UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

BERNARDO ALEXANDRE SANTOS EMERY

# ATRAVÉS DA BRECHA:
# UM ESTUDO PRÁTICO DE ATAQUES CONTRA A PRIVACIDADE EM SISTEMAS DE APRENDIZADO FEDERADO

Ouro Preto
2026

BERNARDO ALEXANDRE SANTOS EMERY

# ATRAVÉS DA BRECHA:
# UM ESTUDO PRÁTICO DE ATAQUES CONTRA A PRIVACIDADE EM SISTEMAS DE APRENDIZADO FEDERADO

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Rodrigo Cesar Pedrosa Silva
Coorientador: Bacharel Ana Luiza Soares

Ouro Preto
2026

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

**FOLHA DE APROVAÇÃO**

**Bernardo Alexandre Santos Emery**

**Através da Brecha: Um Estudo Prático de Ataques ao Aprendizado Federado**

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 2 de Março de 2026

Membros da banca

Rodrigo Cesar Pedrosa Silva (Orientador) - Doutor - Universidade Federal de Ouro Preto
Ana Luiza Almeida Soares (Coorientadora) - Bacharel - Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Ouro Preto
Pedro Henrique Lopes Silva (Examinador) - Doutor - Universidade Federal de Ouro Preto
Arthur Negrão de Faria Martins da Costa (Examinador) - Bacharel - Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Ouro Preto

Rodrigo Cesar Pedrosa Silva, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 02/03/2026

Documento assinado eletronicamente por **Rodrigo Cesar Pedrosa Silva**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 03/03/2026, às 09:27, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1062477** e o código CRC **43FD67B4**.



R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Com todo amor do mundo, à Eldesia, que não está aqui, em carne, para presenciar essa conquista. Mas, que, onde quer que esteja, em espírito e memoria, presencie o fim dessa jornada.

# Agradecimentos

Agradeço a Deus — sob toda forma e definição —, pela possibilidade do mistério e maravilha da vida. “Desvenda os meus olhos, para que eu contemple as Maravilhas dos Teus ensinamentos”.

Agradeço minha família, por toda a paciência, e a vezes, a falta dela também: meus avós Eldesia e Vicente, meus pais Wladmir e Tatiana, meus tios Wilquenio e Wanalyse, e meus irmãos Dimitria, Pietra, Miguel, Bento e Catharine e também tia Luise e primo Sandro.

A Universidade Federal de Ouro Preto, pelo ensino publico, — reitero —, gratuito e de qualidade. Aos professores do DECOM, também pela paciência e compreenssão. Em especial: meu orientador Rodrigo Silva, com grande respeito e consideração, por todas as trocas e atenção. Amanda Nascimento, pela genialidade, conselhos e maestria na sala de aula. Tiago Garcia de Senna Carneiro, pela mentoria no TerraLab e a experiência prática que me permitiu tantas possibilidades. Fernando Cortez Sica, pelo projeto Arduineiros, e a possibilidade de levar a tecnologia para a molecada carente no São Sebastião. José Maria Ribeiro Neves, que não tive aulas com, mas travei contato no Centro Espirita do São Sebastião durante o projeto Arduineiros, e possibilita esse espaço de fé, cultura e aprendizado para os residentes do bairro. Elton José da Silva, pelo projeto literário e as baitas trocas, que você esteja bem e saudável! Guilherme Tavares de Assis, de longe um dos melhores professores que conheci. Também outros professores espetaculares: Jadson Castro, Marco Antonio Moreira de Carvalho e Guillermo Cámara Chávez, e os restantes, que apesar de não termos tido trocas profundas, fazem um grande trabalho. Ao pessoal do CSI UFOP, pela infraestrutura que possibilitou esse trabalho, e possibilita tantos outros. Em especial ao Arthur Negrão, por responder todas as dúvidas e prestar assistência, rápido e assertivo, apesar dos meus milhões de e-mails. Ao grupo de Aprendizado Federado, por suportarem meu jorro de informações e reclamações durante as reuniões, assim como todo apoio, conselhos e *feedbacks*. Ao meu professor do ensino médio, Andre Oshima Roberto, por plantar a semente do senso crítico e conseguir, diferente de muitos, se manter um adulto no meio de crianças, mesmo quando o abismo também olhava para ti.

A Jonas Geiping criador do *framework Breaching*, pelas trocas e auxilio durante a execução desse trabalho. Assim como a sua propria contribuição cientifica, permintindo a existencia desse estudo. Aos amigos: Luan Prestes, Camila Oliveira, Ana Luiza, Emanuel Xavier, Rafael Nepomuceno, Higor Duarte, Lucas Sanches, Amanda Rampazzo, Leo Barbosa, Caio Martins, Jimi Dias Maggi, Amós Gonzaga e Gabriel Souza. Ao Otto, Ravi e Hariel. Por toda luz na minha vida.

Ao leitor, espero que consiga tirar algo proveitoso daqui, e que a montanha-russa de palavras não seja uma jornada tortuosa.

*O mais difícil não é um ser bom e proceder honesto; dificultoso, mesmo, é um saber definido o que quer, e ter o poder de ir até no rabo da palavra. Guimarães Rosa, Grande Sertão Veredas.*

# Resumo

A partir da metade da década passada, os estudos de aprendizado de máquina e inteligência artificial ganharam destaque para além da comunidade científica. Essas técnicas são aptas a afetar a sociedade de maneira profunda, tanto positiva quanto negativamente. Diante do crescimento da quantidade de dados necessários para o funcionamento dessas técnicas, assim como a própria natureza dos dados — muitas vezes sensíveis, privados e contidos em silos de diversas organizações, entidades, empresas e até mesmo em dispositivos de usuários singulares —, o Aprendizado Federado surge como uma proposta capaz de distribuir a carga de armazenamento e processamento desses dados, através da descentralização dos mesmos. Essa abordagem também postula que as informações sensíveis e privadas utlizadas em um treinamento podem ser utlizadas de forma segura, dado que apenas as atualizações do treinamento são compartilhados. Nesse contexto emergente e cada vez mais importante, o estudo das vulnerabilidades dessa técnica possui importância sumária para avaliar a segurança do método, propondo defesas e mitigações as técnicas adversárias. A existência de referências práticas das técnicas de reconstrução auxiliam tanto o âmbito acadêmico, quanto o mundo real, que utilizam o Aprendizado Federado para impactar o dia-a-dia da sociedade, uma vez, que, por mais que o conhecimento possa ser disrrupitivo quando utilizado por agentes maliciosos, é através das brechas e do reconhecimento das falhas que é possivel formular processos mais seguros, assim como atestar cientifica e empiricamente a eficácia de um determinado sistema diante de seus adversários.

**Palavras-chave**: Aprendizado Federado. Ataques de Reconstrução. Aprendizado de Máquina. *Python*. *PyTorch*.

# Abstract

Past the middle of last decade, the studys over Machine Learning and Artificial Inteligence earned emphasis beyond the cientific community. Those tecniques are capable of affect society in profound ways, both positively and negatively. Before the raising amount of data required for the operation of those tecniques, as well the own nature of those data: often sensitive and private, contained in silos of various organizations, entitys, companys and even over devices of singular users, the Federated Learning appears as a capable proposal to both aliviate the burden of storing and processing this data — throught the distribution of the distribution of the processing in a descentralized way —, as well the capability of treat those sensitive data securely. In this emerging context, more and more significant, the study of the vulnerabilities of this tecnique throught reconstruction attacks, supports both the cientific enviroment and the real world that utilizes the Federated Learning to impact the daily life of society. It is trought the breahcs and recognition of failures that a safer approach may become possible, as well afirm emperic and cientificly how well a given system performs against its adversaries.

**Keywords**: Federated Learning, Reconstruction Attacks, Machine Learning. *Python*. *PyTorch*.

# Sumário

# 1 Introdução

O termo Aprendizado Federado (AF) surge através dos trabalhos (KONEčNý; MCMAHAN; RAMAGE, 2015; MCMAHAN et al., 2023). A principal vantagem dessa abordagem, é a separação entre o treinamento do modelo e o acesso direto aos dados brutos, — uma vez que cada nó da rede treina um modelo local, com seu próprio conjunto de dados. Ou seja, apenas a atualização desses modelos é comunicada ao servidor central. Isso divide a carga de processamento entre os nós locais, barateando o treinamento. Essa técnica também é proposta como uma forma segura para o tratamento de dados privados. Isso é afirmado por (MCMAHAN et al., 2023) no contexto do uso de dados sensíveis de celulares para treinar modelos de ferramentas de sugestões de fotos e texto, inclusive, apropriando-se da capacidade de processamento ociosa dos dispositivos móveis dos usuários.

Em suma, o AF é uma técnica amplamente utilizada em diversos cenários e setores da sociedade e indústria, como os da tecnologia (MCMAHAN et al., 2023) e saúde (JOCHEMS et al., 2016). O estudo de suas falhas é também sumariamente importante, uma vez que dados sensíveis e privados podem ser apossados por agentes maliciosos. Como diz o ditado: a melhor defesa é o ataque. Através do conhecimento prático de como os ataques funcionam e que vulnerabilidades exploram, é possível propor maneiras de dificultar ou até mesmo impedir o vazamento de dados; estabelecendo científica e factualmente se o método é uma via segura ou não.

Os estudos que se configuram como base desse trabalho promovem ataques a partir daquele que detêm os resultados (modelos) produzidos pela técnica do AF. Definidos geralmente como servidores, ou coordenadores, ao longo da literatura (MCMAHAN et al., 2023; FOWL et al., 2022a; RODRIGUEZ-BARROSO et al., 2023). Sendo comumente, as grandes indústrias, organizações e entidades. Dessa forma, trabalhos como (FOWL et al., 2022a; FOWL et al., 2022b; WEN et al., 2022), ao promover ataques a partir do servidor, assumem a questão da segurança e privacidade sob os olhos dos clientes — ou usuários.

Estudos como (VEALE; BINNS; EDWARDS, 2018) demonstram que o Aprendizado de Máquina, de forma geral, é utilizado para gerar brechas em leis como a Regulamento Geral sobre a Proteção de Dados (LGPD) (PARLIAMENT; COUNCIL, 2016). Enquanto a indústria convenientemente estabelece o AF como uma técnica segura (MCMAHAN et al., 2023), estudos como (CHEN; SU; XU, 2017) atestam que o método permitiu o vazamento de dados privados de indivíduos famosos através de técnicas de reconstrução. De forma alarmante, a eficácia dos ataques apresentados em (FOWL et al., 2022a), denunciam a capacidade de abuso por parte dos servidores, atestando que a maioria dos *frameworks* de AF sequer contêm implementações para auditar a confiabilidade dos sistemas em uso.

Esse presente estudo propõem uma maneira eficaz de conduzir um ataque adversário contra um modelo treinado através da técnica de Aprendizado Federado. Através de implementações públicas, escolhidas pela popularidade e também diversidade de funcionalidades, os fundamentos necessários para a compreensão do assunto, assim como o processo da coleta de dados e desempenho dos ataques foram detalhados ao longo desse trabalho.

## 1.1 Objetivos

O principal objetivo desse estudo é estabelecer um procedimento prático e eficaz para a coleta de dados necessários para que um ataque adversário contra um sistema de Aprendizado Federado possa ser executado, para estabelecer um cenário onde a segurança de um determinado sistema possa ser colocado a prova, avaliando quão bem uma determinada configuração protege a privacidade dos usuários utilizando dados sensíveis num ambiente de Aprendizado Federado.

De forma geral, para alcançar o objetivo, são estabelecidos os seguintes pontos secundários como marcos:

- Definir os fundamentos de Aprendizado e Aprendizado Federado.
- Estabelecer os fundamentos de Ataques Adversários contra sistemas de Aprendizado Federado.
- Produzir o material necessário para a utilização de uma implementação pública de um sistema de Aprendizado Federado, assim como de ataques adversários.
- Através das implementações escolhidas, estabelecer um procedimento que abranja os passos necessários para que um ataque possa ser conduzido, desde a coleta de dados, a execução de fato.
- Analisar os resultados obtidos durante a experimentação e levantar as possibilidades e minúcias das técnicas executadas.

## 1.2 Organização do Trabalho

No Capítulo 2 será estabelecido o que é o Aprendizado Federado e sua diferença em relação ao método centralizado. Da mesma forma, será definido como funcionam as técnicas de ataque a esse modelo, os tipos de ameaça, os atores e seus comportamentos durante o processo. Também serão apresentadas as implementações disponíveis sobre o tema e seu manuseio.

A seguir, no Capítulo 3, são descritos os experimentos que estabelecem os procedimentos empíricos para que um ataque adversário possa ser conduzido contra um sistema de Aprendizado

Federado. Diversos modelos e ataques são utilizados, realizando uma comparação dos resultados e como uma determinada configuração pode impactar positiva ou negativamente a eficácia de um ataque adversário.

Por fim em 4 serão apresentadas as considerações finais, assim como as possibilidades que os procedimentos e experimentos descritos no Capítulo anterior deixam em aberto para novas possibilidades e avanços no tema.

# 2 Referencial Teórico

Esse Capítulo irá apresentar os conceitos básicos para a compreensão do tema. A Seção 2.1 busca definir o que é o Aprendizado Federado, suas origens teóricas e algumas das definições utilizadas em sua execução. Na Seção 2.2, da mesma forma, serão apresentadas diversas taxonomias de Ataques ao Aprendizado Federado. Na Seção 2.1.1 é apresentado o *framework Flower*, utilizado amplamente para o treinamento de modelos através da arquitetura de Aprendizado Federado. Finalmente a Seção 2.3, apresenta o *framework Breaching*, que implementa uma coletânea de ataques eficazes contra a privacidade dessa arquitetura. Essas duas implementações serão a base central para a condução dos experimentos e definição do procedimento de coleta de dados e ataque adversários.

## 2.1 Aprendizado e Aprendizado Federado

Em (KONEčNý; MCMAHAN; RAMAGE, 2015), estudo percursor do Aprendizado Federado, é realizada uma análise da estrutura de otimização de soma finita, como argumento central para um dos problemas que essa nova arquitetura busca solucionar. Essa estrutura é expressa pela seguinte Equação (2.1):

$$\min_{\omega \in \mathbb{R}^d} f(\omega) \quad \text{onde} \quad f(\omega) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} f_i(\omega) \tag{2.1}$$

Sendo:

$n$**:** total de amostras no conjunto de treinamento.

$\omega \in \mathbb{R}^d$**:** vetor de parâmetros do modelo (em espaço $d$-dimensional)

$f(\omega)$**:** objetivo de minimização a qual $f_i(\omega)$ busca reduzir o erro médio sobre o conjunto de treinamento.

Essa estrutura em 2.1 é conhecida como **minimização empírica** (VAPNIK, 1991). Ela cobre regressões lineares e logísticas, Máquinas de Vetores de Suporte (MVS), assim como algoritmos complexos como Campos Aleatórios Condicionais (CAC) e Redes Neurais (KONEčNý; MCMAHAN; RAMAGE, 2015).

De maneira geral, $f_i(\omega)$ é tipicamente especificado por uma **função de perda** $\ell$ que depende de um par de entrada e saída $\{x_i, y_i\}$ onde $f_i(\omega) = \ell(\omega; x_i, y_i)$ (KONEčNý; MCMAHAN; RAMAGE, 2015). Ou seja, $\omega$, definido anteriormente como o vetor de parâmetros do modelo, é ajustado através dos pares $\{x_i, y_i\}$, para **minimizar** a perda média do modelo entre as suas previsões e os valores reais dessas previsões.

Essa é a base da teoria do Aprendizado: dado um conjunto de amostras $n$, rotuladas para uma determinada **tarefa**, produzir um modelo estatístico capaz de realizar tal tarefa com a menor taxa de erro possível, e de tal maneira, que também seja possível que tal modelo possa reajustar seus parâmetros $\omega \in \mathbb{R}^d$ para sempre minimizar a possibilidade de erro em previsões futuras.

Ainda assim, a medida que o volume de dados cresce, o custo de armazenamento e processamento desses dados aumenta proporcionalmente. É necessário ressaltar também que a medida que o protagonismo da tecnologia cresce diante da sociedade, as preocupações sobre seu uso indevido e regulamentações para sua aplicação também.

Isso resulta em duas necessidades primarias que devem ser atendidas; a primeira, otimizar o treinamento e processamento dos dados, diante do aumento da carga e a segunda, proteger os dados sensíveis que precisam de ser utilizados para realizar determinadas tarefas, algumas importantes e essenciais, como o auxílio na análise e decisão médica, e outras banais e mercadológicas, como a retenção de um usuário através do aprimoramento da usabilidade e engajamento de uma aplicação eletrônica.

Os autores em (KONEčNý; MCMAHAN; RAMAGE, 2015) então propõem uma variação dessa estrutura de forma que seja possível operar em cenários distribuídos, dividindo a carga de processamento e armazenamento entre nós computacionais. O que resolve a primeira necessidade, e categoricamente afirmam que a natureza do treinamento, realizado através dessa estrutura, também viabiliza a segurança necessária para que dados sensíveis possam ser utilizados sem riscos a privacidade. Essa estrutura é expressa a seguir pela Equação 2.2:

$$f(\omega) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(\omega) \quad \text{onde} \quad F_k(\omega) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(\omega) \tag{2.2}$$

Comparando a expressão da Equação (2.1) e Equação (2.2), nota-se a adição da variável $k$ e um somatório extra para a composição dos parâmetros representados por $f(\omega)$. Nessa variação, $n$, definido anteriormente como o total de amostras no conjunto de treinamento está dividido entre $k$ **clientes**.

Ou seja, na abordagem expressa pela Equação (2.1), os dados estão centralizados em um determinado servidor que não é necessariamente o dono destes dados. Já na Equação (2.2), cada $k$ é o real detentor de seu conjunto $n_k$, e calcula seu próprio $F_k(\omega)$ através desses dados, gerando um modelo local que não conhece outros $F_{k_i}(\omega)$ nem seus dados $n_{k_i}$. Uma vez que todo

cliente possui seu conjunto de parâmetros, o somatório $\sum_{k=1}^{K} \frac{n_k}{n} F_k(\omega)$ é executado. Essa expressão significa que $f(\omega)$ é calculado através da **agregação** de diferentes $F_k(\omega)$, ou seja, os parâmetros de um modelo final são gerados através dos parâmetros locais de cada modelo em $k$.

Sendo:

$K$**:** número total de nós/clientes na rede federada.

$\mathcal{P}_k$**:** conjunto de dados no $k$-ésimo nó (partição $k$).

$n_k$**:** número de amostras no nó $k$ ($n = \sum_{k=1}^{K} n_k$).

$F_k(\omega)$**:** função de perda média no nó $k$.

A expressão na Equação (2.2) é essencialmente, a base do Aprendizado Federado. Dado um conjunto de clientes, contendo seu próprio conjunto de dados, coordenados por um servidor central, que ao fim de uma rodada de treinamentos locais, recebe os parâmetros de tais treinamentos, os agrega em um novo modelo e redistribui tal modelo de volta aos clientes. Ao longo de algumas rodadas, é produzido um modelo de inteligência artificial eficaz, gerado sem a centralização de dados. Consequentemente, o processamento necessário para a sua produção também é descentralizado, e a privacidade dos dados originais, é supostamente assegurada.

### 2.1.1 *Flower*

*Flower* é um *framework* que implementa diversos recursos para treinar, analisar e avaliar modelos utilizando a arquitetura de Aprendizado Federado (FLOWER, 2020a). Essa ferramenta contém uma documentação consistente e extensa disponibilizando rapidamente ao usuário um ambiente capaz de simular o aprendizado federado. Através do guia de inicialização rápida (FLOWER, 2020c) é possível ilustrar os conceitos introduzidos na Seção anterior de forma prática.

Esse guia disponibiliza ao usuário um repositório (FLOWER, 2020d) que treina um modelo *CNN* simples, utilizando a biblioteca *PyTorch* e o *Dataset CIFAR10*. Por padrão, o exemplo utiliza um total dez clientes — cinco para treinamento e cinco para avaliação —, instanciados na mesma máquina em que é executado em forma de *threads*. Realizando um total três iterações (*epochs*) e cinco rodadas (*rounds*) percorrendo o conjunto de treinamento. A implementação disponibilizada é relativamente simples, uma vez que grande parte do comportamento está abstraída em módulos internos e gira em torno de três *scripts*: *task.py*, *client_app.py* e *server_app.py* que são detalhados a seguir.

O *script task.py* define:

- o modelo que será treinado.
- a maneira como o conjunto de dados (*dataset*) será particionado, transformado e disponibilizado durante o treinamento e avaliação.
- a função de treino e avaliação que os clientes irão decorar.

Observe a função de treinamento:

Listagem 2.1: Método *train* no *script task.py*

```
64 def train(net, trainloader, epochs, lr, device):
65     """Train the model on the training set."""
66     net.to(device)  # move model to GPU if available
67     criterion = torch.nn.CrossEntropyLoss().to(device)
68     optimizer = torch.optim.Adam(net.parameters(), lr=lr)
69     net.train()
70     running_loss = 0.0
71     for _ in range(epochs):
72         for batch in trainloader:
73             images = batch["img"].to(device)
74             labels = batch["label"].to(device)
75             optimizer.zero_grad()
76             loss = criterion(net(images), labels)
77             loss.backward()
78             optimizer.step()
79             running_loss += loss.item()
80     avg_trainloss = running_loss / len(trainloader)
81     return avg_trainloss
```

Essa função:

**Assinatura da função (linha 64):** recebe o modelo *net*, o conjunto de dados (*trainloader*), a quantidade de iterações (*epochs*), a taxa de aprendizado (*lr*) e o dispositivo (CPU/GPU) na qual o treinamento irá ocorrer (*device*).

