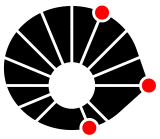
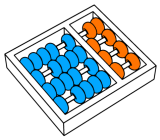




Simulando o custo ambiental de cenários em aprendizado federado

M. Casanova *T. Zorzetto* *V. Dominguite*

Relatório Técnico - IC-PFG-24-23
Projeto Final de Graduação
2024 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Simulando o custo ambiental de cenários em aprendizado federado

Matheus Casanova* Tobias Zorzetto* Victor Dominguite*

Resumo

O crescimento exponencial de dispositivos móveis e o aumento da complexidade de modelos de aprendizado de máquina impulsionaram o interesse em abordagens descentralizadas, como o aprendizado federado. Essa metodologia permite o treinamento colaborativo de modelos sem a necessidade de transferência de dados sensíveis, promovendo maior privacidade e eficiência. No entanto, o impacto do aprendizado federado no consumo energético dos dispositivos móveis ainda é um desafio subexplorado, especialmente em cenários de uso real. Além disso, ainda não existem ferramentas difundidas para medir o impacto desse treinamento distribuído em diferentes dispositivos.

Diante disso, este projeto apresenta o desenvolvimento de uma aplicação *Android* projetada para permitir o treinamento on-device de modelos de aprendizado de máquina utilizando *datasets* customizados. A aplicação coleta métricas detalhadas de consumo energético durante o processo de treinamento, fornecendo subsídios para a análise de impacto do aprendizado federado em dispositivos móveis. Além disso, foi implementada uma simulação de aprendizado federado baseada nas medições obtidas, com o objetivo de avaliar o impacto ambiental e de desempenho dessa abordagem em diferentes cenários.

Assim, este trabalho fornece meios para medição do impacto energético do aprendizado federado em cenários customizados. Os resultados demonstram a viabilidade da aplicação para auxiliar pesquisadores e desenvolvedores na análise de *trade-offs* entre desempenho, consumo energético e impacto ambiental em arquiteturas de aprendizado federado.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

Sumário

1 Introdução

O aprendizado federado tem se destacado como uma alternativa ao treinamento centralizado de modelos de aprendizado de máquina, oferecendo maior privacidade ao permitir que os dados permaneçam nos dispositivos dos clientes. Contudo, essa abordagem enfrenta desafios relacionados à limitação de recursos computacionais dos dispositivos móveis, que assumem uma carga de processamento anteriormente atribuída aos servidores. Essa transferência de computação pode impactar significativamente o consumo de energia e outros recursos sistêmicos desses dispositivos, tornando essencial uma análise detalhada para avaliar os custos e a viabilidade dessa solução distribuída.

Apesar de seu potencial, as simulações de aprendizado federado frequentemente utilizam perfis genéricos ou simplificados para estimar o consumo de recursos dos dispositivos clientes, o que compromete a precisão dos resultados. Além disso, carece-se de ferramentas acessíveis e práticas que permitam aos pesquisadores obter perfis reais de consumo de energia e outros recursos diretamente de dispositivos móveis, dificultando a realização de simulações mais realistas e representativas.

Este trabalho propõe uma solução para essa lacuna: uma aplicação Android capaz de realizar o treinamento on-device de modelos de aprendizado de máquina personalizados, extraindo métricas detalhadas de consumo de recursos sistêmicos. A aplicação se destina a pesquisadores e desenvolvedores que necessitam de perfis de consumo realistas para simulações ou estudos relacionados ao aprendizado federado. Com uma interface configurável, o aplicativo permite a seleção de modelos e datasets, parametrização do treinamento e armazenamento das métricas coletadas em um banco de dados na nuvem, facilitando sua posterior análise e utilização.

As principais contribuições deste trabalho incluem: (i) o desenvolvimento de uma aplicação flexível que suporta a configuração de modelos e datasets personalizados para o treinamento on-device; (ii) a coleta de métricas detalhadas, como tempo de treinamento, consumo de energia e tamanho dos modelos, diretamente de dispositivos reais; (iii) a criação de ferramentas complementares para simulações em aprendizado federado, utilizando os perfis coletados; e (iv) a disponibilização de códigos de referência para a geração de modelos e datasets compatíveis, incentivando a replicação e a adoção da abordagem proposta por outros pesquisadores.

O restante do relatório está organizado da seguinte forma: a próxima seção detalha o processo de geração dos modelos e datasets para uso no dispositivo móvel. Em seguida, apresenta-se a aplicação Android desenvolvida, descrevendo seu fluxo de execução e principais aspectos de implementação. Posteriormente, discute-se o ambiente de simulação criado, com ênfase na integração dos perfis coletados e nas análises realizadas. Por fim, conclui-se o trabalho com uma síntese das contribuições alcançadas e uma reflexão sobre as perspectivas futuras.

2 Projeto da solução

O projeto visa desenvolver uma aplicação que seja capaz de treinar qualquer modelo e dataset de forma embarcada em dispositivos móveis, extraindo métricas de gasto de energia advindas desse procedimento. Para atingir esse nível de generalização, o primeiro passo seria desenvolver uma etapa que precede a aplicação, na qual os modelos e datasets possam ser gerados de forma padronizada. A utilização de modelos para inferência em smartphones é algo cada vez mais difundido atualmente, mas o treinamento desses modelos na borda é algo ainda pouco explorado. Dessa forma, é importante a utilização de frameworks e métodos que permitam a embarcação do modelo, com a capacidade de executar um treinamento localmente no dispositivo de maneira otimizada.

Com isso em mente, é necessário também que o app, etapa intermediária, seja capaz de realizar o download dos arquivos gerados, e desserializá-los de forma que os dados sejam interpretáveis e manipuláveis dentro da aplicação. Além disso, a interface da aplicação deve ser intuitiva para o usuário final, permitindo a realização do fluxo de execução de maneira clara e fluida. Ela deve facilitar a obtenção das métricas obtidas, garantindo que possam ser acessadas e visualizadas remotamente. Com esses passos bem estabelecidos, é essencial que uma etapa posterior seja fornecida, na qual simulações de aprendizado federado sejam realizadas com os dados obtidos previamente.

Todas essas etapas, com seus respectivos requisitos, englobaram os principais desafios enfrentados no desenvolvimento do projeto. Para abordá-los, diversas soluções foram implementadas, sendo mais detalhadas nas seções seguintes deste relatório. De maneira geral, buscou-se fazer com que a aplicação Android fosse projetada com uma interface intuitiva e configurável, a partir de um sistema de *cards* sequenciais, permitindo a seleção de modelos e datasets personalizados, bem como a parametrização do treinamento. A coleta de métricas foi viabilizada utilizando recursos disponíveis no Android, como o BatteryManager, complementados por técnicas indiretas para estimar energia consumida em dispositivos onde APIs avançadas não estavam disponíveis. Além disso, códigos de referência foram desenvolvidos em Python para gerar modelos no formato TensorFlow Lite e datasets binários, garantindo a padronização no formato e usabilidade com a aplicação Android. Para integrar os perfis em simulações de aprendizado federado, utilizou-se o framework Flower, possibilitando o uso realista dos dados coletados.

Na Figura 1, é apresentado um diagrama contendo o fluxo completo de uso dos artefatos produzidos neste projeto. Inclui desde a geração do modelo e serialização dos datasets, passando pelo treinamento on-device com o uso da aplicação e, por fim, realizando o uso dos dados gerados para a execução da simulação. Além disso, é explicitado os momentos em que os dados transitam entre a nuvem e os dispositivos: após a etapa de "pré-processamento", o modelo gerado e as *features* e *labels* em formato binário são enviadas para um repositório online (como Github ou Google Drive), a partir do qual seja possível gerar links de download para o arquivo. Além disso, após a etapa de treinamento no aplicativo, as métricas geradas são enviadas ao banco de dados remoto (Firebase) para futuro uso pela simulação.

