UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Estudo Comparativo do Desempenho de Bibliotecas para Redes Neurais Convolucionais em Diferentes Microarquiteturas de GPU

Dissertação de Mestrado

Felipe de Almeida Florencio

São Cristóvão – Sergipe

2020

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Felipe de Almeida Florencio

Estudo Comparativo do Desempenho de Bibliotecas para Redes Neurais Convolucionais em Diferentes Microarquiteturas de GPU

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de mestre em Ciência da Computação.

Orientador(a): Edward David Moreno Ordonez

São Cristóvão – Sergipe

2020

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL
UNIVERSIDADE FEDERAL DE SERGIPE

Florencio, Felipe de Almeida
F632e Estudo comparativo do desempenho de bibliotecas para redes neurais convolucionais em diferentes microarquiteturas de GPU / Felipe de Almeida Florencio ; orientador Edward David Moreno Ordonez. - São Cristóvão, 2020.
146 f. : il.

Dissertação (mestrado em Ciência da Computação) – Universidade Federal de Sergipe, 2020.

1. Computação. 2. Inteligência artificial. 3. Redes neurais (computação). I. Ordonez, Edward David Moreno orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE SERGIPE
PRÓ-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA
COORDENAÇÃO DE PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Ata da Sessão Solene de Defesa da Dissertação do Curso de Mestrado em Ciência da Computação-UFS. Candidato: Felipe de Almeida Florencio

Em 27 dias do mês de julho do ano de dois mil e vinte, com início às 14h00min, realizou-se na Sala virtual https://meet.jit.si/defesaPROCC.florencio da Universidade Federal de Sergipe, na Cidade Universitária Prof. José Aloísio de Campos, a Sessão Pública de Defesa de Dissertação de Mestrado do candidato **Felipe de Almeida Florencio**, que desenvolveu o trabalho intitulado: **"** *Estudo Comparativo do Desempenho de Bibliotecas para Redes Neurais Convolucionais em Diferentes Microarquiteturas de GPU*", sob a orientação do Prof. Dr. **Edward David Moreno Odonez**. A Sessão foi presidida pelo Prof. Dr. **Edward David Moreno Odonez** (PROCC/UFS), que após a apresentação da dissertação passou a palavra aos outros membros da Banca Examinadora, Prof. Dr. **Ricardo José Paiva de Britto Salgueiro** (PROCC/UFS) e, em seguida, ao Prof. Dr. **Ricardo dos Santos Ferreira** (UFV - Universidade Federal de Viçosa). Após as discussões, a Banca Examinadora reuniu-se e considerou o mestrando (a) aprovado "*(aprovado/reprovado)*". Atendidas as exigências da Instrução Normativa 01/2017/PROCC, do Regimento Interno do PROCC (Resolução 67/2014/CONEPE), Resolução nº 25/2014/CONEPE e da Portaria nº 413 de 27 de maio de 2020 (Banca por videoconferência) que regulamentam a Apresentação e Defesa de Dissertação, e nada mais havendo a tratar, a Banca Examinadora elaborou esta Ata que será assinada pelos seus membros e pelo mestrando.

Cidade Universitária "*Prof. José Aloísio de Campos*", 27 de julho de 2020.
"participação à distância por videoconferência"

Prof. Dr. Edward David Moreno Ordonez
(PROCC/UFS)
Presidente

Prof. Dr. Ricardo José Paiva de Britto Salgueiro
(PROCC/UFS)
Examinador Interno

Prof. Dr. Ricardo dos Santos Ferreira
(UFV)
Examinador Externo

Felipe de Almeida Florencio
Candidato

Eu dedico esse trabalho a minha família, minha companheira, meus amigos, meus professores e a todos os cientistas e filósofos que contribuiram com a ciência da computação.

Agradecimentos

À minha família, por todo apoio e incentivo dado para que eu me dedicasse exclusivamente a vida acadêmica mesmo com todas as dificuldades financeiras.

À minha companheira, por estar ao meu lado tanto nos momentos bons quanto nos momentos difíceis durante a graduação. Foi uma pessoa essencialmente importante na minha linha de pesquisa por colaborar profundamente na minha formação quanto cientista com todo o seu acúmulo sobre biologia e principalmente sobre neurologia e darwinismo.

Às minhas amigas e aos meus amigos que me deram todo apoio e me fizeram enxergar o papel social que eu deveria exercer quanto um cientista da computação.

Aos meus colegas do PROCC por todo o suporte técnico, científico e emocional.

Ao meu orientador Edward, por toda a orientação acadêmica e de vida que me fizeram persistir no sonho de seguir a carreira acadêmica.

A todas e todos aqueles que lutaram e lutam por uma educação pública, gratuita e de qualidade. Sem essas pessoas minha formação quanto cientista não seria possível.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - pelo suporte financeiro. Código de Financiamento 001.

Nós só podemos ver um pouco do futuro, mas o suficiente para perceber que há muito a fazer
(Alan Turing)

Resumo

Contexto: A popularização da *Deep Learning* e da *Deep Inference* impulsionou o desenvolvimento de ferramentas para implementação de Redes Neurais Convolucionais (*CNNs*, do inglês *Convolutional Neural Networks*) como bibliotecas específicas para o desenvolvimento de *CNNs*, a popularização também estimulou o desenvolvimento de *GPUs* com recursos de aceleração de *CNNs*. Os desenvolvedores e cientistas que trabalham com *CNNs* precisam de estudos científicos experimentais que apontem qual a biblioteca mais adequada para determinada microarquitetura de *GPU*. **Objetivos:** Comparar o desempenho das bibliotecas *CNTK*, *PyTorch*, *TensorFlow 1.15* e *TensorFlow 2.2* em diferentes microarquiteturas de *GPU* (*Kepler*, *Maxwell*, *Pascal* e *Turing*) utilizando a *CNN LeNet-5* e o *dataset MNIST*. **Metodologia:** Inicialmente, foi realizado um mapeamento sistemático como forma de identificar e sistematizar os principais *benchmarks*, em seguida foi realizado um experimento avaliando o impacto da *API Keras* no desempenho das bibliotecas utilizadas no estudo comparativo e por último foram realizados quatro experimentos comparando as bibliotecas, cada experimento foi realizado em um ambiente acelerado por uma *GPU* com uma microarquitetura diferente. **Resultados:** Para o mapeamento sistemático, foi identificado que as arquiteturas clássicas de *CNN* como *LeNet-5* e *AlexNet* são as mais utilizadas como *benchmarks*, também mostrou que os *datasets* mais utilizados em *benchmarking* são o *ImageNet*, *MNIST* e *CIFAR-10*. O experimento sobre o impacto da *API Keras* mostrou que a *API* impacta negativamente no desempenho de todas as bibliotecas testadas. No estudo comparativo de desempenho das bibliotecas, a biblioteca *PyTorch* apresentou o pior desempenho e as bibliotecas *CNTK*, *TensorFlow 1.15* e *TensorFlow 2.2* alternaram entre os três menores tempos de execução. **Conclusão:** Os resultados evidenciam que a biblioteca *PyTorch* apresenta um baixo nível de utilização da *GPU* e utiliza uma grande quantidade de memória se comparada as outras bibliotecas que apresentaram um desempenho superior.

Palavras-chave: Biblioteca de Aprendizagem Profunda, Rede Neural Convolucional, Avaliação de Desempenho, GPU, Graphic Processing Unit, Benchmarking.