**Linhas 67-68:** definição a função de perda do treinamento (*criterion*) e o método de otimização (*optimizer*).

**Linhas 71-79:** itera os lotes do conjunto de dados durante as *epochs*, calculando a perda do treinamento, a cada iteração.

**Linhas 80-81:** por fim, produtos da perda são agregados na variável *running_loss*, e no fim da função a média da perda é calculada em relação ao total de lotes treinados no conjunto de dados e essa média é retornada.

Indo além, é possível através do algoritmo, encontrar parte da expressão matemática na Equação (2.2):

$$F_k(\omega) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(\omega) \tag{2.3}$$

Cada $k$ cliente:

- possui um conjunto de treinamento $n_k$ representado pela variável *trainloader*
- utiliza uma função de perda $f_i(\omega)$ representado pela variável *criterion*
- calcula $\omega$, representado por *running_loss* a cada iteração de $\sum_{i \in \mathcal{P}_k}$ representado pelo *loop* de repetição.
- obtêm um gradiente $F_k(\omega)$ representado pela variável *avg_trainloss*, gerado através da divisão $\frac{1}{n_k}$, representado pela operação *running_loss / len(trainloader)*.
- retorna $F_k(\omega)$ ao servidor para que a outra parte da equação seja realizada.

Como descrito anteriormente, esse método *train* em *task.py* é decorado pelo processo do cliente implementado através do *script client_app.py*. Observe a função *train* nesse *script*:

Listagem 2.2: Método *train* no *script client_app.py*

```
15 @app.train()
16 def train(msg: Message, context: Context):
17     """Train the model on local data."""
18
19     # Load the model and initialize it with the received weights
20     model = Net()
21     model.load_state_dict(msg.content["arrays"].to_torch_state_dict())
22     device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
23     model.to(device)
24
25     # Load the data
26     partition_id = context.node_config["partition-id"]
27     num_partitions = context.node_config["num-partitions"]
28     trainloader, _ = load_data(partition_id, num_partitions)
29
30     # Call the training function
31     train_loss = train_fn(
32         model,
33         trainloader,
34         context.run_config["local-epochs"],
35         msg.content["config"]["lr"],
36         device,
37     )
```

```
38
39      # Construct and return reply Message
40      model_record = ArrayRecord(model.state_dict())
41      metrics = {
42          "train_loss": train_loss,
43          "num-examples": len(trainloader.dataset),
44      }
45      metric_record = MetricRecord(metrics)
46      content = RecordDict({"arrays": model_record, "metrics": metric_record})
47      return Message(content=content, reply_to=msg)
48
```

Essa função:

**Linhas 20-21:** instancia o modelo e carrega os últimos parâmetros recebidos do servidor.

**Linhas 31-37:** invoca o método decorado *train* do arquivo *task.py* e recebe a média que foi calculada no fim do treinamento.

**Linhas 41-46:** retorna o resultado do treinamento ($F_k(\omega)$) para que o servidor agregue o modelo com as atualizações de todos os clientes.

Por fim, o *script server_app.py* contem apenas uma única função:

Listagem 2.3: Método *main* no *script server_app.py*

```
14 @app.main()
15 def main(grid: Grid, context: Context) -> None:
16     """Main entry point for the ServerApp."""
17
18     # Read run config
19     fraction_train: float = context.run_config["fraction-train"]
20     num_rounds: int = context.run_config["num-server-rounds"]
21     lr: float = context.run_config["lr"]
22
23     # Load global model
24     global_model = Net()
25     arrays = ArrayRecord(global_model.state_dict())
26
27     # Initialize FedAvg strategy
28     strategy = FedAvg(fraction_train=fraction_train)
29
30     # Start strategy, run FedAvg for `num_rounds`
31     result = strategy.start(
32         grid=grid,
33         initial_arrays=arrays,
34         train_config=ConfigRecord({"lr": lr}),
```

```
35          num_rounds=num_rounds,
36      )
37
38      # Save final model to disk
39      print("\nSaving final model to disk...")
40      state_dict = result.arrays.to_torch_state_dict()
41      torch.save(state_dict, "final_model.pt")
```

Nesse trecho é possível observar:

**Linhas 19-21:** carrega as configurações do treinamento.

**Linhas 24-25:** instancia o modelo global, e o estado atual de seus parâmetros é salvo para que os clientes o recebam durante o treinamento.

**Linha 28:** define a estratégia do treinamento.

**Linhas 31-41:** inicializa o treinamento e salva o modelo final ($F(\omega)$) em disco.

Grande parte do funcionamento da arquitetura e protocolo é abstraída. O processo de agregação e redistribuição é gerenciado pela chamada do método *strategy.start()*. O resto da Equação (2.2) está abstraído nessa parte, mas a sua implementação está disponível na documentação de *flower* (FLOWER, 2020b) em especial o método *aggregate_train* que realiza o somatório $\sum_{k=1}^{K} \frac{n_k}{n} F_k(\omega)$.

## 2.2 Ataque ao Aprendizado Federado

Por ser uma variante do aprendizado de máquina, o Aprendizado Federado herda diversas características do processo centralizado original, sob a diferença da divisão de nós e não transmissão dos dados brutos. Logo, as mesmas fraquezas e métodos de ataque do modelo original podem ser adaptadas ao modelo federado. Denominados como modelos de ameaça, os autores categorizam as possibilidades entre ameaças internas e ameaças externas.

Ameaças externas tem como foco as vulnerabilidades criadas no segmento da comunicação dos modelos federados; quando os clientes enviam as atualizações para que o coordenador central realize a agregação dos mesmos, expondo suas atualizações através da rede na qual se comunicam. Já no caso interno, um ou mais nós, ou o próprio coordenador, comportam-se de forma maliciosa.

Ameaças internas são consideradas mais perigosas que as externas, uma vez que o agente malicioso tem acesso ao sistema em si e pode se aproveitar de suas vulnerabilidades. Os autores

de (RODRIGUEZ-BARROSO et al., 2023) e a maioria dos estudos realizados na área tem foco nas ameaças internas e as categoriza da seguinte forma:

**Estrutura dos ataques:** são geralmente categorizados como ataques bizantinos ou ataques *Sybil*.

**Ataques bizantinos:** consistem no envio de atualizações arbitrárias ao coordenador, comprometendo o modelo global durante esse processo.

**Ataques *Sybil*:** são ataques de natureza colaborativa, sob os quais um ou mais nós, ou até mesmo nós fictícios comprometem o modelo central de forma disruptiva.

**Diferenciação do protagonismo entre cliente e servidor:** um coordenador malicioso é considerado mais perigoso que um cliente malicioso, uma vez que o primeiro possui mais informações e conhecimento da arquitetura federada do que os clientes.

**Comportamento do adversário:** podem ter um comportamento passivo ou ativo, generalizando a motivação entre prejudicar o modelo (ataque ao modelo) ou obter dados sensíveis do aprendizado (ataque a privacidade).

**Comportamento passivo, ou adversário "honesto, mas curioso":** é aquele que respeita os protocolos e não tenta interferir no treinamento, mas mesmo assim busca obter informações restritas.

**Comportamento ativo, ou adversário malicioso:** é aquele que interfere, de maneira a prejudicar o modelo, ou injetar tarefas secundárias durante o treinamento, para comprometer a privacidade de maneira eficaz.

## Estratégias de Ataques

As principais formas de ataques publicadas são geralmente classificadas da seguinte forma:

**Ataque analítico:** em redes neurais há um termo definido por *bias*, que, em tradução literal significa viés ou tendência. Esse termo é um parâmetro constante adicionado aos neurônios da rede, para criar um deslocamento que evita que a rede somente utilize o ponto de origem durante uma previsão. Em (PHONG et al., 2018) é feita a descoberta de que ataques analíticos baseados em gradientes podem ser realizados respeitando esse deslocamento como a informação necessária para a reconstrução do dado, uma vez que o gradiente sempre carrega essa constante como ponto determinante para o ataque.

**Ataques recursivos:** É comentado por (FOWL et al., 2022a) como uma extensão do ataque analítico. Como afirma (ZHU; BLASCHKO, 2021), o ataque recursivo não depende do *bias* para ser aplicado. O método utiliza técnicas matemáticas de álgebra linear para realizar a inversão de gradientes, através da recursão das camadas computacionais.

**Ataques baseados em otimização:** essa técnica se baseia no processo de minimizar a distância entre gradientes ou outros dados públicos que possam reconstruir o dado privado do treinamento. Ou seja, um servidor malicioso ao obter a atualização de um cliente, pode simultaneamente, conduzir um treinamento de otimização que busca encontrar a mesma função de perda que gerou o gradiente do cliente. Dessa forma, é possível reconstruir o dado privado do cliente de maneira silenciosa, comprometendo a privacidade e obtendo o dado bruto que gerou o gradiente inicial.

No caso dos ataques analíticos (PHONG et al., 2018) e recursivos (ZHU; BLASCHKO, 2021), só é possível obter resultados de uma única instância dos dados. No caso de múltiplas instâncias que realizam a agregação dos gradientes durante a atualização, essa estratégia só consegue obter a média das entradas, tornando-se incapaz de identificar o dado individual da agregação. Ataques de Otimização operam melhor na inversão do modelo, mas a medida que o tamanho do *batch* cresce, o desempenho da reconstrução cai severamente, uma vez que a maioria dos ataques não conseguem reconstruir os dados individuais agregados ao *batch*.

A seguir, dois ataques da família de otimização serão abordados, uma vez que foram os ataques utilizados nos experimentos do Capítulo 3 posterior.

### 2.2.1 *Beyond Infering Class Representatives*

Essa publicação foi uma das primeiras a oferecer um modelo de ameaça silencioso, assumindo a perspectiva de um servidor adversário. Em (WANG et al., 2019), são exploradas diversos comportamentos de um servidor, que realiza a reconstrução dos dados de seus clientes de maneira ativa ou passiva, mas sempre silenciosamente, sem comprometer o treinamento ou o modelo, utilizando o conceito de *Generative Adversarial Networks* (GAN) (GOODFELLOW et al., 2014), em tradução livre, Rede Generativa Adversária. Essa rede consegue gerar amostras que são praticamente indistinguíveis das amostras reais ou de treinamento. Seu funcionamento expresso através da Equação (2.4) a seguir:

$$\begin{aligned}\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}}\big[\log D(x)\big] \\ + \mathbb{E}_{z\sim p_z}\big[\log(1-D(G(z)))\big],\end{aligned} \tag{2.4}$$

Sendo:

$G$**:** gerador $G$, representa a rede que iriá gerar a amostra indistinguível.

$D$**:** discriminador $D$, responsável por tentar distinguir os dados reais dos falsos.

$z$**:** amostra $z$ de $G$ gerada de uma distribuição conhecida e utilizada como entrada, por exemplo, gaussiana ou uniforme.

$p_{data}$**:** distribuição de dados reais.

$p_z$**:** distribuição de dados falsos.

$\min_G V(D,G)$**:** $G$ minimiza $V(D,G)$, ou seja, engana $D$, aproximando $z$ do dado real.

$\max_D V(D,G)$**:** $D$ maximiza $V(D,G)$, ou seja, acerta ao classificar os dados reais e falsos.

Temos então aqui um **jogo minimax** (MM), onde se busca encontrar o Equilíbrio Nash como ponto ideal de convergência; $D$ é incapaz de distinguir o dado real do falso e $G$ gera dados falsos indistinguíveis de dados reais (GOODFELLOW et al., 2014).

### 2.2.2 *Deep Leakage from Gradients*

Diferente do ataque descrito na Subseção 2.2.1 anterior, o ataque *Deep Leakage* não utiliza uma GAN para reconstruir um gradiente real. Ao invés disso, esse ataque gera um dado e um rótulo falso, e através das técnicas de otimização, aproxima o dado e rotulo falsos dos reais a medida que gera um novo gradiente através dos dados falsos (ZHU; LIU; HAN, 2019).

Dada a expressão (2.5):

$$\nabla W_{i,t} = \frac{\partial \ell(F(x_{t,i}, W_t), y_{t,i})}{\partial W_t} \tag{2.5}$$

Essa expressão é similar a apresentada em (2.2), sendo o segundo trecho da mesma. Um gradiente $W$ de um cliente $t$ é gerado através de uma função de perda $F()$ utilizando um par ordenado $x_{t,i}, y_{t,i}$ onde $i$ é o conjunto de treinamento.

Através da premissa que $W_t$ e $F()$ são obrigatoriamente compartilhados num sistema de Aprendizado Federado, os autores de (ZHU; LIU; HAN, 2019) propõem as seguintes expressões:

$$\nabla W' = \frac{\partial \ell(F(x'_{t,i}, W_t), y'_{t,i})}{\partial W_t} \tag{2.6}$$

Onde $W'$, $x'$ e $y'$ são respectivamente um gradiente, um dado de entrada e um rótulo dessa entrada falsificados (*dummies*). Caso esse gradiente $W'$ seja otimizado de forma que se aproxime do $W$ real, consequentemente $x'$ e $y'$ também se aproximarão de seu respectivo pares reais. A expressão da Equação (2.7) a seguir, então é o objetivo de minimização do ataque:

$$x'*, y'* = \underset{x',y'}{argmin} \|\nabla W' - \nabla W\|^2 = \underset{x',y'}{argmin} \| \frac{\partial \ell(F(x', W), y')}{\partial W} - \nabla W\|^2 \tag{2.7}$$

Nessa Equação (2.7) temos então que $x'*$ e $y'*$ serão escolhidos para minimizar as distâncias — representada por $\|$ e $\|$ —, entre os gradientes $W'$ e $W$, expandindo a função de perda $\frac{\partial \ell(F(x', W), y')}{\partial W} - \nabla W$. Ou seja, $W'$ será gerado através da otimização sequencial de $x'$ e $y'$ de forma que o gradiente $W'$ de $x'$ e $y'$ se aproxime cada vez mais do dado original. Uma vez que a convergência seja atingida, esses dados foram reconstruídos.

A vantagem dessa abordagem é que nenhum passo extra é necessário, como a definição e treinamento da GAN como no ataque *Beyond Inferring* da Seção 2.2.1 anterior.

## 2.3 *Framework Breaching*

O *framework Breaching* (GEIPING, 2021a) implementa uma série de ataques de reconstrução, apresentando alta eficácia em comprometer a privacidade dos ambientes de Aprendizado Federado. Nas publicações associadas ao *framework*, são propostas modificações nas estruturas dos ataques federados vigentes à época. Seus autores assumem a perspectiva da privacidade sob os olhos do usuário. Isso é, observam como um servidor malicioso pode aplicar modificações sutis ao modelo para violar a privacidade dos usuários. Esses estudos propõem implementações capazes de obter resultados em cenários reais, em que os volumes de dados podem ter escalas industriais, em diferença aos outros trabalhos especificados aqui. A seguir, um resumo dos trabalhos relacionados:

***Robbing the FED* (FOWL et al., 2022a):** propõe uma técnica denominada "**módulo de impressão**". Esse módulo é uma camada oculta que um servidor malicioso adiciona aos parâmetros do modelo que são compartilhados com os clientes, possibilitando distinguir dados individuais dos conjuntos agregados, em diferença as estratégias anteriores eram capazes apenas de obter a média das entradas. Nos trechos finais do estudo, há um tópico voltado a questão ética do trabalho, uma vez que os resultados são profundamente mais eficazes que outros trabalhos até então. Ressaltando que disponibilidade do código-fonte pode ser apropriada para fins maliciosos, os autores afirmam que o papel fundamental do estudo é informar a comunidade sobre o real estado da privacidade no âmbito do Aprendizado Federado.

***Decepticons* (FOWL et al., 2022b):** expõe que grande parte da literatura tem foco apenas nos casos de uso visuais, enquanto os cenários reais geralmente operam no contexto textual. Os autores então propõe a injeção de parâmetros vetoriais no modelo para a reconstrução dos dados textuais. Além do chamado de ética, os autores propõem mitigações que poderiam ser realizadas. É também denunciado que a maioria dos *frameworks* de Aprendizado Federado utilizados pelas indústrias não implementam formas de verificar os parâmetros e comportamento do servidor e que leis poderiam ser criadas para garantir a proteção dos usuários diante das ameaças e abusos possíveis.

***Fishing for User Data* (WEN et al., 2022):** propõe uma nova estratégia para reconstruir dados no cenário visual, injetando um gradiente conhecido que se sobressai na agregação do *batch*. Isso leva a uma atualização previsível do modelo, uma vez que a agregação é dominada por uma única classe, permitindo a desagregação dos dados reais do gradiente injetado. Esse método de forma alguma altera o modelo, caindo na categoria de servidor "honesto, mas curioso". Mitigações são propostas sem atentar para questões éticas. Os autores também afirmam que tais técnicas, ao passo que fortalecem a privacidade do usuário, diminuem a acurácia do modelo treinado.

### 2.3.1 Estrutura do Repositório

O repositório tem como base um diretório nominal, estruturando subsequentemente:

***analysis*:** disponibiliza diversas métricas para avaliar o desempenho dos ataques.

***attacks*:** implementa diversos ataques modulares de inversão de gradientes.

***cases*:** estrutura diversos casos de uso, como comportamento do servidor, usuário, arquitetura do modelo e *dataset*.

***config*:** possibilita através da biblioteca *hydra* um conjunto de configurações que operam nas implementações definidas anteriormente para facilitar e estender as possibilidades de execução.

#### Dependências

As principais dependências de *Breaching* estão listadas a seguir:

***conda*:** é um programa que tem como o foco o gerenciamento de ambientes *python* (ANACONDA, 2012). É uma extensão semelhante à solução nativa da linguagem (*pyenv*) que tem como diferencial a possibilidade de utilizar uma versão diferente da nativa do sistema.

***PyTorch*:** é a principal biblioteca para a utilização e cálculo de tensores (META, 2016a), — que são matrizes ou vetores multidimensionais —, principalmente utilizados nas técnicas de aprendizado de máquina e redes neurais.

***Hydra*:** é um *framework* que permite a configuração hierárquica de um programa através da composição e sobrecarga de arquivos ou parâmetros de linha de comando (META, 2020). Um tutorial apresentando as capacidades e minucias desse *framework* está disponível no apêndice A.

Existem, além dessas uma gama de outras dependências, mas que operam em segundo plano. Uma vez que não são explicitamente necessárias para a execução e compreensão do projeto, não serão abordadas.

### 2.3.2 Configuração

Durante o início da Subseção 2.3.1 foi observado que a configuração do *Framework* está implementada na pasta *config*. Observe:

Listagem 2.4: Trecho do arquivo *cfg.yaml*

```
1 # @package _global_
2 # Configuration defaults
3 # Settings are separated into case, attack analysis
4 defaults:
5   - case: 2_single_imagenet
6   - attack: invertinggradients
7   - _self_
```

- A chave *defaults* é uma chave reservada em *Hydra*.
- Através dela, seus filhos representam um diretorio (chave), e um arquivo (valor) a qual o arquivo de configuração em questão irá compor uma configuração completa.

Em paralelo, agora, observe os possíveis grupos de configuração:

Listagem 2.5: Trecho da saída do argumento *--help* declarando os grupos de configuração

```
    $ python3 simulate_breach.py --help
    == Configuration groups ==
    Compose your configuration from those groups:

    attack: _default_optimization_attack, analytic, april_analytic,
beyondinfering, clsattack, decepticon, deepleakage, imprint, invertinggradients,
legacy, modern, multiscale_ghiasi, rgap, sanitycheck, seethroughgradients, tag,
wei
    case: 0_sanity_check, 10_causal_lang_training, 1_single_image_small, 2_
↪single_imagenet, 4_fedavg_small_scale, 5_small_batch_imagenet, 6_large_batch_
↪cifar, 8_industry_scale_fl, 9_bert_training
    case/data: Birdsnap, CIFAR10, CIFAR100, ImageNet, ImageNetAnimals,
TinyImageNet, cola, random-tokens, shakespeare, stackoverflow, wikitext
    case/data/db: LMDB, none
    case/impl: default
    case/server: honest-but-curious, malicious-fishing, malicious-model-cah,
malicious-model-rtf, malicious-transformer
```

```
case/user: local_gradient, local_updates, multiuser_aggregate
```

- A chave *attack* tem como valor padrão o arquivo *invertinggradients* e pode ser combinada com cada uma das saídas produzidas em 2.5. A Tabela no Apêndice B.11, expõe a gama de ataques implementados, assim como a estratégia utilizada.
- Da mesma forma, a chave *case* tem como padrão o arquivo *2_single_imagenet* e suas possibilidades podem ser visualizadas através da tabela B.10 no Apêndice B. É possível notar também que o grupo *case* possui ainda mais quatro subgrupos, compostos pelos itens a seguir:

  ***data*:** grupo de *datasets*, implementados no *framework*. Esse grupo também possui o subgrupo *db*, utilizado para habilitar o uso de banco de dados com os *datasets*. A tabela B.2, resume as opções disponíveis de *datasets*, junto de sua modalidade e tarefa a qual o conjunto de dados se destina. Cada uma das entradas representam arquivos de configuração em *breaching/config/case/data*. Os parâmetros que cada *dataset* podem apresentar se encontram disponíveis nas tabelas do Apêndice B.3, Apêndice B.4 e Apêndice B.5.