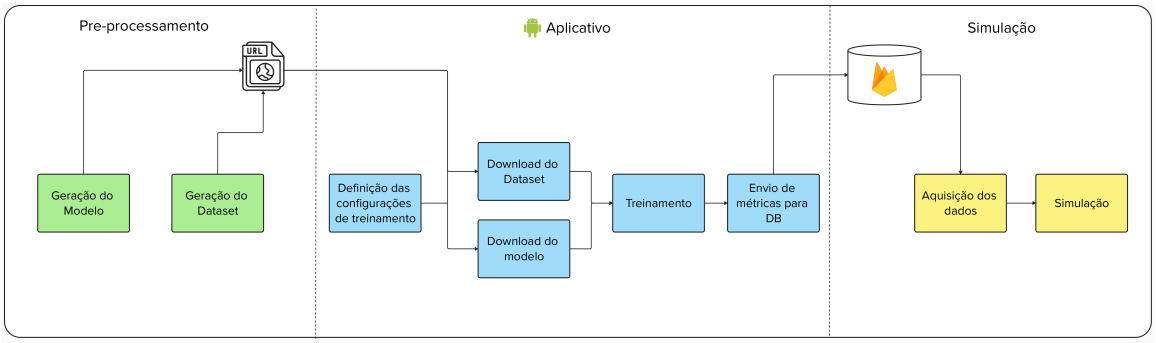


Figura 1: Diagrama de projeto da solução completo.

3 Geração de Modelo e Dataset

O treinamento on-device depende da geração prévia do modelo de aprendizado de máquina em um formato compatível e do fornecimento do dataset em um formato aceitável pela aplicação. A primeira etapa para que se possa fazer o uso da aplicação consiste justamente na geração tanto do modelo TFLite configurado para suportar aprendizado on-device quanto dos arquivos de *features* e *labels* do dataset, serializados em formato binário. Além disso, esses artefatos devem ser disponibilizados online para que um link de download possa ser fornecido à aplicação Android. A seguir, estão detalhadas as etapas de geração do modelo, criação do dataset e sua disponibilização remota.

3.1 Geração de Modelo

O modelo utilizado no treinamento on-device foi desenvolvido utilizando o framework TensorFlow, explorando majoritariamente arquiteturas leves e otimizadas para dispositivos móveis. Após a definição da arquitetura e do treinamento inicial em um ambiente desktop, o modelo foi convertido para o formato TFLite (TensorFlow Lite), que permite execução eficiente em hardware limitado. O modelo gerado é configurado com suporte para ajuste de pesos (treinamento incremental), utilizando APIs específicas da biblioteca TensorFlow Lite.

Para executar o treinamento on-device, é utilizado o interpretador de modelos TFLite fornecido pela biblioteca do Tensorflow, disponível para Java Android. O interpretador deve ser capaz de chamar o método de treinamento do modelo, o qual deve ser exposto como uma *signature* na conversão do modelo Keras para TFLite. As *signatures* são definidas por um *wrapper* (decorador) de funções da própria biblioteca do Tensorflow (`@tf.function`). Esse procedimento está exemplificado no código de referência disponível em [1], no repositório "OnDeviceTrainingToolkit", mais especificamente no arquivo *model_definition.py* de qualquer um dos dois exemplos presentes no diretório *model_generation*, além de também estar explicitado no tutorial fornecido em [2]. A função de treino recebe dois parâmetros: um batch de *features* e suas *labels* correspondentes, realizando, com isso, um passo de atualização dos pesos.

Uma questão importante a se prestar atenção se refere à definição de um *batch size* fixo na construção do modelo Keras. Nos testes realizados, notou-se que o interpretador de modelos TFLite utilizado no aplicativo não é capaz de processar dados quando o *batch size* é variável. Portanto, para que o modelo TFLite seja gerado esperando uma entrada com um *batch size* fixo pré-determinado e, consequentemente, opere corretamente na aplicação, é necessário que a definição do modelo Keras seja feita especificando-se o parâmetro de *batch size*.

Os arquivos utilizados para definição e treinamento do modelo Keras e sua conversão para TFLite estão disponíveis em [1], mais especificamente, no diretório *model_generation* do repositório *OnDeviceTrainingToolkit*. Essa implementação utilizou como base o exemplo fornecido em [2].

3.2 Geração de Dataset

O dataset gerado deve ser fragmentado em dois arquivos finais para ser utilizado na aplicação: um para as *features* e outro para as *labels*. Para garantir compatibilidade com a aplicação Android, que realiza o carregamento dos dados diretamente na memória durante a execução, é necessário que os dados sejam serializados como arquivos binários contendo apenas as *features* ou *labels*. Isto é, não deve haver nenhum tipo de header, apenas os dados organizados sequencialmente em formato binário. Por exemplo, para 100 amostras de *features* com dimensões 20 por 20 representadas em floats de 32 bits, será gerado um arquivo de exatamente $100 \cdot 20 \cdot 20 \cdot 32 = 1.280.000$ bits.

Foi utilizada a biblioteca *Numpy* para realizar a conversão para binário. Primeiramente, um subconjunto do dataset completo foi adicionado a um *Numpy array*, contendo então apenas os dados que seriam utilizados para treinamento no dispositivo. Em seguida, com o uso do método `to_bin`, foi possível salvar as *features* e *labels* em arquivos `.bin`, que poderão ser baixados pelo aplicativo e utilizados para o treinamento on-device.

Há também a alternativa de salvar apenas uma única *feature* para ser usada no aplicativo. Como o intuito da aplicação é de obter o perfil de consumo energético do dispositivo durante o treinamento on-device e não necessariamente obter um modelo com bom desempenho, então é viável utilizar uma mesma *feature* replicada para realizar o treinamento. Conforme melhor detalhado na seção 4.2, na aplicação, há a opção de se usar uma única *feature*. Nesse caso, a *feature* fornecida é replicada por um total de $batch\_size \cdot num\_batches$, não sendo necessário passar à aplicação todo o dataset.

3.3 Acesso remoto

Para que a aplicação pudesse acessar tanto o modelo TFLite, quanto o dataset para a realização do treinamento, foi feito o upload desses dados online, de forma que a aplicação pudesse fazer seu download.

Assim, com fornecimento do *link* de download, a aplicação é capaz de carregar em memória os dados necessários para o treinamento. Foi testado o Github e Google Drive como repositórios de armazenamentos de dados. Vale ressaltar que é necessário fornecer o link de download dos dados brutos.

4 Aplicação Android

Contextualizada a forma como os modelos e *datasets* são gerados e compartilhados, pode-se explicar como o aplicativo de treinamento on-device utiliza esses recursos. Assim, Esta seção foca em descrever de forma aprofundada o corpo geral da aplicação *Android* desenvolvida, desde sua interface e funcionalidades até os desafios encontrados em seu desenvolvimento.

Foi escolhido o desenvolvimento para a plataforma *Android* para tornar a aplicação o mais acessível possível. Isso pois dispositivos de diversas marcas utilizam deste sistema operacional, garantindo, assim, um maior leque de possibilidades para os usuários. Além disso, o ambiente *Android* fornece maior abertura no que se trata de funcionalidades e compatibilidades disponíveis, quando comparado ao iOS.

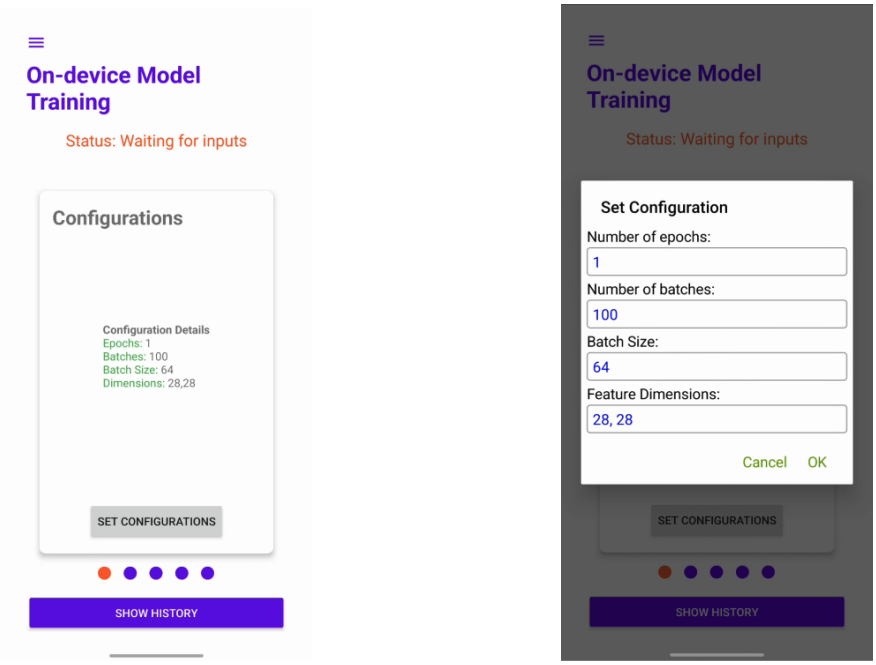
4.1 O Propósito

Como descrito de forma breve anteriormente, o objetivo da aplicação é servir como uma ferramenta que permite aos usuários fazerem o download de seus modelos e *datasets* remotos, no formato descrito na seção 3, realizarem o treinamento de acordo com as configurações desejadas além de visualizar e extrair as métricas de consumo energético relacionadas a cada um dos procedimentos realizados.

