k} \sum_{i \in \mathscr{D}_k} \ell(w; x_i, y_i). \tag{9}$$

Assim, o objetivo global corresponde à minimização da média ponderada das perdas locais, refletindo a contribuição de cada cliente proporcionalmente ao seu volume de dados.

Cada cliente realiza treinamento local a partir do modelo global recebido, utilizando métodos de otimização como o Gradiente Descendente Estocástico (SGD):

$$w_k \leftarrow w_k - \eta \nabla F_k(w_k), \tag{10}$$

onde:

- η é a taxa de aprendizado;
- o treinamento ocorre por E épocas locais;
- apenas os parâmetros atualizados são enviados ao servidor.

A execução de múltiplas épocas locais reduz a necessidade de comunicação frequente com o servidor, tornando o método eficiente em cenários distribuídos.

O fluxo do algoritmo FedAvg pode ser descrito da seguinte forma:

1. O servidor inicializa e envia o modelo global $w^{(t)}$ aos clientes;
2. Cada cliente realiza treinamento local utilizando seus dados $\mathscr{D}_k$;
3. Os clientes retornam os modelos atualizados $w_k^{(t+1)}$ ao servidor;
4. O servidor agrega os modelos por média ponderada:

$$w^{(t+1)} = \sum_{k=1}^{K} \frac{n_k}{n} w_k^{(t+1)}. \tag{11}$$

Esse processo é repetido iterativamente até a convergência do modelo global.

4.3.2. Considerações Práticas

Em cenários reais, especialmente quando os dados são não independentes e não identicamente distribuídos (non-IID), podem surgir desafios adicionais:

- divergência entre modelos locais (*client drift*);
- impacto da heterogeneidade dos dados na convergência;
- trade-off entre comunicação e desempenho.

Apesar dessas limitações, o Federated Averaging constitui a base matemática do Aprendizado Federado, sendo uma abordagem simples, eficiente e amplamente aplicável em cenários com restrições de privacidade, como aplicações médicas e institucionais.

Algorithm 1 Federated Averaging (FedAvg)

Inicializar modelo global $w^{(0)}$
for cada rodada federada $t = 0, 1, 2, \ldots, T$ **do**
    Servidor seleciona subconjunto de clientes $\mathscr{S}_t$
    Servidor envia $w^{(t)}$ para todos $k \in \mathscr{S}_t$
    for all clientes $k \in \mathscr{S}_t$ **em paralelo do**
        Inicializar $w_k \leftarrow w^{(t)}$
        for $e = 1$ até E **do**
            Atualizar $w_k \leftarrow w_k - \eta \nabla F_k(w_k)$
        end for
        Enviar $w_k^{(t+1)}$ ao servidor
    end for
    Atualizar modelo global:

$$w^{(t+1)} = \sum_{k \in \mathscr{S}_t} \frac{n_k}{\sum_{j \in \mathscr{S}_t} n_j} w_k^{(t+1)}$$

end for

O algoritmo *Federated Averaging* (FedAvg) constitui o principal método de treinamento no contexto do Aprendizado Federado, sendo responsável por coordenar o aprendizado colaborativo entre múltiplos clientes de forma descentralizada. Sua execução ocorre por meio de um processo iterativo organizado em rodadas federadas, nas quais o modelo global é progressivamente refinado a partir das contribuições locais de cada participante.

Inicialmente, o servidor central define um modelo global $w^{(0)}$, que pode ser inicializado aleatoriamente ou a partir de um treinamento prévio. Em cada rodada federada t, o servidor seleciona um subconjunto de clientes $\mathscr{S}_t$ e envia a esses participantes a versão atual do modelo global $w^{(t)}$. Essa seleção pode ser motivada por critérios como disponibilidade, capacidade computacional ou restrições de comunicação.

Após receber o modelo global, cada cliente $k \in \mathscr{S}_t$ realiza o treinamento local utilizando exclusivamente seus dados privados $\mathscr{D}_k$. O processo de treinamento é tipicamente conduzido por meio de métodos de otimização baseados em gradiente, como o Gradiente Descendente Estocástico (SGD), sendo executado por um número fixo de épocas locais E. Ao final dessa etapa, cada cliente obtém um modelo atualizado $w_k^{(t+1)}$, que reflete as características específicas de sua distribuição local de dados.

Em seguida, os clientes enviam ao servidor apenas os parâmetros atualizados de seus modelos, sem compartilhar os dados utilizados no treinamento. O servidor então realiza a etapa de agregação, combinando os modelos locais por meio de uma média ponderada, na qual o peso de cada cliente é proporcional ao número de amostras disponíveis em sua base de dados. Essa estratégia permite que clientes com maior volume de dados exerçam maior influência na construção do modelo global.

A atualização global pode ser formalmente descrita pela equação:

$$w^{(t+1)} = \sum_{k \in \mathscr{S}_t} \frac{n_k}{\sum_{j \in \mathscr{S}_t} n_j} w_k^{(t+1)}. \tag{12}$$

O modelo global atualizado $w^{(t+1)}$ é então redistribuído aos clientes, iniciando uma nova rodada de treinamento. Esse processo iterativo continua até que um critério de parada seja atingido, como a convergência do modelo ou um número máximo de rodadas.

Do ponto de vista conceitual, o FedAvg pode ser interpretado como uma aproximação distribuída da minimização de uma função objetivo global, definida como a média ponderada das funções de perda locais. Essa formulação permite integrar informações provenientes de diferentes fontes de dados, preservando suas particularidades, ao mesmo tempo em que constrói um modelo global mais robusto e generalizável.

No contexto de aplicações reais, como dispositivos móveis ou sistemas de saúde, o algoritmo apresenta vantagens significativas, especialmente no que se refere à preservação da privacidade dos dados e à redução da necessidade de transferência de grandes volumes de informação. No entanto, sua eficiência pode ser impactada por fatores como a heterogeneidade dos dados entre os clientes (cenário non-IID), o custo de comunicação e a variabilidade na capacidade computacional dos participantes.

Em aplicações na área da saúde, como a classificação de células em exames citopatológicos, o FedAvg permite que diferentes instituições colaborem no treinamento de modelos de aprendizado profundo sem a necessidade de compartilhar imagens médicas sensíveis. Cada instituição contribui com o aprendizado a partir de seus próprios dados, enquanto o modelo global se beneficia da diversidade das fontes, resultando em maior robustez e capacidade de generalização em cenários multicêntricos.

4.3.3. Limitações do Aprendizado Federado

Apesar das vantagens associadas ao Aprendizado Federado, sua aplicação prática apresenta desafios relevantes que devem ser considerados.

4.3.3.1. Heterogeneidade dos Dados (non-IID)

Em cenários reais, os dados disponíveis nos diferentes clientes frequentemente não seguem a mesma distribuição, caracterizando um ambiente não independente e não identicamente distribuído (non-IID). Essa heterogeneidade pode levar à divergência entre os modelos locais, fenômeno conhecido como *client drift*, dificultando a convergência do modelo global e impactando negativamente o desempenho.

4.3.3.2. Custo de Comunicação

O Aprendizado Federado depende da troca iterativa de parâmetros entre clientes e servidor. Em ambientes com recursos limitados, como dispositivos móveis ou redes com baixa largura de banda, o custo de comunicação pode se tornar um gargalo significativo. Estratégias como redução de frequência de comunicação, compressão de modelos e quantização são frequentemente utilizadas para mitigar esse problema.

4.3.3.3. Convergência do Modelo

A convergência do FedAvg pode ser mais lenta ou instável em comparação com métodos centralizados, especialmente em cenários com alta heterogeneidade de dados e diferentes capacidades computacionais entre os clientes. A escolha adequada de hiperparâmetros, como taxa de aprendizado e número de épocas locais, é fundamental para garantir a estabilidade do treinamento.

4.3.3.4. Dependência do Servidor Central

Embora os dados não sejam compartilhados, o servidor central ainda desempenha um papel crítico na agregação dos modelos. Isso implica a necessidade de um nível de confiança nessa entidade e pode representar um ponto único de falha ou ataque.

4.3.3.5. Privacidade e Segurança

Apesar de reduzir o risco de exposição de dados brutos, o Aprendizado Federado não é totalmente imune a ataques. Técnicas de inferência podem, em alguns casos, extrair informações sensíveis a partir dos parâmetros compartilhados. Por essa razão, abordagens complementares, como privacidade diferencial e criptografia segura, são frequentemente incorporadas ao processo de treinamento.

4.4. Visão Geral do Processo: Do Tratamento dos Dados até o Compartilhamento do Modelo Global

O desenvolvimento de sistemas inteligentes para análise de lâminas citopatológicas digitais (*Whole Slide Images* – WSI), no contexto do exame de Papanicolaou, envolve um pipeline estruturado que integra técnicas de processamento de imagens, aprendizado profundo e aprendizado federado. Esse fluxo é motivado pelas características intrínsecas das WSIs, que apresentam resolução em escala gigapixel, tornando inviável seu processamento direto por modelos de aprendizado de máquina. Dessa forma, torna-se necessário decompor o problema em etapas bem definidas, capazes de transformar dados brutos em representações computacionais eficientes e informativas.