  ***impl*:** grupo específicos as opções de implementação, como parâmetros da biblioteca *PyTorch* e comportamento do ataque. Os parâmetros de implementação definem recursos como uso de aceleração gráfica, comportamento dos tensores, validações e *benchmarks*. Estes são apresentados na tabela B.1 no Anexo B.

  ***server*:** grupo de configuração que define o comportamento do servidor durante uma simulação, disponível na tabela do Apêndice B.9.

  ***user*:** grupo de configuração que define o comportamento de agregação de usuário/cliente, assim como quantidade do *batch* utilizado durante o treinamento e qual cliente será atacado durante uma simulação. A Tabela B.6, define o comportamento do cliente ao submeter atualizações ao servidor. Cada uma das entradas representam arquivos de configuração em *breaching/config/case/user*. Cada um dos parâmetros possíveis também está especificado na tabela do Apêndice B.

### 2.3.3 Tutorial — Utilizando *Breaching*

Essa Seção visa, através dos *scripts* implementados no repositório, definir as noções gerais do funcionamento do *framework*. O repositório contem três arquivos centrais, além dos exemplos, que definem a maneira de operar o *framework*:

***simulate_breach.py*:** implementa uma estrutura completa, que simula um ambiente de Aprendizado Federado, instanciando um servidor, usuários e modelo. Uma vez instanciados, esse *script* realiza uma iteração do protocolo, gerando um *payload* que contem todos os dados necessários para a execução de um determinado ataque. No fim da execução, analisa a eficácia do ataque e também disponibiliza *logs* da execução.

***minimal_example.py*:** propõe uma estrutura mínima, sem instanciar algum tipo de servidor, cliente ou modelo e simula uma única iteração do protocolo federado para ilustrar como um ataque pode ser conduzido utilizando dados externos.

***minimal_example_robbing_the_fed.py*:** semelhante à implementação anterior, é um modelo básico que propõe uma forma de verificar um modelo e sua arquitetura contra o método *Robbing the feds* proposto em (FOWL et al., 2022a).

**Análise da implementação *simulate_breach.py***

No README o autor do *framework* recomenda que a implementação em *simulate_breach.py* pode ser utilizada para verificar a instalação do projeto através do comando abaixo, que executa o caso de reconstrução mais simples com uma única iteração:

```
$ python simulate_breach.py dryrun=True
```

A implementação contem dois métodos: "*main_laucher*" e "*main_process*". O ataque padrão utiliza o caso de uso *2_single_imagenet* que requer o *dataset imagenet* (tabela B.10). Esse *dataset* só pode ser baixado manualmente e ocupa um espaço de disco considerável. Sendo assim, o caso de uso *1_single_image_small* é utilizado no lugar do caso padrão através da sobrecarga:

```
$ python simulate_breach.py dryrun=True case=1_single_image_small
```

Uma vez que o objeto de configuração é instanciado, a execução continua através da função *main_process*. Quando a simulação é finalizada, um arquivo de registro é salvo no diretório *outputs/default/<DATA>/<HORA>*. Um exemplo de saída com sucesso está disponível no apêndice D.

Esse *script* é útil para testar os diversos cenários e ataques implementados pelo *framework*. Através da dependência *Hydra*, é possível sobrecarregar os diversos parâmetros implementados nos arquivos de configuração, selecionando diferentes ataques, modelos, servidores e clientes. É possível também definir a quantidade de iterações que serão realizadas durante a simulação, assim como a quantidade do *batch* do gradiente.

### Análise da implementação *minimal_example.py*

Conforme descrito no início da Seção, esse *script* não utiliza a estrutura de configuração apresentada anteriormente. Dessa forma, é proposto um caso de uso capaz de executar os ataques a modelos customizados e reais. No trecho a seguir, as variáveis são inseridas manualmente para ilustrar como um modelo customizado pode ser utilizado no ataque:

Listagem 2.6:

```
8  import torch
9  import torchvision
10 import breaching
11
12
13 class data_cfg_default:
14     modality = "vision"
15     size = (1_281_167,)
16     classes = 1000
17     shape = (3, 224, 224)
18     normalize = True
19     mean = (0.485, 0.456, 0.406)
20     std = (0.229, 0.224, 0.225)
21
22
23 transforms = torchvision.transforms.Compose(
24     [
25         torchvision.transforms.Resize(256),
26         torchvision.transforms.CenterCrop(224),
27         torchvision.transforms.ToTensor(),
28         torchvision.transforms.Normalize(mean=data_cfg_default.mean, std=data_
   ↪cfg_default.std),
29     ]
30 )
31
```

***data_cfg_default*:** são os conjuntos de metadados relativos a um *dataset*, necessários durante a normalização, para que o *dataset* seja utilizado pela rede escolhida.

- *size* é o tamanho total de imagens no *dataset*.
- *classes* é a quantidade rótulos.
- *shape* representa a quantidade de canais (3) e o tamanho das imagens (224x224).
- *mean* é o valor de normalização dos canais.
- *std* é o valor de normalização dos *pixels* do *dataset*.

***transforms*:** é utilizado para converter as imagens do *dataset* em tensores que podem ser utilizados por um modelo.

Listagem 2.7:

```
32
33 def main():
34     setup = dict(device=torch.device("cpu"), dtype=torch.float)
35
36     # This could be your model:
37     model = torchvision.models.resnet152(pretrained=True)
38     model.eval()
39     loss_fn = torch.nn.CrossEntropyLoss()
40
41     # And your dataset:
42     dataset = torchvision.datasets.ImageNet(root="~/data/imagenet", split="val",
   transform=transforms)
43     datapoint, label = dataset[1200]  # This is the owl, just for the sake of
   this experiment
44     labels = torch.as_tensor(label)[None, ...]
45
46     # This is the attacker:
47     cfg_attack = breaching.get_attack_config("invertinggradients")
48     attacker = breaching.attacks.prepare_attack(model, loss_fn, cfg_attack,
   setup)
```

- Durante o método *main()* é definido o objeto *setup* que define como o modelo irá operar.
- O modelo então é definido, utilizando uma rede *resnet152* pré-treinada, assim como a função de perda do treinamento.
- Um *dataset* é utilizado para gerar o gradiente que deverá ser reconstruído durante o ataque.
- Por fim o ataque também é configurado para a execução.

Uma vez que as variáveis necessárias são instanciadas, uma iteração do treinamento federado é simulada, e o ataque, por fim, é executado:

Listagem 2.8:

```
49
50     # ## Simulate an attacked FL protocol
51     # Server-side computation:
52     server_payload = [
53         dict(
54             parameters=[p for p in model.parameters()], buffers=[b for b in
   model.buffers()], metadata=data_cfg_default
```

```
        )
    ]
    # User-side computation:
    loss = loss_fn(model(datapoint[None, ...]), labels)
    shared_data = [
        dict(
            gradients=torch.autograd.grad(loss, model.parameters()),
            buffers=None,
            metadata=dict(num_data_points=1, labels=labels, local_
↪hyperparams=None,),
        )
    ]

    # Attack:
    reconstructed_user_data, stats = attacker.reconstruct(server_payload,
shared_data, {}, dryrun=False)

    # Do some processing of your choice here. Maybe save the output image?
```

- A variável *server_payload* representa os parâmetros enviados ao cliente pelo modelo.
- Já *shared_data*, são os gradientes computados pelo cliente e devolvidos ao servidor num cenário real.
- Por fim, através desses dados, o ataque é executado, retornando o produto da reconstrução, e deixando em aberto ao usuário a decisão de como manusear tais dados.

# 3 Experimentos e Resultados

No Capítulo anterior foram definidos vários dos recursos e conhecimentos necessários para manipular os *Frameworks Flower* e *Breaching*. Nesse Capítulo, estão descritos os experimentos realizados e resultados obtidos, utilizando os ataques de *Breaching* contra modelos treinados através de *Flower*, na Seção 3.1. Por fim a Seção 3.2 apresenta diversas modificações introduzidas em *Breaching* na forma de correções e melhorias, impactando na usabilidade direta do repositório.

## 3.1 Bem-me-quer, Mal-me-quer — Um Experimento Prático

Através das estruturas estabelecidas no Capítulo 2 anterior nas subseções 2.1.1 e 2.3, essa Seção visa utilizar tais recursos para estabelecer um caso de uso real: dado um sistema federado, — instanciado e treinado utilizando o *framework Flower* —, é possível produzir uma cadeia de ataques contra o modelo gerado, para averiguar sua segurança e privacidade?

### 3.1.1 Modificações Realizadas nos *Frameworks*

Esse experimento utilizará a implementação do guia de inicialização rápida definida no Capítulo anterior (2.1.1) para coletar os gradientes que deverão ser reconstruídos em *Breaching* utilizando como base o *script minimal_example.py* (2.3.3).

A estratégia mais simples do aprendizado federado é o método *FedSGD*, que envia o gradiente resultante do treinamento ao servidor, sem realizar nenhuma operação a mais. Já a estratégia *FedAVG*, implementada por padrão no exemplo utilizado em *flower*, envia a média escalar dos gradientes computados durante o treinamento. Por uma questão prática, para reduzir o experimento ao caso mais básico, a função de treino implementada no arquivo *task.py* em *flower* será modificada de forma que o gradiente computado, assim como o dado real utilizado e todas as informações necessárias sejam salvas antes da agregação realizada pelo servidor. Essa modificação não é aceitável num ambiente real, mas atende a esse primeiro experimento, que busca viabilizar a possibilidade de reconstruir um gradiente treinado através do *framework Flower* utilizando a implementação em *breaching*. As modificações podem ser contempladas na forma de *diffs* no apêndice F.

Durante a primeira *epoch* do primeiro *batch*, os parâmetros e *buffers* do servidor devem

ser salvos **antes** que o gradiente seja calculado. Da mesma forma, os gradientes e os *buffers* do cliente são salvos **após** o cálculo realizado pela função *loss.backward()*. Por praticidade, os clientes irão sobrescrever esses arquivos, e o último cliente a executar a primeira iteração que terá o dado salvo de fato. De qualquer maneira, nesse estágio inicial, não é importante salvar mais que um gradiente e seu dado real.

A seguir, serão apresentadas as modificações realizadas no *script minimal_example.py*, é possível também contemplar o algoritmo na íntegra no apêndice F.

Listagem 3.1: Modificações no *script minimal_example.py*

```
8  import torch
9  import torchvision
10 import breaching
11 import argparse
12
13
14 class data_cfg_default:
15     modality = "vision"
16     size = (1_281_167,)
16     classes = 1000
17     shape = (3, 3224, 3224)
18     normalize = True
19     mean = (0.485, 0.456, 0.406)
20     std = (0.229, 0.224, 0.225)
21
22
23 transforms = torchvision.transforms.Compose(
24     [
25         torchvision.transforms.Resize(256),
26         torchvision.transforms.CenterCrop(224),
27         torchvision.transforms.ToTensor(),
28         torchvision.transforms.Normalize(mean=data_cfg_default.mean, std=data_
   ↪cfg_default.std),
29     ]
18
19
20 def main():
21     parser.add_argument('--attack', default='invertinggradients')
22     parser.add_argument('--gradient-path', required=True)
23     parser.add_argument('--parameters-path', required=True)
24     parser.add_argument('--public-buffers-path', required=True)
25     parser.add_argument('--shared-buffers-path', required=True)
26     parser.add_argument('--true-data-path', required=True)
27     parser.add_argument('--provide-labels', action='store_true', default=False)
28     parser.add_argument('--provide-public-buffers', action='store_true',
   default=False)
29     parser.add_argument('--provide-shared-buffers', action='store_true',
   default=False)
30     args = parser.parse_args()
31
```

```
    setup = dict(device=torch.device("cpu"), dtype=torch.float)

    # This could be your model:
    model = torchvision.models.resnet152(pretrained=True)
    model = breaching.cases.models.model_preparation.Net()
    model.eval()
    loss_fn = torch.nn.CrossEntropyLoss()

    # And your dataset:
    dataset = torchvision.datasets.ImageNet(root="~/data/imagenet", split="val", transform=transforms)
    datapoint, label = dataset[1200]  # This is the owl, just for the sake of this experiment
    labels = torch.as_tensor(label)[None, ...]

    # This is the attacker:
    cfg_attack = breaching.get_attack_config(args.attack‘‘invertinggradients’’)
    attacker = breaching.attacks.prepare_attack(model, loss_fn, cfg_attack, setup)

    true_data = torch.load(args.true_data_path)
    images    = true_data["images"]
    labels    = true_data["labels"]

    parameters     = torch.load(args.parameters_path)
    buffers        = torch.load(args.public_buffers_path) if args.provide_public_buffers else None
    shared_buffers = torch.load(args.shared_buffers_path) if args.provide_shared_buffers else None
    gradients      = torch.load(args.gradient_path)

    # ## Simulate an attacked FL protocol
    # Server-side computation:
    server_payload = [
        dict(
            parameters=[p for p in model.parameters()], buffers=[b for b in model.buffers()], metadata=data_cfg_default
            parameters=parameters, buffers=buffers, metadata=data_cfg_default
        )
    ]
    # User-side computation:
    loss = loss_fn(model(datapoint[None, ...]), labels)
    shared_data = [
        dict(
            gradients=torch.autograd.grad(loss, model.parameters()),
            buffers=None,
            metadata=dict(num_data_points=1, labels=labels, local_hyperparams=None,),
```

```
58              gradients=gradients,
59              buffers=shared_buffers,
60              metadata=dict(
61                  num_data_points=1,
62                  labels=None if args.provide_labels == False else labels,
63                  local_hyperparams=None,
64              ),
65          )
66      ]
67
68      # Attack:
69      reconstructed_user_data, stats = attacker.reconstruct(server_payload,
    shared_data, {}, dryrun=False)
70
71      # Do some processing of your choice here. Maybe save the output image?
72      images = images.cpu()
73      reconstructed_images = reconstructed_user_data['data'].cpu()
74      torchvision.utils.save_image(reconstructed_images, f"outputs/reconstructed.
    ↪png", normalize=True)
75      torchvision.utils.save_image(images, f"outputs/true.png", normalize=True)
76      ...
```

Na Listagem acima, foram realizadas as seguintes modificações:

- uma vez que os gradientes estão sendo gerados através do *framework Flower*, o *script minimal_example.py* foi simplificado. As linhas removidas 16, 18 até 29 e 35 até 39 remetem ao manuseio direto do *dataset* para simular uma iteração da estratégia *FedSGD* durante a execução e não são mais necessárias.

- as linhas 11 e 21 até 30 adicionam a utilidade *argparse* para que o *script* receba argumentos na linha de comando durante uma execução. Sendo assim, o *script* passa a receber o nome de um ataque, o caminho para os parâmetros e *buffers* do servidor, assim como o gradiente e os *buffers* do cliente. O dado real também é adicionado para ilustrar e comparar os resultados do experimento. Esses dados introduzidos via linha de comando são consequentemente atribuídos aos locais corretos nas linhas 40 até 64.

- a linha 32 é modificada para aceitar o modelo que será utilizado durante o treinamento em *flower*. O modelo padrão *Net* disponível no guia de inicialização rápida foi adicionado ao módulo *model_preparation* de *breaching*, que implementa os modelos utilizados no *framework*. Sendo assim, basta alterar as últimas linhas para utilizar qualquer outro modelo caso o mesmo seja modificado durante o treinamento.

- por fim, as linhas 69 até 77 salvam os resultados do ataque em disco.

Através dessas modificações, é possível então gerar os gradientes através do *Flower* e atacá-los com *Breaching*. A seguir, alguns ataques serão realizados contra diversos modelos, utilizando uma *epoch* e um *round* durante o treinamento.

### 3.1.2 Configurações e metadados dos Ataques Executados

Essa Subseção visa apresentar algumas minúcias relativas aos ataques executados durante o experimento. Tais ataques têm o embasamento teórico descrito no Capítulo anterior, nas subseções 2.2.1 e 2.2.2. As listagens 3.2 e 3.3 a seguir são saídas que *Breaching* proporciona durante a execução do ataque.

Listagem 3.2: Metadados do Ataque *Deep Leakage*

```
Attacker (of type OptimizationJointAttacker) with settings:
    Hyperparameter Template: deep-leakage

    Objective: Euclidean loss with scale=1.0 and task reg=0.0
    Regularizers:
    Augmentations:

    Optimization Setup:
        optimizer: L-BFGS
        signed: None
        step_size: 1.0
        boxed: False
        max_iterations: 1200
        step_size_decay: None
        langevin_noise: 0.0
        warmup: 0
        grad_clip: None
        callback: 100
```

Listagem 3.3: Metadados do Ataque *Beyond Infering*

```
Attacker (of type OptimizationBasedAttacker) with settings:
    Hyperparameter Template: beyond-infering

    Objective: Euclidean loss with scale=1.0 and task reg=0.0
    Regularizers: Total Variation, scale=0.2352. p=2 q=1.25.
    Augmentations:

    Optimization Setup:
        optimizer: L-BFGS
        signed: None
        step_size: 1.0
        boxed: True
```

```
22          max_iterations: 400
23          step_size_decay: None
24          langevin_noise: 0.0
25          warmup: 0
26          grad_clip: None
27          callback: 100
```

Para ambos os metadados, nas linhas 12-14, estão os hiperparâmetros disponíveis para aprimorarem os resultados de um ataque. *Objetive* é objetivo de distância euclidiana com variação de escala e regularização da tarefa. *Regularizers* são regularizadores de variação (TV), com variação de escala, e "p" e "q" sendo os exponentes de magnitude, e normalização de canais, respectivamente. Por fim *Augmentations* representam hiperparâmetros indisponíveis para os ataques escolhidos. Agora, as linhas 16-26 apresentam parâmetros da função de perda em uso, os mais pertinentes sendo *optimizer* objetivo de perda, no caso, L-BFGS (META, 2016b). A chave step_size é semelhante ao *learning rate* de um treinamento, é o valor de ajuste a cada nova computação da perda. Já *max_iterations* simboliza a quantidade total em que o ataque, ou o objetivo será executado para realizar a reconstrução. Por fim *callback* é numero de iterações que o ataque irá realizar antes de realizar alguma saída. A iteração no valor do *callback* será impressa na tela, observando o atual valor de convergência.

### 3.1.3 Ataque Contra o Modelo Padrão *Net* Implementado em *Flower*

Nessa Subseção, está descrita a primeira execução do experimento. Os ataques descritos anteriormente serão executados contra o modelo "*Net*", implementado por padrão no guia de inicialização rápida em *flower*. Em 3.4 a implementação do modelo é exposta, assim como informações de sua arquitetura que *Breaching* disponibiliza durante a execução do ataque. Em sequência, os resultados de cada ataque são expostos em 3.1.3.1 e 3.1.3.2.

Listagem 3.4: Implementação do modelo *Net*

```
class Net(torch.nn.Module):
"""Model (simple CNN adapted from 'PyTorch: A 60 Minute Blitz')"""

def __init__(self):
    super(Net, self).__init__()
    self.conv1 = nn.Conv2d(3, 6, 5)
    self.pool = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(6, 16, 5)
    self.fc1 = nn.Linear(16 * 5 * 5, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, 10)

def forward(self, x):
```

```
        x = self.pool(torch.nn.functional.relu(self.conv1(x)))
        x = self.pool(torch.nn.functional.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = torch.nn.functional.relu(self.fc1(x))
        x = torch.nn.functional.relu(self.fc2(x))
        return self.fc3(x)

Model architecture loaded with 62,006 parameters and 0 buffers.
Overall this is a data ratio of      20:1
for target shape [1, 3, 32, 32] given that num_queries=1.
```

Esse modelo é uma Rede Neural Convolucional (CNN) bem simples cujo objetivo é classificar imagens, especificamente configurado para o padrão do *dataset CIFAR10*. Tendo como implementação as seguintes camadas:

***conv1*:** camada convolucional que recebe imagens em três canais (RGB), produzindo 6 filtros de saída. Esses filtros detectam padrões na imagem, e produzem um mapa de características (*feature map*) de uma determinada imagem.

***pool*:** camada de redução dimensional, após as transformações realizadas pela camada anterior, essa camada é responsável por reduzir as dimensões espaciais da imagem sendo processada pela metade. Ou seja, o *dataset CIFAR10* tem dimensão 32x32 e após o processamento nessa camada, o dado em uso é reduzido para 16x16.

***conv2*:** segunda camada convolucional, recebe os 6 filtros em *conv1* e aplica novos filtros, totalizando 16.

***fc*:** camadas totais conectadas (*full connected layer*), são os "neurônios" da rede de fato. Combinam as características extraídas anteriormente e são responsáveis por mapear as classes conforme o dado recebido.

***buffers*:** esse modelo não possui camadas de normalização (*bn*), sendo assim, o mesmo não utiliza nenhuma informação em seus *buffers*.

***data ratio*:** proporção entre o número total de parâmetros do modelo (62.006) e o tamanho de dados de entrada: 1 imagem, 3 canais, 32 de largura e 32 de altura ([1, 3, 32, 32]). Significando haver aproximadamente 20 parâmetros para cada valor de entrada que o modelo recebe.

#### 3.1.3.1 Resultados do Ataque *Deep Leakage* Contra o Modelo *Net*

A seguir, essa Subseção irá exibir o resultado da primeira execução do experimento. O Ataque *Deep Leakage* foi usado para reconstruir os gradientes coletados durante o treinamento

em *Flower*, conforme a implementação na Listagem E.1. Na Listagem 3.5, a Figura 3.1 a esquerda é o resultado da reconstrução e a Figura 3.2 a direita, a imagem original. Nessa execução, apenas os parâmetros do modelo antes da geração do gradiente foram utilizados, uma vez que os *buffers* públicos e compartilhados são irrelevantes devido à arquitetura do modelo.