4.2 Interface de usuário

A interface do *App* foi desenvolvida em um sistema de cards. Cada card representa uma etapa do processo. Em cada um dos cards foram colocados valores padrões para auxiliar o usuário no preenchimento dos campos.

A primeira etapa é onde as configurações para o treinamento são selecionadas. O card *Configurations* é o responsável por essa etapa. A página com o card pode ser visualizada na Figura 2a. As configurações escolhidas são exibidas constantemente no card.



(a) (b)

Figura 2: **(a)** A primeira etapa no fluxo de uso da aplicação: o card de configurações do treinamento. **(b)** Popup de input para as configurações de treinamento, exibido após clicar-se em *SET CONFIGURATIONS* no card de configurações.

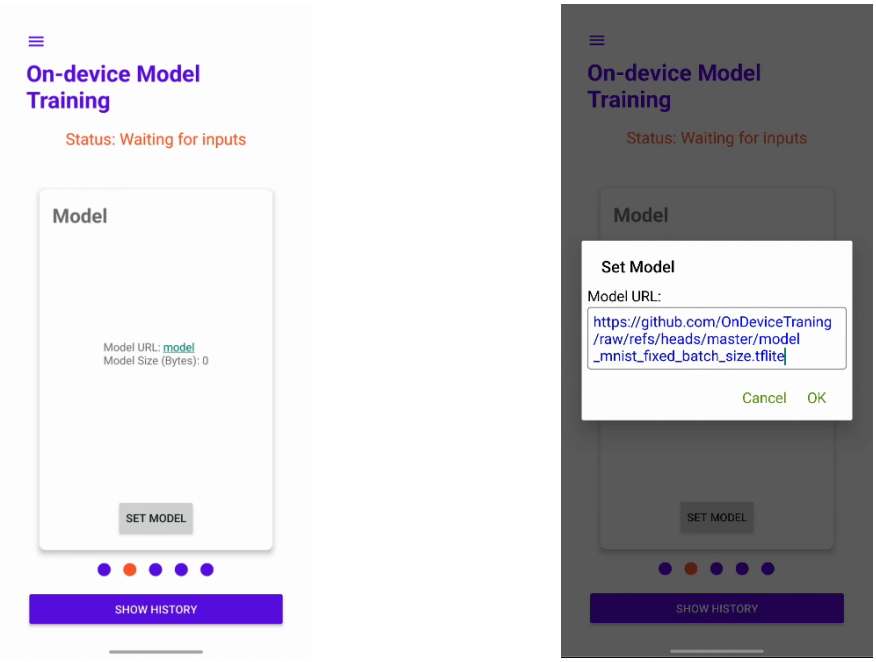
Ao clicar no botão *SET CONFIGURATIONS* um *pop-up* é exibido com campos a serem preenchidos, como pode ser observado na Figura 2b. Os valores atribuídos em cada um dos campos serão utilizados como parâmetros para o treinamento. Como pode ser visto na figura, é possível configurar o número de épocas, número de batches, o tamanho de cada batch, e as dimensões dos dados. Todos os campos, com exceção das dimensões dos dados, aceitam apenas números naturais. As dimensões dos dados são números naturais separados por vírgula. Para exemplificar, se cada dado for uma imagem 28x28 com três canais, então o campo deve ser preenchido com 28,28,3. Não existe limite no número de dimensões.

É importante notar que a **multiplicação entre o batch size e o batch number deve resultar no número total de amostras**. Caso contrário, não será possível realizar o treinamento posteriormente.

Ao clicar em *OK*, esses valores são salvos, podendo ser checados no próprio card.

Para passar para a próxima etapa, o usuário deve deslizar para a esquerda. Caso seja necessário alterar algo feito anteriormente, basta deslizar para a direita. Para facilitar a navegação, elementos visuais circulares foram inseridos abaixo dos cards, exibindo em laranja a etapa atual e as já concluídas.

O próximo card, que pode ser visualizado na Figura 3a, é o encarregado de fazer o download do modelo.



(a) (b)

Figura 3: **(a)** Card de download do modelo, exibindo seu link e tamanho em *bytes* **(b)** Popup de input para o link de download do modelo, exibido após clicar-se em *SET MODEL* no card "Model".

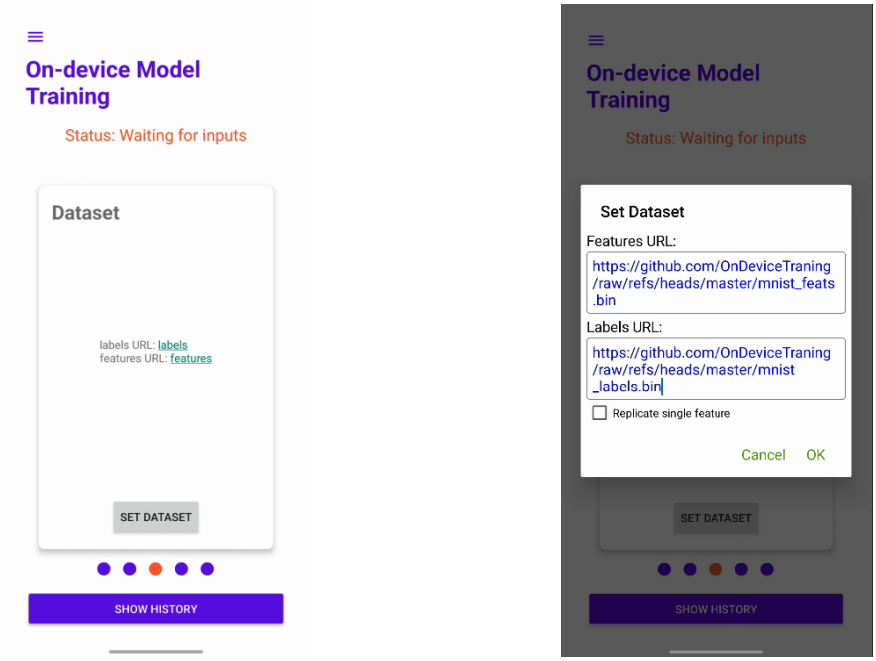
Ao clicar no botão *SET MODEL*, outro *popup* para input surge (Figura 3b). Nele, é possível colocar a URL para o modelo *TFLite* que será utilizado no treinamento. Clicando em *OK*, a aplicação dará início ao download do modelo. Ele é armazenado em um arquivo temporário, e passará para a memória principal apenas no momento do treinamento. Tais detalhes serão descritos de forma apropriada na próxima seção. O progresso do download é exibido em uma barra de progresso acima do card. O download terá terminado quando a mensagem *Model Downloaded* for exibida.

O card possui um elemento clicável, em verde, que permite acessar o link fornecido pelo usuário. No caso, como se trata de um link de download de arquivo, o modelo será baixado no armazenamento persistente do dispositivo, não tendo efeito no fluxo de uso da aplicação. No mesmo local da mensagem *Model Downloaded*, é possível visualizar o tempo que levou para fazer o download do modelo e o seu tamanho total. Tais medidas são importantes pois podem ser utilizadas para calcular o gasto energético atrelado a transferência dos bytes do modelo, e posteriormente do dataset, através da rede.

