Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Desenvolvimento e análise de uma biblioteca de diferenciação automática para ambientes distribuídos

Luã Nowacki Scavacini Santilli

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Daniel Macêdo Batista

São Paulo
2023

O conteúdo deste trabalho é publicado sob a licença CC BY 4.0
(Creative Commons Attribution 4.0 International License)

Agradecimentos

What I cannot create, I do not understand
— Richard Feynman

Ao IME, pela oportunidade; à internet, pelo tácito; ao meu orientador, pela demanda; aos meus amigos, pelas discussões; e à minha família, pelo suporte.

Resumo

Luã Nowacki Scavacini Santilli. **Desenvolvimento e análise de uma biblioteca de diferenciação automática para ambientes distribuídos**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Nos últimos anos, a arquitetura de hardware para desenvolvimento de redes neurais enfatizou o uso de GPU's ou clusters de GPU's, o que pode ser visto pelo tamanho e arquitetura dos modelos no *benchmark* de Meta, 2024. A natureza paralela de redes neurais fez com que esse tipo de arquitetura paralela fosse de grande utilidade. Por exemplo, nota-se a AlexNet Krizhevsky *et al.*, 2012, que utilizou uma arquitetura com duas GPU's em paralelo para estabelecer o estado da arte para o ImageNet. A prevalência de GPU's, contudo, é cara e escalar para modelos ou sessões de treinamento maiores escala substancialmente o preço desse tipo de treinamento. O trabalho atual propõe uma forma de escalar treinamento através de redes de treinamento colaborativo com o foco em CPU's. Para fazer isso, uma biblioteca de diferenciação automática com o foco em ambientes distribuídos é desenvolvida na linguagem C. O nome da biblioteca é NfNN (Networking for Neural Networks). O desenvolvimento da biblioteca mostrou-se valioso no quesito de compreensão de como as técnicas de diferenciação automática e aprendizagem profunda funcionam, assim de como elas são implementadas em bibliotecas de uso geral. A biblioteca está disponível publicamente como software livre, permitindo que melhorias sejam realizadas por qualquer pessoa. Foi possível observar que as abordagens de treinamento distribuído propostas são efetivas, e que a biblioteca desenvolvida é capaz de realizar treinamento de redes neurais em ambientes distribuídos.

Palavras-chave: Diferenciação Automática. Sistemas Distribuídos. Aprendizado de Máquina.

Abstract

Luã Nowacki Scavacini Santilli. **Development and evaluation of an automatic differentiation library for distributed environments**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

In recent years, hardware architecture for neural network development has emphasized the use of GPUs or GPU clusters. This can be seen by the size and description of the architectures on the ImageNet benchmark available on Meta, 2024. The parallel nature of neural networks has made this type of parallel architecture highly useful. For instance, we can observe AlexNet Krizhevsky *et al.*, 2012, which used a two-GPU architecture in parallel to establish the state of the art in ImageNet. However, the prevalence of GPUs is expensive, and scaling to larger models or training sessions substantially increases the cost of this type of training. The current work proposes a way to scale training through collaborative training networks with a focus on CPUs. To achieve this, a library for automatic differentiation with a focus on distributed environments is developed in C. The name of the library is NfNN (Networking for Neural Networks). The development of the library proved valuable in terms of understanding how automatic differentiation and deep learning techniques work, as well as how they are implemented in general-purpose libraries. The library is publicly available as free software, allowing improvements to be made by anyone. It was possible to observe that the proposed distributed training approaches are effective, and that the developed library is capable of performing neural network training in distributed environments.

Keywords: Automatic Differentiation. Distributed Systems. Machine Learning.

Lista de figuras

Lista de tabelas

Sumário

Introdução

A biblioteca de diferenciação automática descrita nesta monografia NfNN (Networking for Neural Networks) é desenvolvida com o foco na compreensão das técnicas e abordagens utilizadas neste tipo de biblioteca. Aprendizado de máquina tende a este tipo de biblioteca porque um dos métodos mais comuns de treinamento de redes neurais envolve o cálculo de gradientes de funções de perda, o que é facilitado pelo uso de diferenciação automática.

Além do desenvolvimento da biblioteca de diferenciação automática em C, também desenvolvemos e testamos dois algoritmos de otimização distribuída. A arquitetura testada é uma simples rede inteiramente conectada com duas camadas para o conjunto de dados do MNIST LeCun *et al.*, 1998. Dois ambientes foram testados, uma numa máquina local e outro num Datacenter de máquinas. Os resultados mostram que os dois algoritmos de otimização distribuída testados são capazes de melhorar o treinamento no ambiente de Datacenter. O algoritmo de otimização centralizada síncrona melhorou a precisão no conjunto de validação final em cerca de 5% mantendo o tempo de execução do treinamento razoavelmente constante. Enquanto que o algoritmo de otimização centralizada assíncrona manteve a precisão no conjunto de validação final em menos de 1% da base, mas reduziu o tempo de execução do treinamento em cerca de 70%.

Nos próximos capítulos, primeiro apresentamos conceitos iniciais de bibliotecas de diferenciação automática e aprendizagem de máquina, assim como uma discussão de como diferentes bibliotecas são organizadas sobre estes conceitos. Em seguida, apresentamos os conceitos de otimização distribuída e alguns algoritmos de otimização. Depois, apresentamos os detalhes de desenvolvimento da biblioteca proposta, seguidos pelos resultados obtidos com os testes de algoritmos de otimização distribuída. Por fim, concluímos o trabalho com uma discussão sobre os resultados obtidos e possíveis trabalhos futuros.

Capítulo 1

Bibliotecas de diferenciação automática

Para diferentes problemas computacionais, existe a necessidade de calcular derivadas parciais de expressões definidas durante o programa. A capacidade de realizar isso de forma automática, conhecida como diferenciação automática, simplifica no desenvolvimento destes tipos de programas.

Neste capítulo, conceitos de bibliotecas de autodiferenciação são apresentados, juntamente com uma revisão de bibliotecas existentes.

1.1 Conceitos iniciais de bibliotecas de diferenciação automática

Nesta seção serão apresentados fatores comuns de bibliotecas de diferenciação automática, estes conceitos serão utilizados para analisar as bibliotecas que existem hoje no campo.

1.1.1 Bibliotecas de aprendizagem profunda vs diferenciação automática

A principal diferença entre bibliotecas de aprendizagem profunda e bibliotecas de diferenciação automática consiste na gama de funcionalidades que cada uma oferece. Bibliotecas de aprendizado profundo oferecem a capacidade de realizar diferenciação automática sobre operações definidas em seus principais objetos, porém também apresentam uma coleção de módulos úteis para o o desenvolvimento de modelos de redes neurais.

Dentre essas capacidades destacamos funções de perda, como erro quadrático médio ou entropia cruzada; otimizadores, como Adam ou SGD; módulos para o carregamento e transformação de conjunto de dados comuns, como MNIST ou ImageNet; e camadas normalmente utilizadas no desenvolvimento de um modelo de aprendizagem profundo,

como lineares ou convolucionais. Exemplos deste tipo de biblioteca são: PyTorch e Tensorflow.

Por outro lado, bibliotecas de diferenciação automática, como Autograd e JAX, focam primariamente no cálculo de gradientes definidos através de operações em tensores.

1.1.2 Definição de diferenciação automática

Diferenciação automática é uma técnica computacional utilizada para calcular derivadas parciais de uma função especificada num programa. Ao contrário de métodos numéricos de diferenciação, que podem sofrer com imprecisões devido à aproximação, e diferenciação simbólica, que pode ser computacionalmente proibitiva, a diferenciação automática decompõe uma determinada função em uma série de operações elementares, cujas derivadas são conhecidas. E através do uso da regra da cadeia, calcula a derivada parcial de uma determinada operação.

Em outras palavras, diferenciação automática é uma forma de simplificar o desenvolvimento de programas que requerem o cálculo de derivadas parciais para operações definidas no programa.

Um exemplo de problema que pode ser resolvido através do uso de diferenciação automática é encontrar o mínimo de uma função. Para isso, é necessário calcular o gradiente da função e utilizar um método de otimização para encontrar o mínimo. O gradiente de uma função é definido como o vetor das derivadas parciais da função. Um dos métodos mais comuns para realizar a otimização é a descida de gradiente estocástica, que consiste em atualizar o valor dos pesos da função de acordo com um estimador do gradiente calculado através de uma amostra do conjunto de dados.

O trecho de pseudo-código 1 mostra como uma biblioteca que implementa diferenciação automática pode ser utilizada para calcular o gradiente de uma função arbitrária. A função l é definida como x^2+xy+y^2, e o gradiente de l é calculado através da função AutoDiferencie, que utiliza diferenciação automática para calcular o gradiente de l em relação a x e y. Esse trecho é baseado na implementação de diferenciação automática como implementada no PyTorch. O resultado é impresso na tela.