Listagem 3.5: Resultado do ataque *deepleakage*

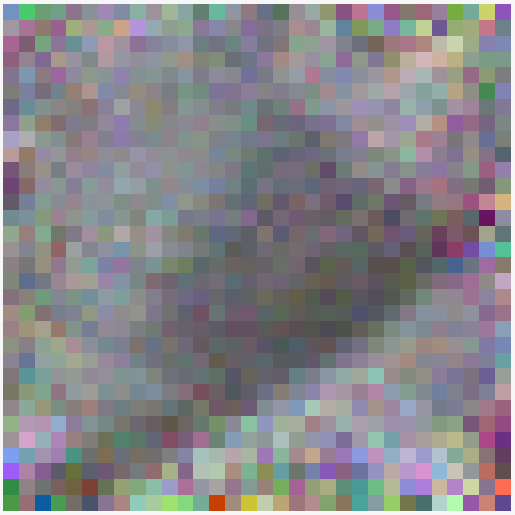

Figura 3.1 – Resultado da reconstrução

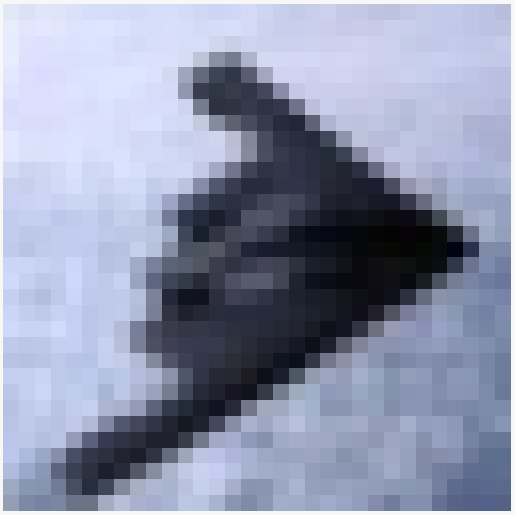

Figura 3.2 – Imagem original

Através das figuras acima na Listagem 3.5, é possível notar que a reconstrução foi parcial. A forma do objeto foi reconstruída, apesar de as cores estarem distantes e ainda existir ruido no resultado.

#### 3.1.3.2 Resultados do Ataque *Beyond Infering* contra o modelo *Net*

Agora, essa Subseção exibe os resultados do ataque *Beyond Infering*, três execuções foram realizadas, utilizando o hiperparâmetro de variação de escala. Na Listagem 3.6, a primeira Figura 3.3 utiliza os valores padrões de *Breaching*. A segunda 3.4 utiliza $1^{-4}$ como valor, expandido para 0.0001. Em seguida, 3.5, utiliza $1^{-3}$, expandido para 0.001. Por fim, 3.6 é a imagem original.

Listagem 3.6: Resultados da reconstrução do ataque *Beyond Infering* sob diferentes configurações de hiperparâmetros.

3.3 — Sem hiperparâmetros | 3.4 — Escala = 0.0001 | 3.5 — Escala = 0.001 | 3.6 — Imagem original

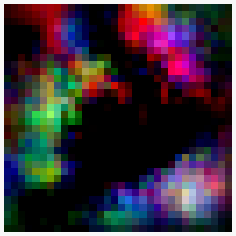
Figura 3.3

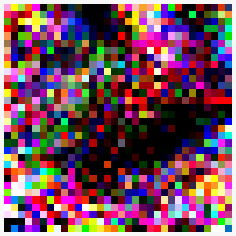
Figura 3.4

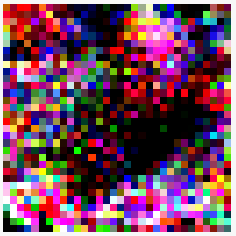
Figura 3.5

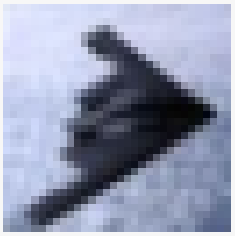
Figura 3.6

Através das figuras acima, se observa que na Figura 3.3, a reconstrução foi parcial, é possível observar a forma da imagem, em preto, e uma alta quantidade de ruído ao redor, ainda que suavizado. Nas figuras 3.4 e 3.5 o resultado de ambos os valores foram piores que o valor padrão, com grande presença de ruído até mesmo no ponto central da imagem.

### 3.1.4 Ataque Contra o Modelo *ConvNet* Implementado em *Breaching*

Nessa Subseção, os ataques *Deep Leakage* e *Beyond Infering* serão executados agora contra o modelo "*ConvNet*", implementado em *Breaching*. É o modelo padrão utilizado durante a simulação pelo framework para o ataque *Deep Leakage* e alguns outros. Como na Seção anterior, o modelo e sua arquitetura estão descritos na Listagem 3.7. Duas variações de configuração são utilizadas, seccionadas por 3.1.4.1 e 3.1.4.2. Os resultados dessas variâncias estão expostos, respectivamente, nas listagens 3.8 e 3.9.

Listagem 3.7: Implementação do modelo *ConvNet*

```
    class ConvNet(torch.nn.Module):
    """ConvNetBN."""

    def __init__(self, width=32, num_classes=10, num_channels=3):
        """Init with width and num classes."""
        super().__init__()
        self.model = torch.nn.Sequential(
            OrderedDict(
                [
                    ("conv0", torch.nn.Conv2d(num_channels, 1 * width, kernel_
↪size=3, padding=1)),
                    ("bn0", torch.nn.BatchNorm2d(1 * width)),
                    ("relu0", torch.nn.ReLU()),
                    ("conv1", torch.nn.Conv2d(1 * width, 2 * width, kernel_
↪size=3, padding=1)),
                    ("bn1", torch.nn.BatchNorm2d(2 * width)),
                    ("relu1", torch.nn.ReLU()),
                    ("conv2", torch.nn.Conv2d(2 * width, 2 * width, kernel_
↪size=3, padding=1)),
                    ("bn2", torch.nn.BatchNorm2d(2 * width)),
```

```
                    ("relu2", torch.nn.ReLU()),
                    ("conv3", torch.nn.Conv2d(2 * width, 4 * width, kernel_
↪size=3, padding=1)),
                    ("bn3", torch.nn.BatchNorm2d(4 * width)),
                    ("relu3", torch.nn.ReLU()),
                    ("conv4", torch.nn.Conv2d(4 * width, 4 * width, kernel_
↪size=3, padding=1)),
                    ("bn4", torch.nn.BatchNorm2d(4 * width)),
                    ("relu4", torch.nn.ReLU()),
                    ("conv5", torch.nn.Conv2d(4 * width, 4 * width, kernel_
↪size=3, padding=1)),
                    ("bn5", torch.nn.BatchNorm2d(4 * width)),
                    ("relu5", torch.nn.ReLU()),
                    ("pool0", torch.nn.MaxPool2d(3)),
                    ("conv6", torch.nn.Conv2d(4 * width, 4 * width, kernel_
↪size=3, padding=1)),
                    ("bn6", torch.nn.BatchNorm2d(4 * width)),
                    ("relu6", torch.nn.ReLU()),
                    ("conv7", torch.nn.Conv2d(4 * width, 4 * width, kernel_
↪size=3, padding=1)),
                    ("bn7", torch.nn.BatchNorm2d(4 * width)),
                    ("relu7", torch.nn.ReLU()),
                    ("pool1", torch.nn.MaxPool2d(3)),
                    ("flatten", torch.nn.Flatten()),
                    ("linear", torch.nn.Linear(36 * width, num_classes)),
                ]
            )
        )

    def forward(self, input):
        return self.model(input)


Model architecture loaded with 733,642 parameters and 1,608 buffers.
Overall this is a data ratio of     239:1
for target shape [1, 3, 32, 32] given that num_queries=1.
```

Esse modelo ainda é uma CNN, mas é possivel notar que sua arquitetura é mais complexa. Possui 671,636 parâmetros a mais que o modelo *Net* utilizado no experimento 3.4 anterior. É possivel notar que esse modelo possui oito camadas de normalização (BN), ou seja, diferente do modelo anterior, esse modelo **é capaz de guardar informações em seus *buffers***. De maneira geral a disposição das camadas é a seguinte:

**camadas *conv*:** possui oito camadas convolucionais, diferente do modelo anterior, o numero de cnais e *feature maps* são expandidos mais profundamente, com uma largura (*width*)

aplicada ao numero de filtros que o modelo utiliza.

**camadas *pool*:** possui duas camadas de redução dimensional, aplicando dessa vez uma redução 3x3, ou seja, um terço da imagem original é redimensionada.

**camadas *ReLU*:** essas camadas introduzem transformações não lineares nos dados em uso do modelo, isso permite que o modelo aprenda padrões mais complexos que o anterior, que aplica apenas transformações lineares através das camadas convolucionais.

**camadas *BN*:** aplica normalizações nas ativações de cada camada, ou seja, calcula média e variancia dessas ativações e aplica escala e deslocamento a cada uma delas. Isso estabiliza o treinamento para *batchs* maiores e pode em alguns casos acelerar a convergência. As transformações que essas camadas aplicam são os dados salvos nos *buffers* do modelo.

#### 3.1.4.1 Resultados dos Ataques *Deep Leakage* e *Beyond Infering* Contra o Modelo *ConvNet* em Modo de Treino

Essa Subseção apresenta os resultados de diversas execuções dos ataques *Deep Leakage* e *Beyond Inferring* contra o modelo *ConvNet* descrito anteriormente. Como ressaltado, esse modelo possui camadas de normalização de *batch*, isso consequentemente resulta em **diferenças** entre gradientes coletados em modo de treino e modo de avaliação.

Gradientes gerados em modo de treino utilizam essa camada para normalizar as ativações do modelo. Isso não acontece quando o modelo é treinado em modo de avaliação. Consequentemente, para *batchs* de tamanho um, não faz muito sentido que essas camadas estejam ativas, uma vez que a normalização seria aplicada em apenas uma imagem, defasando detalhes que podem ser utilizados pelas *feature map's* do modelo.

Sendo assim, a Listagem 3.8 abaixo executa os ataques *Deep Leakage* e *Beyond Infering* contra o modelo em modo de **treino**. As Figuras 3.7 e 3.10 são resultados coletados em execuções que não utilizam os *buffers* compartilhados durante o treinamento. As figuras 3.8 e 3.11 são resultados coletados utlizando tais *buffers*. Por fim as figuras 3.9 e 3.12 são as imagens originais, identicas devido ao gradiente utilizado nos ataques ser o mesmo.

Em relação a utilização ou não desses *buffers* compartilhados, essa opção é instrumentada através das linhas 25 e 29 das modificações apresentadas na Listagem 3.1 no inicio do experimento na Subseção 3.1.1. Essas linhas são, respectivamente, os argumentos *–shared-buffers-path* e *–provide-shared-buffers*, que quando presentes na evocação do *script*, adicionam a informação a instância do modelo utilizado durante o ataque.

Por fim, nessa rodada, em relação ao ataque *Beyond Infering*, os dois ataques executados utilizam o hiperparâmetro de escala padrão de *Breaching*.

Listagem 3.8: Comparação dos resultados executados em relação ao modelo *ConvNet* em **modo de treino**

(1) A direita — Sem *buffers* compartilhados | (2) No centro — Com *buffers* compartilhados | (3) A esquerda — Imagem original

Ataque *Deep Leagake*

Figura 3.7

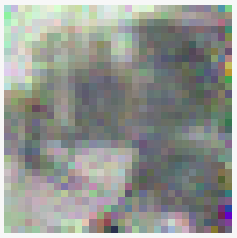

Figura 3.8

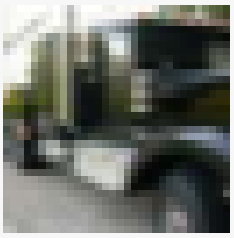

Figura 3.9

Ataque *Beyond Infering*

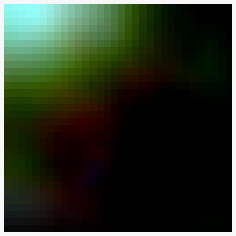

Figura 3.10

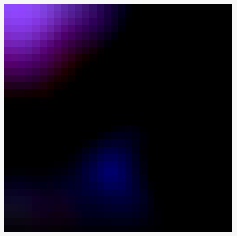

Figura 3.11

Figura 3.12

Através das figuras acima, é possível concluir de maneira geral que tanto os resultados da execução com *buffers* compartilhados (figuras 3.7 e 3.10) quanto sem (figuras 3.8 e 3.11) tiveram performance semelhante entre si. Por mais que os *buffers* compartilhados durante o treinamento estejam disponíveis por conta da arquitetura do modelo possuir camadas de normalização, essas camadas não são necessárias. Uma vez que o tamanho do *batch* desse experimento é de apenas uma imagem, não existe necessidade de aplicar a normalização, dado que o objetivo desse procedimento é normalizar a distribuição dos *pixels* de diferentes imagens de um *batch* durante as ativações. Ou seja, para *batchs* de uma única imagem, faz mais sentido que o treinamento seja conduzido em **modo de avaliação**.

Em relação ao ataque *Deep Leakage* é possível notar que os *pixels* mais claros do caminhão foram normalizados em relação aos *pixels* claros das redondezas na Figura 3.8 em relação a Figura 3.7. Ou seja, as informações dos *buffers* públicos coletados, quando acrescentados ao ataque, reconstrõem a imagem com a normalização que foi aplicada pelas camadas BN durante a ativação. Os resultados desse ataque contra o modelo *ConvNet* (figuras 3.7 e 3.8) apresentam uma distribuição de *pixels* mais semelhante à imagem original do que os resultados (Figura 3.1) do experimento anterior na Subseção 3.1.3.2, utilizando o modelo *Net*. Uma hipótese plausivel

que explique esse comportamento está relacionada com a **quantidade de parâmetros** (*map feature's*) disponíveis no modelo. Conforme detalhado no Capítulo dos fundamentos na Seção 2.2.2 esse ataque utiliza apenas os dados públicos compartilhados entre cliente e servidor durante o treinamento. Uma vez que o modelo possui uma quantidade **maior** de parâmetros, isso significa que esse ataque possui mais informações para utilizar durante a reconstrução dos dados originais.

Em relação ao ataque *Beyond Infering*, os resultados obtidos nessa rodada (figuras 3.10 e 3.11) são menos eficazes que os resultados anteriores na Subseção 3.1.3.2 (figuras 3.3, 3.4 e 3.5). Por algum motivo a GAN não consegue antigir uma convergencia equivalente ao modelo *Net* ainda que a quantidade de *map feature's* e parâmetros do modelo sejam maiores. Isso parece ter a ver com a presença das camadas de normalização BN, e será investigado mais a fundo nas proximas subseções.

#### 3.1.4.2 Resultados do Ataque *Beyond Infering* Contra o Modelo *ConvNet* em Modo de Avaliação

A partir dessa Subseção, apenas o ataque *Beyond Infering* será executado. Essa subssesão em especifico, ainda utiliza o modelo *ConvNet* porém com gradientes coletados em **modo de avaliação**. O ataque *Deep Leakage*, para todos os modelos e configurações aplicadas a partir desse ponto sempre atingem uma boa convergencia, de forma que seus resultados não são mais interessantes como objeto de análise. De maneira geral, é possivel concluir o ataque *Deep Leakage*, tendo como base a utilização das informações publicas compartilhadas durante o treinamento, sempre terá uma boa convergência quando a quantidade de parâmetros e *feature map's* do modelo for grande o suficiente.

Já em relação ao ataque *Beyond Infering*, serão exploradas as condições necessárias para que o mesmo atinja uma convergencia performatica. Sendo assim, ainda explorando a hipótese em relação a presença das camadas de normalização BN, essa Seção executa o ataque *Beyond Infering* contra gradientes coletados em um treinamento em modo de avaliação, de forma que não exista normalização durante as camadas de convergencia e transformações não lineares do modelo *ConvNet*.

Na Listagem 3.9 abaixo, serão expostos os resultados dessa configuração, assim como as conclusões que esses resultados permitem inferir. Da mesma forma como na Subseção 3.1.3.2 utilizando o modelo *Net*, quatro figuras estão disponíveis. As três primeiras figuras (3.13, 3.14 e 3.15), relativas a reconstrução utilizando variações do hiperparametro de regularização de escala e a ultima Figura 3.16 sendo a imagem original. Para todas as reconstruções, como nenhuma normalização está sendo aplicada, os *buffers* públicos são irrelevantes, sendo assim, esses dados não foram compartilhados com o modelo durante o ataque.

Listagem 3.9: Execução do ataque *Beyond Infering* contra o gradiente gerado em **modo de avaliação**

3.13 — Sem hiperparâmetros | 3.14 — Escala = 0.0001 | 3.15 — Escala = 0.001 | 3.16 — Imagem original

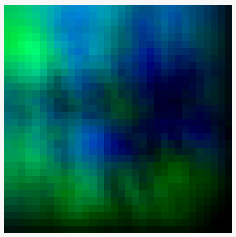

Figura 3.13

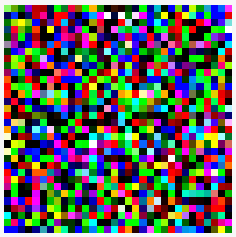

Figura 3.14

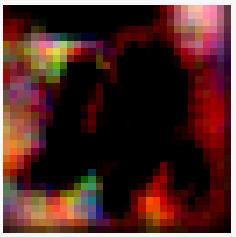

Figura 3.15

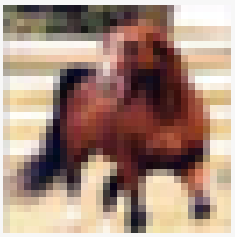

Figura 3.16

Mesmo em modo de avaliação, o ataque *Deep Leekage* não conseguiu reconstruir os gradientes. Diferente dos resultados do modelo *Net* na subessão 3.1.3.2, uma convergência aproximada foi atingida utilizando hiperparametro de escala em $1^{-3}$ (Figura 3.15) ao passo que o ataque contra o gradiente em *Net* atingiu um resultado semelhante utilizando os valores padrões em *Breaching* (Figura 3.3). Isso é esperado, uma vez que o exemplo desse ataque em *Breaching* (GEIPING, 2021b), só atinge convergencia quando a escala é aplicada. Ainda assim, o resultado apresentado aqui é pior em relação ao resultado exposto no exemplo. De qualquer maneira, o resultado foi melhor em relação a rodada passada 3.8, em que os gradientes foram gerados em modo de treino. Ou seja, a GAN conseguiu atingir uma convergencia melhor quando as camadas de normalização não foram utilizadas nas ativações.

### 3.1.5 Ataque contra os modelos *Net* e *ConvNet* utilizando valores de transformação customizados

Uma vez que existe uma diferença entre os resultados expostos na Subseção 3.9 passada e os apresentados em *Breaching* utilizando o modelo *ConvNet* uma análise comparativa das implementações foi realizada.

Dessa comparação, é possível notar a implementação em *Flower* (FLOWER, 2020e) utiliza valores de transformação dos *datapoints* **diferentes** dos que *Breaching* utiliza (GEIPING, 2021c).

- *Flower* utiliza os valores (0.5, 0.5, 0.5) para cada canal para *std* e *mean*.
- *Breaching* utiliza os valores (0.491, 0.482, 0.446) para cada canal em *mean*, que estão próximos dos valores em *flower*.
  - já os valores de *std* para cada canal em *Breaching* são (0.24703224003314972, 0.24348513782024384 e 0.26158785820007324), um pouco menos que a metade dos valores padrões em *Flower*.

Sendo assim, essa Subseção irá explorar os efeitos que essas transformações tem quando aplicadas aos gradientes coletados durante o treinamento em *Flower*. Ambos os modelos *Net* e *ConvNet* serão utilizados na próxima rodada de experimentos. O treinamento foi conduzido com o modelo em **modo de avaliação** e os *buffers* não foram compartilhados durante os ataques para nenhum dos modelos.

A Listagem 3.10 abaixo apresenta o trecho em que as modificações são aplicadas em *Flower*:

Listagem 3.10: Modificação na transformação do *datapoint* no *script task.py*

```
pytorch_transforms = Compose([ToTensor(), Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5))])
```

```
pytorch_transforms = Compose([
    ToTensor(),
    Normalize(
        mean=[0.4914672374725342, 0.4822617471218109, 0.4467701315879822],
        std=[0.24703224003314972, 0.24348513782024384, 0.26158785820007324]
    )
])
```

Em sequência, a próxima Listagem 3.11 exibe os resultados dessas transformações em relação ao modelo *Net*:

Listagem 3.11: Resultados da reconstrução do ataque *Beyond Inferring* com transformação customizada sob diferentes configurações de hiperparâmetros contra um gradiente coletado utilizando o modelo *Net*

(1) — Sem hiperparâmetros | (2) — Escala = 0.0001 | (3) — Escala = 0.001 | (4) — Imagem original

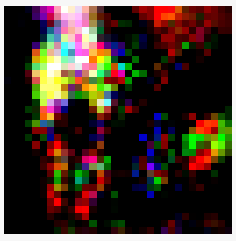

Figura 3.17

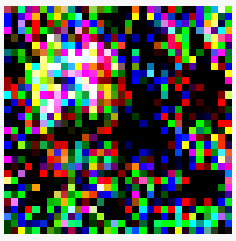

Figura 3.18

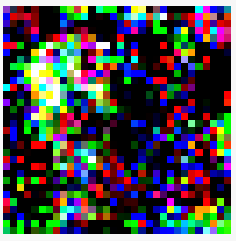

Figura 3.19

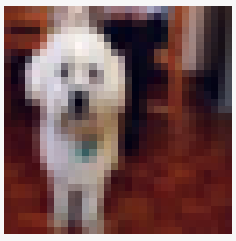

Figura 3.20

Comparando com os resultados do primeiro experimento com o modelo *Net* em 3.1.3.2: a Figura 3.17 consegue reconstruir a cor mas apresenta ruidos na forma, ao passo que o fundo fica defasado. Já a Figura 3.3, reconstrói a forma do avião praticamente sem ruídos. Tanto os pares 3.18 e 3.19 possuem grande taxa de ruído, da mesma forma que as figuras 3.4 e 3.5 no experimento passado. De maneira geral, a transformação não imbuiu nenhuma melhoria nos resultados obtidos, aparentando, na verdade ter piorado a performance do ataque.