A etapa inicial do pipeline consiste no pré-processamento das lâminas digitais, conforme ilustrado na Figura 4.3. Nessa fase, cada WSI é subdividida em pequenas regiões denominadas *patches*, com dimensões fixas, como 512×512 pixels. Essa abordagem reduz significativamente a complexidade computacional do problema, ao mesmo tempo em que preserva padrões locais relevantes para a análise citopatológica. Adicionalmente, regiões sem conteúdo celular, como áreas em branco ou com baixa densidade de informação, são removidas, resultando em um conjunto de dados mais compacto e adequado ao treinamento de modelos de aprendizado profundo.

Após a extração inicial dos patches, realiza-se uma etapa de seleção baseada em critérios estatísticos, conforme apresentado na Figura 4.4. Nem todos os patches contribuem igualmente para o aprendizado do modelo, sendo comum a presença de regiões

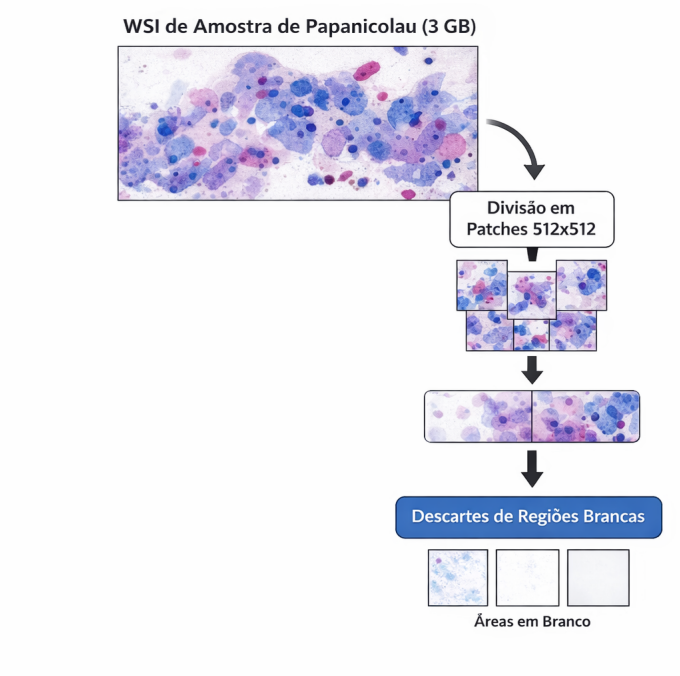

Figura 4.3. Pipeline de pré-processamento de lâminas citológicas digitais (WSI). A imagem completa é subdividida em patches e regiões sem conteúdo celular são descartadas, reduzindo o volume de dados e aumentando a eficiência do processamento.

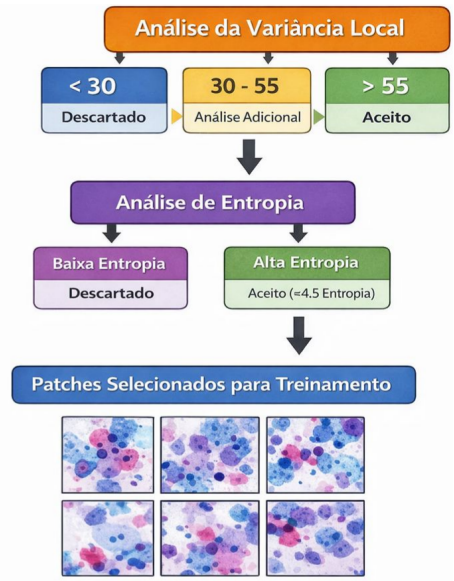

Figura 4.4. Seleção de patches baseada em variância local e entropia. Regiões com baixo conteúdo informacional são descartadas, enquanto patches mais informativos são selecionados para o treinamento.

com baixa variabilidade visual ou conteúdo irrelevante. Para mitigar esse problema, são utilizadas métricas como variância local e entropia, que permitem quantificar o nível de informação presente em cada região. Patches com baixa complexidade são descartados, enquanto regiões com maior riqueza estrutural, tipicamente associadas à presença de cé-

lulas e possíveis alterações morfológicas, são mantidas. Esse processo contribui para aumentar a densidade informacional do conjunto de dados e reduzir o impacto de ruídos no treinamento.

Uma vez definidos os patches relevantes, inicia-se o processamento local dos dados e o treinamento dos modelos de aprendizado profundo em cada instituição participante, conforme ilustrado na Figura 4.5. Esse processamento inclui a organização dos dados, normalização das imagens e treinamento de redes neurais convolucionais. Cada instituição atua como um cliente independente, operando sobre seus próprios dados, sem a necessidade de compartilhamento direto de informações sensíveis.

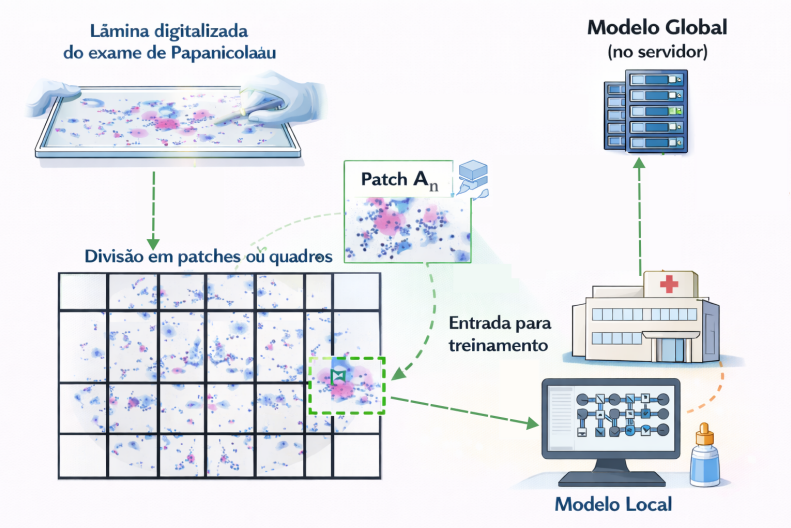

Figura 4.5. Pipeline de processamento local e treinamento em ambiente federado. Cada instituição realiza o treinamento de modelos com seus próprios dados e compartilha apenas os parâmetros aprendidos.

Após o treinamento local, os dados originais permanecem nas instituições de origem, e apenas os parâmetros do modelo são compartilhados. Conforme ilustrado na Figura 4.6, esses parâmetros representam padrões visuais aprendidos ao longo do treinamento, codificados numericamente na forma de pesos da rede neural. Esses pesos não contêm informações explícitas sobre os dados originais, o que contribui para a preservação da privacidade e atende aos requisitos regulatórios associados ao uso de dados médicos.

Na etapa seguinte, os parâmetros dos modelos locais são enviados a um servidor central, onde ocorre o processo de agregação, conforme apresentado na Figura 4.7. Nesse processo, os modelos treinados de forma independente são combinados por meio de algoritmos como o *Federated Averaging*, que realiza uma média ponderada dos parâmetros recebidos. O resultado é um modelo global que incorpora o conhecimento distribuído entre múltiplas instituições.

Após a agregação, o modelo global atualizado é redistribuído às instituições participantes, iniciando uma nova rodada de treinamento local. Esse processo iterativo continua até que o modelo atinja estabilidade ou desempenho satisfatório. Esse mecanismo permite explorar dados distribuídos e heterogêneos, aumentando a capacidade de genera-

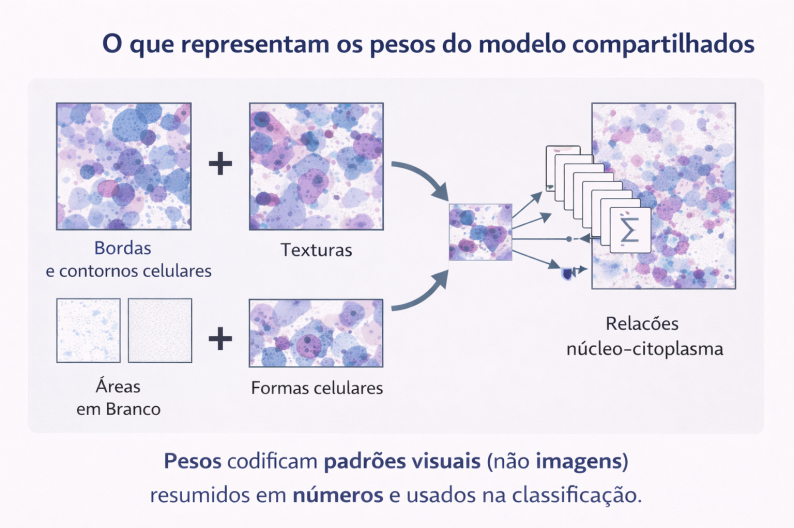

Figura 4.6. Representação conceitual dos parâmetros aprendidos por modelos de aprendizado profundo. Os pesos codificam padrões visuais relevantes sem armazenar diretamente os dados originais.