Algoritmo 1 Exemplo de Uso de Diferenciação Automática

```
1: l ← (x^2 + xy + y^2)          ▷ Define a expressão
2: AutoDiferencie(l)             ▷ Calcula o gradiente
3: print(x.grad)                 ▷ Imprime ∂l/∂x
4: print(y.grad)                 ▷ Imprime ∂l/∂y
```

1.1.3 Operações em tensores

Uma das ideias principais de bibliotecas de aprendizado de máquina é a capacidade de se trabalhar com tensores. Tensores, neste contexto, são uma generalização do conceito de matrizes para um número maior de dimensões. O valor dessa abordagem é a de que a utilização de uma mesma operação em um conjunto de dados pode aumentar a performance de um programa.

Esse tipo de abordagem é conhecido como Single-Instruction Multiple Data (SIMD). E a formulação de um problema de aprendizado de máquina como um problema de realização de múltiplas operações deste tipo permite a utilização de hardware especializado para esse tipo de problema. GPUs e vetorização na CPU, e.g. SSE4.2, AVX, AVX-2, são exemplos de hardware que podem aumentar a velocidade de execução.

Vale notar a razão pela vantagem de uma abordagem (SIMD): hardware moderno (CPU) tende a ser dividido em um front-end e um backend. O front-end tem como objetivo decodificar as instruções e o backend de executá-las. A utilização de SIMD permite que o trabalho do front-end seja reaproveitado para um conjunto maior de dados.

1.1.4 Descida de gradiente

Antes de apresentar o valor de bibliotecas de diferenciação automática, é necessário discutir o valor do algoritmo de descida de gradiente, dada a ubiquidade do algoritmo no contexto de aprendizado de máquina Sun *et al.*, 2020.

Descida de gradiente é um algoritmo de otimização amplamente utilizado para encontrar o mínimo local de uma função diferenciável. O princípio por trás do método é simples: a partir de um ponto inicial, o algoritmo toma passos sucessivos na direção oposta ao gradiente (ou aproximado gradiente) da função no ponto atual, porque esta é a direção de declive mais acentuado. O tamanho dos passos é determinado por um parâmetro chamado taxa de aprendizado. No Algoritmo 2, apresentamos um pseudocódigo simples para ilustrar o procedimento de descida de gradiente para minimizar uma função $f(x)$.

A Figura 1.1 ilustra o processo de descida de gradiente. O valor da função origina no círculo preto. A cada iteração do algoritmo o gradiente da função para cada parâmetro é calculado e esse gradiente é utilizado para atualizar os parâmetros, com o propósito de reduzir o valor da função.

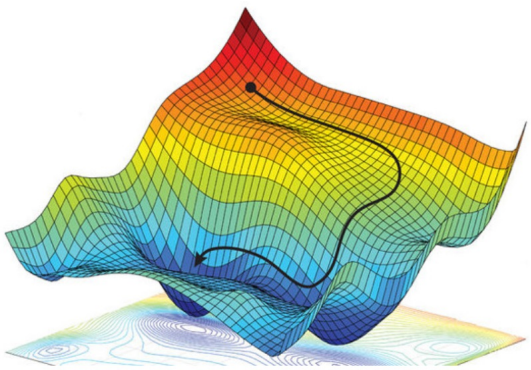

Figura 1.1: *Ilustração do algoritmo de descida de gradiente. Originalmente publicado em Amini et al., 2019*

.

1.1.5 Avaliação imediata vs. avaliação preguiçosa

Algumas bibliotecas de diferenciação automática utilizam de uma abordagem imediata, em que o cálculo de operações em tensores é realizado sobre a linha de código em que a

Algoritmo 2 Descida de Gradiente Simples

```
1: function DescidaDeGradiente(f, ∇f, x_inicial, α, max_iteracoes)
2:     x ← x_inicial
3:     for i = 1 to max_iteracoes do
4:         g ← ∇f(x)                          ▷ Calcula o gradiente da função em x
5:         x ← x − α · g                      ▷ Atualiza o ponto atual
6:     end for
7:     return x
8: end function
```

operação é definida. Outras bibliotecas utilizam de uma abordagem preguiçosa, em que o cálculo das operações é realizado apenas quando necessário, por exemplo, quando o gradiente de uma função é necessário para o cálculo de uma operação posterior. Essa abordagem preguiçosa normalmente requer a construção de um grafo computacional para que a inferência ou retropropagação seja realizada.

Esta segunda abordagem é mais eficiente, pois permite que uma série de otimizações sejam executadas de forma a aumentar a performance do programa. Porém, a necessidade de compilar um modelo antes de sua execução diminui a ergonomia de desenvolvimento [1] ao dificultar a depuração do estado de um modelo uma vez que os resultados das expressões não ocorrem imediatamente.

1.1.6 Capacidade de fusão de kernel

Bibliotecas que implementam o paradigma de avaliação preguiçosa podem mais facilmente utilizar do mecanismo de fusão de kernel. Esta é uma técnica que permite que várias operações distintas sejam combinadas para uma mais eficiente. No contexto de desenvolvimento de uma GPU essa técnica também pode diminuir a quantidade de transferências de memória para a GPU Wang *et al.*, 2010.

1.1.7 Alocação de memória

Alocação de memória pode ser um problema significativo na performance de uma aplicação. Esse problema é ainda mais marcante para bibliotecas de aprendizagem profunda, dado a escassez de memória disponível para GPUs e dado que um padrão recorrente para tentar melhorar a performance de um modelo consiste em aumentar o tamanho do lote de dados utilizados para o treinamento Paszke *et al.*, 2019.

Contudo, essa questão deve ser ponderada com a ergonomia do uso da biblioteca, uma vez que a alocação de memória pode ser uma complicação para o usuário da biblioteca.

Esse dilema fez com que diferentes bibliotecas adotassem diferentes estratégias para a

[1] Ergonomia de desenvolvimento se refere a facilidade de realizar alguma forma de desenvolvimento através do uso de outras facilidades de código atingindo um resultado igual ou similar ao proposto anteriormente. Neste caso ergonomia indica que a não necessidade de compilar um modelo diminui a quantidade de passos necessários para testar um modelo.

alocação de memória. Por exemplo, a biblioteca PyTorch utiliza um mecanismo de contagem de referências ao invés de simples coleta de lixo. Isso permite que a memória seja liberada imediatamente após o uso, o que pode ser importante para aplicações que utilizam de muita memória Paszke *et al.*, 2019. Já a biblioteca JAX realiza a pre-alocação de 75% da memória da GPU para reduzir a necessidade de alocar memória durante a execução do programa Authors, 2023.

1.2 Análise de diferentes bibliotecas de diferenciação automática

Nesta secção é apresentada uma análise de como as bibliotecas de autodiferenciação lidam com as escolhas apresentadas anteriormente. Outras características interessantes também são ressaltadas.

1.2.1 PyTorch

PyTorch é uma biblioteca que ganhou espaço em publicações no meio acadêmico e na indústria. Ela por padrão apresenta um método de avaliação imediata; não possui a capacidade de fusão de kernel; e implementa um alocador de tensores com o objetivo de saturar as operações na GPU enquanto outras são realizadas na CPU Paszke *et al.*, 2019.

No entanto, vários esforços nos últimos anos foram realizados com o objetivo de permitir que os modelos desenvolvidos em PyTorch sejam mais eficientes. Entre eles o desenvolvimento de um módulo que é capaz de compilar modelos parcialmente, a fim de também permitir a fusão de kernel Foundation, 2023.

A implementação da biblioteca é realizada majoritariamente em C++, enquanto que os módulos de alto nível e a apresentação da biblioteca são desenvolvidos em Python Parvat *et al.*, 2017.

1.2.2 TensorFlow

Tensorflow é uma biblioteca de aprendizado de máquina desenvolvida pelo Google posteriormente ao DistBelief Dean *et al.*, 2012. Historicamente, uma iteração de inferência ou retropropagação requereria a construção de um grafo computacional, portanto era preguiçosa por padrão; no artigo em que a biblioteca é descrita são levantadas diversas considerações sobre a capacidade de aprendizagem distribuída Abadi *et al.*, 2016.

Em termos de alocação de memória, a biblioteca aloca toda a memória disponível da GPU Thakur, 2022. A implementação da biblioteca é realizada principalmente em C++, enquanto que a interface de alto nível é desenvolvida em Python Parvat *et al.*, 2017.

1.2.3 JAX

JAX é uma biblioteca de diferenciação automática focada no conceito de transformações de alto nível. A biblioteca é desenvolvida em Python e utiliza de um compilador JIT (just-

in-time) para gerar código em C++ e CUDA Bradbury *et al.*, 2018.