Finalmente 3.1.5 apresenta os resultados da transformação em relação ao modelo *Conv-*

*Net*:

Listagem 3.12: Resultados da reconstrução do ataque *Deep Leakage* com transformação customizada sob diferentes configurações de hiperparâmetros contra um gradiente coletado utilizando o modelo *ConvNet*

3.21 — Sem hiperparâmetros | 3.22 — Escala = 0.0001 | 3.23 — Escala = 0.001 | 3.24 — Imagem original

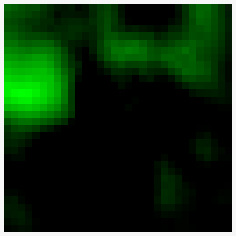

Figura 3.21

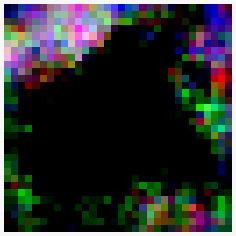

Figura 3.22

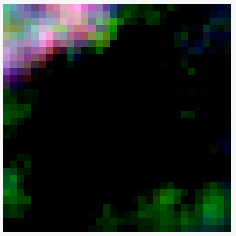

Figura 3.23

Figura 3.24

Em relação à execução passada, com os valores de transformação padrão que *Flower* aplica nos *datapoints*, diferente dos resultados das figuras 3.13, 3.14 e 3.15, os resultados dessa roda, representados por 3.21, 3.22 e 3.23 possuem bem menos ruidos. A melhor taxa de convergência foi a Figura 3.22 com escala $1^{-4}$ ao passo que os resultados com transformação padrão tiveram convergência melhor com a Figura 3.15 em escala $1^{-3}$. De maneira geral, a GAN em *Beyond Inferring* conseguiu reduzir os ruidos, mas os resultados apresentados ainda são piores que os disponíveis em *Breaching* (GEIPING, 2021b). As transformações parecem que auxiliam e influenciam nos valores de escala, mas apenas estas não são suficientes para que o ataque *Beyond Inferring* apresente uma boa convergência.

### 3.1.6 Ataque contra o *ConvNet* instanciado com o dobro de parâmetros

Aprofundando as comparações, foi observado que durante as simulações em *Breaching* o modelo *ConvNet* é instanciado com a variavel *width* com o dobro do valor padrão definido em sua implementação. Sendo assim, nessa Subseção, a proxima rodada de experimentos irá avaliar o desempenho do ataque *Deep Leakage* utilizando o dobro da largura durante a definição do modelo. Essa variavel pode ser observada durante a descrição da arquitetura do modelo na Subseção 3.1.4, onde é possivel notar que seu valor é multiplicado por dois e depois por quatro na definição das camadas convolucionais.

Observando a listagem abaixo, o efeito que uma largura maior produz no modelo é de uma quantidade de parâmetros e *buffers* **maiores** presentes no modelo:

Listagem 3.13: Comparação de valores de parâmetros do modelo *Convnet* instanciados com largura 32 e 64

Largura 32

```
Model architecture loaded with 733,642 parameters and 1,608 buffers.
Overall this is a data ratio of     239:1
for target shape [1, 3, 32, 32] given that num_queries=1.
```

Largura 64

```
Model architecture loaded with 2,904,970 parameters and 3,208 buffers.
Overall this is a data ratio of     946:1
for target shape [1, 3, 32, 32] given that num_queries=1.
```

Sendo assim, as listagens a seguir apresentam os resultados do ataque *Deep Leakage* contra o modelo *ConvNet* utilizando o valor 64 em sua largura. Para ambas as listagens apresentadas, os gradientes foram coletados num treinamento realizado em **modo de avaliação** e os ataques foram executados sem compartilhar os *buffers* do treinamento.

Listagem 3.14: Resultados da reconstrução do ataque *Deep Leakage* **sem transformação customizada** sob diferentes configurações de hiperparâmetros e **largura do modelo com valor em 64**

3.25 — Sem hiperparâmetros | 3.26 — Escala = 0.0001 | 3.27 — Escala = 0.001 | 3.28 — Imagem original

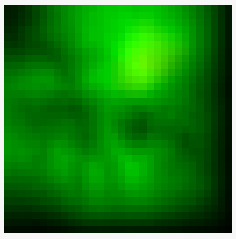

Figura 3.25

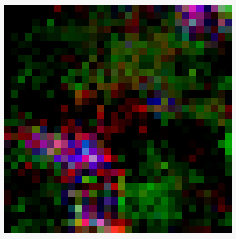

Figura 3.26

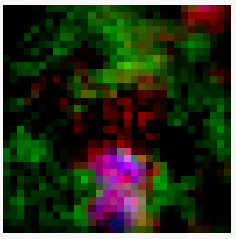

Figura 3.27

Figura 3.28

Em relaçao as figuras acima:

- para todas as Figuras 3.25, 3.26 e 3.27, a presença de ruido parece ser menor que nos resultados passados, ainda assim, a forma do animal parece estar distorcida, e suas cores não foram convergidas.
- até a cor verde, predominante, está em um tom mais escuro que a imagem original, essa configuração parece não ter surtido um efeito interessante para esse *datapoint*.

Listagem 3.15: Resultados da reconstrução do ataque *Deep Leakage* **com transformação customizada** sob diferentes configurações de hiperparâmetros e **largura do modelo com valor em 64**

3.29 — Sem hiperparâmetros | 3.30 — Escala = 0.0001 | 3.31 — Escala = 0.001 | 3.32 — Imagem original

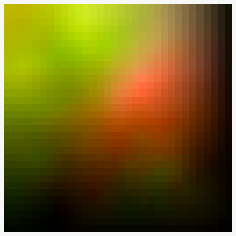

Figura 3.29

Figura 3.30

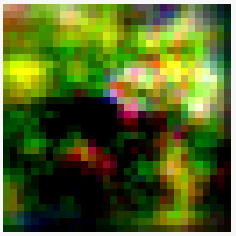

Figura 3.31

Figura 3.32

Os resultados apresentados nessa execução, com largura 64 e transformação customizada apresentam um resultado melhor em relação a todas as outras passadas. A Figura 3.29 com hiperparâmetro padrão, não consegue recuperar as formas corretamentes, mas apresenta uma distribuição de cor semelhante a imagem original. As Figuras 3.30 e 3.31 se aproximam tanto a forma quanto a distribuição das cores. É possivel concluir que a GAN foi mais eficaz com a quantidade de parâmetros maiores e que uma transformação adequada ajuda na convergencia quando os hiperâmetros são ajustados. Ainda assim, os resultados apresentados ainda são piores que os exibidos em *Breaching* (GEIPING, 2021b).

### 3.1.7 Ataque Contra o Modelo *ConvNet Small*

Essa Subseção 3.1.7, assim como a próxima 3.1.8, introduzem dois outros modelos disponíveis em *Breaching*. Ambos não possuem camadas de normalização, e apresentam uma quantidade **massiva** de parâmetros. Ainda assim, em termos de arquitetura, ambos são menos complexos que o modelo *ConvNet* e mais complexos que o modelo *Net*. O modelo *ConvNet Small* abaixo é utilizado em *Breaching* com largura tamanho 256, quatro vezes a mais que a utilizada por *ConvNet* no experimento anterior.

Os experimentos descritos nessa Subseção seguem o mesmo padrão da anterior, uma rodada utilizando os valores de transformação padrão e outra utilizando os valores definidos em *Breaching*. Para ambos os casos, os gradientes foram coletados durante um treinamento realizado em **modo de avaliação**.

Listagem 3.16: Implementação do modelo *ConvNet Small*

```
class ConvNetSmall(torch.nn.Module):
  """ConvNet without BN."""
```

```
    def __init__(self, width=32, num_classes=10, num_channels=3):
        """Init with width and num classes."""
        super().__init__()
        self.model = torch.nn.Sequential(
            OrderedDict(
                [
                    ("conv0", torch.nn.Conv2d(num_channels, 1 * width, kernel_
↪size=3, padding=1)),
                    ("relu0", torch.nn.ReLU()),
                    ("conv1", torch.nn.Conv2d(1 * width, 2 * width, kernel_
↪size=3, padding=1)),
                    ("relu1", torch.nn.ReLU()),
                    ("conv2", torch.nn.Conv2d(2 * width, 4 * width, kernel_
↪size=3, stride=2, padding=1)),
                    ("relu2", torch.nn.ReLU()),
                    ("pool0", torch.nn.MaxPool2d(3)),
                    ("conv3", torch.nn.Conv2d(4 * width, 4 * width, kernel_
↪size=3, stride=2, padding=1)),
                    ("relu3", torch.nn.ReLU()),
                    ("pool1", torch.nn.AdaptiveAvgPool2d(1)),
                    ("flatten", torch.nn.Flatten()),
                    ("linear", torch.nn.Linear(4 * width, num_classes)),
                ]
            )
        )

    def forward(self, input):
        return se,lf.model(input)
```

Sobre a arquitetura do modelo *ConvNet Small*

- possui uma camada convolucional a mais que o modelo *Net*, ao passo que quatro a menos que o modelo *ConvNet*.
- não possui camadas de normalização como o modelo *Net*.
- possui camadas de transformação não lineares, na mesma proporção de camadas convolucionais.

Listagem 3.17: Comparação das arquiteturas dos modelos *Net*, *ConvNet* e *ConvNet Small*

*Net*

```
Model architecture loaded with 62,006 parameters and 0 buffers.
Overall this is a data ratio of      20:1
for target shape [1, 3, 32, 32] given that num_queries=1.
```

*ConvNet*

Largura 32

```
Model architecture loaded with 733,642 parameters and 1,608 buffers.
Overall this is a data ratio of     239:1
for target shape [1, 3, 32, 32] given that num_queries=1.
```

Largura 64

```
Model architecture loaded with 2,904,970 parameters and 3,208
buffers.
Overall this is a data ratio of     946:1
for target shape [1, 3, 32, 32] given that num_queries=1.
```

*ConvNet Small com largura em **256***

```
Model architecture loaded with 15,355,402 parameters and 0 buffers.
Overall this is a data ratio of    4999:1
for target shape [1, 3, 32, 32] given that num_queries=1.
```

Essa configuração é a que mais contem parâmetros e *map feature's* até então. Observe resultados abaixo:

Listagem 3.18: Resultados do ataque *Deep Leakage* contra o modelo *ConvNet Small* **sem transformação customizada** sob diferentes configurações de hiperparâmetros

3.33 — Sem hiperparâmetros | 3.34 — Escala = 0.0001 | 3.35 — Escala = 0.001 | 3.36 — Imagem original

Figura 3.33

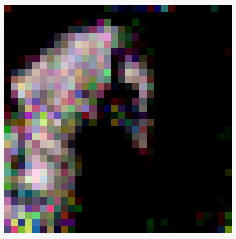

Figura 3.34

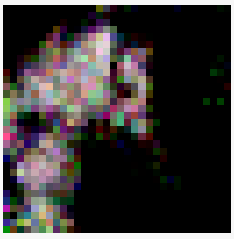

Figura 3.35

Figura 3.36

- as três figuras 3.33, 3.34 e 3.35 recuperam parcialmente a forma e coloração do cavalo, mas não o cenário no fundo.
- o melhor resultado é a segunda Figura 3.34 com escala $1^{-3}$ e o pior, a primeira Figura 3.33 com escala padrão.
- quando comparada com o experimento passado, essa rodada de execução teve resultados **melhores** utilizando os valores de transformação padrões em *Flower*.

Listagem 3.19: Resultados do ataque *Deep Leakage* contra o modelo *ConvNet Small* **com transformação customizada** sob diferentes configurações de hiperparâmetros

3.37 — Sem hiperparâmetros | 3.38 — Escala = 0.0001 | 3.39 — Escala = 0.001 | 3.40 — Imagem original

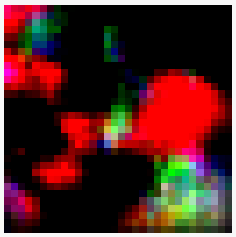

Figura 3.37

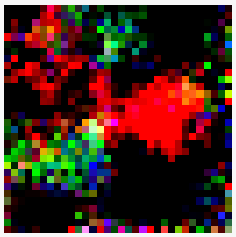

Figura 3.38

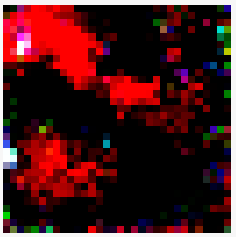

Figura 3.39

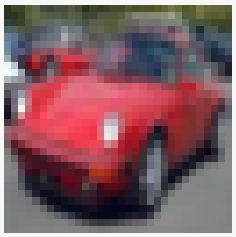

Figura 3.40

- nas figuras 3.37, 3.38 e 3.39 as cores do carro foram recuperadas, mas a convergência não foi boa.
- quando comparada com o experimento passado, essa rodada de execução teve resultados **piores** utilizando os valores de transformação em *Breaching*.

### 3.1.8 Ataque Contra o Modelo *ConvNet Beyond*

Por fim, a última rodada de execução utiliza o modelo descrito na própria publicação do ataque *Beyond Inferring* (WANG et al., 2019) e implementado em *Breaching*. Da mesma forma que os experimentos anteriores, serão executadas duas rodadas, utilizando as transformações padrões em *flower* e outra utilizando as de *Breaching*. Os gradientes foram coletados durante um treinamento em **modo de avaliação** e os *buffers* do treinamento são irrelevantes nesse cenário devido ao modelo não possuir camadas BN.

Listagem 3.20: Implementação e metadados do modelo *ConvNet Beyond*

```
    model = torch.nn.Sequential(
                OrderedDict(
                    [
                        ("conv1", torch.nn.Conv2d(channels, 32, 3, stride=2,
padding=1)),
                        ("relu0", torch.nn.LeakyReLU()),
                        ("conv2", torch.nn.Conv2d(32, 64, 3, stride=1,
padding=1)),
                        ("relu1", torch.nn.LeakyReLU()),
                        ("conv3", torch.nn.Conv2d(64, 128, 3, stride=2,
padding=1)),
                        ("relu2", torch.nn.LeakyReLU()),
                        ("conv4", torch.nn.Conv2d(128, 256, 3, stride=1,
padding=1)),
```

```
                        ("relu3", torch.nn.LeakyReLU()),
                        ("flatt", torch.nn.Flatten()),
                        ("linear0", torch.nn.Linear(12544, 12544)),
                        ("relu4", torch.nn.LeakyReLU()),
                        ("linear1", torch.nn.Linear(12544, classes)),
                        ("softmax", torch.nn.Softmax(dim=1)),
                    ]
                )
            )

Model architecture loaded with 206,047,306 parameters and 0 buffers.
Overall this is a data ratio of   67073:1
for target shape [1, 3, 32, 32] given that num_queries=1.
```

Em relação a arquitetura desse modelo:

- possui o maior número de parâmetros e *feature map's* entre todos os modelos testados, implementados por valores fixos nas camadas convolucionais e lineares.
- possui numero de camadas convolucionais e não lineares semelhantes ao modelo *ConvNet Small* apresentado na Subseção passada.
- utiliza *LeakyReLU* ao invés de *ReLU*, isso significa que esse modelo aceita gradientes **negativos**.
- utiliza um *softmax* na camada final, utilizado geralmente para classificação multi-classe.

Finalmente, os últimos resultados do experimento:

Listagem 3.21: Resultados da reconstrução do ataque *Deep Leakage* **sem transformação customizada** sob diferentes configurações de hiperparâmetros

3.41 — Sem hiperparâmetros | 3.42 — Escala = 0.0001 | 3.43 — Escala = 0.001 | 3.44 — Imagem original

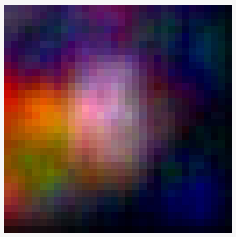

Figura 3.41

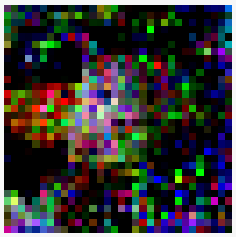

Figura 3.42

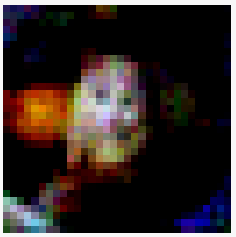

Figura 3.43

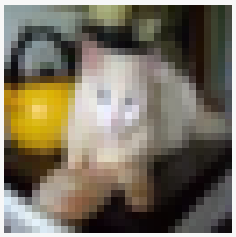

Figura 3.44

Listagem 3.22: Resultados da reconstrução do ataque *Deep Leakage* **com transformação customizada** sob diferentes configurações de hiperparâmetros

3.45 — Sem hiperparâmetros | 3.46 — Escala = 0.0001 | 3.47 — Escala = 0.001 | 3.48 — Imagem original

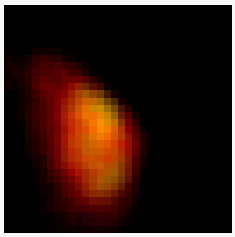

Figura 3.45

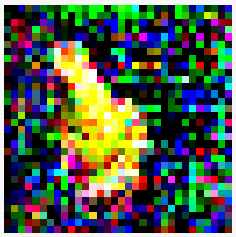

Figura 3.46

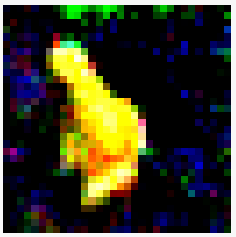

Figura 3.47

Figura 3.48

Para ambos os casos, os resultados são semelhantes:

- as figuras com hiperparâmetro de escala padrão 3.41 e 3.45 são as menos efetivas, sendo a primeira a melhor, reconstruindo o gato e até mesmo a cor amarela do fundo parcialmente.
- as figuras com escala $1^{-4}$ 3.42 e 3.46 reconstroem a forma central apresentando ruídos, ainda que 3.42 apresente resultado melhor.
- as figuras com escala $1^{-3}$ 3.43 e 3.47 apresentam o melhor resultado, sendo 3.43 mais eficiente pela presença de mais detalhes.

### 3.1.9 Análise final da eficácia do ataque *Beyond Infering*

O intuito dessa Seção é concentrar os resultados demonstrados nas subseções anteriores em relação ao ataque *Deep Leakage* e as diferentes configurações utilizadas. De maneira geral, existe uma disparidade entre os resultados, e as hipóteses se contradizem. Uma vez que os gradientes são coletados em *flower*, os *datapoints* que cada cliente utiliza na treinamento são atribuidos de maneira aleatoria. Isso torna a comparação ineficaz, uma vez que as reconstruções analisadas diferem a cada nova rodada.

**Modelo *Net*:** os resultados apresentados pelas figuras 3.3, 3.4, 3.5 com transformação padrão e os resultados apresentados pelas figuras, 3.17, 3.18 e 3.19 com transformação customizada tiveram um resultado semelhante: esse modelo tem maior eficácia quando o hiperparâmetro de escala se encontra no valor padrão. Também, de maneira geral, não parece que a mudança de transformação ofereceu melhorias, e nesse caso específico, parece ter surtido efeito negativo.

**Modelo *ConvNet*:** os resultados são variados, o melhor caso de uso foi durante a rodada em que o modelo utilizou largura em valor 64 com as transformações disponíveis em *Breaching* na

Subseção 3.1.6, especificamente as figuras 3.29, 3.30 e 3.31, que se aproximam de fato com o exemplo publicado em (GEIPING, 2021b).

**Modelo *ConvNet Small*:** o melhor resultado da experimentação desse modelo foi utilizando as transformações padrões em *Flower* através da Figura 3.38.

**Modelo *ConvNet Beyond*:** o último modelo apresentou bons resultados utilizando ambas as transformações, em especial, as figuras 3.42 e 3.43.

Em suma, através dos resultados obtidos é possível reconhecer que o ataque funciona, mas não é possível extrair dos casos um padrão que racionalize as condições ideais para que a GAN tenha uma boa convergência.

## 3.2 Modificações, Melhorias e Correções realizadas em *Breaching*

Na condução desse estudo, durante o manuseio do *framework* diversas modificações foram propostas, desde a correção de erros de execução, como incrementos na documentação e melhorias em diversos trechos do repositório. Dessa forma, esse estudo tem impacto nos usuários do *Framework*. Algumas das modificações introduzidas serão descritas nas subseções a seguir.