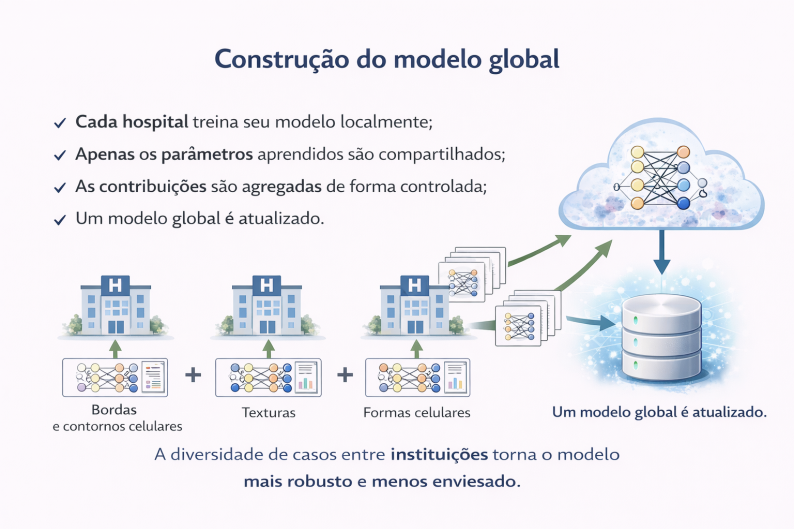

Figura 4.7. Processo de agregação no aprendizado federado. Os parâmetros dos modelos locais são combinados para formar um modelo global sem acesso aos dados originais.

lização do modelo e reduzindo o impacto de variações entre diferentes contextos clínicos.

Ao final desse processo, o modelo global pode ser utilizado como ferramenta de apoio à decisão clínica, auxiliando especialistas na triagem citopatológica. A combinação entre aprendizado profundo e aprendizado federado permite desenvolver soluções robustas, escaláveis e compatíveis com requisitos de privacidade, sendo especialmente adequada para aplicações em saúde que envolvem múltiplas instituições e dados sensíveis.

Em síntese, o pipeline apresentado integra de forma coerente as etapas de pré-processamento, seleção de dados, aprendizado local e agregação federada, constituindo

uma abordagem eficiente para análise de WSIs em cenários reais de diagnóstico assistido por inteligência artificial.

4.5. Implementação Prática em Python: Simulação Federada com CNN

Esta seção apresenta a implementação prática de uma simulação de aprendizado federado aplicada à classificação de imagens citopatológicas. O objetivo é mostrar, de forma progressiva e didática, como diferentes tecnologias podem ser integradas para construir um ambiente distribuído, reprodutível e compatível com o treinamento de Redes Neurais Convolucionais (CNNs).

A implementação proposta combina três elementos principais. O primeiro é o Docker, utilizado para padronizar o ambiente de execução e permitir que servidor e clientes sejam executados em contêineres isolados. O segundo é a biblioteca Flower, responsável por coordenar o processo de aprendizado federado, incluindo a comunicação entre clientes e servidor e a agregação dos modelos locais. O terceiro é o PyTorch, utilizado para a construção, treinamento e avaliação da rede neural convolucional empregada no experimento.

Ao longo desta seção, o leitor será conduzido desde os conceitos fundamentais das ferramentas utilizadas até a organização prática do projeto. Inicialmente, discutimos o papel do Docker, do *Dockerfile* e do arquivo `docker-compose.yaml` na criação da arquitetura distribuída. Em seguida, apresentamos o conjunto de dados utilizado, sua divisão entre diferentes clientes e a forma como os volumes Docker permitem simular instituições distintas. Por fim, descrevemos a implementação dos clientes federados e do servidor, destacando as principais funções responsáveis pelo carregamento dos dados, treinamento local, agregação dos parâmetros, avaliação global e registro do histórico de treinamento.

Essa organização permite compreender o fluxo completo da simulação federada: cada cliente acessa apenas seus próprios dados, treina localmente uma CNN e envia ao servidor somente os parâmetros atualizados do modelo. O servidor, por sua vez, agrega essas atualizações e constrói um modelo global, sem acessar diretamente as imagens dos clientes. Dessa forma, a implementação reflete os princípios centrais do aprendizado federado: colaboração entre instituições, preservação da privacidade, reprodutibilidade experimental e escalabilidade da solução.

4.5.1. Docker: Conceitos Fundamentais e Aplicação no Projeto

O Docker é uma plataforma de virtualização leve baseada em contêineres que permite empacotar aplicações e dependências em ambientes isolados e reprodutíveis. Diferentemente de máquinas virtuais completas, os contêineres compartilham o kernel do sistema operacional, tornando sua execução mais eficiente.

Neste minicurso, o Docker é utilizado para padronizar o ambiente de execução de experimentos com CNNs e Aprendizado Federado. Como esses sistemas dependem de múltiplas bibliotecas e configurações específicas, o Docker permite encapsular todo o ambiente em uma única imagem, garantindo consistência entre diferentes máquinas.

Uma imagem Docker é definida por um *Dockerfile*, que descreve a instalação de

dependências e a configuração do ambiente. A partir dessa imagem, são instanciados os contêineres executáveis.

No cenário federado, o Docker garante que todos os clientes utilizem o mesmo ambiente, evitando inconsistências. Além disso, permite simular múltiplos clientes em uma única máquina, facilitando testes controlados.

Outra vantagem é a facilidade de implantação: contêineres podem ser rapidamente distribuídos e executados em diferentes instituições. Isso também contribui para a reprodutibilidade científica, permitindo que experimentos sejam replicados sob as mesmas condições.

4.5.2. Biblioteca Flower para Aprendizado Federado

A biblioteca *Flower* é um framework em Python para desenvolvimento e execução de sistemas de Aprendizado Federado, permitindo o treinamento distribuído sem compartilhamento de dados brutos.

Neste minicurso, a Flower é utilizada para implementar o algoritmo *Federated Averaging (FedAvg)*. A biblioteca abstrai a comunicação entre servidor e clientes, permitindo foco na modelagem e nos experimentos.

Sua arquitetura segue o modelo cliente-servidor:

- **Servidor**: coordena o treinamento, distribui o modelo global e agrega as atualizações;
- **Cliente**: executa o treinamento local e retorna os parâmetros atualizados.

A Flower é compatível com frameworks como PyTorch e TensorFlow, permitindo adaptação de modelos existentes. Também suporta simulações locais e execuções distribuídas reais.

Outro diferencial é a flexibilidade na definição de estratégias de agregação, possibilitando lidar com desafios como dados não-IID e heterogeneidade entre clientes.

No contexto de imagens citopatológicas, a Flower permite simular múltiplas instituições colaborando sem compartilhar dados sensíveis, atendendo requisitos de privacidade como a LGPD.

Além disso, sua integração com Docker facilita a criação de ambientes reprodutíveis e distribuíveis.

4.5.3. PyTorch: Framework para Aprendizado Profundo

O *PyTorch* é um dos principais frameworks de aprendizado profundo, amplamente utilizado pela sua flexibilidade e integração com Python.

Uma de suas principais características é o uso de grafos computacionais dinâmicos (*define-by-run*), que facilita depuração e experimentação.

Os dados são representados por tensores com suporte a GPU, essenciais para o treinamento eficiente de modelos de visão computacional.

Modelos são definidos por meio da classe `nn.Module`, permitindo a construção modular de redes neurais. Camadas convolucionais, funções de ativação e operações de *pooling* são facilmente integradas.

O treinamento envolve propagação direta, cálculo da perda e retropropagação, com otimização via algoritmos como SGD ou Adam, utilizando o mecanismo automático de gradientes (*autograd*).

No aprendizado federado, o PyTorch é utilizado para implementar os modelos locais treinados em cada cliente. Cada cliente treina seu modelo com dados próprios e compartilha apenas os parâmetros com o servidor.

O framework também oferece ferramentas como `DataLoader` e a biblioteca `torchvision`, que inclui modelos pré-treinados e utilidades para processamento de imagens.

Dessa forma, o PyTorch fornece a base para implementação das arquiteturas de redes neurais utilizadas neste projeto.

4.5.4. Ambiente de Desenvolvimento

O desenvolvimento de sistemas modernos de aprendizado de máquina, especialmente em cenários complexos como o aprendizado federado aplicado a imagens citopatológicas, exige ambientes computacionais consistentes, reprodutíveis e escaláveis. Nesse contexto, o uso de contêineres surge como uma solução fundamental para garantir que aplicações sejam executadas de forma padronizada em diferentes máquinas e infraestruturas.

O Docker é uma plataforma que permite empacotar aplicações e todas as suas dependências em unidades chamadas *containers*. Esses containers são ambientes isolados que incluem bibliotecas, frameworks, configurações e código necessários para a execução do sistema. Dessa forma, elimina-se o problema clássico de incompatibilidade entre ambientes (por exemplo, diferenças de versões de bibliotecas ou sistemas operacionais), garantindo que o software funcione da mesma forma em qualquer máquina.

No contexto deste projeto, que envolve processamento de imagens de lâminas digitais (WSI), redes neurais convolucionais e aprendizado federado, o Docker desempenha um papel essencial. Isso ocorre porque o pipeline envolve múltiplos componentes (pré-processamento, treinamento local, agregação global, etc.), frequentemente executados em diferentes instituições. Assim, o uso de containers garante reprodutibilidade experimental, facilidade de implantação e escalabilidade da infraestrutura.

Além disso, ao considerar que imagens WSI possuem dimensões extremamente elevadas (podendo atingir resoluções de gigapixels), torna-se inviável processá-las diretamente sem uma infraestrutura adequada. Nesse cenário, o Docker permite organizar pipelines eficientes e distribuídos, facilitando o gerenciamento dos recursos computacionais necessários.

4.5.4.1. Dockerfile

O *Dockerfile* é um arquivo de texto que contém um conjunto de instruções utilizadas para construir uma imagem Docker. Essa imagem funciona como um "molde" que define como o container será criado e executado.