Como a mensagem acima do card desaparece após outro processo ser iniciado (como o download das features e labels, ou o treinamento em si) o tamanho do modelo é exibido de forma permanente no card, até que seja feito o download de um modelo com tamanho distinto.

Deslizando para a próxima etapa, encontra-se o card cujo objetivo é realizar o download

das features e labels a serem utilizadas no treinamento do modelo. Ele pode ser examinado na Figura 4a.



(a) (b)

Figura 4: **(a)** Card "Dataset", de download das *features* e *labels*. **(b)** Popup de input para para os links de download das *features* e *labels*, exibido após clicar-se em *SET DATASET* no card "Dataset".

Clicando no botão *SET DATASET*, é possível atribuir as URLs para os binários das features e dos labels, como observa-se na Figura 4b. Em especial, nesse *popup* existe um checkbox *Replicate single feature*. Esse checkbox serve para o caso em que uma única feature está presente no objeto da URL. Assim, as features e os labels serão replicados para um total de $batchsize \cdot batchnumber$ amostras, meras cópias da amostra original, que serão utilizadas por completo no treinamento. Clicando em *OK*, caso o checkbox esteja selecionado, será feito o download da feature e do label e, em seguida, esses serão replicados. Caso não esteja selecionado, o download será feito normalmente.

Assim como a etapa de download do modelo, essa etapa possui barra de progresso do download, tempo total de download, e objetos clicáveis, em verde, para baixar as features ou labels no armazenamento do dispositivo. Diferentemente do modelo, que é armazenado em um arquivo temporário até o momento do treinamento, as feautres e labels são armazenados em buffers diretamente na memória principal.

A próxima etapa consiste no treinamento. O card Training é o responsável por esse processo, que pode ser visto na Figura 5. Nele, o botão *START TRAINING* dará início ao trainmento. Quando completado, a mensagem vermelha de status acima será atualizada para refletir tal estado, exibindo também o tempo total do treinamento, em milissegundos.

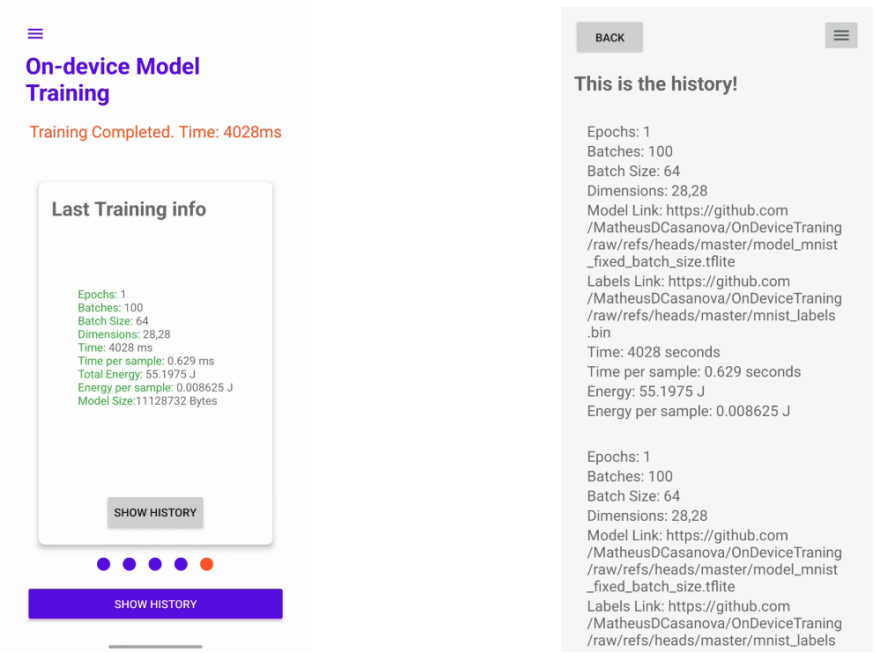
Só é possível iniciar com o modelo, features, e labels, baixados, caso contrário, a mensagem de status irá avisar sobre essa necessidade. Ao fim do treinamento, as métricas são enviadas diretamente para o banco de dados Firebase [3] configurado no build da aplicação. No README [4] do repositório do App está descrito como fazer o build da aplicação utilizando uma configuração própria de banco de dados Firebase.

Figura 5: Card "Training", responsável pelo início da execução do treinamento após clicar-se em "START TRAINING".

Concluído o treinamento, o card seguinte, Last Training Info (Figura 6a), fornece as métricas obtidas no procedimento. Além das configurações atribuídas pelo usuário, exibe também o tempo total de treinamento, o tempo de treinamento por amostra, energia total gasta, energia gasta por amostra, e o tamanho do modelo treinado.

Como foi possível observar desde o início da navegação no aplicativo, na seção inferior está presente o botão *SHOW HISTORY*, em roxo. Ao clicar nesse botão, a tela de History é exibida. A Figura 6b mostra como as métricas para cada um dos treinamentos já realizados no App são exibidas, da mais nova à mais antiga.

O botão no canto superior direito permite ao usuário escolher entre limpar o histórico ou compartilhá-lo. O histório é compartilhado no formato *json*. A aplicação utiliza da ferramenta padrão do Android para compartilhamento, permitindo, então, o envio do arquivo *json* para diversas aplicações. O botão *BACK* no canto superior esquerdo direciona ao início do App.



(a) (b)

Figura 6: **(a)** Card que exibe as métricas exibidas no último treinamento realizado na aplicação. **(b)** Página do histórico contendo as métricas de todos os treinamentos realizados no aplicativo deste dispositivo.

4.3 Detalhes e desafios de implementação

Esta seção descreve os principais detalhes técnicos e desafios enfrentados na implementação dos componentes mais importantes da aplicação Android, destacando as etapas críticas no desenvolvimento e a solução de problemas encontrados. O código foi implementado em Java 8 [5]. As dependências do código podem ser encontradas no arquivo *gradle* no repositório da aplicação [6].

4.3.1 Frontend

A interface foi desenvolvida utilizando o Material como visual base [7]. Foram criados arquivos XML que fazem as definições das interfaces, fornecendo referências que permitem o acesso aos componentes visuais definidos no código fonte Java. As mudanças de estado, animações, e fluxo, são manipulados de acordo com o comportamento definido em diferentes componentes e métodos no código fonte.

4.3.2 Download

Esse componente é o responsável por fazer o download dos artefatos atrelados a cada uma das URLs fornecidos como input pelo usuário na aplicação. O download é feito utilizando

o protocolo HTTP e, para o funcionamento das barras de progresso, os bytes que vão chegando são tratados como uma stream de dados.

No caso do modelo, os bytes são armazenados, conforme chegam, em um arquivo temporário (isto é, que irá durar enquanto a aplicação estiver rodando. Quando for necessário o modelo, esse arquivo é lido pelo interpretador do TFLite.

Quando se trata do dataset (features e labels), os dados vão sendo armazenados em buffers, na memória principal, de acordo com a chegada. Caso a funcionalidade *Replicate single feature* seja habilitada, é nesse componente que serão geradas as réplicas, de tal forma que a geração de cópias se torna transparente para os demais componentes.

4.3.3 Treinamento

O treinamento on-device foi feito de maneira genérica, para acomodar diferentes modelos e datasets. A estrutura inicial do código foi desenvolvida com base no texto "On-Device Training with LiteRT" [2].

A generalização do treinamento se dá principalmente pela parametrização das configurações de execução, permitindo ao usuário alterar os valores de batch size, número de batches, número de épocas e dimensões de features, as quais podem variar, não só nos valores de cada dimensão, mas também no número de dimensões. Além disso, outro aspecto chave foi a padronização na geração do modelo TFLite e na serialização das *features* e *labels*.

O modelo, gerado conforme descrito na seção 3.1, contém as definições das *signatures* necessárias para rodar o treinamento quando o arquivo TFLite é carregado pelo interpretador. As *features* e *labels*, em formato binário e cuja geração está detalhada na seção 3.2, podem ser carregadas diretamente na memória do dispositivo e iteradas sobre, percorrendo os bytes conforme as dimensionalidades definidas nas configurações.