Por padrão a biblioteca prealoca 75% da memória da GPU para diminuir a necessidade de alocar memória durante a execução do programa Authors, 2023. A biblioteca também implementa um mecanismo de avaliação preguiçosa, o que permite a utilização do mecanismo de fusão de kernel Wang *et al.*, 2010. Assim como a capacidade de JIT compilation, que permite que a biblioteca utilize de otimizações de código que não seriam possíveis em tempo de execução.

1.2.4 GGML

GGML é uma implementação em C de uma biblioteca de operações em tensores. Vem ganhando popularidade por sua velocidade de inferência em grandes modelos a partir do enfoque em otimizar o hardware a nível do consumidor Gerganov, 2023. A biblioteca é desenvolvida em C e requer que o usuário prealoque a memória necessária para a execução do programa. A biblioteca implementa um mecanismo de avaliação preguiçosa.

Vale ressaltar que a biblioteca utiliza um mecanismo de arenas de memória para alocar e desalocar memória de forma mais eficiente, apesar de que como um grafo computacional é construído e a avaliação é preguiçosa, é possível estabelecer um limite superior para a quantidade de memória que será utilizada durante a execução de inferência dado um tamanho de lote.

Outro destaque é que essa biblioteca foi desenvolvida inicialmente sem o auxílio de um grande time de desenvolvedores financiados por uma grande empresa de tecnologia, como é o caso das bibliotecas TensorFlow e JAX, pelo Google; e PyTorch, pela Meta.

1.2.5 TinyGrad

TinyGrad é uma implementação de bibliotecas de diferenciação automática com o objetivo de simplicidade. A princípio a biblioteca tentava ser implementada em menos do que 1000 linhas de Python. Com o tempo esse limite arbitrário foi perdido.

A arquitetura do TinyGrad é preguiçosa, com uma API similar ao PyTorch. Todas as operações a nível de tensor são implementadas com 4 diferentes tipos de operação. São estas: Unárias (como funções de ativação), Binárias (combinam dois tensores, e.g. Adição, Subtração, ...) Reduções (convertem um tensor numa forma menor do mesmo - somas, médias) e Alterações no formato (como reshapes)

Essa simplicidade de alto nível permite que seja substancialmente mais simples de portar a biblioteca para um novo acelerador. Aceleradores são considerados como formatos específicos de hardware que são utilizados para acelerar a velocidade das operações, exemplos são GPUs, TPUs, FPGAs e AVX-2 Hotz, 2023.

1.3 Discussão

Nesta seção são discutidas as escolhas feitas pelas bibliotecas de diferenciação automática e como elas afetam a ergonomia e performance da biblioteca. Neste contexto de

diferentes bibliotecas, o sucesso de performance em termos de inferência da biblioteca GGML sugere que existe um espaço para melhorias através do uso de uma implementação e interface focada em C. TensorFlow e Pytorch são primeiramente desenvolvidas em C++, mas a interface de alto nível é desenvolvida com enfoque em Python. Além disso, a maior parte destas bibliotecas trata de performance com foco no uso de GPU. Assim, com o objetivo de estudar as características num ambiente de CPU e melhor entender o funcionamento de uma biblioteca de diferenciação automática, é válida a implementação de uma nova biblioteca.

Capítulo 2

Conceitos básicos de treinamento distribuído

A ideia de sistemas distribuídos toma como base o aproveitamento do poder computacional de diferentes processadores conectados via uma rede para diminuir o tempo de execução de um problema computacional. O problema é dividido em problemas menores que são resolvidos separadamente por cada um dos processadores. Uma das vantagens de tal sistema é o custo. Uma coleção de computadores com CPU tradicional pode ser mais barato do que a utilização de um supercomputador. Essa ideia é conhecida como escalabilidade horizontal, isto é, a adição de mais máquinas para aumentar o poder computacional do sistema; contra a escalabilidade vertical, que consiste em adicionar mais poder computacional a uma máquina já existente.

2.1 Aprendizado federado vs distribuído

Quando busca-se realizar aprendizado em um sistema distribuído, dois modelos se destacam: aprendizado federado e aprendizado distribuído. A principal diferença entre aprendizado federado e distribuído reside na motivação de cada uma das abordagens. Aprendizado federado é motivado pela privacidade dos dados, enquanto que aprendizado distribuído, pela eficiência no treinamento de modelos Zhang *et al.*, 2021.

Essas diferentes motivações permitem que hipóteses distintas sejam assumidas em cada uma das abordagens. Por exemplo, aprendizado distribuído pode assumir que as distribuições de dados ocorrem de maneira razoavelmente uniforme entre os nós, enquanto que o aprendizado federado não pode assumir isso, pois os dados são privados e não podem ser compartilhados.

A Figura 2.1 ilustra a diferença entre as duas abordagens. No aprendizado federado, cada um dos nós possui um conjunto de dados distinto. O objetivo é realizar o treinamento de um modelo de aprendizado de máquina com todos os dados disponíveis, sem que os dados sejam compartilhados entre os nós. Já no aprendizado distribuído, os dados são distribuídos entre os nós e o objetivo é realizar o treinamento com uma melhora na eficiência computacional.

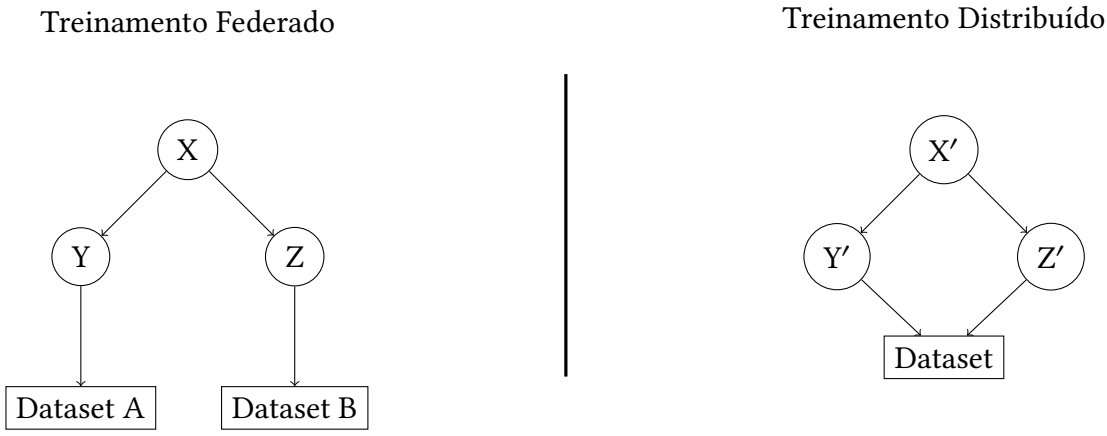

Figura 2.1: *Treinamento federado vs distribuído*

2.2 Hiperparâmetros

Uma das abordagens mais simples de treinamento distribuído consiste em realizar experimentos com diferentes hiperparâmetros em máquinas distintas. Nota-se que esse problema de seleção de hiperparâmetros é embaraçosamente paralelo, isto é, pode-se realizar experimentos com diferentes hiperparâmetros em paralelo com mínimos requerimentos de sincronização.

Por exemplo, máquina (1) realiza determinado experimento com uma taxa de aprendizado de 0,1 enquanto que a máquina (2) realiza determinado experimento com uma taxa de aprendizado igual a 0,01.

2.3 Paralelismo de modelo vs paralelismo de dados

Uma das principais distinções entre as estratégias de treinamento distribuído é a diferença entre o paralelismo a nível do modelo e o paralelismo a nível dos dados. Nota-se que abordagens híbridas são possíveis.

A abordagem de paralelismo a nível do modelo particiona este em máquinas ou nós distintos. Para realizar a inferência ou retropropagação, é necessário que os sinais do modelo sejam transmitidos entre máquinas. Essa abordagem é mais utilizada para realizar o treinamento de redes neurais profundas, nas quais o número de parâmetros do modelo supera o permitido em uma só máquina ou GPU.

Já na abordagem a nível de dados, o modelo é replicado em diferentes máquinas e cada uma delas recebe uma parte dos dados para realizar o treinamento. Isso torna necessário algum mecanismo de comunicação para que os diferentes nós da rede atuem de maneira colaborativa com o objetivo de minimizar o tempo de treinamento.

Para ambientes de grande latência, a abordagem de paralelismo a nível de dados sugere-se mais proveitosa, tanto por ser mais simples em termos de implementação, quanto por existirem mecanismos que diminuem a necessidade de sincronizar o modelo em cada iteração de inferência ou retropropagação.