Abstract

Context: The popularization of Deep Learning and the Deep Inference spurred the development of tools for the implementation of Convolutional Neural Networks (CNNs) as specific libraries for the development of CNNs, the popularization also stimulated the development of GPUs with CNN acceleration capabilities. Developers and scientists working with CNNs need experimental scientific studies that point out which library is most suitable for a particular GPU microarchitecture. **Objective:** Compare the performance of the CNTK, PyTorch, TensorFlow1.15 and TensorFlow 2.2 libraries in different GPU microarchitectures (Kepler, Maxwell, Pascal and Turing) using LeNet-5 CNN and the MNIST dataset. **Method:** Initially, a systematic mapping was carried out as a way to identify and systematize the main benchmarks, then an experiment was carried out evaluating the impact of the Keras API on the performance of the libraries used in the comparative study and lastly, four experiments were carried out comparing the libraries, each experiment was performed in a computational environment accelerated by a GPU with a different microarchitecture. **Results:** For the systematic mapping, it was identified that the classic CNN architectures such as LeNet-5 and AlexNet are the most used as benchmarks, also showed that the most used data benchmarking sets are ImageNet, MNIST and CIFAR-10. The experiment on the impact of the Keras API showed that the API negatively impacts the performance of all the tested libraries. In the comparative study of the libraries' performance, the PyTorch library presented the performance performance and the CNTK, TensorFlow 1.15 and TensorFlow 2.2 libraries alternated between the three shorter execution times. **Conclusion:** The results show that the PyTorch library presents a low level of use of GPU and uses a large amount of sec memory compared to other libraries that showed a higher performance.

Keywords: Deep Learning Library, Convolutional Neural Network, Performance Evaluation, GPU, Graphic Processing Unit, Benchmarking.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

API	Application Programming Interface - Interface de Programação de Aplicativos
AWS	Amazon Web Service
CIFAR-10	Canadian Institute For Advanced Research Dataset
CNN	Convolutional Neural Network - Rede Neural Convolucional
CNTK	Microsoft Cognitive Toolkit
CPU	Central Processing Unit - Unidade Central de Processamento
CUDA	Compute Unified Device Architecture
cuDNN	CUDA Deep Neural Network library
DDR	Double Data Rate SDRAM
DGA	Directed Acyclic Graph - Grafo Acíclico Direcionado
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit - Unidade de Processamento Gráfico
GQM	Goal Question Metric
HPC	High-Performance Computing - Computação de Alta Performance
MLP	MultiLayer Perceptron - Perceptron Multicamadas
MNIST	Modified National Institute of Standards and Technology database
NCS	Neural Computer Stick
PC	Personal Computer - Computador Pessoal
RAM	Random Acess Memory - Memória de Acesso Aleatório
RNA	Rede Neural Artificial
SLM	Systematic Literature Mapping
TSMC	Taiwan Semiconductor Manufacturing Company, Limited

Sumário

1 Introdução

Nesse capítulo há uma apresentação geral do trabalho, por isso são tratados: a motivação, a problemática e hipótese, os objetivos, a metodologia e a estrutura do documento.

1.1 Motivação

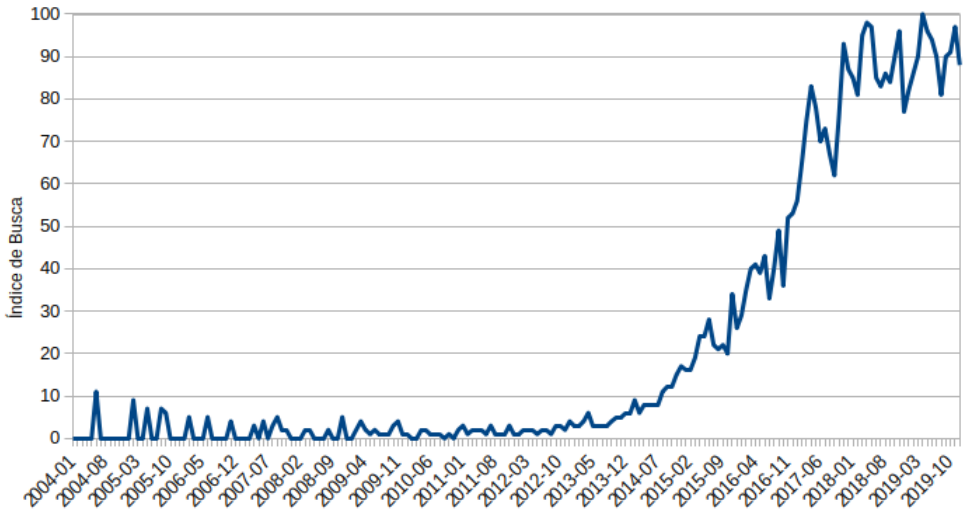
A Aprendizagem Profunda, também conhecida pela expressão em inglês *Deep Learning*, é uma área da inteligência artificial que permite que os computadores aprendam com a experiência e entendam o mundo em termos de uma hierarquia de conceitos, com cada conceito definido por meio de sua relação com conceitos mais simples. Se representarmos essa hierarquia de conceitos em um grafo, iríamos precisar de diversas camadas formando um grafo profundo, por isso o nome de *Deep Learning* (GOODFELLOW; BENGIO; COURVILLE, 2016).

A ideia da aprendizagem profunda é treinar uma Rede Neural Artificial (*RNA*) de múltiplas camadas em um conjunto de dados para permitir que ele lide com tarefas do mundo real. Embora os conceitos teóricos por trás não sejam novos, a aprendizagem profunda teve um grande interesse na última década devido a vários fatores, incluindo a sua aplicação bem-sucedida na resolução de vários problemas (muitos com potenciais comerciais), o desenvolvimento de novas arquitetura de computadores com maior nível de paralelismo, o nascimento das Redes Neurais Convolucionais (*CNN*) e a maior acessibilidade a computadores de alta performance (SHATNAWI et al., 2018) (VERHELST; MOONS, 2017) (RASCHKA, 2015).

As Redes Neurais Convolucionais desempenham um papel importante na história do *Deep Learning*. As *CNNs* são um exemplo-chave de uma aplicação bem sucedida de estudos do cérebro para aplicações de aprendizado de máquina. Elas também foram alguns dos primeiros modelos profundos a ter um bom desempenho, muito antes de modelos profundos arbitrários serem considerados viáveis. As redes convolucionais também foram uma das primeiras redes neurais a resolver aplicações comerciais e permanecem na vanguarda das aplicações comerciais

(GOODFELLOW; BENGIO; COURVILLE, 2016). Por exemplo, na década de 1990, o grupo de pesquisa em rede neural da *AT & T* desenvolveu uma rede convolucional para leitura de cheques (LECUN et al., 1998). Posteriormente, vários sistemas de reconhecimento de escrita e escrita baseados em redes convolucionais foram implementados pela *Microsoft* (SIMARD; STEINKRAUS; PLATT, 2003).

Figura 1 – Busca por termo no *Google Trends*



Fonte: Autoria Própria

O crescimento do interesse por redes neurais convolucionais pode ser observado analisando os resultados da pesquisa pelo termo "*Convolutional Neural Network*"no *Google Trends* (GOOGLE LLC, 2018)[1]. A Figura 1 ilustra o gráfico do índice de buscas do termo no *Google* de Janeiro de 2004 até Dezembro de 2019. O índice é normalizado dentro de uma escala de 0 a 100, o que mostra que durante a pesquisa realizada em Dezembro de 2018 o termo estava em seu ponto mais alto na busca.