4.3.4 Métricas

As métricas são geradas através de uma comparação entre o estado do dispositivo antes e depois do treinamento. O estado consiste no tempo e na carga da bateria registrados pelo smartphone. Para calcular o tempo total de treinamento, trivialmente basta subtrair do tempo pós o tempo pré-treinamento.

O cálculo da energia gasta já não é tão trivial e requer uma importante suposição. A primeira suposição é que toda a energia que o dispositivo gastou entre o início e o fim do processo foi totalmente dedicada ao treinamento. Como se sabe, a realidade é que o dispositivo possui diversos processos rodando em plano de fundo que, embora possa ser irrisório dependendo do caso, consumem energia. Assim, considera-se o caso em que não há processos de fundo que consumam energia considerável quando comparada com o gasto do treinamento.

A carga da bateria fornecida pela API do dispositivo utiliza como unidade microAmpere-hora. Dessa forma, é necessário utilizar a API do dispositivo para extrair também a voltagem do dispositivo durante o treinamento para que seja possível calcular a energia gasta em Joules. A voltagem é fornecida em millivolts. Convertendo a carga e a voltagem para

as unidades internacionais de medida, é possível calcular a energia em Joules através da relação:

$$E = Q \cdot V \tag{1}$$

4.3.5 Banco de dados

O banco de dados foi configurado para a aplicação Android utilizando d serviço Real Time Database [8], fornecido pelo Firebase. Este serviço consiste em um banco de dados NoSQL na nuvem, no qual os dados são armazenados como JSON e sincronizados em tempo real para cada cliente conectado.

As métricas de treinamento são enviadas para o banco de dados configurado logo após a finalização de um treinamento. A Figura 7 mostra os dados de alguns treinamentos armazenados no banco de dados. Os dados são advindos de diferentes dispositivos que realizaram o treinamento através da aplicação.

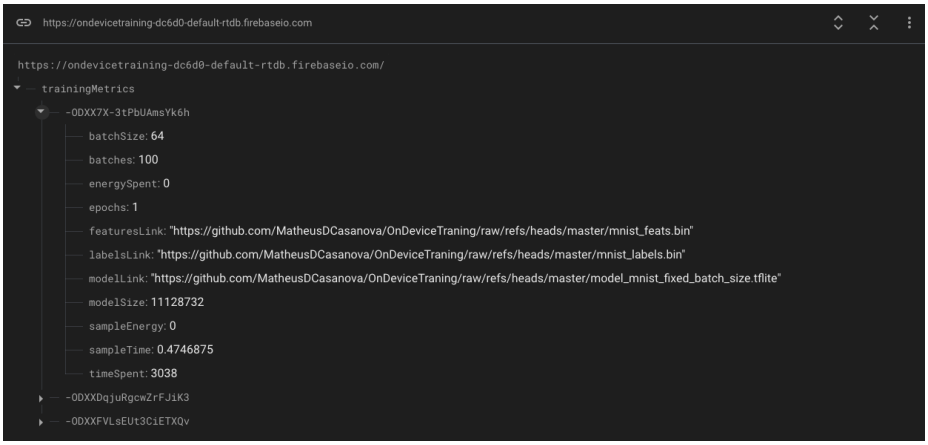


Figura 7: Interface do Real Time Database, no qual se observa dados armazenados advindos de diferentes clientes.

No momento, o banco de dados a ser utilizado pela aplicação é configurado no build da aplicação. Para estabelecer a conexão com o próprio banco de dados, é possível seguir os passos descritos no README do repositório do App, On-Device-Training [4]. Esse processo envolverá realizar um build próprio da aplicação.

Tendo em vista a atual complexidade envolvida no processo de conexão com um banco de dados próprio, surge a proposta de melhoria na aplicação. Deseja-se que, no futuro, seja possível configurar uma conexão com um novo banco através da própria interface de usuário, sem a necessidade de clonar, alterar o código fonte com a nova configuração, e realizar um novo build da aplicação.

4.4 Fluxo de uso da aplicação

Na Figura 8, é possível observar um fluxograma, sumarizando todo o processo descrito anteriormente de uso da aplicação, partindo da página inicial, ilustrando até a finalização do treinamento e exportação do histórico. Vale notar que a inserção no banco de dados não está presente no fluxograma, pois elas são transparentes ao usuário, não pertencendo ativamente ao fluxo de uso em si.

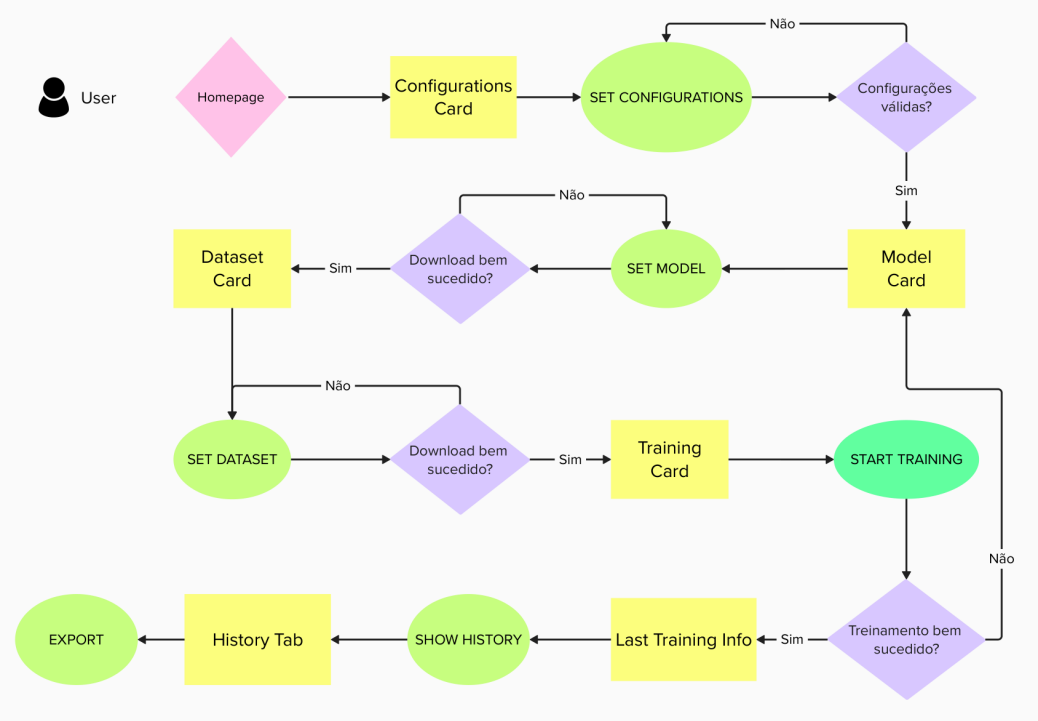


Figura 8: Fluxograma de uso da aplicação.

5 Simulação do Aprendizado Federado

Conforme descrito anteriormente, o aplicativo desenvolvido integra-se a uma instância específica de um banco de dados do *Firebase*, permitindo o armazenamento dos dados de cada execução realizada pelos dispositivos conectados. Esses dados podem ser utilizados posteriormente para simular o comportamento de um sistema distribuído de aprendizado federado, avaliando métricas como o consumo acumulado de energia e o desempenho do modelo federado.

Os usuários do aplicativo são livres para realizar suas próprias simulações da maneira como julgarem mais conveniente para o seu propósito. No entanto, este projeto também disponibiliza um repositório de referência que serve como exemplo prático para a simulação, chamado *OnDeviceTrainingToolkit*, disponível em [1], mais especificamente, no diretório *flower-simulation*. Essa implementação será utilizada como parte dos experimentos apre-

sentados na seção 6 deste relatório.

O código que implementa essa simulação foi escrito em Python, utilizando como bibliotecas principais *flower* [9], *hydra* [10] e *firebase-admin* [11]. Assim, os próximos itens desta seção têm como objetivo detalhar como essas bibliotecas são aplicadas para implementar os diferentes componentes do pacote.