Dado o foco em ambientes com grande latência deste trabalho, os experimentos e testes desenvolvidos focam no paralelismo ao nível de dados. Grande latência aqui é definida pela utilização de um protocolo de redes como TCP para a comunicação entre nós, oposto a comunicação realizada por meio de memória compartilhada (num ambiente de várias GPUs em uma mesma máquina, por exemplo).

A Figura 2.2 ilustra a diferença entre as duas abordagens. Na abordagem de paralelismo a nível de modelo, o modelo é particionado em diferentes máquinas. Para realizar a retropropagação, é necessário que os sinais do modelo sejam transmitidos entre máquinas. Já na abordagem de paralelismo a nível de dados, o modelo é replicado em diferentes máquinas e cada uma delas recebe uma parte dos dados para realizar o treinamento. Isso torna necessário algum mecanismo de comunicação para que os diferentes nós da rede atuem de maneira colaborativa com o objetivo de minimizar o tempo de treinamento.

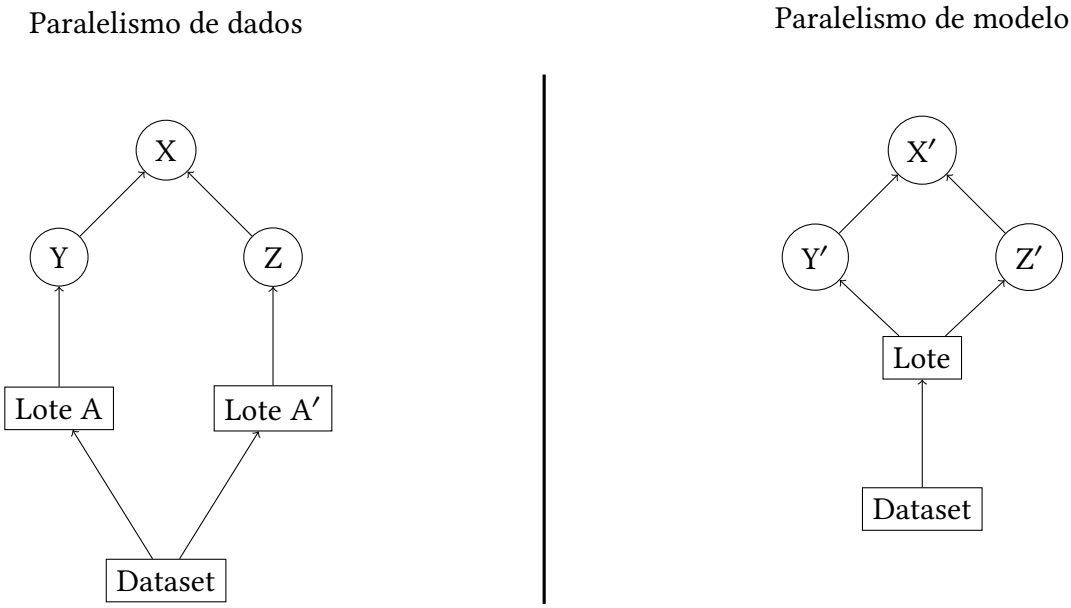

Figura 2.2: *Paralelismo de dados vs paralelismo de modelo*

2.4 Otimização centralizada vs descentralizada

Em termos de paralelismo a nível de dados, podemos elencar duas abordagens: otimização centralizada, na qual um nó central é responsável por atualizar pesos do modelo e otimização descentralizada, em que múltiplos nós são responsáveis por atualizar os pesos do modelo.

Em otimização centralizada, um nó recebe os gradientes de outros nós da rede, e os utiliza para atualizar os pesos do modelo. Depois disso esse nó central envia os pesos atualizados para os outros nós da rede. A forma como essa atualização é realizada é definida por uma outra subdivisão de sincronização, que será abordada adiante.

Assim, cada um dos nós é responsável por calcular um estimador do gradiente, efetivamente permitindo o aumento do tamanho de um *minibatch*.

Já em otimização descentralizada, os nós também são responsáveis por atualizar a sua versão do modelo. Isso permite que mais de uma iteração de retropropagação e atualização seja realizada por nó antes da sincronização. Preocupações adicionais devem ser empregadas para garantir que os gradientes estimados por cada um dos nós não tendam a ser divergentes Xin *et al.*, 2020.

Para ilustrar a diferença entre as duas abordagens, podemos utilizar a parte esquerda da Figura 2.2 como exemplo. No caso de otimização centralizada o nó X atua como o otimizador e recebe os gradientes de Y e Z para atualizar os pesos do modelo. Já em otimização descentralizada, os nós Y e Z são também responsáveis por atualizar os pesos do modelo, no entanto ainda é necessário que os nós se comuniquem eventualmente para garantir que os pesos do modelo sejam atualizados de maneira consistente. Esta comunicação é representada pelo nó X.

2.5 Otimização síncrona vs assíncrona

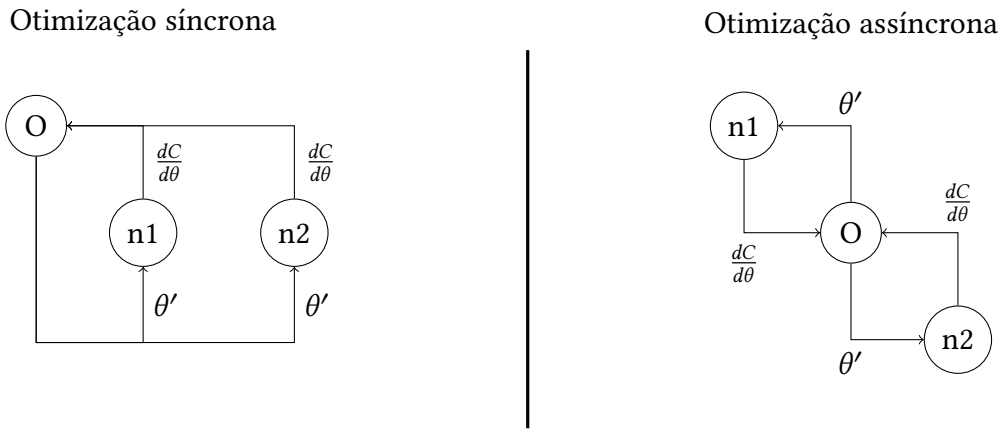

Figura 2.3: *Otimização síncrona vs assíncrona*

Uma outra subdivisão de sistemas de aprendizagem distribuída consiste na forma como os trabalhadores operam.

Para otimização síncrona, consideramos a existência de um nó central responsável por atualizar os pesos do modelo. Neste caso, o nó central espera receber os n gradientes dos trabalhadores para atualizar os pesos do modelo. Isso permite que os n trabalhadores auxiliem na geração de um melhor estimador do gradiente, permitindo um aumento na taxa de aprendizado McCandlish *et al.*, 2018.

Essa abordagem, contudo, gera eventuais estagnações dos trabalhadores pois requer que os os n nós terminem uma iteração antes de receberem os pesos atualizados. E portanto é mais adequada para cenários em que os diferentes nós da rede tenham uma performance razoavelmente uniforme, tanto em termos de *hardware* quanto em termos de taxa de comunicação.

Ainda considerando o caso de um nó central, um sistema de otimização assíncrona recebe o gradiente de um dos n nós, atualiza os pesos do modelo, e envia os novos pesos para o mesmo nó. Isso possibilita que os trabalhadores não tenham que esperar o mais

lento, assim aumentando a taxa de atualização do nó central. Porém, ao permitir que os diferentes nós atuem assincronamente, o impasse de nós suspensos a cada iteração é substituído pela necessidade de assegurar que os diferentes pesos ao longo da rede não estejam obsoletos.

A Figura 2.3 ilustra a diferença entre os dois tipos de otimização. No caso síncrono, o nó central espera receber os gradientes de todos os nós antes de atualizar os pesos do modelo. Já no caso assíncrono, o nó central recebe o gradiente de um dos nós, atualiza os pesos do modelo e envia os novos pesos para o mesmo nó.

Capítulo 3

Desenvolvimento da biblioteca

No desenvolvimento de software várias decisões de projeto devem ser tomadas. Essas decisões vão desde a linguagem de programação a ser utilizada até especificações do problema a ser resolvido. Este capítulo descreve as decisões relacionadas à biblioteca para autodiferenciação que foi desenvolvida.

3.1 Escolha de linguagem

Baseado na análise das bibliotecas no Capítulo 1 podemos observar uma dualidade entre bibliotecas de aprendizado profundo. Por um lado, as bibliotecas são implementadas em linguagens que permitem um maior controle sobre o uso de memória e performance, como C e C++, porém disponibilizam uma interface de alto nível para o usuário em uma outra linguagem, como Python.