No contexto deste projeto, o *Dockerfile* é fundamental para garantir a padronização do ambiente tanto do cliente quanto do servidor na arquitetura de aprendizado federado. Isso assegura reprodutibilidade, facilidade de implantação e consistência entre diferentes máquinas e instituições.

De forma geral, um *Dockerfile* descreve:

- a imagem base do sistema;
- a organização do ambiente interno;
- a instalação de dependências;
- a cópia dos arquivos da aplicação;
- o comando de inicialização do container.

Dockerfile do cliente O código a seguir apresenta o *Dockerfile* utilizado no cliente:

```
FROM python:3.11
WORKDIR /app

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

COPY requirements.txt /app/requirements.txt

RUN pip install --upgrade pip && \
    pip install --no-cache-dir -r /app/requirements.txt

COPY . /app

CMD ["python", "app.py"]
```

A instrução `FROM` define a imagem base do container. No caso, utiliza-se `python:3.11`, uma imagem oficial que já contém o interpretador Python instalado, servindo como ponto de partida para o ambiente da aplicação.

A linha `WORKDIR` define o diretório de trabalho dentro do container. Ao utilizar `/app`, todos os comandos subsequentes passam a ser executados nesse diretório, o que organiza melhor a estrutura do projeto.

As instruções **ENV** configuram variáveis de ambiente importantes. A variável `PYTHONDONTWRITEBYTECODE=1` impede a geração de arquivos `.pyc`, evitando arquivos desnecessários no container. Já `PYTHONUNBUFFERED=1` garante que a saída do Python seja exibida imediatamente no terminal, facilitando o monitoramento e a depuração.

A instrução **COPY** copia o arquivo `requirements.txt` da máquina local para o container. Esse arquivo contém todas as dependências Python necessárias para o funcionamento da aplicação.

O comando **RUN** executa instruções durante o processo de construção (*build*) da imagem Docker. Inicialmente, o comando `pipinstall--upgradepip` atualiza o gerenciador de pacotes do Python para uma versão mais recente, garantindo compatibilidade com as bibliotecas utilizadas no projeto. Em seguida, o comando *pip install –no-cache-dir -r /app/requirements.txt* instala todas as dependências listadas no arquivo `requirements.txt`, evitando o armazenamento de arquivos temporários de cache, o que contribui para a redução do tamanho final da imagem.

No contexto deste projeto, as principais dependências instaladas são:

- `flwr==1.8.0`: biblioteca *Flower*, responsável por implementar a infraestrutura de aprendizado federado, permitindo a comunicação entre servidor e clientes e a execução de estratégias como o FedAvg;
- `numpy==1.26.4`: biblioteca fundamental para computação numérica em Python, amplamente utilizada para manipulação eficiente de arrays e operações matemáticas vetorizadas;
- `torch==2.2.2`: framework de aprendizado profundo (*PyTorch*), utilizado para definição, treinamento e avaliação das redes neurais convolucionais empregadas no projeto;
- `torchvision==0.17.2`: extensão do PyTorch voltada para processamento de imagens, fornecendo transformações, datasets e modelos pré-treinados úteis para tarefas de visão computacional;
- `pillow==10.3.0`: biblioteca para manipulação de imagens em Python, utilizada para leitura, conversão e pré-processamento das imagens de entrada.

Em conjunto, essas dependências compõem a base tecnológica necessária para a execução do pipeline de aprendizado federado com redes neurais convolucionais, incluindo desde o processamento de dados até o treinamento distribuído dos modelos.

A instrução **COPY** `. /app` copia todo o conteúdo do projeto para dentro do container, incluindo o código-fonte da aplicação.

Por fim, a instrução **CMD** define o comando que será executado quando o container for iniciado. Nesse caso, o comando `pythonapp.py` inicia a aplicação cliente.

Dockerfile do servidor As instruções iniciais são idênticas às do cliente, pois tanto cliente quanto servidor utilizam o mesmo ambiente base em Python e seguem o mesmo processo de instalação de dependências e organização do código.

A principal diferença está na instrução `EXPOSE`. Essa linha indica que o container utilizará a porta `8080` para comunicação. Embora não publique automaticamente a porta, essa instrução documenta a intenção da aplicação e facilita a integração com ferramentas como o *docker-compose*.

A instrução final `CMD` mantém o mesmo comportamento, iniciando o servidor através do comando *python app.py*.

Síntese Os *Dockerfiles* apresentados seguem uma estrutura simples e eficiente: definem uma base Python, configuram o ambiente, instalam dependências, copiam o código e iniciam a aplicação. Essa padronização é essencial no contexto de aprendizado federado, pois garante que todos os nós (clientes e servidor) operem sob condições controladas e reprodutíveis, reduzindo problemas de incompatibilidade e facilitando a implantação distribuída.

4.5.4.2. Arquivo docker-compose.yaml

O arquivo `docker-compose.yaml` é utilizado para definir, configurar e gerenciar aplicações compostas por múltiplos containers de forma integrada. Enquanto o *Dockerfile* é responsável por descrever como uma imagem individual deve ser construída, o *docker-compose* atua em um nível superior, permitindo a orquestração de diversos serviços que, em conjunto, compõem um sistema distribuído.

No contexto deste projeto de aprendizado federado, essa ferramenta desempenha um papel central, pois permite representar toda a arquitetura distribuída de forma declarativa, clara e reprodutível. Em particular, o arquivo define explicitamente:

- um container responsável pelo servidor federado, que coordena o treinamento global;
- múltiplos containers representando os clientes (instituições participantes), cada um com seus próprios dados;
- a rede de comunicação entre os diferentes componentes do sistema;
- os volumes de dados locais, garantindo isolamento entre os clientes.

O uso do `docker-compose.yaml` traz diversas vantagens práticas:

- permite inicializar toda a infraestrutura com um único comando;
- garante isolamento e independência entre os clientes;
- facilita a simulação de cenários reais de aprendizado federado com dados distribuídos;

- promove reprodutibilidade experimental e padronização do ambiente.

O código a seguir apresenta a configuração utilizada neste projeto:

```yaml
services:
  fl-server:
    build: ./server
    container_name: fl-server
    volumes:
      - ./server:/app
      - ./data_sbcas/data/global_test:/app/global_test
    ports:
      - "8080:8080"
    networks:
      - fl-net

  fl-client-1:
    build: ./client
    container_name: fl-client-1
    volumes:
      - ./client:/app
      - ./data_sbcas/data/client_1:/app/data
    depends_on:
      - fl-server
    networks:
      - fl-net

  fl-client-2:
    build: ./client
    container_name: fl-client-2
    volumes:
      - ./client:/app
      - ./data_sbcas/data/client_2:/app/data
    depends_on:
      - fl-server
    networks:
      - fl-net

  fl-client-3:
    build: ./client
    container_name: fl-client-3
    volumes:
      - ./client:/app
      - ./data_sbcas/data/client_3:/app/data
    depends_on:
      - fl-server
    networks:
      - fl-net

```



```
46 networks:
47   fl-net:
48     driver: bridge
```

A diretiva `services` define os diferentes serviços que compõem a aplicação. Cada serviço corresponde a um container que será instanciado durante a execução do sistema.

O serviço denominado `fl-server` representa o servidor central do aprendizado federado. A instrução `build` aponta para o diretório `./server`, onde se encontra o respectivo *Dockerfile*. O campo `container_name` atribui um nome explícito ao container, facilitando sua identificação durante a execução e depuração.

A seção `volumes` é responsável por mapear diretórios da máquina local para dentro do container. No caso do servidor, o diretório `./server` é associado a `/app`, garantindo que o código-fonte esteja acessível internamente. Além disso, o diretório `./data_sbcas/data/global_test` é mapeado para `/app/global_test`, permitindo acesso a dados globais utilizados em avaliação.

A diretiva `ports` realiza o mapeamento de portas entre o host e o container. O valor `"8080:8080"` indica que a porta 8080 da máquina local será conectada à porta 8080 do container, tornando o servidor acessível externamente.

A seção `networks` conecta o serviço à rede virtual `fl-net`, permitindo comunicação com os demais containers.

Os serviços `fl-client-1`, `fl-client-2` e `fl-client-3` representam os diferentes clientes do sistema federado. Todos utilizam a mesma configuração base, sendo construídos a partir do diretório `./client`. Essa padronização garante que todos os clientes operem sob o mesmo ambiente computacional.

Cada cliente possui um nome distinto, definido por `container_name`, permitindo sua identificação individual. A principal diferença entre eles está nos volumes de dados associados. Por exemplo, cada cliente acessa um diretório específico, como `./data_sbcas/data/client_1`, `./data_sbcas/data/client_2` e `./data_sbcas/data/client_3`. Essa separação é essencial para simular o cenário real de aprendizado federado, no qual cada instituição mantém seus próprios dados localmente, sem compartilhamento direto.

A diretiva `depends_on` indica que os clientes dependem do serviço `fl-server`. Isso garante que o servidor seja inicializado antes dos clientes, assegurando o funcionamento correto da comunicação entre os nós.

Todos os serviços estão conectados à mesma rede `fl-net`, o que permite a comunicação direta entre servidor e clientes de forma transparente.

Por fim, a seção `networks` define a rede utilizada pelos containers. O driver `bridge` é o padrão do Docker para redes locais, permitindo que os containers se comuniquem como se estivessem em uma rede privada isolada.