5.1 Configuração da Simulação

Para a configuração dinâmica de parâmetros, o usuário define as características de cada rodada em um arquivo *YAML*, chamado *config.yaml*.

A imagem da figura 9 apresenta um exemplo de configuração para uma simulação. Nesse exemplo, o modelo é especificado pelo link fornecido no parâmetro *model_use*, enquanto o dataset é dividido em 10 partições, conforme definido pelo parâmetro *partitions*. O número de rounds e mínimo numero de clientes por round também são definidos. Essa configuração permite ao usuário ajustar tanto o modelo quanto os dados utilizados de forma flexível, facilitando a realização de diferentes experimentos de aprendizado de máquina distribuído.

```
1   firebase_database_url: https://ondevicetraining-dc6d0-default-rtdb.firebaseio.com/
2   model_use: https://github.com/MatheusDCasanova/OnDeviceTraning/raw/refs/heads/master/model_mnist_fixed_batch_size.tflite
3   device_config_path: /home/tobias/flower-simulation/deviceConfigurations.json
4   simulation_result_path: /home/tobias/flower-simulation/simulation-result.csv
5
6   partitions: 300
7   min_client_per_round: 2
8   num_rounds: 3
9
0   backend_config:
1     client_resources:
2       num_cpus: 8
3   gpu_backend_config:
4     client_resources:
5       num_gpus: 1
```

Figura 9: Exemplo de arquivo *config.yaml*

Esse arquivo de configuração é empregado em dois momentos distintos durante a execução do sistema. Primeiramente, ele é processado pela biblioteca *yaml* da coleção do *Python* para a aquisição dos dados provenientes do aplicativo. Em seguida, o arquivo é carregado pela biblioteca *hydra*, que utiliza as informações para configurar e executar a simulação. Os detalhes de cada um desses momentos serão explorados nos itens seguintes.

5.2 Aquisição dos Dados do Aplicativo

Para que a simulação possa utilizar os dados coletados pela aplicação, um *script* em *python* chamado *firebase_retriever.py* foi criado.

Nesse código, a biblioteca *firebase-admin* se conecta à instancia do *firebase* de interesse a partir de um certificado de acesso da instância que deve ser adicionado ao repositório local no caminho *secrets/service-account.json*. Para obter o certificado para a conta de interesse, um tutorial pode ser encontrado na referência [12].

Após a conexão, o *script* busca as informações da base de dados da instância definida pelo link no arquivo de configuração. Faz-se, então, um filtro dos dados para que sejam isoladas

as informações das execuções em dispositivos nos quais foi utilizado o modelo da URL definida pelo parâmetro de configuração *model_use*. Por fim, esses dados são salvos para um arquivo *JSON* de nome definido nas configurações a partir do campo *device_config_path*.

5.3 Implementação da Simulação com *Flower.ai*

Agora, após terem sido definidas as configurações e obtidos os perfis de execução dos dispositivos de interesse, pode-se, finalmente, simular como um sistema de aprendizado federado em larga escala se comportaria em um cenário mais próximo da realidade.

Nessa etapa, o *framework* de aprendizado federado *flower.ai* é utilizado. O número de clientes é definido pelo valor de partições definidos no arquivo de configurações e as características de cada um desses clientes, como número de épocas a se treinar, gasto de energia por amostra, tempo de treino por amostra, entre outros, são definidos a partir da escolha aleatória do resultado de uma das execuções coletadas pelo aplicativo.

O servidor, por sua vez, fica responsável pela agregação dos pesos do modelo, a partir do uso da estratégia FedAvg [13]. Porém, ainda mais importante para esse projeto, o servidor é responsável por guardar as métricas de energia total consumida por cada cliente em cada rodada. Dessa forma, ao fim da execução, um arquivo (de nome definido pelo campo *simulation_result_path* na configuração) é criado com todos esses dados, através de uma tabela com as informações de *round*, *client_id*, *energy*, *time*, *epochs*. Isso permite calcular o gasto total de energia do sistema, mas também permite análises diferentes como custo por *round* ou em relação ao tempo de treinamento. Além disso, os *logs* da simulação são enviados para o subdiretório *outputs*, que pode ser utilizado para visualizar como as métricas de performance do modelo evoluíram junto com as métricas de gasto de energia e tempo de treinamento.

Mais detalhes de como esses *scripts* podem ser executados e também modificados para usos alternativos podem ser vistos no *README* do repositório.

6 Experimentos

Os experimentos realizados neste trabalho têm como objetivo avaliar o impacto energético e de desempenho do treinamento embarcado de modelos de aprendizado de máquina em dispositivos móveis, utilizando diferentes *datasets* e configurações. Para isso, adotou-se um fluxo experimental comum, seguido pela análise específica de cada *dataset*, incluindo o MNIST e o CIFAR-10.

6.1 Fluxo geral dos experimentos

O fluxo experimental seguiu as seguintes etapas principais:

1. **Preparação dos modelos e *datasets***: Os modelos foram convertidos para o formato TensorFlow Lite, e os *datasets* foram processados em um formato binário otimizado para carregamento em dispositivos móveis. O código específico dessa etapa pode ser encontrado no repositório *OnDeviceTrainingToolkit*, disponível na organização referenciada em [1]. Mais especificamente, no diretório *model_generation*. Abaixo desse,

há diretórios de exemplos de código para cada experimento executado. Mais especificamente, *example-mnist* e *example-cifar10*, que contém os arquivos para gerar os modelos, *features* e *labels* para os experimentos executados com o dataset MNIST e CIFAR-10, respectivamente. Esses exemplos contém os mesmos quatro arquivos: *dataset_definition.py*, *model_definition.py*, *constants.py* e *main.py*.

No arquivo *dataset_definition.py*, é feito o carregamento e pré-processamento do dataset, por exemplo, normalizando as imagens e constituindo o *tf.data.Dataset* que será utilizado no treinamento local, já com os batches formados. A principal função desse arquivo, que será chamada em outros momentos nesse passo inicial é a *get_processed_ds()*.

Já no arquivo *model_definition.py*, há a definição da classe *MyModel*, que contém tanto o *backbone* do modelo, isto é, a sequência de camadas e operações em um *forward pass* do modelo, quanto a definição das funções que serão convertidas como *signatures* do modelo TFLite e poderão ser chamadas pelo interpretador no aplicativo. O *backbone* é definido na função *_get_model()*, ressaltando que, nessa etapa, deve ser fornecido o batch size fixo da camada de entrada do modelo. As funções definidas como *signatures* foram: *train* (responsável por um passo de atualização dos pesos com uso de um batch), *infer* (roda uma inferência do modelo, devolvendo tanto os logits, quanto as probabilidades para cada classe), *save* (responsável por salvar os pesos do modelo) e *restore* (responsável por designar novos valores aos pesos do modelo, com base em valores salvos anteriormente).

No arquivo *main.py*, o treinamento local é executado, utilizando o dataset e modelos definidos anteriormente nos outros arquivos, além das configurações determinadas no arquivo *constants.py*. Após o treinamento local, os dados de treinamento são salvos em arquivos binários, conforme descrito na seção 3.2 e o modelo é convertido para o formato TFLite e salvo no arquivo correspondente, conforme descrito em 3.1. Com os arquivos das features, labels e do modelo TFLite em mãos, o próximo passo foi realizar o upload desses para algum espaço online que permitisse download, conforme descrito na seção 3.3.

2. **Configuração e treinamento on-device**: Primeiramente, foi feita a configuração dos parâmetros de treinamento, como tamanho do *batch*, número de épocas e número de *batches*. Em seguida, deve-se obter as URLs de download das *features* e *labels*, assim como do modelo TFLite, para serem carregadas no app, conforme melhor descrito na seção 4.2. No caso de upload para o Github, os links possuíam um formato semelhante ao seguinte: *https://github.com/PFG-Federated-Learning/OnDeviceTraning/raw/refs/heads/master/model_mnist_fixed_batch_size.tflite*. Com os links em mãos, os cards de modelo e dataset podem ser preenchidos com as respectivas URLs de download. O quarto card é onde ocorre de fato o treinamento do modelo, ao clicar-se no botão de "START TRAINING".