Uma das vantagens dessa abordagem é que o uso de uma linguagem como Python permite uma série de conveniências para o usuário da biblioteca, como a não necessidade de gerenciamento manual de memória; a disponibilidade de uma série de pacotes e módulos pré-desenvolvidos, que permitem facilmente a utilização de plotagem, leitura de conjunto de dados entre outros; e a capacidade de desenvolvimento interativo, a partir do uso de ferramentas como notebooks.

No entanto, o sucesso de bibliotecas de inferência que também disponibilizam a interface em um linguagem de baixo nível, como GGML sugere que existe um possibilidade de desenvolvimento para uma biblioteca de diferenciação automática neste contexto. E, como será abordado nas próximas seções, os métodos de uso comum de bibliotecas de aprendizagem profunda permitem que diferentes estratégias de desenvolvimento sejam adotadas para melhorar a ergonomia do uso numa linguagem de gerenciamento manual de memória. Dessa forma, a linguagem escolhida para o desenvolvimento foi C.

3.2 Alocação de memória

Como mencionado na Seção 1.1.7 a alocação de memória é um problema crucial no desenvolvimento de uma biblioteca de aprendizagem profunda. Sendo necessário minimizar o impacto da latência na alocação de memória assim como a ergonomia para o usuário da biblioteca.

Analisando os padrões comuns de uso de memória no processo de treinamento de uma biblioteca de aprendizagem profunda, é possível identificar que existem dois ciclos de utilização principal de memória: a alocação dos parâmetros do modelo e otimizadores e a alocação de memória intermediária para a execução de inferência e retropropagação.

A alocação dos parâmetros do modelo é um processo que ocorre apenas uma vez, antes do início do treinamento, e o tamanho da memória alocada é proporcional ao tamanho do modelo. Ao final do treinamento, espera-se que esta memória seja salva num arquivo para que possa ser reutilizada posteriormente. E no contexto de aprendizagem distribuída, é de extrema utilidade que essa memória seja facilmente enviada e recebida entre diferentes nós de uma rede.

Já a alocação dentro de um ciclo de treinamento ou inferência pode ser tanto um processo que ocorre uma única vez, quanto um processo que ocorre a cada iteração. Essa diferença depende da criação ou não de um grafo computacional antes da execução de uma iteração. No entanto, em ambos os casos, também é possível definir um limite superior da quantidade de memória utilizada no processo visto os hiperparâmetros do modelo como o tamanho do lote e o número de transformações dentro do ciclo.

Dada a escolha da biblioteca de expandir o grafo computacional de um tensor a cada operação, a alocação de memória intermediária ocorre a cada iteração do ciclo de treinamento ou inferência. Porém, a capacidade de estabelecer um limite superior para a quantidade de memória utilizada no processo permite a utilização de uma estratégia de alocação de memória que é simultaneamente rápida e ergonômica.

Essa estratégia, chamada de alocação por arena ou regiões de memória, consiste na alocação de uma grande quantidade de memória de uma só vez e na utilização de um ponteiro para indicar a posição atual de alocação dentro dessa região. A cada alocação, o ponteiro é incrementado de acordo com o tamanho da alocação e a cada desalocação, o ponteiro é decrementado. Isso traz duas vantagens principais: a primeira é que a alocação de memória é um processo rápido, de velocidade similar a alocação de pilha, pois não é necessário realizar uma chamada ao sistema operacional para cada alocação, e a segunda é que a memória fica alocada de forma contígua, o que diminui o impacto da latência de acesso à memória; facilita a utilização de instruções de vetorização e permite a utilização de instruções de prefetching para melhorar o desempenho do acesso à memória Hanson, 1990.

Além de sua velocidade, a alocação por arena também é ergonômica, pois permite que o usuário da biblioteca não precise se preocupar em desalocar os tensores alocados intermediariamente durante o processo de treinamento ou inferência, pois a desalocação ocorre de forma automática ao final do ciclo. O usuário somente tem que se preocupar em definir um limite superior para a quantidade de memória utilizada antes de um loop e em

desalocar a região no final de um loop.

3.3 Diferenciação automática

Como exposto na Seção 1.1.5, no que diz respeito a forma da construção de um grafo computacional, existem duas maneiras comuns de implementar diferenciação automática, de forma imediata ou preguiçosa. Dada a facilidade de depuração no contexto de bibliotecas que criam o grafo computacional de forma imediata, esta foi a escolha para a biblioteca.

A nível de implementação, isso significa que em qualquer operação realizada sobre diferentes tensores da biblioteca, um grafo computacional é expandido. Por exemplo, considere o seguinte trecho de código:

```
1    nfnn_tensor *A = NfNN_Const(Mem, NfNN_Dim2(1, 1), 2.0f);
2    nfnn_tensor *B = NfNN_Const(Mem, NfNN_Dim2(1, 1), 3.0f);
3    nfnn_tensor *C = NfNN_Mul(Mem, A, B);
```

Nesse caso são definidos dois tensores A e B como matrizes $1x1$ com o valores 2 e 3 respectivamente. Logo depois, é executada a operação *NfNN_Mul*. Esta operação aloca um novo tensor C sobre a arena de memória *Mem*. Além disso, ela salva no tensor C uma operação de adição sobre A e B. Essa operação salva permite que após a execução de *NfNN_AutoGrad_Backward* os gradientes de $\frac{\partial C}{\partial A}$ e $\frac{\partial C}{\partial A}$ sejam calculados. Considere agora o seguinte trecho:

```
1    NfNN_AutoGrad_Backward(Mem, C);
2    assert(A->Gradient.Data[0] == 3.0f);
3    assert(B->Gradient.Data[0] == 2.0f);
```

Este exemplo continua o acima, e mostra o cálculo dos gradientes depois da chamada de *NfNN_AutoGrad_Backward*. Como $C = A * B$ temos que $\frac{\partial C}{\partial A} = B$ e $\frac{\partial C}{\partial B}$.

O cálculo da retropropagação deve ser realizado sobre um tensor que pode ser convertido para um escalar, ou seja, somente possui uma dimensão significativa. Isso permite que os gradientes para cada um dos valores do modelo seja calculado. Cada gradiente do modelo pode ser visto como a sensibilidade de alteração no valor de C com respeito ao parâmetro.

Para calcular a derivada parcial de um elemento θ_k do modelo podemos utilizar a regra da derivativa total Parr e Howard, 2018. Considere que C constitui um valor escalar, como o resultado de uma função de custo, também considere que conhecemos todas as derivadas parciais de S em relação a C, dessa forma, podemos expressar a regra da derivativa total como:

$$\frac{\partial f(S_1, \ldots, S_{n+1})}{\partial \theta_k} = \sum_{i=1}^{n+1} \frac{\partial f}{\partial S_i} \frac{\partial S_i}{\partial \theta_k}$$

O algoritmo de retropropagação pode ser definido como o cálculo em ordem de derivadas parciais em relação a um escalar. Considere que queremos calcular $\frac{\partial C}{\partial x}$ para a expressão

$C = \sigma(x) + x$, que possui grafo computacional ilustrado na Figura 3.1.

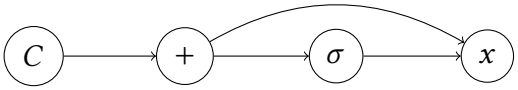

Figura 3.1: *Grafo computacional para* $C = \sigma(x) + x$

Esse caso indica que precisamos primeiro achar a ordem topológica sobre as operações antes de executar o cálculo de diferenciação em relação a um parâmetro.

Para achar a ordem topológica podemos implementar um simples algoritmo de ordenação topológica, expresso no Algoritmo 3.

Algoritmo 3 Ordenação topológica

```
    L ← lista vazia que irá conter os elementos ordenados
    S ← conjunto de todos os nós com marcação ‘não visitado‘
 3: function VISIT(node)
        if node está marcado como ‘visitado‘ then
            return
 6:     end if
        marque node como ‘visitado‘
        for cada nó m com uma aresta de node para m do
 9:         VISIT(m)
        end for
        adicione node ao início de L
12: end function
    for cada nó n em S do
        VISIT(n)
15: end for
    return L                                  ▷ Lista ordenada topologicamente
```

Uma vez tendo uma lista L de tensores para utilizar a regra da derivativa total, podemos iterar para cada um dos tensores, e dependendo do tipo de operação, calcular o gradiente. Considerando C como um tensor que pode ser convertido para um escalar, podemos ilustrar o algoritmo de retropropagação como o Algoritmo 4.

Algoritmo 4 Retropropagação

```
    L ← TopologicalSort(C)
    C.Grad ← 1.0
    for cada tensor t em L do
 4:     for cada tensor d que t depende do
            d.Grad + = t.Grad ∗ Deriv(t, d)
        end for
    end for
```

As dependências de t podem ser calculadas pelo tipo de operação em que t foi definido. Por exemplo, se t for gerado pela expressão $t = A + B$ temos que t depende de dois tensores,

A e B. Similarmente, caso t seja o resultado de uma função de ativação, como uma ReLU, terá somente um tensor como de dependência.