Síntese O arquivo `docker-compose.yaml` implementa uma arquitetura completa de aprendizado federado em ambiente controlado. O servidor atua como coordenador global do treinamento, enquanto os clientes realizam o aprendizado local utilizando seus próprios dados. A separação via volumes garante o isolamento dos dados, e a rede compartilhada possibilita a troca de informações entre os nós.

Toda essa infraestrutura pode ser inicializada de forma simples com o comando:

```
docker-compose up
```

Essa abordagem torna o ambiente altamente reprodutível, escalável e adequado para experimentos distribuídos, aproximando o cenário experimental de aplicações reais em ambientes multicêntricos.

4.5.5. Dados Utilizados no Projeto

Neste projeto, foi utilizado o dataset *Single Cell Conventional Pap Smear Images*, composto por imagens citológicas individuais de exames de Papanicolau.

Para simular um cenário de aprendizado federado, os dados foram organizados de forma distribuída, utilizando volumes Docker associados a diferentes clientes. Cada cliente acessa apenas seu subconjunto local de dados, enquanto o conjunto de teste é mantido separado na pasta `global_test`, utilizada para avaliação do modelo global.

4.5.5.1. Organização e Particionamento dos Dados

O dataset foi estruturado em dois conjuntos principais: *Training* e *Test*. O conjunto de treinamento foi dividido entre três clientes, enquanto o conjunto de teste foi mantido íntegro para avaliação global.

O processo de preparação dos dados consiste em:

- validação da estrutura do dataset (consistência entre classes de treino e teste);
- criação dos diretórios dos clientes e do conjunto `global_test`;
- cópia dos dados de teste para avaliação global;
- divisão dos dados de treinamento entre os clientes.

A divisão dos dados de treinamento é realizada de forma balanceada por classe. Para cada classe, as imagens são embaralhadas e distribuídas entre os clientes:

```
random.shuffle(images)
client_splits = split_evenly(images, NUM_CLIENTS)

for client_idx, split_imgs in enumerate(client_splits, start=1):
```



```
    for img_path in split_imgs:
        dst = OUTPUT_DIR / f"client_{client_idx}" / cls /
         ↪  img_path.name
        safe_link_or_copy(img_path, dst, COPY_FILES)
```

Essa estratégia garante que cada cliente possua uma amostra representativa de todas as classes, mantendo uma distribuição aproximadamente balanceada.

Ao final, a estrutura dos dados segue o padrão esperado pela arquitetura Docker, em que cada cliente acessa seu diretório local (`/app/data`) e o servidor utiliza o conjunto `/app/global_test` para avaliação do modelo global.

Síntese A organização adotada permite simular um ambiente federado controlado, no qual os dados permanecem distribuídos entre os clientes, enquanto a avaliação do modelo é realizada de forma centralizada e consistente.

4.5.6. Implementação dos Clientes Federados

No contexto do aprendizado federado, os clientes são responsáveis por executar o treinamento local do modelo utilizando apenas os dados disponíveis em sua própria instituição. Diferentemente de uma abordagem centralizada, os dados brutos não são enviados ao servidor. Cada cliente treina localmente, calcula atualizações nos parâmetros do modelo e compartilha apenas esses parâmetros com o servidor federado.

No presente projeto, os clientes são implementados a partir do mesmo código-fonte, sendo diferenciados apenas pelo volume de dados montado no container Docker. Assim, os serviços `fl-client-1`, `fl-client-2` e `fl-client-3` simulam instituições distintas, cada uma com seu próprio conjunto local de imagens.

4.5.6.1. Configuração e Hiperparâmetros

A primeira parte do código define os principais hiperparâmetros utilizados pelo cliente federado. Esses valores controlam desde o local dos dados até o número de épocas locais executadas em cada rodada federada.

```
DATA_DIR = Path("/app/data")
SERVER_ADDRESS = "fl-server:8080"

BATCH_SIZE = 16
LOCAL_EPOCHS = 1
LEARNING_RATE = 1e-3
VAL_RATIO = 0.2
RANDOM_SEED = 42

NUM_RETRIES = 20
RETRY_SLEEP = 3
```



```
INITIAL_SLEEP = 5

DEVICE = torch.device("cuda" if torch.cuda.is_available() else
 ↪  "cpu")
```

A variável `DATA_DIR` indica o diretório local onde os dados do cliente são montados dentro do container. O endereço `SERVER_ADDRESS` define o servidor federado responsável por coordenar o treinamento. O tamanho do lote é definido por `BATCH_SIZE`, enquanto `LOCAL_EPOCHS` determina quantas épocas de treinamento local são executadas a cada rodada federada. A taxa de aprendizado `LEARNING_RATE` controla a magnitude das atualizações realizadas pelo otimizador.

A proporção `VAL_RATIO=0.2` indica que 20% dos dados locais são separados para validação. A semente aleatória `RANDOM_SEED` garante reprodutibilidade na divisão entre treino e validação. Por fim, as variáveis `NUM_RETRIES`, `RETRY_SLEEP` e `INITIAL_SLEEP` controlam o mecanismo de tentativa de conexão com o servidor, importante em ambientes baseados em Docker, nos quais os containers podem iniciar em tempos diferentes.

4.5.6.2. Inspeção dos Dados Locais

Antes de iniciar o treinamento, o cliente verifica se o volume de dados foi corretamente montado. Essa etapa é realizada pela função `inspect_data_dir()`, que identifica as subpastas de classes e contabiliza a quantidade de arquivos em cada uma delas.

```
def inspect_data_dir(data_dir: Path) -> None:
    if not data_dir.exists():
        raise FileNotFoundError(f"Pasta de dados não encontrada:
         ↪  {data_dir}")

    class_dirs = sorted([p for p in data_dir.iterdir() if
     ↪  p.is_dir()],
                        key=lambda x: x.name)

    if not class_dirs:
        raise ValueError(f"Nenhuma subpasta de classe encontrada
         ↪  em {data_dir}")

    total_files = 0

    for class_dir in class_dirs:
        num_files = len([p for p in class_dir.iterdir() if
         ↪  p.is_file()])
        total_files += num_files
```



```
        log(f"{class_dir.name}: {num_files} arquivos")

    log(f"Total de arquivos em /app/data: {total_files}")
```

Essa verificação é importante porque cada cliente depende de um volume Docker específico. Caso o volume esteja vazio, incorreto ou sem subpastas de classe, o treinamento não deve prosseguir. Além disso, essa função permite confirmar se a distribuição de dados entre os clientes está correta.

4.5.6.3. Carregamento, Transformações e Divisão dos Dados

A função `get_dataloaders()` é responsável por carregar as imagens, aplicar transformações e criar os carregadores de dados utilizados no treinamento e na validação.

```
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225],
    ),
])
```

No conjunto de treinamento, são aplicadas transformações de aumento de dados (data augmentation), como espelhamento horizontal e rotação aleatória. Essas operações ajudam a reduzir o overfitting e tornam o modelo mais robusto a pequenas variações nas imagens.

Para validação, são utilizadas apenas transformações determinísticas:

```
val_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225],
    ),
])
```

As imagens são redimensionadas para 224×224, tamanho esperado pela arquitetura ResNet-18. A normalização utiliza as médias e desvios padrão do ImageNet, pois o modelo empregado foi previamente treinado nessa base.

O carregamento dos dados é feito com `ImageFolder`, que assume uma estrutura em que cada subpasta representa uma classe:

```
base_dataset = datasets.ImageFolder(root=str(DATA_DIR))
class_names = base_dataset.classes
num_classes = len(class_names)
```

Em seguida, os dados locais são divididos em treino e validação:

```
dataset_size = len(base_dataset)
val_size = max(1, int(dataset_size * VAL_RATIO))
train_size = dataset_size - val_size

generator = torch.Generator().manual_seed(RANDOM_SEED)

train_subset, val_subset = random_split(
    base_dataset,
    [train_size, val_size],
    generator=generator,
)
```

A divisão é feita localmente em cada cliente. Portanto, cada instituição simulada possui sua própria partição de treino e validação. Esse comportamento é coerente com o cenário federado, pois os dados não são centralizados.

Como o conjunto de treino e o conjunto de validação usam transformações diferentes, foi criada uma classe auxiliar chamada `TransformedSubset`:

```
class TransformedSubset(torch.utils.data.Dataset):
    def __init__(self, subset, transform):
        self.subset = subset
        self.transform = transform

    def __len__(self):
        return len(self.subset)

    def __getitem__(self, idx):
        image, label = self.subset[idx]
        image = image.convert("RGB")
        image = self.transform(image)
        return image, label
```

Por fim, são criados os *DataLoaders*:


```
train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=0,
)

val_loader = DataLoader(
    val_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=0,
)
```

O `shuffle=True` é utilizado no treino para embaralhar as amostras a cada época. Na validação, o embaralhamento não é necessário, pois o objetivo é apenas avaliar o desempenho do modelo.