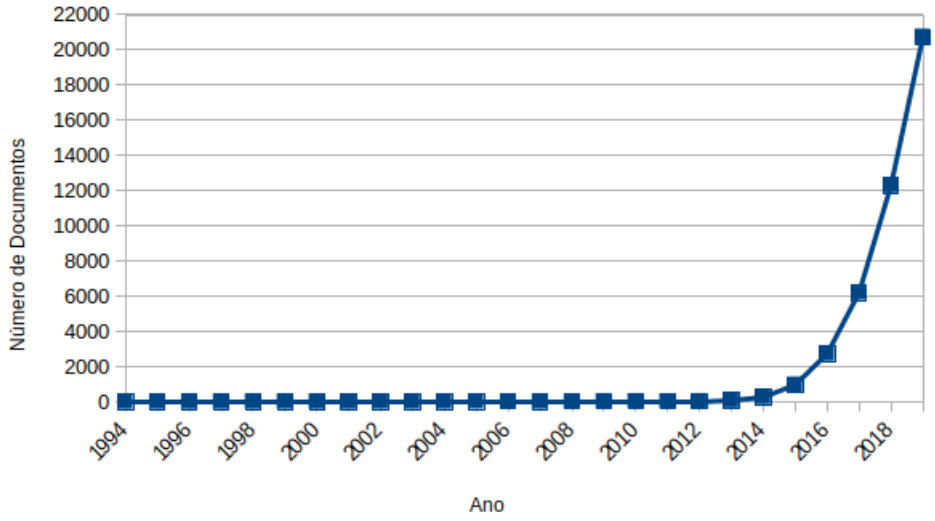
No campo científico o crescimento das pesquisas na área pode ser analisado pelo número de documentos na base científica *Scopus* (ELSEVIER B.V., 2018), base que reúne documentos de várias conferências e periódicos. A Figura 2 mostra o número de documentos publicados por ano desde 1994, o ponto máximo em 2019.

No campo industrial, a base do WIPO (2018)[2] mostra o número de depósito de patentes ao longo dos anos para a expressão *Convolutional Neural Network*. Na Figura 3 mostra o número de patentes por ano, podemos observar que o pico de depósito de patentes foi em 2019. Não

[1] *Google Trends* é um *website* do *Google* que analisa a popularidade das principais consultas de pesquisa na Pesquisa *Google* em várias regiões e idiomas. O *website* usa gráficos para comparar o volume de pesquisa de diferentes consultas ao longo do tempo

[2] *World Intellectual Property Organization (WIPO)* é o fórum global para serviços de propriedade intelectual, política, informação e cooperação. A sua é liderar o desenvolvimento de um sistema internacional de propriedade intelectual (IP) equilibrado e eficaz que permita a inovação e a criatividade para o benefício de todos (WIPO, 2019).

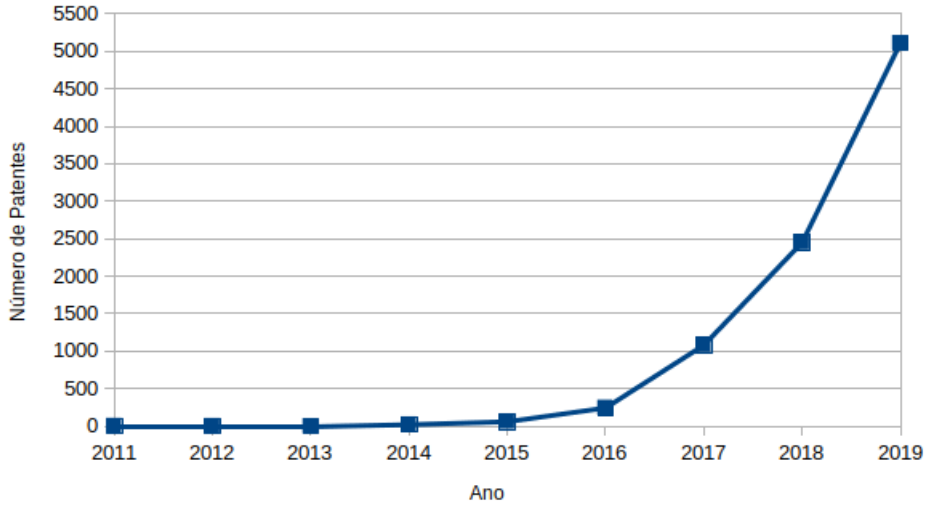
Figura 2 – Número de documentos no *Scopus*



Fonte: Autoria Própria

foram encontrados registros de patentes anteriores ao ano de 2011, esse gráfico também não ilustra o número de patentes no ano de 2020.

Figura 3 – Número de patentes no *Wipo*



Fonte: Autoria Própria

Em consequência do crescimento do estudo e uso das *CNNs*, houve o desenvolvimento de arquiteturas de *hardware* paralelo específicos para Redes Neurais Convolucionais e o aprimoramento de arquiteturas computacionais já consolidadas como *Graphics Processing Units* (*GPU*), largamente utilizado em serviços de nuvem como *Microsoft Azure*, *Google Cloud* e *Amazon Web Services* (*AWS*) (WALTER et al., 2020) (AMAZON WEB SERVICE, 2020) (NVIDIA CORPORATION, 2019a) . Houve também a chegada de várias bibliotecas *opensource* no

mercado com foco em *Deep Learning* e *Deep Inference*.

1.2 Problemática e Hipótese

Existem várias *bibliotecas* de *Deep Learning*, como o *TensorFlow*, *Theano*, *CNTK*, *Caffe*, *Torch*, *Neon* e *PyTorch*. Cada um dessas bibliotecas possui diferentes características de desempenho e aplicam diferentes técnicas para otimizar sua implementação de algoritmos. Portanto, embora o mesmo algoritmo seja implementado em diferentes bibliotecas, o desempenho de cada implementação pode variar muito (BAHRAMPOUR et al., 2015) (SHATNAWI et al., 2018).

Por conta da variedade de bibliotecas *open source* disponíveis no mercado e do largo uso de *GPUs* para treinamento e inferência de modelos *CNN*, os desenvolvedores e cientistas que trabalham com Redes Neurais Convolucionais precisam de estudos científicos experimentais que apontem qual a biblioteca mais adequada para determinada microarquitetura de *GPU*. Diante disso, esse trabalho visa investigar o desempenho e as limitações de diferentes bibliotecas em diferentes microarquiteturas de *GPU*.

Algumas perguntas de pesquisa podem ser feitas:

Q1 - Há diferença de desempenho entre as bibliotecas?

Q2 - Quais são os *benchmarks* disponíveis para a avaliação de desempenho das bibliotecas?

Q3 - Quais são as causas de uma possível diferença de desempenho entre as bibliotecas?

Não foram levantadas hipóteses para a segunda e a terceira questão de pesquisa. Para a primeira questão de pesquisa, foram levantadas as seguintes hipóteses:

$H0^{Q1}$ - Não há diferenças estatísticas entre o desempenho das bibliotecas.

$H1^{Q1}$ - Há Diferenças estatísticas entre o desempenho das bibliotecas.

O presente estudo é focado em sistemas computacionais acelerados por *GPUs*. Para responder as questões, códigos que utilizam bibliotecas para *CNNs* foram executados em diferentes sistemas computacionais com diferentes microarquiteturas de *GPU*, afim de identificar deficiências e proficiências das bibliotecas.