3.4 Otimizadores

Para implementar otimizadores, como SGD ou Adam, precisamos manter uma lista dos parâmetros do modelo. Dessa forma é necessário implementar uma função que adiciona os parâmetros de uma rede neural no otimizador, uma função para realizar o update baseado nos gradientes de um dado tensor, e uma função para zerar os gradientes.

Para a biblioteca NfNN, decidimos utilizar uma lista ligada, isso permite que diferentes tensores sejam adicionados nesta lista. Essa forma também permite uma iteração pelos diferentes tensores da lista.

O código a seguir demonstra a interface para a utilização do otimizador Adam. São demonstrados as funções para criar um otimizador, para adicionar parâmetros ao otimizador, para zerar os gradientes presentes e para realizar um passo de otimização depois do cálculo dos gradientes.

```
1    // Cria um otimizador do tipo Adam com taxa de aprendizado Lr
2    // Numero de trabalhadores NW definida com Beta1 e Beta2
3    nfnn_optimizer *Optim = NfNN_Optimizer_Adam(Mem, Lr, NW, Beta1, Beta2);
4    // Adiciona os pesos W1 e W2 no otimizador
5    NfNN_Optimizer_AddParam(Mem, Optim, W1);
6    NfNN_Optimizer_AddParam(Mem, Optim, W2);
7    // Zera os gradientes
8    NfNN_Optimizer_ZeroGrad(Optim);
9    // Realiza uma iteracao de otimizacao
10   NfNN_Optimizer_Step(Optim);
```

3.5 Implementação de um servidor de parâmetros

Um servidor de parâmetros, no contexto de otimização distribuída, consiste num computador que recebe os gradientes de diferentes trabalhadores, junta esses gradientes de alguma forma, realiza uma iteração de otimização e envia os novos pesos da rede para os trabalhadores. Como discutido anteriormente, isso pode ser feito de forma síncrona (esperando todos os trabalhadores enviarem os gradientes para atualizar o modelo) ou de forma assíncrona (realizando um passo de otimização assim que algum dos trabalhadores enviar um gradiente).

Para implementar um servidor de parâmetros é preciso criar uma interface que seja capaz de lidar com os dois cenários.

Segue abaixo uma implementação de otimização síncrona:

```
1    // Cria um servidor de parâmetros para um Ip em determinada porta
2    nfnn_parameter_server *Server = NfNN_ParameterServer_Create(Mem, Ip, Port);
3    // Espera a conexão dos trabalhadores no servidor
4    for (u32 G = 0; G < NumberOfWorkers; G++)
5    {
6       NfNN_ParameterServer_AddWorker(Mem, Server);
```

```
    }
    // Envia todos os pesos W1 e W2 para todos os trabalhadores
    // com o modelo de broadcasting
    NfNN_ParameterServer_BroadcastWeights(Mem, Server, W1);
    NfNN_ParameterServer_BroadcastWeights(Mem, Server, W2);
    // Espera todos os trabalhadores finalizarem
    NfNN_ParameterServer_WaitWorkers(Mem, Server, NumberOfWorkers);
    // Espera os gradientes
    NfNN_ParameterServer_AwaitGradient(Mem, Server, W1);
    NfNN_ParameterServer_AwaitGradient(Mem, Server, W2);
```

Segue abaixo uma implementação de otimização assíncrona:

```
    // Segue inicialização conforme o exemplo síncrono
    // Espera um trabalhador começar a enviar um gradiente
    nfnn_platform_socket Sock = NfNN_ParameterServer_AwaitForWorker(Mem, Server);

    // Recebe os gradientes W1 e W2
    NfNN_Network_RecvAddGradient(Mem, Sock, W1);
    NfNN_Network_RecvAddGradient(Mem, Sock, W2);
    // Realiza um passo de otimizacao
    NfNN_Optimizer_Step(Optim);
    // Envia os novos pesos para o trabalhador
    NfNN_Network_SendTensor(Mem, Sock, W1);
    NfNN_Network_SendTensor(Mem, Sock, B1);
```

Para implementar a espera de leitura de gradientes foi utilizado o mecanismo de select, que possibilita verificar se um socket de um conjunto está pronto para ser lido. Esse mecanismo não é o mais eficiente, como pode ser visto na análise de bibliotecas como nginx ou redis, porém é mais portável (funciona tanto em windows como em linux) Gammo *et al.*, s.d.

3.6 Diferentes backends

A natureza de bibliotecas de aprendizagem profunda aproveita em muito o poder computacional de diferentes máquinas. Isso implica que para ter uma melhor performance é necessário aproveitar tanto características do hardware em específico (como a disponibilidade de instruções vetorizadas) como em computação heterogênea (uso de GPUs através de cuda).

A biblioteca proposta é desenvolvida de forma que todas as funções que requerem o cálculo efetivo (como soma ou multiplicação de matrizes) fiquem reservadas num determinado arquivo. Dessa forma, a implementação de um novo backend consiste em reimplementar cada uma das funções neste único arquivo.

Capítulo 4

Experimentos realizados

Este capítulo relata os experimentos realizados para atestar o funcionamento da biblioteca desenvolvida e os resultados obtidos. A biblioteca está disponível como software livre sob a licença MIT em https://github.com/luatil/NfNN.

4.1 Descrição do conjunto de dados

Para realizar a validação da biblioteca e teste dos algoritmos de otimização distribuída utilizamos o conjunto de dados do MNIST conforme exposto em LeCun *et al.*, 1998. Essa escolha foi realizada pela simplicidade do conjunto de dados, o que permitiu que resultados razoáveis fossem obtidos com um modelo simples.

O conjunto de dados de treinamento consiste de 60000 imagens em preto e branco (1 canal) com dimensões 28 de altura por 28 de largura (pixels). Ele apresenta imagens de dígitos do zero até o 9 escritos a mão, digitalizados. Apesar de ser considerado um conjunto simples, foi de extrema utilidade para os testes da biblioteca.

4.2 Descrição do modelo

O modelo utilizado foi uma rede neural com duas camadas lineares com 32 neurônios de largura na primeira camada e 10 na segunda. Ambas as camadas contém um viés. A função de ativação utilizada foi a ReLU *Rectified Linear Unit*. A última camada contém uma função *logsoftmax* que transforma os resultados das camadas lineares em log-probabilidades.

A criação do modelo é definida nos exemplos a partir das duas funções *CreateModel* e *Forward* apresentadas a seguir:

```
1
2   static model
3   CreateModel(nfnn_memory_arena *Mem, nfnn_random_state *Random)
4   {
5      model Result = {0};
6
7      Result.W1 = NfNN_Matrix(Mem, Random, 784, 32);
```

```
 8       Result.B1 = NfNN_Matrix(Mem, Random, 1, 32);
 9       Result.W2 = NfNN_Matrix(Mem, Random, 32, 10);
10       Result.B2 = NfNN_Matrix(Mem, Random, 1, 10);
11
12       return Result;
13    }
14
15    static nfnn_tensor *
16    Forward(nfnn_memory_arena *Mem, model Model, nfnn_tensor *Input)
17    {
18       nfnn_tensor *L1 = NfNN_MatMul(Mem, Input, Model.W1);
19       nfnn_tensor *L1b = NfNN_Add(Mem, L1, Model.B1);
20       nfnn_tensor *R1 = NfNN_ReLU(Mem, L1b);
21       nfnn_tensor *L2 = NfNN_MatMul(Mem, R1, Model.W2);
22       nfnn_tensor *L2b = NfNN_Add(Mem, L2, Model.B2);
23       return NfNN_LogSoftmax(Mem, L2b, 1);
24    }
```

A inicialização da função *NfNN_Matrix* mantém os valores da matriz entre -1 e $+1$.

4.3 Diferentes abordagens distribuídas

Para treinar este modelo foram utilizadas duas abordagens diferentes de otimização distribuída centralizada: síncrona e assíncrona. A abordagem síncrona consiste em criar um servidor de parâmetros e n trabalhadores. O servidor espera que todos os trabalhadores enviem os gradientes da iteração atual para otimizar o modelo atual e realizar a difusão dos novos pesos para os diferentes trabalhadores. Já a abordagem assíncrona recebe um gradiente, atualiza o modelo, e envia os novos gradientes diretamente para o trabalhador que enviou o gradiente.