### 3.2.1 Correção no Módulo de Métricas de Reconstrução de *Datasets* visuais

Um erro ocorria ao executar o *script simulate_breach.py*, utilizando as configurações padrões com sobrecarga para o caso de uso do *CIFAR10*:

Listagem 3.23: Comando de execução *simulate_breach.py*

```
$ python simulate_breach.py dryrun=True case=1_single_image_small
```

Na linha 51 do método *report* no arquivo *analysis.py* é atribuído o valor da chave *cfg_case.data.vocab_size* à variável *maxlenght*:

Listagem 3.24: Trecho do método *report* onde o erro ocorre

```
47    if reconstructed_user_data["labels"] is not None:
48        test_label_acc = count_integer_overlap(
49            reconstructed_user_data["labels"].view(-1),
50            true_user_data["labels"].view(-1),
51            maxlength=cfg_case.data.vocab_size,
52        ).item()
```

Essa chave, porem, não existe na estrutura *cfg*. Ao realizar uma busca de ocorrência da palavra *maxlenght* dentro do diretório *breaching/config*, obtém-se o seguinte resultado:

Listagem 3.25: Busca de ocorrência da palavra *vocab_size*

```
$ grep -r "vocab_size" breaching/config
breaching/config/case/10_causal_lang_training.yaml:  vocab_size: 50257
breaching/config/case/9_bert_training.yaml:  vocab_size: 30522
breaching/config/case/data/wikitext.yaml:vocab_size: 50257
breaching/config/case/data/stackoverflow.yaml:vocab_size: 50257
breaching/config/case/data/shakespeare.yaml:vocab_size: 50257
breaching/config/case/data/random-tokens.yaml:vocab_size: 50257
breaching/config/case/data/cola.yaml:vocab_size: 30522
```

Através dessa saída é possível notar que a chave *vocab_size* está presente nos arquivos de configuração dos *datasets* textuais, mas não nos visuais. O parâmetro, — nesse modulo em específico —, é usado para avaliar quão bem os rótulos (*labels*) reconstruídos pelo ataque correspondem aos rótulos reais do *dataset*. Essa métrica é valida tanto para *datasets* visuais quanto textuais. No caso do *dataset CIFAR10*, o valor dessa variável deve ser "10"; no caso do *dataset CIFAR100*, "100". Quando questionado quanto a possibilidade de tornar esse parâmetro opcional para a modalidade visual, o autor respondeu que, na verdade, a chave equivalente dessa modalidade é nomeada como *classes*. Alterando o comando em 3.25 obtemos o seguinte resultado:

Listagem 3.26: Busca de ocorrência da palavra *classes*

```
$ grep -r "classes" breaching/config
breaching/config/case/data/CIFAR10.yaml:classes: 10
breaching/config/case/data/CIFAR100.yaml:classes: 100
breaching/config/case/data/cola.yaml:classes: 2
breaching/config/case/data/ImageNetAnimals.yaml:classes: 397
breaching/config/case/data/Birdsnap.yaml:classes: 500
breaching/config/case/data/ImageNet.yaml:classes: 1000
breaching/config/case/data/TinyImageNet.yaml:classes: 200
```

A partir da *issue*, a primeira correção foi realizada no repositório, através do *Pull Request 17*:

Listagem 3.27: *Diff* da correção implementada pelo *Pull Request 17*

```
]

47      if reconstructed_user_data["labels"] is not None:
48  +       if metadata["modality"] ==  "text":
49  +           maxlength = cfg_case.data.vocab_size
50  +       else:
51  +           maxlength = cfg_case.data.classes
```

```
52          test_label_acc = count_integer_overlap(
53          reconstructed_user_data["labels"].view(-1),
54          true_user_data["labels"].view(-1),
51  -       maxlength=cfg_case.data.vocab_size,
55  +       maxlength=maxlength,
```

Caso a modalidade do *dataset* seja do tipo *text*, é utilizada a chave *vocab_size*, caso contrário, a chave *classes*. Após essa mudança, o *script* executa corretamente e consegue prover as métricas de análise, conforme no apêndice D.

### 3.2.2 Correção no Modulo de Métricas de Reconstrução com Aceleração Gráfica

A biblioteca *PyTorch* suporta aceleração gráfica para o cálculo de tensores e o *framework Breaching* está configurado para habilitar esse recurso durante as simulações. Para tal, basta sobrecarregar o parâmetro *case.impl.enable_gpu_acc*:

Listagem 3.28: Comando de execução com aceleração gráfica

```
$ python simulate_breach.py dryrun=True case=1_single_image_small case.impl.
↪enable_gpu_acc=True
```

Esse comando produziu um erro que provem da função *tensor_to_numpy* implementado na biblioteca *PyTorch* e aturado no método *dump_metrics*; linha 287 do arquivo *breaching/utils.py*. Esse método está disponível na íntegra no anexo C e é responsável por escrever as métricas do ataque no arquivo de *output* conforme descrito na Seção de análise do *script simulate_breach.py* no Capítulo anterior (2.3.3). Observe o trecho:

Listagem 3.29: Trecho de interesse do método *dump_metrics*; anexo C

```
283     for metric, val in metrics.items():
284         try:
285             sanitized_metrics[metric] = np.asarray(val).item()
286         except ValueError:
287             sanitized_metrics[metric] = np.asarray(val).tolist()
```

Esse trecho itera um dicionário de items do objeto *metrics*, desestruturado-o em *metric* (chave) e *val* (valor). É possível entender o estado dessas variáveis através da depuração do método com o módulo *pdb*:

Listagem 3.30: Depuração da função *dump_metrics* (Listagem 3.29)

```
    $ python -m pdb simulate_breach.py dryrun=True case=1_single_image_small
case.impl.enable_gpu_acc=True

    > /home/user/breaching/simulate_breach.py(1)<module>()
    -> """
    (Pdb) b breaching/utils.py:283
    Breakpoint 1 at /home/user/breaching/breaching/breaching/utils.py:283
    (Pdb) r
    > /home/user/breaching/breaching/utils.py(283)dump_metrics()
    -> for metric, val in metrics.items():
    (Pdb) p metrics.items()
    dict_items([('mse', 0.1150408387184143), ('psnr', 9.3914794921875), ('lpips',
0.3313041627407074), ('rpsnr', tensor(nan)), ('ssim', tensor(nan)), ('max_ssim',
tensor(nan)), ('max_rpsnr', tensor(nan)), ('order', tensor([0], device='cuda:0
↪')), ('IIP-pixel', 0.0), ('feat_mse', 7.010284264197253e-08), ('parameters',
2904970), ('label_acc', 1.0)])
```

Através da depuração é possível perceber que *val*, a medida que é iterado, pode ser um conjunto de *floats* ou tensores. Quando do tipo tensor, esse pode ser do tipo *tensor(nan)* ou do tipo *tensor([0], device='cuda:0')*. Somente tensores do primeiro tipo podem ser convertidos pelo método *np.asarray*, mas quando são do tipo *CUDA*, é necessário transformá-los para a *CPU* antes que possam ser computados como vetores pelo método *np.asarray*. Com base nesses fatos, a seguinte correção foi implementada através do *Pull Request 18*:

Listagem 3.31: *Diff* da correção implementada pelo *Pull Request 18*

```
284     try:
285 +       if torch.is_tensor(val) and val.is_cuda:
286 +           val = val.cpu()
287         sanitized_metrics[metric] = np.asarray(val).item()
```

Após essa mudança, o *script* executa corretamente e consegue prover as métricas de análise, conforme no apêndice D.

### 3.2.3 Correção de *warnings* com o Otimizador *L-BFGS*

Durante a execução do experimento 3.1.1 utilizando os ataques *Deep Leakage* e *Beyond Infering*, os módulos *Pytorch* exaltaram avisos relativos à conversão de um tensor para seu valor escalar sem antes realizar o processo de *detach*. A investigação do aviso levou ao *Pull Request 27*, realizando o *detach* no fim da execução da função *closure()* durante a seleção do candidato na reconstrução.

### 3.2.4 Correção na Execução do Ataque *RGAP* com Aceleração Gráfica

Ao executar o ataque *RGAP* com aceleração gráfica, um erro ocorria, o *Pull Request 25* movimenta o tensor *torch original_dy_dx* para a *cpu* antes de convertê-lo para *numpy*. As linhas que provem as saídas da reconstrução também foram modificadas, utilizando o modulo *log* no lugar do módulo *print* uma vez que esse ultimo modulo apenas envia as saídas no fim da execução. Isso melhora o *feedback* do usuário em relação ao progresso da reconstrução, uma vez que as saídas são encaminhadas em tempo real ao invés do fim da execução.

### 3.2.5 Expansão dos Atributos da Classe *data_cfg_default*

A implementação dos ataques em *Breaching* em determinados momentos usa o padrão "variável.atributo" e em outros o padrão "variavel[chave]". O primeiro é o padrão de variáveis na forma de objetos, enquanto o segundo é o padrão na forma de dicionários. Isso funciona uma vez que *Breaching* utiliza o tipo *DictConfig* que aceita ambos os padrões durante as simulações. Porem no *script minimal_example.py*, a variável de configuração é definida como uma classe, impedindo a utilização do padrão de dicionário. O *Pull Request 23* modifica a classe para ser do tipo *DictCOnfig* permitindo tanto a abordagem de objetos quanto dicionários.

### 3.2.6 Melhorias na Documentação

Também foram realizadas melhorias na documentação dos parâmetros utilizados em *Breaching*, utilizando arquivos *Markdowns* que anexam ao projeto as tabelas propostas no apêndice B, integradas através dos seguintes *Pull Requests*:

***Pull Request 21*:** implementa a documentação dos parâmetros de implementação, disponíveis no apêndice B.

***Pull Request 22*:** implementa a documentação dos parâmetros de *Datasets*, disponíveis no apêndice B.

***Pull Request 26*:** implementa os parâmetros usuários, disponíveis no apêndice B

# 4 Considerações Finais

Nesse estudo, foi apresentado as bases introdutórias ao tema, além do detalhamento de um projeto real, com impacto no dia-a-dia daqueles que se distendem na questão do Ataque Federado. Correções foram propostas e levadas ao repositório central, produzindo uma melhoria efetiva no funcionamento do mesmo. A Seção 4.1 busca definir os impactos que o estudo geram, assim como os conceitos e ideais que permeiam as motivações do trabalho. Finalizando com 4.2, é relatado os próximos passos e possibilidades futuras em relação ao repositório *Breaching*.

## 4.1 Conclusão

Através dos processos descritos no Capítulo anterior, fica evidente que *Breaching* é um projeto funcional, mas que exige de seu usuário a construção de um modelo mental de seu funcionamento que só pode ser desenvolvido através da experimentação e leitura do código-fonte. Dessa forma, a Seção "Tutorial — Utilizando *Breaching* (2.3.3) oferece ao leitor uma introdução do funcionamento do *Framework*, e seus principais módulos de execução e funcionamento. Da mesma forma, o apêndice A oferece ao leitor uma introdução da dependência *Hydra* e seu funcionamento geral, o que pode ser expandido para a utilização das simulações implementadas em *Breaching*.

As correções e melhorias em 3.2 tem impacto direto no projeto e extrapolam esse estudo em si, uma vez que permitem aos usuários do *Framework* um manuseio efetivo, sem erros nos locais em que as mudanças foram aplicadas:

- A correção implementada na Subseção 3.2.1 corrigem erros durante a execução da análise dos ataques utilizados em *datasets* visuais.
- A correção implementada na Subseção 3.2.2 corrige erros na execução da análise de ataques utilizando aceleração gráfica.
- A correção implementada na Subseção 3.2.3 permita uma saída sem erros para os ataques que utilizam o otimizador *L-BGGS*.
- A correção implementada na Subseção 3.2.4 permite a execução do ataque *RGAP* com aceleração gráfica e melhor *feedback* das saídas.
- A correção implementada na Subseção 3.2.5 permite a execução do *script minimal_exemple.py* sem erros quando um módulo de ataque utiliza padrões diferentes para acessar os atributos de uma variável.

- As melhorias implementadas na Subseção 3.2.6 enriquerecem o *framework* com o tabelamento de alguns dos parâmetros utilizados.

Essas correções evitam que usuários percam tempo depurando o código e consigam rapidamente obter os resultados desejados, uma vez que os casos de uso podem ser executados sem erros.

Por fim, o experimento "Bem me quer, Mal me quer" na Seção 3.1, mostram que o *Framework* pode ser utilizado em conjunto com qualquer outra estrutura que utilize a biblioteca *PyTorch* para produzir modelos treinados através da arquitetura de Aprendizado Federado.

O objetivo desse estudo foi atingido, um processo de coleta dos dados está demonstrado na Subseção 3.1.1 através da Listagem E.1. Ainda que o ponto da coleta não seja ideal num cenário real — fato abordado a seguir em 4.2, que também oferece um referencial teórico para atingir esse cenário —, a implementação pode ser extrapolada para outros pontos em *Flower*, ou qualquer outra implementação de Aprendizado Federado que utilize *pytorch*. Uma *pipeline* de ataques podem ser facilmente construída utilizando *Breaching*, como demonstra a implementação na Listagem 3.1. O argumento *–attack* pode receber todos os ataques do repositório, contemplados através da tabela B.11 para averiguar a segurança de um determinado treinamento de Aprendizado Federado.

Houveram também descobertas, em relação ao tipo de modelo em uso, e como a presença de certas camadas, como a de normalização, podem afetar a eficácia do ataque, e do treinamento em si. Principalmente para o caso onde o tamanho do *batch* é de uma única imagem como demonstrado na Subseção 3.1.4.

## 4.2 Trabalhos Futuros

O experimento na Seção 3.1 simula uma situação onde um cliente ataca a si mesmo, e não onde um servidor ataca um cliente. Entretanto, isso pode ser alcançável através da sobrecarga da estratégia utilizada pela implementação em *Flower* (FLOWER, 2020b). O método *aggregate_train()* realiza a agregação dos gradientes gerados pelos clientes uma vez que são enviados ao servidor, sendo assim, o servidor pode sobrecarregar esse método para coletar os parâmetros, *buffers* e gradientes de cada cliente antes da agregação e aplicar as técnicas de reconstrução utilizado os dados coletados. O procedimento é o mesmo que descritor no experimento da seção 3.1, uma vez que os clientes enviam o *state dict* do modelo treinado ao servidor.

Esse estudo realizou apenas um caso de uso com *batchs* de uma imagem e pode ser expandido para *batchs* maiores progressivamente, reiterando o valor mínimo que torna a reconstrução impossível para os ataques. Da mesma forma, esse estudo não averiguou a eficácia de ataques mais robustos, capazes de reconstruir *batchs* maiores que um. Isso pode ser alcançável

aumentando o numero do *batch_size* que a variável *trainloader* recebe no *script task.py*, de maneira graduativa em multiplos de dois.

Também nenhum ataque que modifica parâmetros de forma maliciosa e que apresentam os melhores resultados na literatura foi explorada. O *script minimal_example_robbing_the_fed.py* em *Breaching* pode ser utilizado como base para aplicar alcaçar esse caso de uso. A modalidade textual e tabular também não foi contemplada, permitindo espaço para novas contribuições, para tal, é possivel escolher um guia em *flower* que instancie um treinamento usando essas modalidades (FLOWER, 2020a), e da mesma forma, observar quais ataques são eficazes para esse cenário através dos exemplos em *Breaching* (GEIPING, 2021d).

É preciso ressaltar também que os resultados obtidos não foram mensurados através de nenhuma métrica, como as utilizadas em *Breaching* durante uma simulação. Experimentos que se aproveitam desse recurso podem oferecer um respaldo mais consistente, uma vez que cada métrica tem embasamento cientifico em sua composição. *Breaching* oferece um módulo (GEIPING, 2021a) que utiliza diversas métricas quando uma simulação é executada utilizando o *script simulate_breach.py* e a implementação pode ser extrapolada para o *script* minimo.

A análise final em relação às condições ótimas para o uso do ataque *Beyond Infering* na Subseção 3.1.9 foram inconclusivos. Na condução dos experimentos o fazer funcionar e o *saber* funcionar se misturaram; o ataque funciona, e obteve resultados que remetem o dado original. Mas é justamente a não conclusão que estabelece não um fracasso, mas uma possibilidade de estudo que possa analisar e determinar as condições ideias de um ataque, otimizando um cenário que busque sempre os adversários mais plausíveis de promoverem brechas a segurança de uma configuração de Aprendizado Federado.

O tema com certeza é vasto e ainda pode ser enriquecido de diversas formas.

# Referências

ANACONDA. *Conda*. 2012. <https://www.anaconda.com/>. Accessed: 2025-5-17.

CHEN, Y.; SU, L.; XU, J. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *arXiv preprint arXiv:1705.05491*, 2017. Disponível em: <https://arxiv.org/abs/1705.05491>.

FLOWER. *Flower*. 2020. <https://flower.ai>. Accessed: 2025-5-17.

FLOWER. *Flower*. 2020. <https://flower.ai/docs/framework/ref-api/flwr.serverapp.strategy.FedAvg.html#flwr.serverapp.strategy.FedAvg>. Accessed: 2025-5-17.

FLOWER. *Flower - Get Started*. 2020. <https://flower.ai/docs/framework/tutorial-series-get-started-with-flower-pytorch.html>. Accessed: 2025-5-17.

FLOWER. *Flower - Get Started - Source Code*. 2020. <https://github.com/adap/flower/blob/main/examples/quickstart-pytorch>. Accessed: 2025-5-17.

FLOWER. *Flower - Get Started - Source Code*. 2020. <https://github.com/adap/flower/blob/main/examples/quickstart-pytorch/pytorchexample/task.py#L36>. Accessed: 2025-5-17.

FOWL, L.; GEIPING, J.; CZAJA, W.; GOLDBLUM, M.; GOLDSTEIN, T. Robbing the fed: Directly obtaining private data in federated learning with modified models. *arXiv preprint arXiv:2110.13057*, 2022. Disponível em: <https://arxiv.org/abs/2110.13057>.

FOWL, L.; GEIPING, J.; REICH, S.; WEN, Y.; CZAJA, W.; GOLDBLUM, M.; GOLDSTEIN, T. Decepticons: Corrupted transformers breach privacy in federated learning for language models. *arXiv preprint arXiv:2201.12675*, 2022. Disponível em: <https://arxiv.org/abs/2201.12675>.

GEIPING, J. *Breaching*. 2021. <https://github.com/JonasGeiping/breaching>. Accessed: 2025-5-17.

GEIPING, J. *Breaching*. 2021. <https://github.com/JonasGeiping/breaching/blob/main/examples/Beyond%20Inferring%20Class%20Representatives%20-%20Optimization-based%20Attack%20-%20ConvNet%20CIFAR-10.ipynb>. Accessed: 2025-5-17.

GEIPING, J. *Breaching*. 2021. <https://github.com/JonasGeiping/breaching/blob/main/breaching/config/case/data/CIFAR10.yaml>. Accessed: 2025-5-17.

GEIPING, J. *Breaching*. 2021. <https://github.com/JonasGeiping/breaching/blob/main/examples>. Accessed: 2025-5-17.

GOODFELLOW, I. J.; POUGET-ABADIE, J.; MIRZA, M.; XU, B.; WARDE-FARLEY, D.; OZAIR, S.; COURVILLE, A.; BENGIO, Y. Generative adversarial nets. In: GHAHRAMANI, Z.; WELLING, M.; CORTES, C.; LAWRENCE, N.; WEINBERGER, K. (Ed.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2014. v. 27. Disponível em: <https://proceedings.neurips.cc/paper_files/paper/2014/file/f033ed80deb0234979a61f95710dbe25-Paper.pdf>.

JOCHEMS, A.; DEIST, T. M.; van Soest, J.; EBLE, M.; BULENS, P.; COUCKE, P.; DRIES, W.; LAMBIN, P.; DEKKER, A. Distributed learning: Developing a predictive model based on data from multiple hospitals without data leaving the hospital – a real life proof of concept. *Radiotherapy and Oncology*, v. 121, n. 3, p. 459–467, 2016. ISSN 0167-8140. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167814016343365>.

KONEčNý, J.; MCMAHAN, B.; RAMAGE, D. Federated optimization:distributed optimization beyond the datacenter. *arXiv preprint arXiv:1511.03575*, 2015. Disponível em: <https://arxiv.org/abs/1511.03575>.

MCMAHAN, H. B.; MOORE, E.; RAMAGE, D.; HAMPSON, S.; ARCAS, B. A. y. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629*, 2023. Disponível em: <https://arxiv.org/abs/1602.05629>.

META. *PyTorch*. 2016. <https://pytorch.org/>. Accessed: 2025-5-17.

META. *PyTorch L-BFGS*. 2016. <https://docs.pytorch.org/docs/stable/generated/torch.optim.LBFGS.html>. Accessed: 2026-1-29.

META. *Hydra*. 2020. <https://hydra.cc/>. Accessed: 2025-5-17.

PARLIAMENT, E.; COUNCIL. *Regulation (EU) 2016/679 of the European Parliament and of the Council*: General data protection regulation (gdpr). 2016. <https://eur-lex.europa.eu/eli/reg/2016/679/oj>. Accessed: 2025-5-17.

PHONG, L. T.; AONO, Y.; HAYASHI, T.; WANG, L.; MORIAI, S. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, v. 13, n. 5, p. 1333–1345, 2018.