1.3 Objetivos

O principal objetivo deste trabalho é avaliar (segundo tempo de execução, temperatura do *hardware* e utilização do *hardware*) diferentes bibliotecas de aprendizagem profunda, verificando

o comportamento durante a execução das fases de treinamento e de inferência de redes neurais convolucionais.

Seguindo a formalização de definição de objetivo do modelo GQM (*Goal Question Metric*), proposto por Basili, Caldiera e Rombach (1994), o **objetivo principal** pode ser reescrito como: **Analisar** diferentes bibliotecas de aprendizado profundo, **com a finalidade de** compará-los **com relação a** tempo de execução, utilização do *hardware* e temperatura do *hardware* **do ponto de vista** de pesquisadores, cientistas e desenvolvedores que trabalham com *deep learning* **no contexto de** redes neurais convolucionais executando em sistemas computacionais com diferentes microarquiteturas de *GPU*.

São **objetivos específicos**:

- Identificar os principais *benchmarks*;
- Avaliar o impacto do uso da *API Keras* no desempenho da biblioteca *TensorFlow*;
- Identificar qual biblioteca possui melhor desempenho em ambientes computacionais com GPUs com microarquitetura *Kepler*;
- Identificar qual biblioteca possui melhor desempenho em ambientes computacionais com GPUs com microarquitetura *Maxwell*;
- Identificar qual biblioteca possui melhor desempenho em ambientes computacionais com GPUs com microarquitetura *Pascal*;
- Identificar qual biblioteca possui melhor desempenho em ambientes computacionais com GPUs com microarquitetura *Turing*;
- Identificar as possíveis causas para os índices de desempenho de cada biblioteca;

1.4 Metodologia Geral

A primeira parte da dissertação consiste na realização de um mapeamento sistemático sobre *benchmarking* de bibliotecas para Redes Neurais Convolucionais. Desta forma, ter-se-á uma compreensão sobre como está o estado da arte (produção acadêmica relacionados ao tema) nesta área de pesquisa. Com isto, pretende-se ter um embasamento teórico suficiente para o desenvolvimento do trabalho, e identificar os principais *benchmarks*.

Foi realizado um estudo preliminar (FLORENCIO et al., 2019) comparando o desempenho das bibliotecas *TensorFlow* e *PyTorch* em um *PC* que utilizava uma *GPU* com arquitetura *Pascal*. Esse estudo preliminar teve como resultado o melhor desempenho da biblioteca *PyTorch*, o que estimulou a investigação dessa biblioteca nos experimentos ao longo da dissertação.

A partir dos *benchmarks* identificados no estudo bibliográfico e do resultado do estudo preliminar foram realizados estudos experimentais *in vitro* - realizado em ambiente fechado com condições controladas. Os experimentos foram realizados seguindo o método *GQM* de Basili, Caldiera e Rombach (1994).

Antes de realizar o estudo comparativo entre as bibliotecas, foi necessário realizar um estudo experimental sobre o impacto do uso da *API* de alto nível *Keras* no desempenho das bibliotecas. Algumas bibliotecas suportam e até estimulam (no caso do *TensorFlow*) a utilização da *API Keras* para facilitar a implementação de *CNNs*, pois a *Keras* proporciona uma implementação de alto nível de *CNNs* enquanto executam as bibliotecas como *back-end*. Esse estudo foi realizado para auxiliar a definir se os códigos seriam implementados com a *API Keras*[3].

Em seguida, foram realizados quatro experimentos que consistem na execução do modelo *LeNet-5* (com o processo de aprendizado e teste) implementado com cada uma das bibliotecas avaliadas (*CNTK*, *PyTorch*, *TensorFlow 1* e *TensorFlow2*). O primeiro Experimento 1 foi realizado em um *PC* acelerado por uma *GPU* com microarquitetura *Maxwell*, os Experimentos 2, 3 e 4 foram realizados utilizando o sistema *Google Colab* (GOOGLE LLC, 2020b). O Experimento 2 foi realizado com a aceleração de uma *GPU* com microarquitetura *Kepler*, o Experimento 3 foi realizado com a aceleração de uma *GPU* com microarquitetura *Pascal* e o Experimento 4 foi realizado com a aceleração de uma *GPU* com microarquitetura *Turing*.

Ao final de cada experimento, os dados coletados foram analisados usando-se da aplicação de testes estatísticos. Essa análise viabilizou a identificação da biblioteca que apresentou melhor desempenho em cada sistema e a identificação das deficiências e proficiências de cada biblioteca.

1.5 Estrutura do Documento

Para facilitar a navegação e melhor entendimento, este documento está estruturado em capítulos e seções, que são:

- Capítulo 1 - Introdução: apresenta a motivação, a problemática e hipótese, os objetivos e a metodologia do trabalho;
- Capítulo 2 - Fundamentação Teórica: apresenta a fundamentação teórica sobre Redes Neurais Artificiais, Redes Neurais Convolucionais, *GPU* e das quatro bibliotecas utilizadas;
- Capítulo 3 - Estado da Arte: apresenta um mapeamento sistemático de trabalhos científicos sobre *Benchmarking* de bibliotecas para Redes Neurais Convolucionais situando o estado da arte;

[3] Apenas as bibliotecas, *CNTK* e *TensorFlow* possuem suporte para o *Keras*

- Capítulo 4 - Metodologia Experimental: apresenta o estudo experimental sobre o impacto do uso da ferramenta *Keras* no desempenho das bibliotecas *TensorFlow 1*, *TensorFlow 2* e *CNTK*;
- Capítulo 5 - Metodologia Experimental: apresenta a definição, o planejamento e a execução dos quatro experimentos com base na metodologia experimental utilizada;
- Capítulo 6 - Resultados do Experimento: Apresentam os resultados e discussões dos experimentos com base nos dados coletados e nos testes estatísticos utilizados;
- Capítulo 7 - Conclusão: apresenta as conclusões e trabalhos futuros dos experimentos desenvolvidos ao longo da dissertação;

2

Fundamentação Teórica

Neste capítulo são tratados conceitos envolvidos na temática de Redes Neurais, *GPUs* e das bibliotecas utilizadas na execução dos experimentos. Deve-se ressaltar que não pretende-se apresentar um estudo extensivo sobre todos estes temas, são apresentados apenas os pontos fundamentais para o entendimento do conteúdo a ser abordado na dissertação.

2.1 Redes Neurais Convolucionais

As Redes Neurais Convolucionais (*CNNs*) são um tipo de rede neural *feed-forward* biologicamente inspirada, é projetada para imitar o comportamento de um córtex visual animal (FRANCO, 2016). O conceito básico das *CNNs* remonta a 1979 quando Fukushima (1979) propôs uma rede neural artificial incluindo células simples e complexas que eram muito semelhantes às camadas de convolução e agrupamento das *CNN* modernas.