3. **Coleta de métricas**: Durante o treinamento, foram coletadas métricas detalhadas, incluindo consumo de energia (total e por amostra), tempo de execução e tamanho do modelo. O histórico dos últimos treinamentos pode ser visto no último card, contendo essas informações. Essas métricas também são enviadas para o banco de

dados remoto do Firebase, a partir do qual os dados podem ser recuperados para a execução da simulação.

4. **Simulação de aprendizado federado**: Com base nos perfis coletados, foram realizadas simulações de aprendizado federado utilizando o framework Flower, avaliando o impacto do treinamento distribuído em diferentes cenários.

As subseções a seguir detalham os experimentos específicos realizados com os *datasets* MNIST e CIFAR-10, incluindo suas características, configurações e os principais resultados obtidos.

6.2 Experimento com o *dataset* MNIST

6.2.1 Descrição do *dataset*

O *dataset* MNIST contém 60.000 imagens de treino e 10.000 imagens de teste, cada uma representando dígitos manuscritos em escala de cinza, com dimensões de 28×28 pixels. Esse *dataset* é amplamente utilizado como *benchmark* para modelos de aprendizado de máquina devido à sua simplicidade e representatividade. O MNIST foi baixado da biblioteca *tensorflow-datasets* (conforme o exemplo referenciado em [14]). Das 60 mil amostras de treinamento, foram utilizadas 6400.

6.2.2 Configurações do experimento

Para o experimento com o MNIST, foi utilizado um modelo de rede neural convolucional simples de camadas convolucionais e densas. Os principais parâmetros de treinamento foram:

- **Tamanho do *batch***: 64
- **Número de *batches***: 100
- **Número de épocas**: Variável, entre 5 e 10.
- **Dimensões das *features***: 28x28

6.2.3 Resultados obtidos

Para esse modelo em específico foram coletadas métricas em 3 dispositivos diferentes. A Tabela 1 mostra a quantidade de épocas treinadas em cada dispositivo e quanto foi o gasto médio de energia e tempo de treinamento por amostra.

Utilizando dos perfis energéticos dos dispositivos da Tabela 1, foi feita uma simulação de aprendizado federado com três configurações distintas. Os resultados da simulação foram dispostos na Tabela 2.

Focando em apenas uma das configurações da Tabela 2, mais especificamente na que possui 20 clientes por round e 100 clientes totais, foi possível gerar um gráfico relacionando a energia total acumulada com a loss ao longo das 5 rodadas. Ele pode ser visto na Figura 10.

Dispositivo	Épocas	Energia/Amostra (J)	Tempo/Amostra (ms)
1	6	0,023	8,327
2	7	0,010	2,483
3	6	0,010	4,617

Tabela 1: Resultados para os treinamentos do MNIST em dispositivo.

Model size (MB)	**Rounds**	**Clientes/Round**	**Clientes Totais**	**Energia Acumulada (J)**	**CO2 (g)**	**Loss Final**
11,13	5	10	50	767,1270235	0,008203963	0,07997134
11,13	5	10	100	791,5462555	0,008465112	0,103275775
11,13	5	20	100	1433,442095	0,015329803	0,086795797

Tabela 2: Resultados para simulação de aprendizado federado do MNIST.

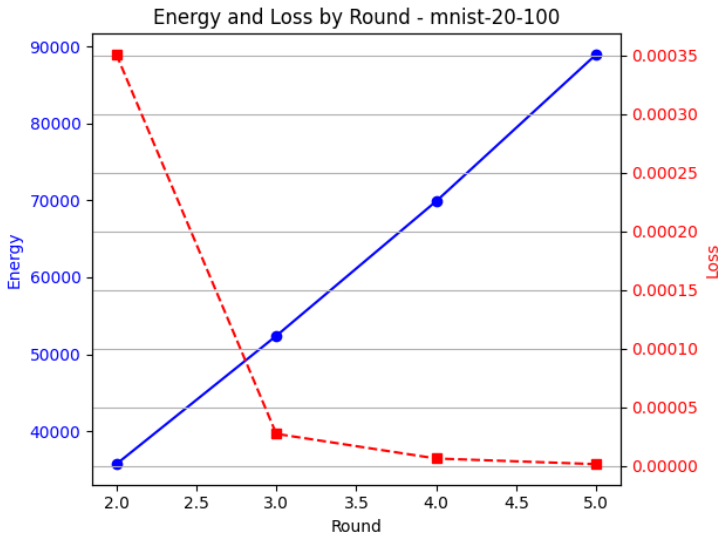


Figura 10: Gráfico com a relação energia total acumulada x loss ao longo das 5 rodadas de aprendizado federado no MNIST.

6.3 Experimento com o *dataset* CIFAR-10

6.3.1 Descrição do *dataset*

O *dataset* CIFAR-10 contém 60.000 imagens coloridas distribuídas em 10 classes, com 50.000 imagens para treino e 10.000 para teste. Dessas 50 mil amostras, foram utilizadas apenas 1024. Cada imagem possui dimensões de 32×32 pixels (com três canais - RGB), tornando este *dataset* mais desafiador devido à maior complexidade visual e dimensionalidade dos dados.

6.3.2 Configurações do experimento

Para o experimento com o CIFAR-10, foi utilizado um modelo de rede neural convolucional mais complexo, com um maior número de camadas e parâmetros, o qual seguiu a estrutura estabelecida no artigo referenciado em [17]. Os parâmetros de treinamento definidos foram:

- **Tamanho do *batch***: 32
- **Número de *batches***: 32
- **Número de épocas**: Variável, entre 5 e 10.
- **Dimensões das *features***: 32x32x3

6.3.3 Resultados obtidos

A tabela 3 abaixo apresenta os dados para os 3 diferentes dispositivos que coletaram métricas utilizando o aplicativo com esse modelo.

Dispositivo	Épocas	Energia/Amostra (J)	Tempo/Amostra (ms)
1	8	0,631	108.109
2	7	0,119	45,624
3	5	0,288	97,354

Tabela 3: Resultados para os treinamentos do CIFAR10 em dispositivo.

Model size (MB)	**Rounds**	**Clientes/Round**	**Clientes Totais**	**Energia Acumulada (J)**	**CO2 (g)**	**Loss Final**
28,97	5	10	50	15701,61434	0,167913064	1,9807E-05
28,97	5	10	100	17273,13989	0,184718958	1,81782E-05
28,97	5	20	100	38319,05761	0,409784002	9,16808E-06

Tabela 4: Resultados para simulação de aprendizado federado do CIFAR-10.

Focando em apenas uma das configurações da Tabela 4, mais especificamente na que possui 20 clientes por round e 100 clientes totais, foi possível gerar um gráfico relacionando a energia total acumulada com a loss ao longo das 5 rodadas. Ele pode ser visto na Figura 11.

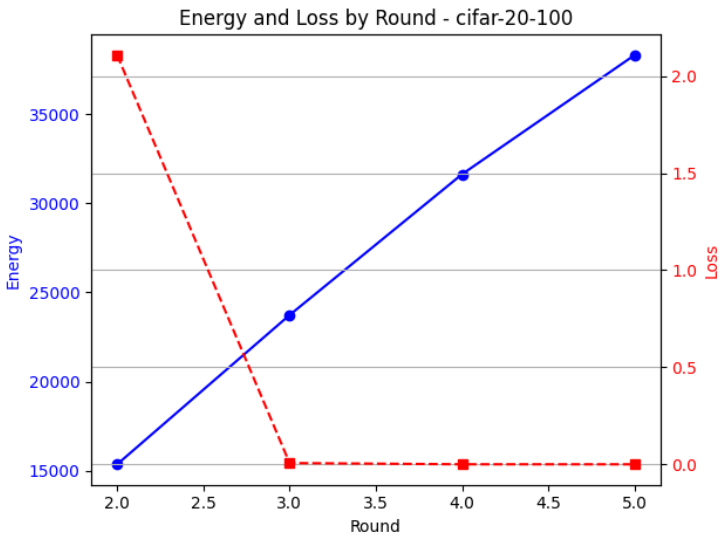


Figura 11: Gráfico com a relação energia total acumulada x loss ao longo das 5 rodadas de aprendizado federado no CIFAR-10.