RODRIGUEZ-BARROSO, N.; JIMéNEZ-LóPEZ, D.; LUZóN, M. V.; HERRERA, F.; MARTíNEZ-CáMARA, E. Survey on federated learning threats: Concepts, taxonomy on attacks and defences, experimental study and challenges. *Information Fusion*, Elsevier BV, v. 90, p. 148–173, fev. 2023. ISSN 1566-2535. Disponível em: <http://dx.doi.org/10.1016/j.inffus.2022.09.011>.

VAPNIK, V. Principles of risk minimization for learning theory. MIT Press, 1991.

VEALE, M.; BINNS, R.; EDWARDS, L. Algorithms that remember: model inversion attacks and data protection law. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, The Royal Society, v. 376, n. 2133, p. 20180083, out. 2018. ISSN 1471-2962. Disponível em: <http://dx.doi.org/10.1098/rsta.2018.0083>.

WANG, Z.; SONG, M.; ZHANG, Z.; SONG, Y.; WANG, Q.; QI, H. Beyond inferring class representatives: User-level privacy leakage from federated learning. In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. [S.l.: s.n.], 2019. p. 2512–2520.

WEN, Y.; GEIPING, J.; FOWL, L.; GOLDBLUM, M.; GOLDSTEIN, T. Fishing for user data in large-batch federated learning via gradient magnification. *arXiv preprint arXiv:2202.00580*, 2022. Disponível em: <https://arxiv.org/abs/2202.00580>.

ZHU, J.; BLASCHKO, M. B. R-{gap}: Recursive gradient attack on privacy. In: *International Conference on Learning Representations*. [s.n.], 2021. Disponível em: <https://openreview.net/forum?id=RSU17UoKfJF>.

ZHU, L.; LIU, Z.; HAN, S. Deep leakage from gradients. In: WALLACH, H.; LAROCHELLE, H.; BEYGELZIMER, A.; ALCHé-BUC, F. d'; FOX, E.; GARNETT, R. (Ed.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2019. v. 32. Disponível em: <https://proceedings.neurips.cc/paper_files/paper/2019/file/60a6c4002cc7b29142def8871531281a-Paper.pdf>.

# Apêndices

# APÊNDICE A – *Tutorial — Hydra*

O framework *Hydra* é construído em cima de uma biblioteca chamada *OmegaConf*. Essa bibliotéca é utilizada por outros projetos importantes na área de aprendizado de máquina como, por exemplo: *PyTorch Lightning* e *Fairseq*. Ela opera a manipulando a sintaxe *YAML* que se define como uma linguagem de serialidade de dados amigável ao usuário. Ou seja, os objetos de configurações do *framework* são instancias do tipo *DictConfig* da biblioteca *OmegaConf* definidos através da linguagem *YAML*.

Na pagina inicial da documentação do *Hydra* é possível obter um rápido vislumbre das funcionalidades que o *framework* implementa. O necessário para a compreensão de *Breaching* será desenvolvido a seguir:

### Inicialização

***@hydra.main*:** é implementada através do recurso functools.wrapper da linguagem; que permite a criação de *decorators* customizados. O arquivo *simulate_breach.py* utiliza esse método para inicializar as variavéis do ataque.

```python
from omegaconf import DictConfig, OmegaConf
import hydra

@hydra.main(version_base=None)
def my_app(cfg: DictConfig) -> None:
    print(OmegaConf.to_yaml(cfg))

if __name__ == "__main__":
    my_app()
```

***Compose API*:** semelhante ao método anterior, podem, inclusive, ser utilizados juntos. A composição pode gerar um objeto de configuração em qualquer local do código, mas depende de algum tipo de inicialização. É útil para trechos que não podem utilizar o *wrapper* @*hydra.main*, como, por exemplo, *jupyter notebooks*. Essa *API* é utilizada pelos exemplos de *Breaching*, implementados em *.pynb*, utilizando o método *hydra.initialize* no lugar do *decorator*.

```
from hydra import compose, initialize
from omegaconf import OmegaConf

if __name__ == "__main__":
    # context initialization
    with initialize(version_base=None):
        cfg = compose()
        print(OmegaConf.to_yaml(cfg))

    # global initialization
    initialize(version_base=None)
    cfg = compose()
    print(OmegaConf.to_yaml(cfg))
```

Através da instrução "*with*" da linguagem, que opera semelhante a um bloco *try/catch*, o objeto *cfg* é inicializado e utilizado somente ao longo da existência do bloco. Mas o método *initialize* pode ser utilizado normalmente também, como demonstra o segundo bloco do exemplo.

**NOTA:** A partir desse ponto o seguinte *script* será utilizado como modelo, alterações serão indicadas como pequenas *diffs*:

*Script* modelo para a descrição das funcionalidades

```
1   from hydra import compose, initialize, main
2   from omegaconf import OmegaConf, DictConfig
3
4   @main(version_base=None)
5   def my_app(cfg: DictConfig) -> None:
6       print(OmegaConf.to_yaml(cfg))
7
8   if __name__ == "__main__":
9       my_app()
10      # context initialization
11      with initialize(version_base=None):
12          cfg = compose()
13          print(OmegaConf.to_yaml(cfg))
14
15      # global initialization
16      initialize(version_base=None)
17      cfg = compose()
18      print(OmegaConf.to_yaml(cfg))
```

Execuções serão ilustradas na forma de entradas e saídas de comandos de terminal:

```
$ COMANDO
OUTPUT
```

### Definição

Conforme descrito no início da Seção A, *Hydra* gera objetos do tipo *DictConfig*. Na Subseção anterior (A), é esclarecido que o *Hydra* sempre deve ser inicializado. Agora, será descrito como esses objetos podem ser definidos:

**Objeto vazio:** caso o script modelo seja executado, será retornado um objeto vazio:

Execução do *script* modelo A

```
$ python my_app.py
{}

{}

{}
```

**Arquivos:** é possível definir e carregar arquivos *YAML* (*.yaml* e *.yml*) para serem utilizados diretamente no código. É dessa maneira que *Breaching* implementa toda a sua configuração.

```
4   @main(version_base=None, config_path="./conf", config_name="cfg")
5   def my_app(cfg: DictConfig) -> None:
6       print(OmegaConf.to_yaml(cfg))
7
8   if __name__ == "__main__":
9       my_app()
10      # context initialization
11      with initialize(version_base=None, config_path="./conf"):
12          cfg = compose(config_name="cfg")
13          print(OmegaConf.to_yaml(cfg))
14
15      # global initialization
16      initialize(version_base=None, config_path="./conf")
17      cfg = compose(config_name="cfg")
```

Os atributos *config_name* e *config_path* foram acrescentados aos métodos *@hydra.main*, *(*initialize) e *compose*. Uma vez adicionados, *Hydra* irá buscar os arquivos de configuração

**relativo ao diretorio do *script*** que invocou os métodos de inicialização (note o ponto "." em *config_path*).

**Arquivo vazio:** uma vez que os atributos *config_name* e *config_path* são definidos, já não é mais possível inicializar um objeto vazio sem que o arquivo exista no caminho especificado.

```
        $ python my_app.py
        Cannot find primary config 'cfg'. Check that it's in your
config search path.

        Config search path:
                provider=hydra, path=pkg://hydra.conf
                provider=main, path=file:///media/work/bernardoemery/
↪breaching
                provider=schema, path=structured://

        Set the environment variable HYDRA_FULL_ERROR=1 for a complete
stack trace.
        $ mkdir -p conf
        $ touch conf/cfg.yaml
        $ python my_app.py
        {}

        {}

        {}
```

**Arquivo simples:** a sintaxe *YAML* define uma estrutura de pares "CHAVE: VALOR" e possui capacidade para aninhar chaves dentro de chaves (*nest*).

```
        $ printf "CHAVE: VALOR\nPAI:\n  FILHO: VALOR\n" > conf/cfg.
↪yaml
        $ python my_app.py
        CHAVE: VALOR
        PAI:
          FILHO: VALOR

        CHAVE: VALOR
        PAI:
          FILHO: VALOR

        CHAVE: VALOR
        PAI:
          FILHO: VALOR
```

**Grupos de arquivos:** é possível estabelecer uma configuração hierárquica de múltiplos arquivos para popular um objeto de configuração. Para isso é necessário que o arquivo de configuração utilize a chave que deve sempre ser declarada no início da configuração.

```
        $ mkdir -p conf/dir1 conf/dir2
        $ printf "defaults:\n   \n- dir1: file1\n- dir2: file2\n" >
conf/cfg.yaml
        $ printf "CHAVE: VALOR\n" > conf/dir1/file1.yaml
        $ printf "CHAVE: VALOR\n" > conf/dir2/file2.yaml
        $ python my_app.py
        dir1:
          CHAVE: VALOR
        dir2:
          CHAVE: VALOR

        dir1:
          CHAVE: VALOR
        dir2:
          CHAVE: VALOR

        dir1:
          CHAVE: VALOR
        dir2:
          CHAVE: VALOR
```

Note que dois diretórios foram criados: "*dir1*" e "*dir2*". Ao utilizar a chave *defaults:*, as próximas entradas de "CHAVES: VALOR" desse estrutura devem corresponder ao "DIRETÓRIO: ARQUIVO" que se deseja agrupar. A partir do diretório de configuração base cada entrada padrão será buscada pelo *Hydra* a partir dos nomes definidos na configuração carregada.

### Operações

Além de definir arquivos, é possível manipular o *Hydra* através de operadores diretamente na linha de comando — no caso de configurações inicializadas através do método @*hydra.main* — . Se inicializa através da *Compose API* essas operações podem ser realizadas através do parâmetro *overrides*, modificando a função *compose* diretamente no código. Os procedimentos a seguir funcionam para todos os recursos descritos anteriormente.

*Hydra* oferece uma sintaxe vasta e complexa para diversas outras operações além das descritas aqui. Uma vez que não são necessárias para a compreensão de *Breaching*, não serão descritas. A documentação está disponível online.

**Adição:** um parâmetro que não existe durante o processo de inicialização pode ser adicionado através do operador "+".

```
$ mkdir -p conf
$ touch conf/cfg.yaml
$ python my_app.py +PAI.FILHO=VALOR
PAI:
  FILHO: VALOR

{}

{}
```

```
11  with initialize(version_base=None, config_path="./conf"):
12      cfg = compose(config_name="cfg", overrides=["+over=rides"])
13      print(OmegaConf.to_yaml(cfg))
14
15  # global initialization
16  initialize(version_base=None)
17  cfg = compose(config_name="cfg", overrides=["+rides.over=value"])
```

```
$ python my_app.py +PAI.FILHO=VALOR
PAI:
  FILHO: VALOR

over: rides

rides:
  over: value
$ printf "CHAVE: VALOR\n" > conf/cfg.yaml
$ python my_app.py +PAI.FILHO=VALOR
CHAVE: VALOR
PAI:
  FILHO: VALOR

CHAVE: VALOR
over:  rides

CHAVE: VALOR
rides:
  over: value
```

**Sobrecarga:** um parâmetro que pode ser sobrecarregado durante o processo de inicialização basta realizar a operação sem o símbolo de adição. Essa é a função mais utilizada por

*breaching* ao longo de seus exemplos e *templates*.

```
11    with initialize(version_base=None, config_path="./conf"):
12        cfg = compose(config_name="cfg", overrides=[‘‘CHAVE=+over=rides’’])
13        print(OmegaConf.to_yaml(cfg))
14
15    # global initialization
16    initialize(version_base=None)
17    cfg = compose(config_name="cfg", overrides=[‘‘CHAVE=+rides.over=value’’])
```

```
$ mkdir -p conf
$ touch conf/cfg.yaml
$ python my_app.py CHAVE=NEW
Could not override 'CHAVE'.
To append to your config use +CHAVE=NEW
Key 'CHAVE' is not in struct
full_key: CHAVE
object_type=dict
$ printf "CHAVE: VALOR\n" > conf/cfg.yaml
$ python my_app.py CHAVE=NEW
CHAVE: NEW

CHAVE: over

CHAVE: rides
```

**Adição ou Sobrecarga** o operador “++” realiza uma sobrecarga caso o parâmetro exista, ou o cria caso contrário.

**Remoção:** utilizando o operador “~”, um parâmetro pode ser removido.

```
11      with initialize(version_base=None, config_path="./conf"):
12          cfg = compose(config_name="cfg", overrides=[‘‘~CHAVE=over’’])
13          print(OmegaConf.to_yaml(cfg))
14
15      # global initialization
16      initialize(version_base=None)
17      cfg = compose(config_name="cfg", overrides=[‘‘~CHAVE=rides’’])
```

```
        $ mkdir -p conf
        $ touch conf/cfg.yaml
        $ python my_app.py ~CHAVE
        Could not delete from config. 'CHAVE' does not exist.

        Set the environment variable HYDRA_FULL_ERROR=1 for a
complete stack trace.
        $ printf "CHAVE: VALOR\nOVER:\n  RIDE: OVER\n  KEEP: THIS\n"
> conf/cfg.yaml
        $ python my_app.py ~CHAVE ~OVER.RIDE
        OVER:
          KEEP: THIS

        OVER:
          RIDE: OVER
          KEEP: THIS

        OVER:
          RIDE: OVER
          KEEP: THIS
```

### Linha de Comando

Além dos operadores, *flags* podem ser utilizadas também. As configurações de interesse estão descritas abaixo e as restantes podem ser acessadas na online ou através da *flag –hydra-help*.

--***hydra-help*:** descreve todas as *flags* oferecidas pelo *framework*.

--***help*:** exibe a documentação específica da **aplicação**; essa documentação pode ser customizada.

--**cfg:** produz uma saída do objeto de configuração construído através dos arquivos. Deve ser utilizado com um dos parâmetros abaixo.

***job*:** exibe os valores particulares à aplicação.

***hydra*:** exibe os valores particulares ao *Hydra*.

***all*:** junção dos parâmetros acima.

--**package:** usado em conjunção com o parâmetro *--cfg*, exibe os valores específicos do grupo de configuração escolhido. É possível obter os grupos de configuração através do parâmetro *--help* na Seção *Configuration groups*.

--***info*:** exibe um aglomerado de informações ou uma categoria específica, como especificado abaixo:

***all*:** exibe todas as informações, é o padrão quando o comando não é utilizado com nenhum outro argumento.

***config*:** produz uma saída semelhante ao parâmetro *--cfg*.

***defaults*:** exibe as informações de todos os grupos de configuração que utilizam o atributo *defaults* (A).

***defaults-tree*:** exibe a árvore gerada pela composição dos grupos que utilizam o atributo *default*.

***plugins*:** exibe os *plugins* do *Hydra* sendo utilizados pela aplicação.

***searchpath*:** exibe os caminhos que podem ser utilizados para compor os objetos de configuração.

--***multirun*:** possibilita que múltiplas execuções sejam realizadas, com diferentes parâmetros.

--***config-path*:** sobrecarrega o caminho definido pelo modo de inicialização @*hydra.main*.

--***config-name*:** sobrecarrega o nome do arquivo de configuração definido durante a inicialização.

--***config-dir*:** adiciona um diretório além daquele definido durante a inicialização.

Todos os recursos definidos durante essa Seção podem ser utilizados em conjunto. A ordem em qual são declarados pode alterar a saída final.

# APÊNDICE B – Tabelas de Parâmetros

Uma vez que os recursos estão organizados em múltiplos arquivos de configuração, as tabelas abaixo são apresentadas para auxiliar na compreensão das possibilidades propostas por *breaching*.

## Implementação

| Variavel | Descrição | Valores/Tipo | Valor padrão |
|---|---|---|---|
| *shuffle* | Recolhe amostras do *dataset* aleatoriamente | *boolean* | *false* |
| *sample_with_replacement* | Utilizado em conjunto com *shuffle*, amostras são recolhidas sob demanda com reposição caso necessário | *boolean* | *false* |
| *dtype* | Caso o atributo *mixed_precision* seja *True*, o valor dessa variavel deve ser *float*. *dtype* representa o tipo de dado de um tensor. | *torch.dtype* | *torch.float* |
| *non_blocking* | Não utilizado. | *boolean* | *True* |
| *sharing_strategy* | Define a estratégia de multiprocessamento para a manipulação de um mesmo dado em diferentes processos. | *string* | *file_descriptor* |
| *enable_gpu_acc* | Habilita a aceleração gráfica. | *boolean* | *True* |
| *benchmark* | Desabilita o recurso *cuDNN* não-deterministico. | *boolean* | *True* |
| *pin_memory* | — | *boolean* | *True* |
| *threads* | Quantidade máxima de *CPU workes* por *GPU*. | *int* | 0 |
| *persistent_workes* | — | *boolean* | *False* |
| *mixed_precision* | — | *boolean* | *True* |
| *grad_scaling* | — | *boolean* | *True* |
| *JIT* | — | — | *null* |
| *validate_every_nth_step* | — | *int* | 10 |
| *checkpoint* | — | *object* | — |
| *.name* | — | *string* | — |
| *.save_every_nth_step* | — | *int* | 10 |
| *enable_huggingface_offline_mode* | — | *boolean* | *True* |

Tabela B.1 – Parâmetros de implementação utilizados em *Breaching*.

# *Datasets*

| Nome | Modalidade | Tarefa |
|---|---|---|
| *Birdsnap* | *vision* | *classification* |
| *CIFAR10* | *vision* | *classification* |
| *CIFAR100* | *vision* | *classification* |
| *ImageNet* | *vision* | *classification* |
| *ImageNetAnimals* | *vision* | *classification* |
| *TinyImageNet* | *vision* | *classification* |
| *cola* | *text* | *classification* |
| *random*-tokens | *text* | *causal-lm* |
| *shakespeare* | *text* | *causal-lm* |
| *stackoverflow* | *text* | *causal-lm* |
| *wikitext* | *text* | *causal-lm* |

Tabela B.2 – Conjunto de *datasets* implementados em *Breaching*.

| Variavel | Descrição | Valores/Tipo |
|---|---|---|
| *name* | O nome do *dataset*. | *string* |
| *modality* | Modalidade, ou natureza dos dados presentes no *dataset*. | vision<br>text |
| *task* | Tarefa que o modelo deverá executar através do treinamento com o *dataset*. | *classification*<br>*masked-lm*<br>*casual-lm* |
| *path* | Caminho do *dataset* no sistema. | *string* |
| *size* | Valor total de entradas no *dataset* | *int* |
| *default_clients* | Estimativa do número de clientes participando do treinamento. | *int* |
| *partition* | Estratégia de particionamento do *Dataset* durante a sua preparação para o treinamento. | *given*<br>*balanced*<br>*unique-class*<br>*mixup*<br>*feat_est*<br>*ramdom-full*<br>*random*<br>*none* |
| *examples_from_split* | Estratégia de separação dos conjuntos de treinamento ou validação. | *train*<br>*training*<br>*validation* |
| *batch_size* | Numero de amostras processadas em uma iteração durante o treinamento. | *int* |
| *caching* | — | *boolean* |
| *db* | Modulo *LMDB* que utiliza o *dataset* através de um banco de dados. | *none*<br>*LMDB* |

Tabela B.3 – Conjunto de parâmetros comuns aos *datasets* de imagens e texto utilizados em *Breaching*.

| Variavel | Descrição | Valores/Tipo |
|---|---|---|
| *shape* | É uma tripla que representam os valores do "canal", "comprimento" e "altura" das imagens no *dataset*. | *3-tuple* |
| *classes* | Quantidade de rótulos (*labels*) presentes no *dataset*. | *string* |
| *normalize* | Escala os *pixels* das imagens, para um intervalo padrão. | *boolean* |
| *mean* | Média dos valores dos *pixels* durante a normalização. | *3-tuple* |
| *std* | Desvio padrão dos *pixels* durante a normalização. | *3-tuple* |
| *augmentations_train* | Possui os atributos *RandomCrop* e *RandomHorizontalFlip* descristos abaixo. | *object* |
| *.RandomCrop* | Corta a imagem aleatoriamente através do valor especificado para criar variações. | *tuple* |
| *.RandomHorizontalFlip* | Inverte a imagem aleatoriamente. | *float* |
| *augmentations_val* | Possui os atributos *Resize* e *CenterCrop* descritos abaixo. | *object* |
| *.Resize* | Redimensiona a imagem para uma dimensão fixa. | *int* |
| *.CenterCrop* | Corta relativamente ao centro da imagem para avaliações consistentes. | *int* |

Tabela B.4 – Conjunto de parâmetros específicos aos *datasets* de imagens utilizados em *Breaching*.

| Variavel | Descrição | Valores/Tipo |
|---|---|---|
| *shape* | Quando a modalidade tem valor *text*, essa variavel assume o atributo *sequence_lenght*, que representa a quantidade de palavras ou *tokens* presentes no *dataset*. | *int* |
| *vocab_size* | Valor total de *tokens* únicos no dicionário do *dataset*. | *int* |
| *mlm_probability* | Probabilidade de um dado *token* ser mascarado através do recurso *Masked Language Modeling* (MLM). | *float* |
| *disable_mlm* | Desabilita o recurso MLM. | *boolean* |
| *tokenizer* | Modelo responsável por particionar o texto em unidades menores. | *GPT-2*<br>*bert-base-uncased* |

Tabela B.5 – Conjunto de parâmetros específicos aos *datasets* de texto utilizados em *Breaching*.