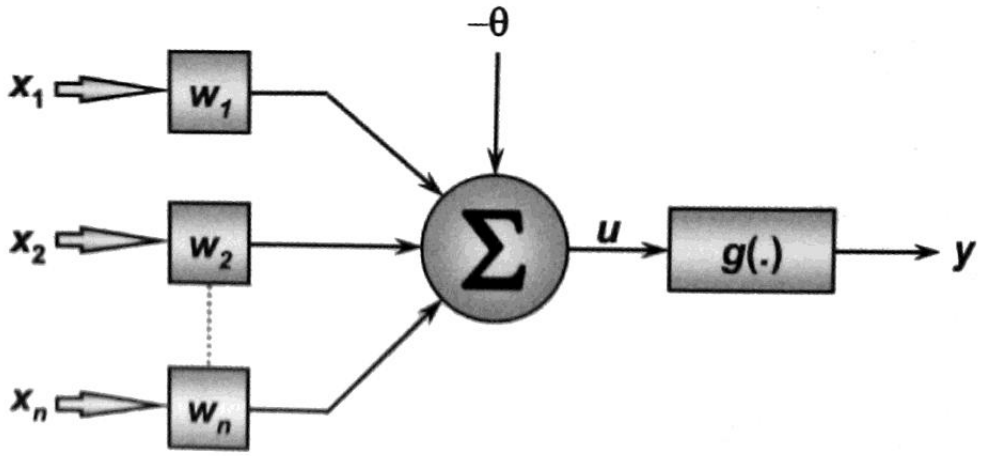
As *CNNs* são compostas por várias camadas de processamento não linear dos dados, onde a saída de cada camada inferior alimenta a entrada da sua camada imediatamente superior (DENG, 2014). Elas utilizam a convolução no lugar da multiplicação de matriz geral em pelo menos uma de suas camadas (GOODFELLOW; BENGIO; COURVILLE, 2016). As camadas das *CNNs* podem ser de três tipos: Camada Convolucional, Camada Convolucional e Camada Densa.

2.1.1 Neurônio Artificial - O Elemento Básico

O primeiro trabalho sobre neurônios artificiais foi elaborado em 1943 por Mcculloch e Pitts (1943). O neurônio proposto pela dupla era bastante simples, era um dispositivo binário: a sua saída poderia ser pulso ou não pulso, as suas várias entradas possuíam ganhos arbitrários podendo ser inibitórias ou excitatórias. A saída do neurônio era determinada a partir do cálculo da soma ponderada das entradas com os respectivos pesos como fatores de ponderação (positivos

nos casos excitatórios e negativo nos casos inibitórios). Se este resultado fosse maior ou igual a um certo limiar, então a saída do neurônio era pulso, caso contrário, era não pulso (FAUSETT, 1994).

Figura 4 – Modelo Matemático de um Neurônio



Fonte:Silva, Spatti e Flauzino (2010), página 34

Na representação de Mcculloch e Pitts (1943) cada neurônio da rede pode ser implementado conforme mostrado na Figura 4. Em Silva, Spatti e Flauzino (2010) é descrito o funcionamento do modelo MCP:

- As entradas $\{X_1, X_2, ..., X_n\}$ representam os sinais recebidos pelo meio externo ou recebidos a partir de outros neurônios
- Os pesos $\{W_1, W_2, ..., W_n\}$ servem para ponderar cada uma das variáveis de entrada do neurônio.
- A soma do produto das entradas pelos seus respectivos pesos (combinação linear), efetuada pelo bloco somador $\sum$, agrega todos os sinais de entrada que foram ponderados pelos respectivos pesos sinápticos a fim de produzir um valor de potencial de ativação.
- O parâmetro θ representa o limiar de disparo do neurônio, é uma variável que especifica qual será o patamar apropriado para que o resultado produzido pelo combinador linear possa gerar um valor de disparo em direção à saída do neurônio.
- A saída **u** representa o pulso de saída do neurônio. Se tal valor é positivo então o neurônio produz um potencial excitatório, caso contrário, o potencial é inibitório.
- A função **g** limita a saída do neurônio dentro de um intervalo de valores razoáveis.
- A saída **Y** representa o valor final produzido pelo neurônio com relação a um conjunto de entradas.

As duas expressões abaixo sintetizam o resultado obtido pelo neurônio artificial do modelo de Mcculloch e Pitts (1943):

$$u = \sum_{i=1}^{n} w_i.x_i - \theta$$

$$y = g(u)$$

O modelo *MCP* é simplificado comparado ao neurônio biológico. O neurônio artificial considera que todos os neurônios de uma mesma camada são disparados sincronicamente, ou seja, todos os neurônios são avaliados ao mesmo tempo. Nos neurônios biológicos não há um mecanismo para sincronizar as ações dos neurônios de acordo com camadas e nem são disparados com tempos discretos. Alguns modelos de neurônios artificiais incorporam mais detalhes do neurônio biológico, mas o modelo de Mcculloch e Pitts (1943)s ainda é largamente usado na computação, pois é capaz de resolver problemas complexos se utilizado em uma RNA (BRAGA; CARVALHO; LUDERMIR, 2012).

A função de ativação (g) é responsável por gerar a saída y do neurônio a partir dos pesos $\{W_1, W_2, ..., W_n\}$ e das entradas $\{X_1, X_2, ..., X_n\}$ (BRAGA; CARVALHO; LUDERMIR, 2012). Algumas das principais funções utilizadas como função de ativação são: função degrau, função sigmoide, linear, gaussiana, logística e tangente hiperbólica (SILVA; SPATTI; FLAUZINO, 2010).

2.1.2 A Camada Convolucional

A camada convolucional consiste em um conjunto de mapas de recursos, que são gerados a partir de uma operação convolucional sobre os dados de entrada ou outro mapa de recursos. Cada camada convolucional define uma representação de entrada de dados em um determinado nível de abstração (FRANCO, 2016).

Em sua forma mais geral, convolução é um operador linear, que a partir de duas funções dadas, resulta numa terceira que mede a soma do produto dessas funções ao longo da região subentendida pela superposição delas em função do deslocamento existente entre elas (GOODFELLOW; BENGIO; COURVILLE, 2016). Ela possui a seguinte forma:

$$\int x(a)w(t-a)da \tag{2.1}$$

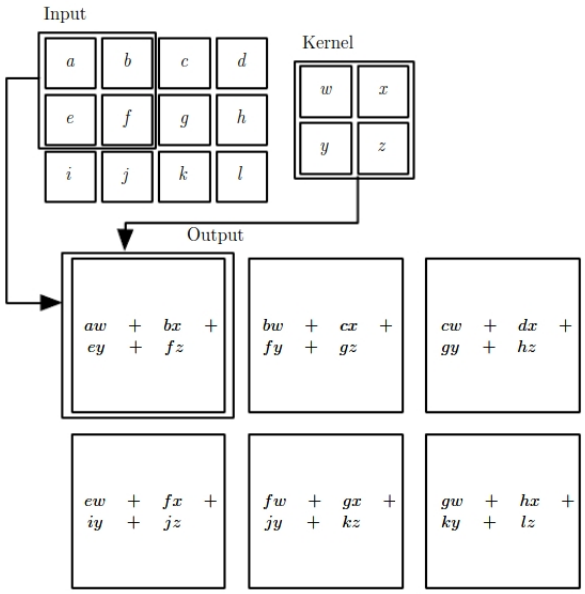
Em que x(t) e w(a) são as duas funções dadas e s(t) a função resultante. A operação de convolução é tipicamente denotada por um asterisco:

$$s(t) = (x * w)(t) \tag{2.2}$$

No caso das redes convolucionais, o primeiro argumento (neste exemplo, a função x) para a convolução é muitas vezes chamado de *input*, e o segundo argumento (neste exemplo,

a função w) chamado de *kernel*. A saída é comumente chamada de mapa de características (GOODFELLOW; BENGIO; COURVILLE, 2016).

Figura 5 – Um Exemplo de Convolução 2-D



Fonte: Goodfellow, Bengio e Courville (2016), página 330

Na Figura 17 há um exemplo de convolução 2-D sem fluxo de *kernel*. A saída está restrita à apenas a posição onde o *kernel* está inteiramente dentro da imagem chamada de "convolução válida". Na figura há setas indicando como o elemento superior esquerdo do da saída é formado aplicando o *kernel* à região superior esquerda correspondente a entrada.