Em ambos os casos a atualização dos pesos foi realizada com o otimizador SGD (Stochastic Gradient Descent) sem *momentum*. Outros otimizadores com melhor desempenho neste e em outros conjuntos de dados, como SGD com *Nesterov* ou Adam, também foram desenvolvidos. Porém, no caso do Adam, o desempenho é tão pronunciado que as eventuais diferenças de performance das abordagens distribuídas ficam mais difíceis de serem percebidas. Dessa forma, optamos por utilizar um otimizador mais simples nos testes das abordagens distribuídas.

A taxa de aprendizado utilizada foi de $0,01$. O tamanho de mini lote utilizado para cada trabalhador foi mantido constante em 32. O uso de otimização síncrona, contudo, efetivamente aumenta o tamanho de um mini lote para $32 * n$, onde n indica o número de trabalhadores.

Apesar dos resultados de McCandlish *et al.*, 2018 em que a taxa de aprendizado pode ser escalada linearmente com o tamanho de um mini lote, optamos por manter a taxa de aprendizado constante para o aumento do número de trabalhadores em otimização síncrona. Isso permite mais claramente verificar que o aumento efetivo de um mini lote através de otimização síncrona centralizada gera melhores estimadores do gradiente, o que implicaria num menor número necessário de iterações para a convergência de um

modelo.

4.4 Descrição dos ambientes de teste

Os experimentos das abordagens distribuídas foram realizados em dois ambientes diferentes. O primeiro deles foi um ambiente local, em que os processos se comunicavam numa mesma máquina. Esse primeiro ambiente é denominado por *Desktop*. A máquina utilizada foi um chip da AMD com 8 núcleos e 16 GB de memória RAM e 4,2 GHz de velocidade de processador.

Já para o teste em ambiente distribuído contratamos 3 servidores de entrada da DigitalOcean. Cada um deles tem uma configuração simples com uma CPU de 2,0 GHz e 1 GB de RAM. Essa configuração limitada dos servidores também guiou o desenvolvimento do modelo, sendo necessário limitar a quantidade de memória a ser utilizada.

Os testes foram realizados medindo o tempo de execução, precisão final e bytes recebidos e enviados para cada uma das diferentes configurações de ambiente, algoritmo e número de trabalhadores.

4.5 Resultados

Tabela 4.1: *Desempenho temporal de métodos síncronos e assíncronos em ambientes de desktop e datacenter. (IC) Intervalo de confiança de 95%*

Configuração	Método	# trab.	Tempo (s)	IC	Precisão final (validação)
Desktop	Sync	1	12	1	83,686157
Desktop	Sync	2	238	5	87,105171
Desktop	Sync	3	260	10	88,499664
Desktop	Async	1	30	5	83,686157
Desktop	Async	2	10	5	83,392136
Desktop	Async	3	9	2	83,274529
Datacenter	Sync	1	230	9	83,686157
Datacenter	Sync	2	235	10	87,105171
Datacenter	Sync	3	232	12	88,508064
Datacenter	Async	1	229	5	83,686157
Datacenter	Async	2	114	5	83,610550
Datacenter	Async	3	74	6	83,795364

Através da análise das tabelas 4.1 e 4.2 podemos avaliar as propriedades dos algoritmos conforme desenvolvidos na biblioteca. Primeiramente, podemos notar a variação da precisão final após 5000 iterações entre os diferentes métodos, tendo um mínimo de 83,27 para otimização assíncrona com 3 trabalhadores no ambiente de Desktop e um máximo de 88,50 para otimização síncrona com 3 trabalhadores no ambiente do Datacenter. Essa variação de cerca de 4 por cento corrobora o exposto em McCandlish *et al.*, 2018 e Langer *et al.*, 2020, no sentido de que um aumento no tamanho do mini lote melhora os estimadores em descida de gradiente estocástica.

Tabela 4.2: *Bytes enviados e recebidos em métodos síncronos e assíncronos em ambientes de desktop e datacenter*

Configuração	Método	# trab.	Bytes Enviados	Bytes Recebidos
Desktop	Sync	1	509.000.004,00	509.000.000,00
Desktop	Sync	2	1.018.000.008,00	1.018.000.000,00
Desktop	Sync	3	1.527.000.012,00	1.527.000.000,00
Desktop	Async	1	509.101.800,00	509.000.000,00
Desktop	Async	2	509.203.600,00	509.000.000,00
Desktop	Async	3	509.305.400,00	509.000.000,00
Datacenter	Sync	1	509.000.004,00	509.000.000,00
Datacenter	Sync	2	1.018.000.008,00	1.018.000.000,00
Datacenter	Sync	3	1.527.000.012,00	1.527.000.000,00
Datacenter	Async	1	509.101.800,00	509.000.000,00
Datacenter	Async	2	509.203.600,00	509.000.000,00
Datacenter	Async	3	509.305.400,00	509.000.000,00

Isso sugere que métodos de otimização síncrona podem ser válidos para diminuir o tempo de convergência de um modelo. Contudo, isso é condicionado ao ambiente testado, pois analisando o tempo em segundos entre os ambientes de Desktop e Datacenter, podemos notar que o tempo necessário para finalizar as 5000 iterações de otimização aumentou consideravelmente para o ambiente Desktop. A Tabela 4.2 indica que esse aumento no tempo de execução foi causado pelo aumento linear na quantidade de bytes recebidas e enviadas entre os trabalhadores. Nota-se que podemos atribuir esse aumento linear pela arquitetura síncrona, que requer que todos os n trabalhadores enviem seus gradientes para o servidor com o propósito de melhorar a qualidade do estimador do gradiente. Porém, ao analisarmos o ambiente do Datacenter percebemos que o tempo de execução teve pouca variação dado o número de trabalhadores, ao mesmo tempo melhorando a precisão final. Essa divergência de resultado entre ambientes traz a hipótese de que a diferente capacidade das máquinas pode viabilizar ou inviabilizar o uso de otimização síncrona.

Em outras palavras, se o tempo para sincronizar as máquinas é grande se comparado ao necessário para realizar uma iteração de retropropagação, como é o caso no ambiente do Desktop, mais iterações de descida de gradiente provavelmente serão mais proveitosas do que a utilização de uma arquitetura distribuída conforme desenvolvida na biblioteca. Porém, no caso em que uma iteração é lenta, proporcionalmente, ao custo de sincronização, como aparenta no caso do datacenter, a melhoria do estimador do gradiente, através do aumento efetivo do tamanho de um mini lote pode ser uma técnica para melhorar a convergência de um modelo.

Focando na questão de otimização assíncrona, primeiro destacamos que o tempo para 1 trabalhador finalizar 5000 iterações é substancialmente mais alto do que no caso síncrono para o ambiente de Desktop. Dada uma semelhante carga computacional nos modelos e troca de bytes, atribui-se essa diferença ao custo de monitorar os trabalhadores como desenvolvido na biblioteca. Esse monitoramento é atualmente realizado com o uso da chamada de *select* e sugere que a mudança para um mecanismo mais eficiente, como *IO Completion Ports* ou *epoll* poderia ser utilizado para melhorar a performance de otimização

assíncrona.

Além disso, percebemos que o aumento no número de trabalhadores em otimização assíncrona tem o efeito de diminuir o tempo de execução de 5000 iterações. Essa diminuição tem uma tendência linear, no sentido que o *throughput* de atualizações por segundo do modelo é proporcional ao número de trabalhadores no sistema. Esse efeito é presente nos dois ambientes testados, e é mais pronunciado no ambiente do Datacenter, no qual a capacidade computacional de cada máquina é mais limitada. Vale ressaltar que esse aumento de performance vem ao custo da precisão final, esse efeito é mais claro, embora ainda diminuto, no caso do Desktop, em que existe uma perda de cerca de 0,2 por cento na precisão final do modelo no conjunto de validação. Isso é devido a atualizações nos pesos do modelo através de gradientes que estão atrasados em termos do sistema, como é descrito em Langer *et al.*, 2020.

4.6 Discussão

Nesta seção, resumimos os principais resultados obtidos nos experimentos realizados e discutimos suas implicações práticas e teóricas.

Primeiramente, observamos que o uso de métodos de otimização distribuída, tanto síncronos quanto assíncronos, demonstrou impactos significativos no desempenho do treinamento dos modelos. Em ambientes de Datacenter, a otimização síncrona mostrou ser particularmente eficaz, melhorando a precisão dos modelos enquanto mantinha um tempo de execução constante. Isto sugere que, em cenários com recursos computacionais limitados, a otimização síncrona pode ser uma abordagem viável para acelerar o treinamento de modelos de aprendizado de máquina.

Por outro lado, em ambientes Desktop, notamos que o custo de sincronização pode superar os benefícios trazidos pelo aumento do tamanho efetivo do mini lote. Isso indica que a eficácia da otimização distribuída pode variar significativamente dependendo da infraestrutura disponível e do custo de comunicação entre os nós.