6.4 Discussão dos resultados

A partir das tabelas 1 e 3, pode-se observar alguns padrões de consumo energético de cada dispositivo utilizado, para cada uma das configurações em que o experimento foi executado. Vale ressaltar que a energia e o tempo por amostra considera o total de vezes que essa amostra aparece no treinamento. Isto é, se o treinamento inclui 5 épocas, então a energia e o tempo por amostra contabilizam as 5 vezes que essa amostra foi vista no treino. Analisando as tabelas, constata-se que o consumo e tempo total do treinamento federado é totalmente vinculado ao gasto energético e poder computacional do dispositivo específico no qual o treinamento está sendo executado. Por exemplo, na tabela 3, vê-se que o dispositivo 3, mesmo com menos épocas, gastou mais do que o dobro de energia por amostra, além de também, levar mais do que o dobro do tempo para processar cada amostra.

As tabelas 1 e 3 também permitem realizar uma análise comparativa entre os dois experimentos. Ao observar-se a segunda linha de cada tabela, tem-se o mesmo dispositivo, executando o treinamento com mesmo número de épocas, distinguindo-se apenas no modelo e dataset utilizados. Com isso, é possível verificar um consumo energético por amostra, assim como tempo de processamento, consideravelmente maior no experimento CIFAR-10, quando comparado ao MNIST. Esse comportamento é esperado, dada a maior complexidade e dimensionalidade das *features* do CIFAR-10, além do uso de um modelo também maior.

O tamanho exato de cada modelo, em megabytes, pode ser observado nas tabelas 2 e 4. Pode-se ver que o modelo utilizado no experimento com o CIFAR-10 é mais de duas vezes maior que aquele utilizado no experimento com o MNIST, o que é diretamente refletido no consumo energético, sendo a energia acumulada, assim como o custo de carbono do treinamento do CIFAR-10 muito superior ao do MNIST. Vale notar que as características

das *features* do dataset também possuem grande influência nessas métricas. No caso, as amostras do CIFAR-10 são mais complexas que as do MNIST e, portanto, requerem maior esforço computacional para serem processadas.

Observando os gráficos de energia acumulada e loss em função das rodadas de aprendizado federado no MNIST e no CIFAR-10, Figuras 10 e 11, respectivamente, verifica-se que uma importante decisão de treinamento pode ser tomada. Essa decisão está atrelada ao custo benefício do treinamento, mais especificamente, ao aumento do gasto acumulado de energia em comparação com a diminuição da loss. Percebe-se, em ambas as Figuras, que, a partir da terceira rodada de treinamento, o gasto de energia mantém um crescimento muito mais acentuado quando comparado com o ritmo de decrescimento da loss. Isso é ainda mais claro no caso do CIFAR-10, Figura 11, em que a variação na loss se dá em casas decimais de impacto irrisório. Dessa forma, os gráficos permitem entender até que ponto o gasto de energia está implicando em melhoras significativas na performance do modelo.

Por fim, os demais usuários podem estender as análises realizadas neste relatório e adicionar outras próprias aos seus experimentos. Exemplo disso é uma comparação entre experimentos que tem um mesmo número de clientes por round. A partir das métricas coletadas nesse trabalho, pôde ser visto que, para experimentos com 10 clientes por round, o resultado obtido pelas métricas foi próximo. Porém, a tabela gerada como saída da simulação (conforme descrito em 5.3) permite a visualização do gasto de energia em cada cliente, possibilitando avaliar como uma variação no número de clientes totais pode dividir a carga de energia consumida em cada dispositivo.

7 Conclusão

Este projeto atingiu seus objetivos ao propor e implementar uma solução integrada para o treinamento on-device de modelos de aprendizado de máquina, a coleta de métricas de consumo energético e a análise de impacto ambiental em cenários simulados de aprendizado federado. A aplicação Android desenvolvida demonstrou ser eficiente e flexível, permitindo o treinamento de modelos personalizados a partir de datasets configuráveis e a extração de métricas relevantes para o contexto de aprendizado distribuído. Além disso, a integração com o banco de dados na nuvem facilitou o armazenamento e a sincronização dessas métricas, possibilitando seu uso em simulações subsequentes.

A simulação de aprendizado federado, conduzida com o uso do framework Flower, permitiu analisar os perfis de consumo energético obtidos e quantificar o impacto ambiental de maneira abrangente. Este aspecto do projeto é particularmente relevante, dado o aumento da utilização de dispositivos móveis em aplicações de aprendizado de máquina distribuído e a necessidade crescente de soluções sustentáveis na área de tecnologia.

A disponibilização dos artefatos, como códigos de referência para geração de modelos e datasets, bem como o exemplo prático de simulação, reforça o caráter replicável e colaborativo da solução desenvolvida. Além disso, o trabalho fornece ferramentas concretas para avaliar o gasto energético de um treinamento federado de maneira generalizada, atendendo a múltiplos cenários, com arquiteturas de modelo e datasets diferentes.

Como perspectivas futuras, destacam-se a possibilidade de aprimorar os mecanismos de

coleta de métricas energéticas no dispositivo móvel, tornar a integração com o banco de dados mais facilmente configurável e a expansão das simulações para incluir cenários mais complexos e diversificados. Também seria interessante explorar o impacto do aprendizado federado em dispositivos móveis de diferentes especificações e capacidades, ampliando a generalização dos resultados obtidos.

Em síntese, o projeto apresentou contribuições significativas tanto do ponto de vista técnico quanto de sustentabilidade, destacando-se como uma solução inovadora e replicável para a obtenção de métricas energéticas do aprendizado de máquina on-device e federado.

Referências

[1] `https://github.com/PFG-Federated-Learning`

[2] `https://ai.google.dev/edge/litert/models/ondevice_training`. (Acessado no segundo semestre de 2024).

[3] `https://firebase.google.com/`. (Acessado no segundo semestre de 2024).

[4] `https://github.com/PFG-Federated-Learning/OnDeviceTraning/tree/master?tab=readme-ov-file`. (Acessado no segundo semestre de 2024).

[5] `https://www.oracle.com/br/java/technologies/javase-jdk8-doc-downloads.html`. (Acessado no segundo semestre de 2024).

[6] `https://github.com/PFG-Federated-Learning/OnDeviceTraning/blob/master/app/build.gradle`. (Acessado no segundo semestre de 2024).

[7] `https://github.com/material-components/material-components-android/tree/release-1.5`. (Acessado no segundo semestre de 2024).

[8] `https://firebase.google.com/docs/database?hl=pt-br`. (Acessado no segundo semestre de 2024).

[9] `https://flower.ai/`. (Acessado no segundo semestre de 2024).

[10] `https://hydra.cc/docs/intro/`. (Acessado no segundo semestre de 2024).

[11] `https://firebase.google.com/docs/admin/setup#python_1`. (Acessado no segundo semestre de 2024).

[12] `https://firebase.google.com/docs/android/setup?hl=pt-br`. (Acessado no segundo semestre de 2024).

[13] `https://flower.ai/docs/framework/ref-api/flwr.server.strategy.FedAvg.html` (Acessado no segundo semestre de 2024).

[14] `https://www.tensorflow.org/datasets/keras_example`. (Acessado no segundo semestre de 2024).

[15] A first look into the carbon footprint of federated learning, Xinchi Qiu and Titouan Parcollet and Javier Fernandez-Marques and Pedro Porto Buarque de Gusmao and Yan Gao and Daniel J. Beutel and Taner Topal and Akhil Mathur and Nicholas D. Lane, 2023

[16] `https://www.gov.br/mcti/pt-br/acompanhe-o-mcti/cgcl/noticias/fator-de-emissao-de-co2-na-geracao-de-energia-eletrica-no-brasil-em-2023-e-o-menor-em-12-anos`. (Acessado no segundo semestre de 2024).

[17] `https://www.geeksforgeeks.org/cifar-10-image-classification-in-tensorflow/`. (Acessado no segundo semestre de 2024).