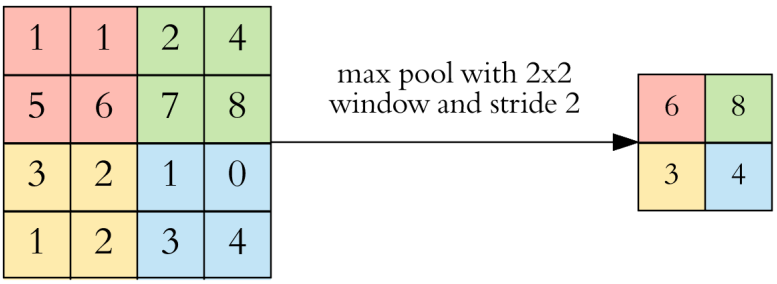
2.1.3 Camada Pooling

A *Pooling Layer* serve para reduzir progressivamente o tamanho espacial da representação, reduzir o número de parâmetros e a quantidade de computação na rede e, portanto, também controlar o *overfitting* (GOOGLE BRAIN TEAM, 2018a).

Um algoritmo de *pooling* comumente usado é o *Max-pooling*, que extrai sub-regiões do mapa de recursos (por exemplo, blocos de 2x2 *pixels*), mantém seu valor máximo e descarta todos os outros valores. (FRANCO, 2016) (GOOGLE BRAIN TEAM, 2018a). Essa extração é mostrada na Figura 6 na qual o *Max-pooling layer* é obtido de uma entrada de mapa de características.

Há também o algoritmo *Average Pooling*, utilizado principalmente nas arquiteturas de *CNN* mais antigas. Esse algoritmo utiliza a média dos valores ao invés de manter o valor máximo.

Figura 6 – Exemplo de um *Max-pooling layer* obtido de uma entrada de mapa de características

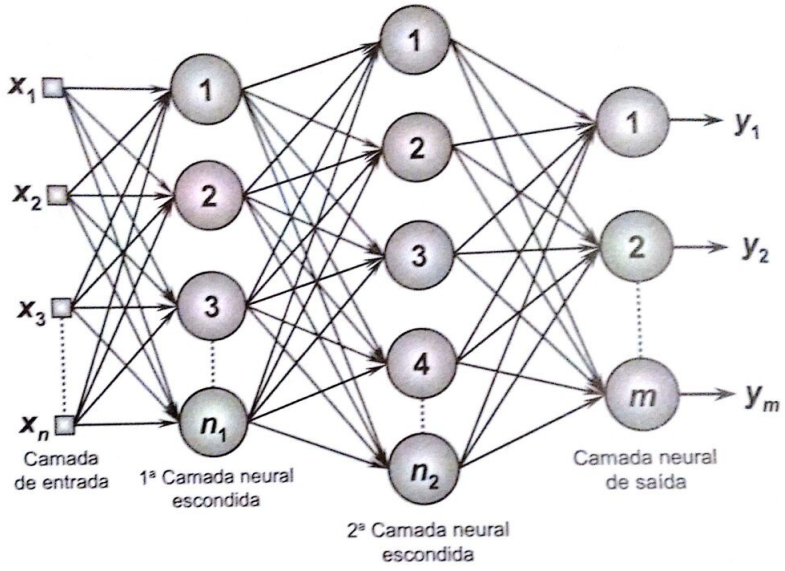


Fonte: Dertat (2017)

2.1.4 Camada Densa

São camadas totalmente conectadas, semelhantes as camadas das redes *Perceptron* Multicamadas ilustrada na Figura 7, que realizam a classificação das características extraídas pelas camadas convolucionais e diminuídas pelas *Pooling Layers*. Em uma camada densa, todos os nós da camada são conectados a todos os nós da camada anterior (RASCHKA, 2015) (FRANCO, 2016) (GOOGLE BRAIN TEAM, 2018a).

Figura 7 – Rede *Perceptron* de múltiplas camadas



Fonte: Silva, Spatti e Flauzino (2010), página 48

2.1.5 A Fase de Aprendizagem

Uma RNA possui duas fases, ou dois processos. A primeira é a fase de aprendizagem, também chamado de fase de treinamento que consiste em três etapas: a RNA é estimulada pelo ambiente, a RNA sofre modificações dos seus parâmetros, a RNA responde de uma maneira diferente ao meio ambiente devido as modificações sofridas. Os parâmetros atualizados são

parâmetros livres das camadas densas. A outra fase é a fase de inferência que é a fase de aplicação do modelo, é a função para a qual a RNA é destinada como classificação e reconhecimento de padrões.

Uma das mais antigas leis para o treinamento de neurônios artificiais é a *Teoria Hebbiana* descrita em Hebb (1949). No processo, quando a saída produzida pela rede está coincidente com a saída desejada, os pesos sinápticos e limiares são incrementados (ajuste excitatório) proporcionalmente aos valores de seus sinais de entrada; quando a saída é diferente do valor desejado, ocorre o ajuste inibitório decrementando os pesos sinápticos e o limiar (SILVA; SPATTI; FLAUZINO, 2010). As regras de aprendizagem estão descritas nas expressões abaixo:

$$w_i^{atual} = w_i^{anterior} + \eta.(t-y).x_i$$

Nas expressões o η representa a taxa de aprendizagem, ou seja, a sensibilidade com a qual os pesos variam em cada atualização. A variável t representa os *targets*, o alvo das amostras para o aprendizado da rede, em outras palavras, representa as saídas esperadas. O w_i são os pesos, a variável y é a saída de cada neurônio e a variável x_i representam os dados de entrada da rede.

A *Teoria de Hebb* foi desenvolvida para o treinamento de um neurônio simples. O processo de cálculo do erro de uma RNA se torna mais complexo com a adição de neurônios e de camadas extras, pois é mais difícil determinar quais pesos causaram erros. Para contornar esse problema é utilizado o método *back-propagation of error* no qual os erros são enviados para trás através da rede. O cálculo do erro é feito pela função de erro de soma de quadrados garantindo que os pesos sejam atualizados sempre que necessário. A função soma dos quadrados dos erros é:

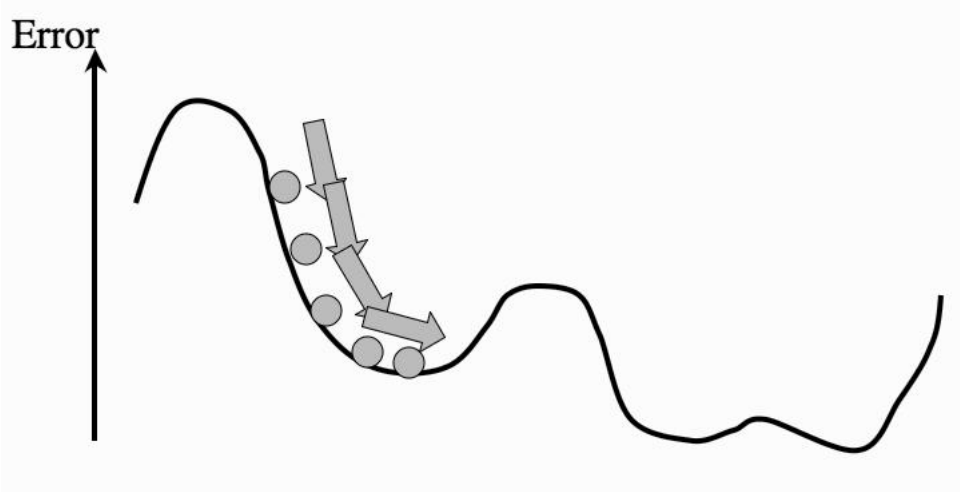
$$E(t,y) = \frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2$$

Na expressão, o N é o número de nós existente na camada de saída, o y_k são as saídas de cada nó e o t_k são os *targets*. A constante $\frac{1}{2}$ poderia ter qualquer outro valor, mas Marsland (2014) recomenda esse valor a fim de simplificar o resultado e compreensão da teoria.