# Usuarios

| Tipo | | |
|---|---|---|
| local_gradient | multiuser_aggregate | local_updates |

Tabela B.6 – Conjunto de usuários (clientes) implementados em *Breaching*.

| Variavel | Descrição | Valores/Tipo |
|---|---|---|
| user_type | Tipo do comportamento de usuário, se refere ao nome do arquivo descrito em B.6 | *string* |
| user_idx | Qual usário no total de clientes será atacado | *int* |
| num_data_points | Tamanho do *batch* que usuário irá realizar o treinamento durante a simulação | *int* |
| provide_buffers | Define se o usuário irá enviar algum *buffer* durante a atualização do modelo | *boolean* |
| provide_labels | Define se o usuário irá enviar os rotulos durante a atualização do modelo | *boolean* |
| provide_num_data_points | Define se o usuário irá enviar a quantidade do *batch* utilizado no treinamento durante a atualização do modelo | *boolean* |
| local_diff_privacy | Objeto que define os valores utilizados pela segurança diferencial local durante o treinamento | *object* |
| .gradient_noise | Ruido adicionado no gradiente antes da atualização | *float* |
| .input_noise | Adiciona um ruido de desvio padrão nos gradientes para o aumento de privacidade. | *float* |
| .distribution | Adiciona um ruido de desvio padrão nos dados de entrada (*Datapoints*) para o aumento de privacidade | *string* |
| .per_example_clipping | — | *float* |

Tabela B.7 – Conjunto de parâmetros comuns a todos os tipos de usuarios utilizados em *Breaching*.

| Variavel | Descrição | Valores/Tipo |
|---|---|---|
| *num_local_updates* | — | *int* |
| *num_data_per_local_update_step* | — | *int* |
| *local_learning_rate* | — | *float* |
| *provide_local_hyperparams* | — | *boolean* |

Tabela B.8 – Conjunto de parâmetros especificos aos usuarios *local_updates* e *multiuser_aggregate* utilizados em *Breaching*.

# Servidores

| Tipo | | | | |
|---|---|---|---|---|
| honest-but-curious | malicious-model-cah | malicious-transformer | malicious-fishing | malicious-model-rtf |

Tabela B.9 – Conjunto de servidores implementados em *Breaching*.

# Casos de Uso

| Nome | Dataset | Servidor | Usuario | Modelo |
|---|---|---|---|---|
| 0_sanity_check | ImageNet | honest-but-curious | local_gradient | linear |
| 1_single_image_small | CIFAR10 | honest-but-curious | local_gradient | ConvNet |
| 2_single_imagenet | Imagenetanimals | honest-but-curious | local_gradient | ResNet18 |
| 4_fedavg_small_scale | Imagenetanimals | honest-but-curious | local_updates | ResNet18 |
| 5_small_batch_imagenet | Imagenetanimals | honest-but-curious | local_gradient | ResNet50 |
| 6_large_batch_cifar | CIFAR100 | honestbutcurious | local_gradient | ResNet3210 |
| 8_industry_scale_fl | Imagenetanimals | honest-but-curious | multiuser_aggregate | ResNet18 |
| 9_bert_training | wikitext | honest-but-curious | local_gradient | bert-base-uncased |
| 10_causal_lang_training | wikitext | honest-but-curious | local_gradient | transformer3 |

Tabela B.10 – Conjunto de casos de uso implementados em *Breaching*.

# Ataques

| Nome | Tipo |
|---|---|
| _default_optimization_attack | default |
| analytic | analytic |
| april_analytic | analytic |
| beyondinfering | optimization |
| clsattack | optimization |
| decepticon | analytic |
| deepleakage | optimization |
| imprint | analytic |
| invertinggradients | optimization |
| modern | optimization |
| rgap | recursive |
| seethroughgradients | optimization |
| tag | optimization |

Tabela B.11 – Conjunto de ataques implementados em *Breaching*.

# APÊNDICE C – *Output* dos parâmetros de configuração - *simulate_breach.py*

```
$ python simulate_breach.py dryrun=True case=1_single_image_small --cfg
job

case:
  data:
    db:
      name: null
    name: CIFAR10
    modality: vision
    task: classification
    path: ~/data
    size: 50000
    classes: 10
    shape:
    - 3
    - 32
    - 32
    normalize: true
    mean:
    - 0.4914672374725342
    - 0.4822617471218109
    - 0.4467701315879822
    std:
    - 0.24703224003314972
    - 0.24348513782024384
    - 0.26158785820007324
    augmentations_train:
      RandomCrop:
      - 32
      - 4
      RandomHorizontalFlip: 0.5
    augmentations_val: null
    default_clients: 10
    partition: balanced
    examples_from_split: validation
    batch_size: 128
    caching: false
  impl:
```

```
    shuffle: false
    sample_with_replacement: false
    dtype: float
    non_blocking: true
    sharing_strategy: file_descriptor
    enable_gpu_acc: false
    benchmark: true
    deterministic: false
    pin_memory: true
    threads: 0
    persistent_workers: false
    mixed_precision: false
    grad_scaling: true
    JIT: null
    validate_every_nth_step: 10
    checkpoint:
      name: null
      save_every_nth_step: 10
    enable_huggingface_offline_mode: true
  server:
    name: honest_but_curious
    pretrained: true
    model_state: default
    provide_public_buffers: true
    has_external_data: false
    num_queries: 1
  user:
    user_type: local_gradient
    user_idx: 0
    num_data_points: 1
    provide_buffers: false
    provide_labels: true
    provide_num_data_points: true
    local_diff_privacy:
      gradient_noise: 0.0
      input_noise: 0.0
      distribution: laplacian
      per_example_clipping: 0.0
  name: single_image_small
  model: ConvNet
  num_queries: 1
attack:
  type: invertinggradients
  attack_type: optimization
  label_strategy: bias-corrected
  text_strategy: run-embedding
```

```
  token_recovery: from-labels
  objective:
    type: cosine-similarity
    scale: 1.0
    task_regularization: 0.0
  restarts:
    num_trials: 1
    scoring: cosine-similarity
  init: randn
  normalize_gradients: false
  optim:
    optimizer: adam
    signed: hard
    step_size: 0.1
    boxed: true
    max_iterations: 24000
    step_size_decay: step-lr
    langevin_noise: 0.0
    warmup: 0
    grad_clip: null
    callback: 1000
  augmentations: null
  differentiable_augmentations: false
  regularization:
    total_variation:
      scale: 0.2
      inner_exp: 1
      outer_exp: 1
  impl:
    dtype: float
    mixed_precision: false
    JIT: null
base_dir: outputs
seed: null
name: default
dryrun: true
num_trials: 100
save_reconstruction: false
```

# APÊNDICE D – Resultado de execução - *simulate_breach.py*

```
$ python simulate_breach.py dryrun=True case=1_single_image_small
xs[2025-10-07 17:32:47,945] ----------------------------------------------------
↪---------
[2025-10-07 17:32:47,953] -----Launching federating learning breach experiment! -
↪-------
[2025-10-07 17:32:47,976] Platform: linux, Python: 3.10.12, PyTorch: 2.8.0+cu128
[2025-10-07 17:32:48,006] CPUs: 10, GPUs: 1 on cisco.
[2025-10-07 17:33:05,523] Computing user update on user 0 in model mode: eval.
[2025-10-07 17:33:08,926] | It: 1 | Rec. loss: 0.4545 |  Task loss: 2.2939 | T:
0.90s
[2025-10-07 17:33:09,036] Optimal candidate solution with rec. loss 0.0099
selected.
[2025-10-07 17:33:09,037] Starting evaluations for attack effectiveness report..
↪.
[2025-10-07 17:33:31,785] METRICS: | MSE: 0.1142 | PSNR: 9.42 | FMSE: 7.5296e-09
| LPIPS: 0.38|
 R-PSNR:  nan | IIP-pixel: 0.00% | SSIM: nan | max R-PSNR:  nan | max SSIM: nan
| Label Acc: 100.00%
[2025-10-07 17:33:31,790] ----------------------------------------------------
↪------
[2025-10-07 17:33:31,791] Finished computations default with total train time:
0:00:43.838180
[2025-10-07 17:33:31,792] -----------------Job finished.-----------------------
↪------
```

# APÊNDICE E – Modificações no método *train* do módulo *task.py*

Listagem E.1: Modificações no método *train* do *script task.py*

```
64 def train(net, trainloader, epochs, lr, device):
65     """Train the model on the training set."""
66     net.to(device)  # move model to GPU if available
67     criterion = torch.nn.CrossEntropyLoss().to(device)
68     optimizer = torch.optim.Adam(net.parameters(), lr=lr)
69     net.train()
70     running_loss = 0.0
71       for epoch in range(epochs):
72       for batch_idx, batch in enumerate(trainloader):
73       +           if epoch == 0 and batch_idx == 0:
74       +               server_parameters = [p.clone().detach() for p in net.
   ↪parameters()]
75       +               torch.save(server_parameters, f"server_parameters.pt")
76       +               server_buffers = [b.clone().detach() for b in net.
   ↪buffers()]
77       +               torch.save(server_buffers, f"server_buffers.pt")
78             images = batch["img"].to(device)
79             labels = batch["label"].to(device)
80             optimizer.zero_grad()
81             loss = criterion(net(images), labels)
82             loss.backward()
83       +           if epoch == 0 and batch_idx == 0:
84       +               gradient = [p.grad.clone().detach() for p in net.
   ↪parameters() if p.grad is not None]
85       +               torch.save(gradient, f"gradient.pt")
86       +               buffers = [b.clone().detach() for b in net.buffers()]
87       +               torch.save(buffers, f"shared_buffers.pt")
88       +               true_data = {"images": images.cpu(), "labels": labels.
   ↪cpu(), "buffers": buffers}
89       +               torch.save(true_data, f"client{true_data.pt")
89             optimizer.step()
90             running_loss += loss.item()
91     avg_trainloss = running_loss / len(trainloader)
92     return avg_trainloss
```

# APÊNDICE F – *Script flower.py* - *minimal_example.py* modificado

```
"""
Example script to run attacks in this repository directly without simulation.
This can be useful if you want to check a model architecture and model gradients
computed/defended in some shape or form
against some of the attacks implemented in this repository, without implementing
your model into the simulation.

All caveats apply. Make sure not to leak any unexpected information.
"""
import torch
import torchvision
import breaching
import argparse
import logging
import sys
from omegaconf import DictConfig
from collections import OrderedDict

data_cfg_default = DictConfig({
    "modality": "vision",
    "task": "classification",
    "shape": [3, 32, 32],
    "classes": 10,
})

def overview(args, setup, model, attacker):
    logging.info(f"\n{setup}")

    num_params, num_buffers = (
        sum([p.numel() for p in model.parameters()]),
        sum([b.numel() for b in model.buffers()]),
    )
    target_information = 1 * torch.as_tensor(data_cfg_default.shape).prod()
    logging.info(
        f"\nModel architecture loaded with {num_params:,} parameters and {num_
↪buffers:,} buffers.\n"
```

```
        f"Overall this is a data ratio of { 1 * num_params / target_information:
↪7.0f}:1 \n"
        f"for target shape {[1, *data_cfg_default.shape]} given that num_
↪queries=1.\n"
    )

    logging.info(f"{attacker}")

    logging.info(f"Client:\n"
                 f"   Provide labels: {args.provide_labels}\n"
                 f"   Provide shared buffers: {args.provide_shared_buffers}\n"
                 f"Server:\n"
                 f"   Provide parameters: {args.provide_parameters}\n"
                 f"   Provide public buffers: {args.provide_public_buffers}\n")


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--attack', default='invertinggradients')
    parser.add_argument('--gradient-path', required=True)
    parser.add_argument('--parameters-path', required=True)
    parser.add_argument('--public-buffers-path', required=True)
    parser.add_argument('--shared-buffers-path', required=True)
    parser.add_argument('--true-data-path', required=True)
    parser.add_argument('--provide-labels', action='store_true', default=False)
    parser.add_argument('--provide-parameters', action='store_true',
default=False)
    parser.add_argument('--provide-public-buffers', action='store_true',
default=False)
    parser.add_argument('--provide-shared-buffers', action='store_true',
default=False)
    args = parser.parse_args()


    setup = dict(device=torch.device("cuda" if torch.cuda.is_available() else
↪"cpu"), dtype=torch.float)


    # This could be your model:
    model = breaching.cases.models.model_preparation.ConvNet()
    model = model.to(setup['device'])
    model.eval()
    loss_fn = torch.nn.CrossEntropyLoss()


    # This is the attacker:
```

```
    cfg_attack = breaching.get_attack_config(args.attack)
    # cfg_attack.regularization.total_variation.scale = 1e-3
    attacker   = breaching.attacks.prepare_attack(model, loss_fn, cfg_attack,
setup)


    # ## Simulate an attacked FL protocol
    # Server-side computation:
    parameters = [p for p in model.parameters()]
    if args.provide_parameters:
        parameters = torch.load(args.parameters_path)
        parameters = [p.to(setup['device']) for p in parameters]


    public_buffers = [b for b in model.buffers()]
    if args.provide_public_buffers:
        public_buffers = torch.load(args.public_buffers_path)
        public_buffers = [b.to(setup['device']) for b in public_buffers]


    server_payload = [
        dict(parameters=parameters, buffers=public_buffers, metadata=data_cfg_
↪default)
    ]

    # User-side computation:
    gradients      = torch.load(args.gradient_path)
    gradients      = [g.to(setup['device']) for g in gradients]

    shared_buffers = None
    if args.provide_shared_buffers:
        shared_buffers = torch.load(args.shared_buffers_path)
        shared_buffers = [b.to(setup['device']) for b in shared_buffers]

    true_data  = torch.load(args.true_data_path)
    images     = true_data["images"]
    labels     = None
    if args.provide_labels:
        labels = true_data["labels"]
        labels = labels.to(setup['device'])

    shared_data = [
        dict(
            gradients=gradients,
            buffers=shared_buffers,
            metadata=dict(
```

```
                num_data_points=1,
                labels=labels,
                local_hyperparams=None,
            ),
        )
    ]

    overview(args, setup, model, attacker)

    # Attack:
    reconstructed_user_data, stats = attacker.reconstruct(server_payload,
shared_data, {}, dryrun=False)

    # Do some processing of your choice here. Maybe save the output image?
    reconstructed_images = reconstructed_user_data['data'].cpu()
    torchvision.utils.save_image(reconstructed_images, f"outputs/custom/{args.
↪attack}.png", normalize=True)
    torchvision.utils.save_image(images.cpu(), 'outputs/custom/true-data.png',
normalize=True)


if __name__ == "__main__":
    logging.basicConfig(stream=sys.stdout, level=logging.INFO, format='
↪%(message)s')
    main()
```

# Anexos

# ANEXO A – *Script minimal_example.py* original

*Script minimal_example.py*, sem as modificações propostas durante o experimento “Bem me quer, Mal me Quer” (nbrefcap3:experiment:flower)

```
"""
Example script to run attacks in this repository directly without simulation.
This can be useful if you want to check a model architecture and model gradients computed/defended in some shape or form
against some of the attacks implemented in this repository, without implementing your model into the simulation.

All caveats apply. Make sure not to leak any unexpected information.
"""
import torch
import torchvision
import breaching


class data_cfg_default:
    modality = "vision"
    size = (1_281_167,)
    classes = 1000
    shape = (3, 224, 224)
    normalize = True
    mean = (0.485, 0.456, 0.406)
    std = (0.229, 0.224, 0.225)


transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.Resize(256),
        torchvision.transforms.CenterCrop(224),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=data_cfg_default.mean, std=data_cfg_default.std),
    ]
)


```

```
def main():
    setup = dict(device=torch.device("cpu"), dtype=torch.float)

    # This could be your model:
    model = torchvision.models.resnet152(pretrained=True)
    model.eval()
    loss_fn = torch.nn.CrossEntropyLoss()

    # And your dataset:
    dataset = torchvision.datasets.ImageNet(root="~/data/imagenet", split="val",
transform=transforms)
    datapoint, label = dataset[1200]  # This is the owl, just for the sake of
this experiment
    labels = torch.as_tensor(label)[None, ...]

    # This is the attacker:
    cfg_attack = breaching.get_attack_config("invertinggradients")
    attacker = breaching.attacks.prepare_attack(model, loss_fn, cfg_attack,
setup)

    # ## Simulate an attacked FL protocol
    # Server-side computation:
    server_payload = [
        dict(
            parameters=[p for p in model.parameters()], buffers=[b for b in
model.buffers()], metadata=data_cfg_default
        )
    ]
    # User-side computation:
    loss = loss_fn(model(datapoint[None, ...]), labels)
    shared_data = [
        dict(
            gradients=torch.autograd.grad(loss, model.parameters()),
            buffers=None,
            metadata=dict(num_data_points=1, labels=labels, local_
↪hyperparams=None,),
        )
    ]

    # Attack:
    reconstructed_user_data, stats = attacker.reconstruct(server_payload,
shared_data, {}, dryrun=False)

    # Do some processing of your choice here. Maybe save the output image?


```

```
73 if __name__ == "__main__":
74     main()
```

# ANEXO B – Método *report* original

Método *report* do arquivo *analysis.py* antes das modifiçãoes implementadas em 3.2.1

```
def report(
    reconstructed_user_data,
    true_user_data,
    server_payload,
    model_template,
    order_batch=True,
    compute_full_iip=False,
    compute_rpsnr=True,
    compute_ssim=True,
    cfg_case=None,
    setup=dict(device=torch.device("cpu"), dtype=torch.float),
):
    log.info("Starting evaluations for attack effectiveness report...")
    model = copy.deepcopy(model_template)  # Copy just in case and discard later
    model.to(**setup)
    metadata = server_payload[0]["metadata"]
    if metadata["modality"] == "text":
        modality_metrics = _run_text_metrics(
            reconstructed_user_data, true_user_data, server_payload, cfg_case,
order_batch
        )
    else:
        modality_metrics = _run_vision_metrics(
            reconstructed_user_data,
            true_user_data,
            server_payload,
            model,
            order_batch,
            compute_full_iip,
            compute_rpsnr,
            compute_ssim,
            cfg_case,
            setup,
        )
    if reconstructed_user_data["labels"] is not None:
        test_label_acc = count_integer_overlap(
            reconstructed_user_data["labels"].view(-1),
            true_user_data["labels"].view(-1),
            maxlength=cfg_case.data.vocab_size,
```

```
        ).item()
    else:
        test_label_acc = 0

    feat_mse = 0.0
    for idx, payload in enumerate(server_payload):
        parameters = payload["parameters"]
        buffers = payload["buffers"]

        with torch.no_grad():
            for param, server_state in zip(model.parameters(), parameters):
                param.copy_(server_state.to(**setup))
            if buffers is not None:
                for buffer, server_state in zip(model.buffers(), buffers):
                    buffer.copy_(server_state.to(**setup))
            else:
                if len(true_user_data["buffers"]) > 0:
                    for buffer, user_state in zip(model.buffers(), true_user_
↪data["buffers"]):
                        buffer.copy_(user_state.to(**setup))

            # Compute the forward passes
            feats_rec = model(reconstructed_user_data["data"].to(device=setup[
↪"device"]))
            feats_true = model(true_user_data["data"].to(device=setup["device
↪"]))
            relevant_features = true_user_data["labels"]
            feat_mse += (feats_rec - feats_true)[..., relevant_features.view(-
↪1)].pow(2).mean().item()

    # Record model parameters:
    parameters = sum([p.numel() for p in model.parameters()])

    if metadata["modality"] == "text":
        m = modality_metrics
        log.info(
            f"METRICS: | Accuracy: {m['accuracy']:2.4f} | S-BLEU: {m['sacrebleu
↪']:4.2f} | FMSE: {feat_mse:2.4e} | "
            + "\n"
            f" G-BLEU: {m['google_bleu']:4.2f} | ROUGE1: {m['rouge1']:4.2f}|
ROUGE2: {m['rouge2']:4.2f} | ROUGE-L: {m['rougeL']:4.2f}"
            f"| Token Acc T:{m['token_acc']:2.2%}/A:{m['token_avg_accuracy']:2.2
↪%} "
            f"| Label Acc: {test_label_acc:2.2%}"
        )

```

```
    else:
        m = modality_metrics
        iip_scoring = " | ".join([f"{k}: {v:5.2%}" for k, v in m.items() if "IIP
↪" in k])
        log.info(
            f"METRICS: | MSE: {m['mse']:2.4f} | PSNR: {m['psnr']:4.2f} | FMSE:
↪{feat_mse:2.4e} | LPIPS: {m['lpips']:4.2f}|"
            + "\n"
            f" R-PSNR: {m['rpsnr']:4.2f} | {iip_scoring} | SSIM: {m['ssim']:2.4f}
| "
            f"max R-PSNR: {m['max_rpsnr']:4.2f} | max SSIM: {m['max_ssim']:2.4f}
| Label Acc: {test_label_acc:2.2%}"
        )

    metrics = dict(
        **modality_metrics,
        feat_mse=feat_mse,
        parameters=parameters,
        label_acc=test_label_acc,
    )
    return metrics
```

# ANEXO C – Método *dump_metrics* original

Método *dump_metrics* do arquivo *utils.py* antes das modificações implementadas em 3.2.2

```
]

def dump_metrics(cfg, metrics):
    """Simple yaml dump of metric values."""

    filepath = f"metrics_{cfg.case.data.name}_{cfg.case.model}_user{cfg.case.
↪user.user_idx}.yaml"
    sanitized_metrics = dict()
    for metric, val in metrics.items():
        try:
            sanitized_metrics[metric] = np.asarray(val).item()
        except ValueError:
            sanitized_metrics[metric] = np.asarray(val).tolist()
    with open(filepath, "w") as yaml_file:
        yaml.dump(sanitized_metrics, yaml_file, default_flow_style=False)
```