4.5.6.4. Construção do Modelo Local

O modelo local é definido pela função `build_model()`. Neste projeto, foi utilizada a arquitetura ResNet-18 com pesos pré-treinados no ImageNet.

```
def build_model(num_classes: int) -> nn.Module:
    weights = models.ResNet18_Weights.DEFAULT
    model = models.resnet18(weights=weights)

    for param in model.parameters():
        param.requires_grad = False

    in_features = model.fc.in_features
    model.fc = nn.Linear(in_features, num_classes)

    return model
```

Inicialmente, todos os parâmetros da rede são congelados. Dessa forma, o treinamento local atualiza apenas a camada final do classificador. Essa estratégia caracteriza uma forma de *transfer learning*, pois reaproveita representações visuais aprendidas em uma grande base de imagens e adapta apenas a saída do modelo ao número de classes do problema.

A camada final original da ResNet-18 é substituída por uma nova camada linear com saída igual ao número de classes identificadas no diretório local.

4.5.6.5. Funções de Treinamento e Avaliação

A função `train()` executa uma época de treinamento local. Ela coloca o modelo em modo de treinamento, percorre os lotes de imagens, calcula a perda, realiza a retropropagação e atualiza os parâmetros treináveis.

```
def train(model, loader, criterion, optimizer):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in loader:
        images = images.to(DEVICE)
        labels = labels.to(DEVICE)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)
        preds = outputs.argmax(dim=1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

    loss = running_loss / total
    acc = correct / total
    return loss, acc
```

A função de perda utilizada é a entropia cruzada, adequada para problemas de classificação multiclasse:

```
criterion = nn.CrossEntropyLoss()
```

Como apenas a camada final foi descongelada, o otimizador Adam recebe somente os parâmetros de `model.fc`:

```
optimizer = torch.optim.Adam(
    model.fc.parameters(),
    lr=LEARNING_RATE
)
```


A avaliação é realizada pela função `evaluate()`, que opera sem cálculo de gradientes:

```
@torch.no_grad()
def evaluate(model, loader, criterion):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in loader:
        images = images.to(DEVICE)
        labels = labels.to(DEVICE)

        outputs = model(images)
        loss = criterion(outputs, labels)

        running_loss += loss.item() * images.size(0)
        preds = outputs.argmax(dim=1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

    loss = running_loss / total
    acc = correct / total
    return loss, acc
```

Essa função retorna duas métricas principais: a perda média e a acurácia. No contexto federado, essas métricas são importantes para acompanhar o comportamento de cada cliente ao longo das rodadas.

4.5.6.6. Conversão dos Parâmetros do Modelo

A comunicação entre PyTorch e Flower exige que os parâmetros do modelo sejam convertidos para listas de arrays NumPy. Essa conversão é feita pela função `get_parameters()`.

```
def get_parameters(model) -> List[np.ndarray]:
    return [val.cpu().numpy() for _, val in
     ↪  model.state_dict().items()]
```

Quando o cliente recebe parâmetros globais enviados pelo servidor, eles precisam ser convertidos de volta para tensores PyTorch. Essa etapa é feita pela função `set_parameters()`.


```python
def set_parameters(model, parameters: List[np.ndarray]) -> None:
    params_dict = zip(model.state_dict().keys(), parameters)
    state_dict = OrderedDict(
        {k: torch.tensor(v) for k, v in params_dict}
    )
    model.load_state_dict(state_dict, strict=True)
```

Essas duas funções são essenciais para o aprendizado federado, pois permitem que o modelo local receba os pesos globais, realize treinamento local e devolva os pesos atualizados ao servidor.

4.5.6.7. Inicialização do Cliente

Após a definição das funções, o cliente executa a sequência de inicialização. Primeiro, verifica o dispositivo disponível, inspeciona os dados, cria os *DataLoaders*, constrói o modelo e define a função de perda e o otimizador.

```python
inspect_data_dir(DATA_DIR)
train_loader, val_loader, num_classes = get_dataloaders()

model = build_model(num_classes).to(DEVICE)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
    model.fc.parameters(),
    lr=LEARNING_RATE
)
```

Esse trecho concentra a preparação do cliente antes de sua conexão com o servidor federado.

4.5.6.8. Implementação do Cliente Flower

A integração com o Flower é feita por meio da classe `FlowerClient`, que herda de `fl.client.NumPyClient`. Essa classe implementa três métodos principais: `get_parameters()`, `fit()` e `evaluate()`.

```python
class FlowerClient(fl.client.NumPyClient):
    def get_parameters(self, config):
        return get_parameters(model)
```

O método `get_parameters()` retorna os parâmetros atuais do modelo local para o servidor.

O método `fit()` representa a etapa de treinamento local em uma rodada federada:

```
def fit(self, parameters, config):
    set_parameters(model, parameters)

    for epoch in range(LOCAL_EPOCHS):
        train_loss, train_acc = train(
            model,
            train_loader,
            criterion,
            optimizer
        )

    val_loss, val_acc = evaluate(model, val_loader, criterion)

    return get_parameters(model), len(train_loader.dataset), {
        "train_loss": float(train_loss),
        "train_acc": float(train_acc),
        "val_loss": float(val_loss),
        "val_acc": float(val_acc),
    }
```

Inicialmente, o cliente recebe os parâmetros globais enviados pelo servidor e atualiza seu modelo local com `set_parameters()`. Em seguida, realiza o treinamento local por `LOCAL_EPOCHS`. Ao final, avalia o modelo em seu conjunto de validação e retorna três informações ao servidor: os parâmetros atualizados, o número de amostras locais usadas no treinamento e um dicionário com métricas locais.

O método `evaluate()` é utilizado quando o servidor solicita a avaliação do modelo global em cada cliente:

```
def evaluate(self, parameters, config):
    set_parameters(model, parameters)

    loss, acc = evaluate(model, val_loader, criterion)

    return float(loss), len(val_loader.dataset), {
        "accuracy": float(acc)
    }
```

Nesse caso, o cliente recebe os pesos globais, atualiza seu modelo local e calcula a perda e a acurácia no conjunto de validação local.

4.5.6.9. Conexão com o Servidor Federado

A conexão com o servidor é feita pela função `fl.client.start_client()`. Antes da conexão, o cliente aguarda alguns segundos para reduzir problemas causados pela inicialização assíncrona dos containers.

```
time.sleep(INITIAL_SLEEP)

for i in range(NUM_RETRIES):
    try:
        fl.client.start_client(
            server_address=SERVER_ADDRESS,
            client=FlowerClient().to_client(),
        )
        break
    except Exception as e:
        log(f"Falha ao conectar: {e}")
        time.sleep(RETRY_SLEEP)
```

Esse mecanismo de repetição é importante porque, em uma arquitetura com Docker Compose, o cliente pode iniciar antes do servidor estar pronto para receber conexões. Assim, o código tenta se conectar várias vezes antes de encerrar.

Síntese A implementação do cliente federado organiza, em um único fluxo, todas as etapas necessárias para o treinamento distribuído: inspeção dos dados locais, carregamento das imagens, aplicação de transformações, construção do modelo, treinamento local, avaliação, conversão de parâmetros e comunicação com o servidor Flower.

A arquitetura adotada utiliza *transfer learning* com ResNet-18, mantendo o extrator de características congelado e treinando apenas a camada final. Essa decisão reduz o custo computacional dos clientes e torna o treinamento mais viável em um ambiente federado. Além disso, a separação dos dados por volumes Docker permite simular um cenário multicêntrico, no qual diferentes instituições colaboram para treinar um modelo global sem compartilhar diretamente suas imagens.

4.5.7. Implementação do Servidor Federado

No aprendizado federado, o servidor atua como o coordenador central do processo de treinamento distribuído. Sua função não é acessar os dados brutos dos clientes, mas organizar as rodadas federadas, receber os parâmetros treinados localmente, agregá-los em um novo modelo global e redistribuir esse modelo atualizado aos clientes.

No presente projeto, o servidor foi implementado com o framework Flower, utilizando uma estratégia customizada baseada no algoritmo *Federated Averaging* (FedAvg). Além da agregação dos modelos locais, o servidor também realiza avaliação em um conjunto global de teste, salva checkpoints a cada rodada e mantém um histórico completo das métricas de treinamento.

4.5.7.1. Configurações e Hiperparâmetros do Servidor

A primeira parte do código define os principais diretórios, arquivos de saída e hiperparâmetros utilizados pelo servidor federado.

```
GLOBAL_TEST_DIR = Path("/app/global_test")
OUTPUT_DIR = Path("/app/output")
CHECKPOINTS_DIR = OUTPUT_DIR / "checkpoints"
HISTORY_JSON = OUTPUT_DIR / "history.json"
HISTORY_CSV = OUTPUT_DIR / "history.csv"

BATCH_SIZE = 16
DEVICE = torch.device("cuda" if torch.cuda.is_available() else
 ↪  "cpu")
```

O diretório `GLOBAL_TEST_DIR` armazena o conjunto de teste global, utilizado exclusivamente para avaliar o modelo agregado após cada rodada federada. Esse conjunto não participa do treinamento dos clientes.

O diretório `OUTPUT_DIR` concentra os arquivos gerados durante a execução, enquanto `CHECKPOINTS_DIR` armazena os modelos salvos ao longo das rodadas. Os arquivos `history.json` e `history.csv` registram o histórico do treinamento em formatos adequados tanto para leitura estruturada quanto para análise tabular.

O hiperparâmetro `BATCH_SIZE=16` define o tamanho dos lotes utilizados na avaliação global. Já a variável `DEVICE` seleciona automaticamente GPU, caso esteja disponível, ou CPU, caso contrário.

```
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
CHECKPOINTS_DIR.mkdir(parents=True, exist_ok=True)
```

Essas instruções garantem que os diretórios de saída existam antes do início do treinamento federado.