O que está por trás da regra de aprendizagem é a minimização do erro de rede por descida do gradiente (usando a derivada da função de erro para reduzir o erro). É um processo de otimização: adaptação dos valores dos pesos para minimizar a função de erro. Na Figura 8 é mostrado o comportamento do erro tendendo ao mínimo local, no melhor caso o erro deve tender ao mínimo global.

Como não pretende-se apresentar um estudo extensivo sobre todos estes temas, não foram apresentados maiores detalhes dobre a minimização do erro e do processo de algoritmo *back-propagation*. Nesta Seção são apresentados apenas os pontos fundamentais para o entendimento do conteúdo a ser abordado na dissertação.

Figura 8 – Erro Tendendo ao Mínimo Local

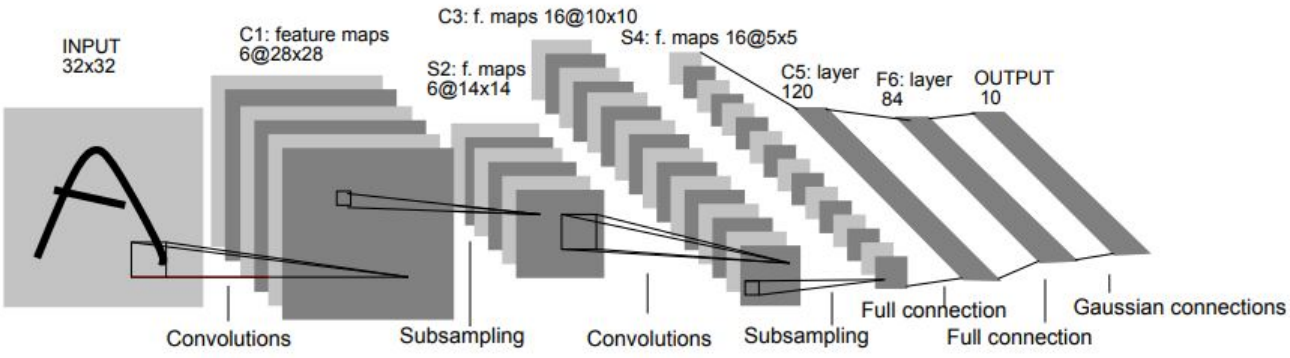


Fonte: Marsland (2014), página 75

2.1.6 A Arquitetura Clássica *LeNet-5*

Lecun et al. (1998) apresentaram a arquitetura de Rede Neural Convolucional *LeNet-5*. Ao desenvolver essa arquitetura, os autores tinham como pretensão obter um melhor desempenho no aprendizado da base de dados *MNIST* (LECUN; CORTES, 2010). Para analisar o desempenho, os autores compararam a taxa de erro resultante da aplicação da *LeNet-5* no *MNIST* com a taxa de erro resultante da aplicação de outros métodos de aprendizado de máquina, incluindo outras arquiteturas *CNN*, para o mesmo problema. A utilização da *LeNet-5* obteve a menor taxa de error, comprovando o desempenho superior da *LeNet-5* com relação aos outros métodos testados.

Figura 9 – Arquitetura da rede *LeNet-5*

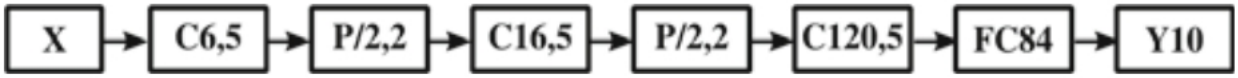


Fonte: Lecun et al. (1998)

A arquitetura *LeNet-5* é ilustrada na Figura 9. Na figura, cada plano é um mapa de características, isto é, um conjunto de unidades cujos pesos são restritos para serem idênticos. Como essa arquitetura de *CNN* foi originalmente proposta para reconhecer dígitos manuscritos

do *MNIST database* (LECUN; CORTES, 2010), a sua entrada é uma imagem 32 × 32 de canal único.

Figura 10 – Representando o *LeNet-5* usando um *DAG*



Fonte: Aghdam (2017), página 98

No *Directed Acyclic Graph (DAG)* ilustrado na Figura 10, *Ca,b* mostra uma camada de convolução com filtros de tamanho $b \times b$ e com a mapas de características. No terceiro e quinto retângulo, *P/a,b* denota uma operação de *pooling* com passada a e tamanho b, *FCa* mostra uma camada totalmente conectada com os neurônios, *Ya* mostra a camada de saída com um neurônio. Essa *CNN* consiste em quatro camadas de *pool* de convolução. Além disso, a última camada de *pooling* é conectada à camada totalmente conectada (AGHDAM, 2017).

Originalmente, Lecun et al. (1998) utilizaram o *Average Pool* na camada de *Pooling*, a função de ativação *Tanh* (exceto na camada de saída) e a função de ativação *Sigmoid* na camada de saída. Em implementações mais modernas da *LeNet-5* é utilizado o *Max Pool* na camada de *Pooling* e a função de ativação *ReLU* (TSANG, 2018).

Na implementação utilizada nessa dissertação, o modelo *LeNet-5* é implementado com o *Average Pool*, conforme o modelo original, e com a função de ativação *ReLU* conforme implementações mais modernas.

Neste capítulo são tratados conceitos envolvidos na *Graphic Processing Unit* (*GPU*) e nas bibliotecas utilizadas ao longo do desenvolvimento dos experimentos. Deve-se ressaltar que não pretende-se apresentar um estudo extensivo sobre todos estes temas, são apresentados apenas os pontos fundamentais para o entendimento do conteúdo a ser abordado na dissertação.

2.2 *Graphic Processing Unit*

Graphic Processing Unit (*GPU*) é um circuito eletrônico projetado para manipular e alterar rapidamente alterar rapidamente a memória e acelerar a criação de imagens em um *buffer* de quadros destinado à saída para um dispositivo de exibição. Tradicionalmente funciona como coprocessadores que executam a renderização de informações gráficas bidimensionais e tridimensionais para exibição em uma tela (WIKIMEDIA FOUNDATION, 2018). No entanto, muitos pesquisadores observaram que a computação de propósito geral com *GPUs* mostrou-se promissora para resolver muitos dos problemas de computação intensiva do mundo, muitas vezes mais rápido que os processadores convencionais (BUCK, 2007).