Também indicamos que a utilização de otimização assíncrona pode ser um importante mecanismo para diminuir o tempo de execução para um determinado número de iterações no sistema, conforme é indicado no caso de otimização assíncrona no Datacenter. Os resultados para o ambiente de Desktop também sugerem que aumentar o número de trabalhadores reduz o tempo para executar todas as iterações.

Além disso, os resultados apontam para a importância de balancear entre precisão e tempo de execução ao escolher uma estratégia de otimização distribuída. Enquanto a otimização síncrona pode levar a modelos mais precisos, ela também pode aumentar o tempo de treinamento, especialmente em ambientes com alta latência de rede.

Capítulo 5

Conclusão

Neste trabalho, desenvolvemos uma biblioteca de diferenciação automática com algumas capacidades de aprendizagem profunda. A biblioteca foi desenvolvida em C, tanto em termos de implementação quanto em termos de interface. O desenvolvimento da biblioteca mostrou-se valioso no quesito de compreensão de como as técnicas de diferenciação automática e aprendizagem profunda funcionam, assim de como elas são implementadas em bibliotecas de uso geral. A utilização de uma linguagem de baixo nível como C também foi importante pois permitiu que a maior parte dos elementos da bibliotecas fossem implementados, de multiplicação de matrizes até a implementação de otimizadores como Adam. Dessa forma, a biblioteca desenvolvida atingiu um de seus objetivos de ilustrar e consolidar o conhecimento adquirido sobre bibliotecas de diferenciação automática.

Além disso, também foram implementados algoritmos de otimização distribuída, que foram testados em ambientes pertinentes, e mostraram-se eficientes dentro da implementação da biblioteca. Como apresentado na seção de resultados, a implementação de otimização centralizada síncrona e assíncrona melhoraram a performance do modelo no ambiente de Datacenter.

Dado o escopo do trabalho, a biblioteca desenvolvida não possui capacidades de aprendizagem profunda tão avançadas quanto as bibliotecas de uso geral. Capacidade aqui se refere tanto a funcionalidades, diferentes tipos de precisão numérica por exemplo; quanto a desempenho (o kernel de multiplicação de matrizes enfatiza simplicidade e não desempenho). Porém, dada a organização da biblioteca, melhorias podem ser implementadas. Como todo o código está disponível como software livre no link https://github.com/luatil/NfNN, estas melhorias são encorajadas.

Futuras pesquisas podem explorar estratégias para reduzir o custo de sincronização em métodos de otimização distribuída, bem como investigar abordagens híbridas que combinem as vantagens dos métodos síncronos e assíncronos. Além disso, seria interessante avaliar a aplicabilidade dessas técnicas em conjuntos de dados maiores e mais complexos, bem como em diferentes arquiteturas de rede e de máquinas. Melhorias de performance nos kernels desenvolvidos, como o de multiplicação de matrizes, são de grande aplicabilidade. Em termos de otimização distribuída, algoritmos de particionamento de dados e aprendizagem federada são a principal área não explorada neste trabalho.

Referências

[Abadi *et al.* 2016] Martin Abadi *et al.* "{Tensorflow}: a system for {large-scale} machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283 (citado na pg. 7).

[Amini *et al.* 2019] Alexander Amini, Ava Soleimany, Sertac Karaman e Daniela Rus. *Spatial Uncertainty Sampling for End-to-End Control*. 2019. arXiv: 1805.04829 [cs.AI] (citado na pg. 5).

[Authors 2023] The JAX Authors. *GPU memory allocation*. en. 2023. URL: https://jax.readthedocs.io/en/latest/gpu_memory_allocation.html (citado nas pgs. 7, 8).

[Bradbury *et al.* 2018] James Bradbury *et al. JAX: composable transformations of Python+NumPy programs*. Versão 0.3.13. 2018. URL: http://github.com/google/jax (citado na pg. 8).

[Dean *et al.* 2012] Jeffrey Dean *et al.* "Large scale distributed deep networks". In: *NIPS*. 2012 (citado na pg. 7).

[Foundation 2023] The PyTorch Foundation. *PyTorch 2.0*. en. 2023. URL: https://pytorch.org/get-started/pytorch-2.0/ (citado na pg. 7).

[Gammo *et al.* s.d.] Louay Gammo, Tim Brecht, Amol Shukla e David Pariag. "Comparing and evaluating epoll, select, and poll event mechanisms". en () (citado na pg. 22).

[Gerganov 2023] Georgi Gerganov. *Tensor library for machine learning*. 2023. URL: http://github.com/ggerganov/ggml (citado na pg. 8).

[Hanson 1990] David R. Hanson. "Fast allocation and deallocation of memory based on object lifetimes". *Software: Practice and Experience* 20.1 (1990), pp. 5–12. DOI: https://doi.org/10.1002/spe.4380200104. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380200104. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380200104 (citado na pg. 18).

[Hotz 2023] George Hotz. *Tinygrad: for something between pytorch and karpathy/micrograd. maintained by tiny corp*. 2023. URL: http://github.com/tinygrad/tinygrad (citado na pg. 8).

[Krizhevsky *et al.* 2012] Alex Krizhevsky, Ilya Sutskever e Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems*. Ed. por F. Pereira, C. J. Burges, L. Bottou e K. Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. url: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (citado nas pgs. iii, v).

[Langer *et al.* 2020] Matthias Langer, Zhen He, Wenny Rahayu e Yanbo Xue. "Distributed training of deep learning models: a taxonomic perspective". en. *IEEE Transactions on Parallel and Distributed Systems* 31.12 (dez. de 2020), pp. 2802–2818. issn: 1045-9219, 1558-2183, 2161-9883. doi: 10.1109/TPDS.2020.3003307 (citado nas pgs. 25, 27).

[LeCun *et al.* 1998] Yann LeCun, Leon Bottou, Yoshua Bengio e Patrick Ha. "Gradient-based learning applied to document recognition". en (1998) (citado nas pgs. 1, 23).

[McCandlish *et al.* 2018] Sam McCandlish, Jared Kaplan, Dario Amodei e OpenAI Dota Team. "An empirical model of large-batch training". *arXiv preprint arXiv:1812.06162* (2018) (citado nas pgs. 14, 24, 25).

[Meta 2024] Meta. *Image Classification on ImageNet*. en. 2024. url: https://paperswithcode.com/sota/image-classification-on-imagenet (citado nas pgs. iii, v).

[Parr e Howard 2018] Terence Parr e Jeremy Howard. *The Matrix Calculus You Need For Deep Learning*. 2018. arXiv: 1802.01528 [cs.LG] (citado na pg. 19).

[Parvat *et al.* 2017] Aniruddha Parvat, Jai Chavan, Siddhesh Kadam, Souradeep Dev e Vidhi Pathak. "A survey of deep-learning frameworks". In: *2017 International Conference on Inventive Systems and Control (ICISC)*. 2017, pp. 1–7. doi: 10.1109/ICISC.2017.8068684 (citado na pg. 7).

[Paszke *et al.* 2019] Adam Paszke *et al.* "Pytorch: an imperative style, high-performance deep learning library". arXiv:1912.01703 (dez. de 2019). arXiv:1912.01703 [cs, stat]. url: http://arxiv.org/abs/1912.01703 (citado nas pgs. 6, 7).

[Sun *et al.* 2020] Shiliang Sun, Zehui Cao, Han Zhu e Jing Zhao. "A survey of optimization methods from a machine learning perspective". *IEEE Transactions on Cybernetics* 50.8 (2020), pp. 3668–3681. doi: 10.1109/TCYB.2019.2950779 (citado na pg. 5).

[Thakur 2022] Ayush Thakur. *How to prevent Tensorflow from fully allocating GPU memory*. en. 2022. url: https://wandb.ai/ayush-thakur/dl-question-bank/reports/How-to-Prevent-TensorFlow-From-Fully-Allocating-GPU-Memory--VmlldzoxOTk1ODE (citado na pg. 7).

[Wang *et al.* 2010] Guibin Wang, YiSong Lin e Wei Yi. "Kernel fusion: an effective method for better power efficiency on multithreaded gpu". In: *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing*. 2010, pp. 344–350. DOI: 10.1109/GreenCom-CPSCom.2010.102 (citado nas pgs. 6, 8).

[Xin *et al.* 2020] Ran Xin, Soummya Kar e Usman A. Khan. "Decentralized stochastic optimization and machine learning: a unified variance-reduction framework for robust performance and fast convergence". *IEEE Signal Processing Magazine* 37.3 (2020), pp. 102–113. DOI: 10.1109/MSP.2020.2974267 (citado na pg. 14).

[Zhang *et al.* 2021] Chen Zhang *et al.* "A survey on federated learning". *Knowledge-Based Systems* 216 (2021), p. 106775. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2021.106775 (citado na pg. 11).