4.5.7.2. Registro do Histórico de Treinamento

O servidor mantém uma lista chamada `history_records`, na qual são armazenadas as métricas de cada rodada federada. Também é mantida a variável `best_global_acc`, responsável por registrar a melhor acurácia obtida no conjunto de teste global.

```
history_records = []
best_global_acc = -1.0
```


A função `save_history()` salva esse histórico em dois formatos: `JSON` e `CSV`.

```
def save_history():
    with open(HISTORY_JSON, "w", encoding="utf-8") as f:
        json.dump(history_records, f, indent=2,
         ↪  ensure_ascii=False)

    fieldnames = [
        "round",
        "train_loss",
        "train_acc",
        "val_loss",
        "val_acc",
        "distributed_accuracy",
        "global_loss",
        "global_acc",
    ]

    with open(HISTORY_CSV, "w", newline="", encoding="utf-8") as
     ↪  f:
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        writer.writeheader()
        for row in history_records:
            writer.writerow(row)
```

Esse mecanismo é importante para reprodutibilidade experimental. As métricas salvas permitem analisar a evolução do treinamento, comparar execuções e identificar a melhor rodada federada.

4.5.7.3. Construção do Conjunto Global de Teste

A função `build_global_test_loader()` é responsável por carregar o conjunto de teste global armazenado em `/app/global_test`.

```
def build_global_test_loader():
    if not GLOBAL_TEST_DIR.exists():
        raise FileNotFoundError(
            f"Diretório global_test não encontrado:
             ↪  {GLOBAL_TEST_DIR}"
        )

    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(
```



```
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225],
        ),
    ])

    dataset = datasets.ImageFolder(
        root=str(GLOBAL_TEST_DIR),
        transform=transform
    )

    loader = DataLoader(
        dataset,
        batch_size=BATCH_SIZE,
        shuffle=False,
        num_workers=0
    )

    return loader, len(dataset.classes)
```

O conjunto global segue a estrutura esperada pelo `ImageFolder`, em que cada subdiretório representa uma classe. As imagens são redimensionadas para 224×224, convertidas para tensores e normalizadas com as médias e desvios padrão do ImageNet, de forma compatível com a ResNet-18 pré-treinada.

```
GLOBAL_TEST_LOADER, NUM_CLASSES = build_global_test_loader()
```

Essa chamada inicializa o carregador global de teste e identifica automaticamente o número de classes do problema.

4.5.7.4. Construção do Modelo Global

O servidor utiliza a mesma arquitetura dos clientes: uma ResNet-18 pré-treinada. Essa consistência é fundamental, pois os parâmetros enviados pelos clientes precisam ser compatíveis com a estrutura do modelo mantido no servidor.

```
def build_model(num_classes: int) -> nn.Module:
    weights = models.ResNet18_Weights.DEFAULT
    model = models.resnet18(weights=weights)

    for param in model.parameters():
        param.requires_grad = False

    in_features = model.fc.in_features
    model.fc = nn.Linear(in_features, num_classes)
```



```
10
11      return model
```

Assim como nos clientes, o backbone convolucional é congelado e apenas a camada final é substituída para se ajustar ao número de classes do conjunto utilizado. Essa abordagem reduz o custo computacional e mantém o processo de treinamento federado mais leve.

4.5.7.5. Atualização dos Parâmetros Globais

Após a agregação federada, os parâmetros do modelo são recebidos em formato NumPy. Para reconstruir o modelo em PyTorch, utiliza-se a função `set_parameters()`.

```
1  def set_parameters(model: nn.Module, parameters:
    ↪  List[np.ndarray]) -> None:
2      params_dict = zip(model.state_dict().keys(), parameters)
3      state_dict = OrderedDict({
4          k: torch.tensor(v) for k, v in params_dict
5      })
6      model.load_state_dict(state_dict, strict=True)
```

Essa função associa cada array recebido ao nome correspondente no `state_dict` do modelo. Em seguida, os pesos são carregados com `strict=True`, garantindo que a estrutura dos parâmetros recebidos seja compatível com a arquitetura esperada.

4.5.7.6. Salvamento de Checkpoints

A função `save_model_checkpoint()` salva os pesos do modelo global em disco.

```
1  def save_model_checkpoint(parameters: List[np.ndarray],
    ↪  filepath: Path):
2      model = build_model(NUM_CLASSES)
3      set_parameters(model, parameters)
4      torch.save(model.state_dict(), filepath)
5      print(f"Checkpoint salvo em: {filepath}", flush=True)
```

A cada rodada, o servidor salva um checkpoint do modelo global agregado. Além disso, quando a acurácia global melhora, o modelo também é salvo como `best_model.pt`. Isso permite recuperar tanto modelos intermediários quanto o melhor modelo obtido durante a execução.

4.5.7.7. Avaliação do Modelo Global

A avaliação global é realizada pela função `evaluate_global_model()`. Ela calcula a perda e a acurácia do modelo agregado utilizando o conjunto `global_test`.

```
@torch.no_grad()
def evaluate_global_model(model: nn.Module, loader: DataLoader):
    criterion = nn.CrossEntropyLoss()
    model.eval()

    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in loader:
        images = images.to(DEVICE)
        labels = labels.to(DEVICE)

        outputs = model(images)
        loss = criterion(outputs, labels)

        running_loss += loss.item() * images.size(0)
        preds = outputs.argmax(dim=1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

    global_loss = running_loss / total
    global_acc = correct / total

    return global_loss, global_acc
```

O uso de `@torch.no_grad()` evita o cálculo de gradientes durante a avaliação, reduzindo o consumo de memória e tornando a inferência mais eficiente. Essa avaliação é importante porque fornece uma medida independente do desempenho do modelo global, sem depender exclusivamente das validações locais dos clientes.

4.5.7.8. Estratégia Federada Customizada

O núcleo do servidor está na classe `FedAvgWithGlobalTest`, que herda da estratégia padrão `FedAvg` do Flower.

```
class FedAvgWithGlobalTest(fl.server.strategy.FedAvg):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.latest_parameters = None
        self.latest_fit_metrics = {}
```


Essa classe adiciona duas variáveis internas. A primeira, `latest_parameters`, armazena os parâmetros globais mais recentes após a agregação. A segunda, `latest_fit_metrics`, guarda as métricas agregadas de treinamento enviadas pelos clientes.

Agregação dos Modelos Locais O método `aggregate_fit()` é executado após os clientes realizarem o treinamento local e enviarem seus parâmetros atualizados ao servidor.

```
def aggregate_fit(self, server_round, results, failures):
    aggregated_parameters, aggregated_metrics =
     ↪  super().aggregate_fit(
        server_round,
        results,
        failures
    )

    if aggregated_parameters is not None:
        self.latest_parameters = aggregated_parameters

        ndarrays = parameters_to_ndarrays(aggregated_parameters)
        round_ckpt = CHECKPOINTS_DIR /
         ↪  f"round_{server_round:03d}.pt"
        save_model_checkpoint(ndarrays, round_ckpt)

    self.latest_fit_metrics = aggregated_metrics or {}

    return aggregated_parameters, aggregated_metrics
```

A chamada `super().aggregate_fit()` executa a agregação FedAvg original do Flower. Em seguida, os parâmetros agregados são armazenados e convertidos para arrays NumPy por meio da função `parameters_to_ndarrays()`.

Cada rodada gera um checkpoint com nome padronizado, como `round_001.pt`, `round_002.pt` e assim por diante. Dessa forma, é possível recuperar o estado do modelo global em qualquer rodada do treinamento.

Avaliação Agregada e Avaliação Global O método `aggregate_evaluate()` é executado após a etapa de avaliação distribuída dos clientes. Além de agregar as métricas locais, esse método reconstrói o modelo global e o avalia no conjunto `global_test`.

```
def aggregate_evaluate(self, server_round, results, failures):
    global best_global_acc

```



```
    aggregated_loss, aggregated_metrics = super().aggregate_evaluate(
        server_round,
        results,
        failures
    )

    if aggregated_metrics is None:
        aggregated_metrics = {}

    global_loss = None
    global_acc = None

    if self.latest_parameters is not None:
        global_ndarrays = parameters_to_ndarrays(self.latest_parameters)
        model = build_model(NUM_CLASSES).to(DEVICE)
        set_parameters(model, global_ndarrays)

        global_loss, global_acc = evaluate_global_model(
            model,
            GLOBAL_TEST_LOADER
        )

        aggregated_metrics["global_loss"] = float(global_loss)
        aggregated_metrics["global_acc"] = float(global_acc)
```

Esse trecho reconstrói o modelo global a partir dos parâmetros agregados na rodada atual. Em seguida, calcula a perda e a acurácia no conjunto global de teste. Essas métricas são adicionadas ao dicionário de métricas agregadas.

O código também verifica se o modelo atual é o melhor obtido até o momento:

```
if global_acc > best_global_acc:
    best_global_acc = global_acc
    best_ckpt = CHECKPOINTS_DIR / "best_model.pt"
    save_model_checkpoint(global_ndarrays, best_ckpt)
```

Caso a acurácia global da rodada atual seja superior à melhor acurácia anterior, o modelo é salvo como `best_model.pt`. Essa estratégia facilita a seleção do melhor modelo final.

Ao final da rodada, as métricas são registradas no histórico:

```
record = {
    "round": server_round,
```

```
    "train_loss":
     ↪  float(self.latest_fit_metrics.get("train_loss",
     ↪  np.nan)),
    "train_acc": float(self.latest_fit_metrics.get("train_acc",
     ↪  np.nan)),
    "val_loss": float(self.latest_fit_metrics.get("val_loss",
     ↪  np.nan)),
    "val_acc": float(self.latest_fit_metrics.get("val_acc",
     ↪  np.nan)),
    "distributed_accuracy": float(
        aggregated_metrics.get("accuracy", np.nan)
    ),
    "global_loss": float(
        global_loss if global_loss is not None else np.nan
    ),
    "global_acc": float(
        global_acc if global_acc is not None else np.nan
    ),
}

history_records.append(record)
save_history()
```

Esse registro reúne métricas locais agregadas, métricas distribuídas de avaliação e métricas globais independentes. Com isso, o servidor mantém uma visão completa da evolução do treinamento federado.

4.5.7.9. Agregação Ponderada de Métricas

A função `weighted_average()` agrega as métricas enviadas pelos clientes considerando o número de amostras de cada um.

```
def weighted_average(metrics):
    total_examples = sum(num_examples for num_examples, _ in
     ↪  metrics)

    if total_examples == 0:
        return {}

    aggregated = {}
    metric_names = set()

    for _, client_metrics in metrics:
        metric_names.update(client_metrics.keys())
```


Inicialmente, a função calcula o número total de amostras utilizadas pelos clientes e identifica quais métricas foram enviadas. Em seguida, calcula a média ponderada de cada métrica.

```
for metric_name in metric_names:
    weighted_sum = 0.0
    valid_examples = 0

    for num_examples, client_metrics in metrics:
        if metric_name in client_metrics:
            weighted_sum += num_examples *
             ↪  float(client_metrics[metric_name])
            valid_examples += num_examples

    if valid_examples > 0:
        aggregated[metric_name] = weighted_sum / valid_examples
```

Essa agregação evita que clientes com poucos dados tenham o mesmo peso estatístico de clientes com muitos dados. Formalmente, para uma métrica m, a agregação ponderada pode ser representada por:

$$\bar{m} = \frac{\sum_{k=1}^{K} n_k m_k}{\sum_{k=1}^{K} n_k},$$

em que n_k representa o número de amostras do cliente k e m_k representa a métrica calculada nesse cliente.

4.5.7.10. Configuração da Estratégia Federada

A estratégia federada é instanciada com parâmetros que definem a participação dos clientes em cada rodada.

```
strategy = FedAvgWithGlobalTest(
    fraction_fit=1.0,
    fraction_evaluate=1.0,
    min_fit_clients=3,
    min_evaluate_clients=3,
    min_available_clients=3,
    fit_metrics_aggregation_fn=weighted_average,
    evaluate_metrics_aggregation_fn=weighted_average,
)
```

O parâmetro `fraction_fit=1.0` indica que todos os clientes disponíveis devem participar do treinamento em cada rodada. De forma semelhante, `fraction_`

`evaluate=1.0` indica que todos os clientes disponíveis também devem participar da avaliação.

Os parâmetros `min_fit_clients=3`, `min_evaluate_clients=3` e `min_available_clients=3` definem que o treinamento federado exige três clientes disponíveis. Isso é coerente com a arquitetura do projeto, composta pelos serviços `fl-client-1`, `fl-client-2` e `fl-client-3`.

As funções `fit_metrics_aggregation_fn` e `evaluate_metrics_aggregation_fn` indicam que as métricas de treinamento e avaliação serão agregadas por média ponderada.

4.5.7.11. Inicialização do Servidor Flower

Por fim, o servidor é inicializado na porta `8080`.

```
fl.server.start_server(
    server_address="0.0.0.0:8080",
    config=fl.server.ServerConfig(num_rounds=3),
    strategy=strategy,
)
```

O endereço `0.0.0.0:8080` permite que o servidor aceite conexões dos containers clientes dentro da rede Docker. O parâmetro `num_rounds=3` define que o treinamento federado será executado por três rodadas. Em cada rodada, os clientes recebem o modelo global, treinam localmente, retornam seus parâmetros e o servidor realiza a agregação.

Síntese A implementação do servidor federado organiza todo o ciclo de treinamento colaborativo. Primeiro, o servidor carrega um conjunto global de teste, define a arquitetura do modelo e inicializa a estratégia FedAvg customizada. Em seguida, a cada rodada, recebe os modelos locais dos clientes, agrega os parâmetros, salva checkpoints, avalia o modelo global e registra as métricas em arquivos de histórico.

Essa implementação vai além de um servidor federado básico, pois adiciona mecanismos importantes para experimentação científica: avaliação global independente, armazenamento do melhor modelo, checkpoints por rodada e histórico estruturado em `JSON` e `CSV`. Esses elementos tornam o experimento mais reprodutível, auditável e adequado para análise posterior no contexto de aplicações médicas sensíveis.

4.6. Conclusão

Neste trabalho, foi apresentada uma abordagem integrada que combina fundamentos teóricos e implementação prática do aprendizado federado aplicado à análise de imagens citopatológicas para a triagem do câncer do colo do útero. Ao longo do texto, discutimos os principais conceitos relacionados ao aprendizado federado, redes neurais

convolucionais e os desafios inerentes ao processamento de dados médicos distribuídos e sensíveis.

A construção do pipeline proposto evidenciou a importância de etapas como o pré-processamento de lâminas digitais, a seleção de regiões relevantes e a organização dos dados em um ambiente federado. A utilização de tecnologias como Docker, Flower e PyTorch permitiu a criação de um ambiente reprodutível, modular e escalável, capaz de simular cenários reais com múltiplas instituições colaborando de forma segura.

A implementação dos clientes e do servidor demonstrou, na prática, como o aprendizado federado pode ser operacionalizado, destacando o fluxo completo de treinamento distribuído, agregação de modelos e avaliação global. Esse processo reforça a viabilidade do uso do aprendizado federado em aplicações médicas, especialmente em contextos onde a privacidade dos dados é um requisito crítico.

Além disso, o trabalho evidencia que a integração entre aprendizado profundo e aprendizado federado permite mitigar desafios importantes, como o compartilhamento de dados sensíveis e a variabilidade entre diferentes fontes de dados. Como resultado, obtém-se um modelo global mais robusto e com maior capacidade de generalização.

Como perspectivas futuras, destacam-se a investigação de cenários com dados não independentes e não identicamente distribuídos (non-IID) mais complexos, a incorporação de técnicas de privacidade diferencial e segurança criptográfica, bem como a avaliação em ambientes reais com múltiplas instituições de saúde. Esses avanços podem contribuir significativamente para a consolidação do aprendizado federado como uma solução prática e confiável para sistemas de inteligência artificial na área da saúde.

Referências

[Bishop and Bishop 2024] Bishop, C. M. and Bishop, H. (2024). *Deep Learning: Foundations and Concepts*. Springer, Cham.

[Diniz et al. 2021] Diniz, D. N., Rezende, M. T., Bianchi, A. G. C., Carneiro, C. M., Ushizima, D. M., Medeiros, F. N. S. d., and Souza, M. J. F. (2021). A hierarchical feature-based methodology to perform cervical cancer classification. *Applied Sciences*, 11(9):4091.

[Goodfellow et al. 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA.

[INCA 2022] INCA (2022). Estimativa 2023: incidência de câncer no brasil. Technical report, Instituto Nacional de Câncer José Alencar Gomes da Silva, Rio de Janeiro.

[LeCun 1989] LeCun, Y. (1989). Generalization and network design strategies. Technical Report CRG-TR-89-4, University of Toronto.

[Liang et al. 2023] Liang, Y., Feng, S., Liu, Q., Kuang, H., Liu, J., Liao, L., Du, Y., and Wang, J. (2023). Exploring contextual relationships for cervical abnormal cell detection. *IEEE Journal of Biomedical and Health Informatics*, 27(8):4086–4097.

[Liang et al. 2021] Liang, Y., Tang, Z., Yan, M., Chen, J., Liu, Q., and Xiang, Y. (2021). Comparison detector for cervical cell/clumps detection in the limited data scenario. *Neurocomputing*, 437:195–205.

[McMahan et al. 2017] McMahan, B., Moore, E., Ramage, D., Hampson, S., and Arcas, B. A. y. (2017). Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR.

[Scheffer 2025] Scheffer, M. C. (2025). Demografia médica no brasil 2025. Acesso em: 12 jan. 2026.

[Teixeira et al. 2023] Teixeira, J. B. A., Rezende, M. T., Diniz, D. N., Carneiro, C. M., Luz, E. J. d. S., Souza, M. J. F., Ushizima, D. M., Medeiros, F. N. S. d., and Bianchi, A. G. C. (2023). Segmentation of cervical nuclei using convolutional neural network for conventional cytology. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization*, 11(5):1876–1888.

[Terra et al. 2023] Terra, D. C., Lisboa, A. C., Rezende, M. T., Carneiro, C. M., and Bianchi, A. G. C. (2023). Shape-based features investigation for preneoplastic lesions on cervical cancer diagnosis. In *Proceedings of the 18th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2023) - Volume 4: VISAPP*, pages 506–513. SCITEPRESS.