A primeira *GPU* foi a *NVIDIA GeForce 256* desenvolvida pela NVIDIA Corporation (1999). Sua arquitetura foi definida para atender um *pipeline* gráfico, ou seja, projetadas para

funcionar como um acelerador. Owens et al. (2008) descrevem as etapas desse *pipeline* e explicam que a entrada para uma *GPU* é uma lista de primitivos geométricos, tipicamente triângulos, em um sistema de coordenadas mundiais em 3-D e que através de muitas etapas, essas primitivas são sombreadas e mapeadas na tela onde são montadas para criar uma imagem final. As etapas do *pipeline* gráfico são:

- Operações de Vértices: As primitivas de entrada são formadas a partir de vértices individuais. Como as cenas típicas têm dezenas a centenas de milhares de vértices e cada vértice pode ser calculado independentemente, esse estágio é adequado para processamento paralelo.
- *Primitive Assembly*: Os vértices são montados em triângulos.
- Rasterização: Processo de determinar quais locais de pixel do espaço da tela são cobertos por cada triângulo.
- Operações de Fragmentos: Utiliza informações de cor dos vértices e possivelmente busca dados adicionais da memória global na forma de texturas, cada fragmento é sombreado para determinar sua cor final. Cada fragmento pode ser calculado em paralelo. Este estágio é tipicamente o estágio mais exigente em computação no *pipeline*.
- Composição: Os fragmentos são montados em uma imagem final com uma cor por pixel.

As *GPUs* modernas evoluíram a partir desse *pipeline* gráfico de função fixa para um processador paralelo programável com poder computacional superior ao de *CPUs multicore*. A *GPU* deixou de ser apenas uma unidade de processamento de aceleração gráfica e passou a ser também um processador programável com alto nível de paralelismo e de largura de banda de memória (LINDHOLM et al., 2008).

A *GPU* divide os recursos do processador entre os diferentes estágios, de modo que o *pipeline* é dividido em espaço, não em tempo (abordagem que se difere da *CPU*). A parte do processador que trabalha em um estágio alimenta sua saída diretamente em uma parte diferente que funciona no próximo estágio.

Um marco para o uso de *GPUs* como uma unidade de processamento de propósito geral foi o lançamento da *NVIDIA Geforce 8800* (NVIDIA CORPORATION, 2007) em 2006. A *GeForce 8800* foi a primeira *GPU* a usar processadores de *threads* escalares em vez de processadores vetoriais combinando linguagens escalares padrão como *C* e eliminando a necessidade de gerenciar registros vetoriais e operações de vetor de programa. Adicionou instruções para suportar *C* e outras linguagens de propósito geral, incluindo aritmética de inteiros, aritmética de ponto flutuante *IEEE 754* e instruções de acesso à memória de carregamento / armazenamento com endereçamento de *bytes*. O lançamento lançamento da *Geforce 8800* foi acompanhado pelo lançamento do *framework CUDA*.

Atualmente existe um grande ecossistemas de *GPU Computing* com centenas de milhões de *GPUs* compatíveis com *CUDA* (NVIDIA CORPORATION, 2018a). Vários desenvolvedores e pesquisadores estão utilizando *GPUs* para diversas aplicações que necessitam de processamento paralelo e isso inclui aprendizagem profunda.

Segundo Hennessy e Patterson (2011) "O temo microarquitetura também é utilizado no lugar de organização [de computadores]", os autores exemplificam o conceito de diferentes microarquiteturas "Dois processadores implementam o conjunto de instruções *x86*, mas têm organizações de *pipeline* e *cache* muito diferentes". A cada etapa evolutiva, as *GPUs NVIDIA* sofrem alterações a nível de organização, ou seja, cada nova microarquitetura de *GPU* implementa um determinado conjunto de instruções de forma diferente, por isso cada geração de *GPUs NVIDIA* é referenciada pelo nome de sua microarquitetura.

2.2.1 Microarquitetura *Kepler*

Kepler é uma microarquitetura de *GPU* desenvolvida pela *NVIDIA* como sucessora da microarquitetura *Fermi*. A *microarquitetura* foi inserida no mercado em Abril de 2012 através do lançamento das *GPUs GK104*, *GK106* e *GK110* e é a primeira microarquitetura da *NVIDIA* a se concentrar em eficiência energética (MUJTABA, 2012).

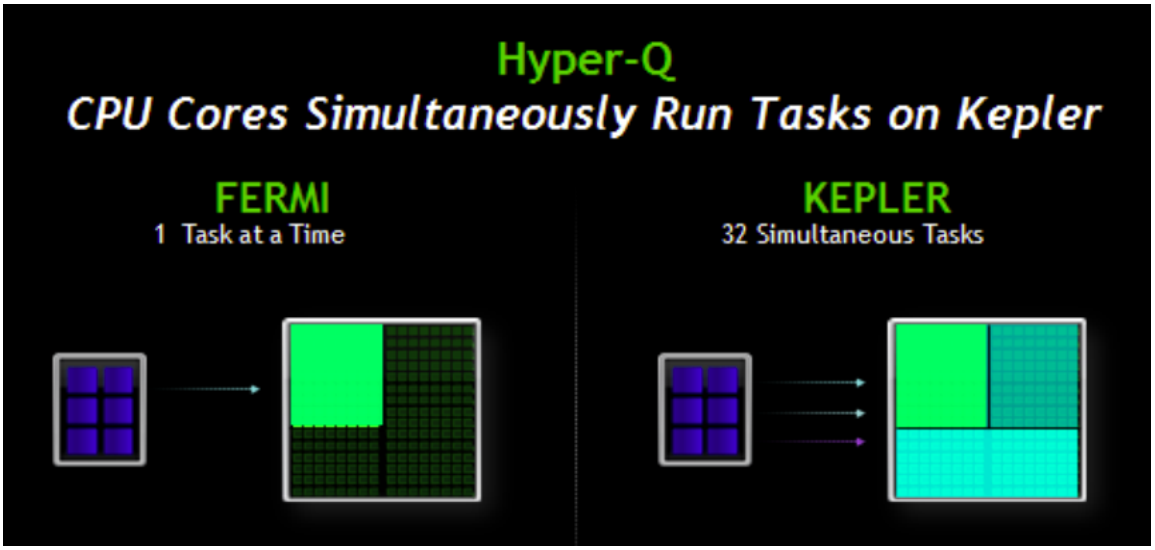
De acordo com Nyland e Jones (2012), a microarquitetura *Kepler* apresenta melhorias com relação a performance, eficiência energética e programabilidade. Para atingir uma melhor eficiência, a *NVIDIA* abandonou o *shader clock* e priorizou o uso de *single clock*, uma vez que aumentar a velocidade do *shader clock* em *GPUs* cada vez menos espaçadas tem um custo muito alto tanto de energia quanto de fabricação. Essa opção por eliminar o *shader clock* fez com que a *NVIDIA* precise operar unidades funcionais duas vezes maiores, porém dois núcleos *Kepler CUDA* consomem 90% da energia de um único núcleo *Fermi CUDA* (SMITH, 2012).

Kuebler e Lake (2020) apontam que melhorias foram proporcionadas através da inserção de algumas tecnologias não existentes em microarquiteturas anteriores. Uma dessas tecnologias é a *Hyper-Q*, que permite que diferentes núcleos de uma *CPU* executem simultaneamente tarefas na *GPU* como ilustrado na Figura 11, em microarquiteturas antigas, eram formadas filas de tarefas para acessar a *GPU*. A tecnologia *Hyper-Q* permite um maior nível de processamento paralelo e reduz as chances da *GPU* ser subutilizada.

A tecnologia *Dynamic Parallelism* inserida nas *GPUs Kepler* habilita *kernels* a iniciar outros *kernels*. Anteriormente, apenas a *CPU* poderia iniciar *kernels*, o que gerava sobrercagas na *CPU* e limitava a capacidade de paralelismo da *GPU*. A Figura 12 ilustra esta tecnologia.

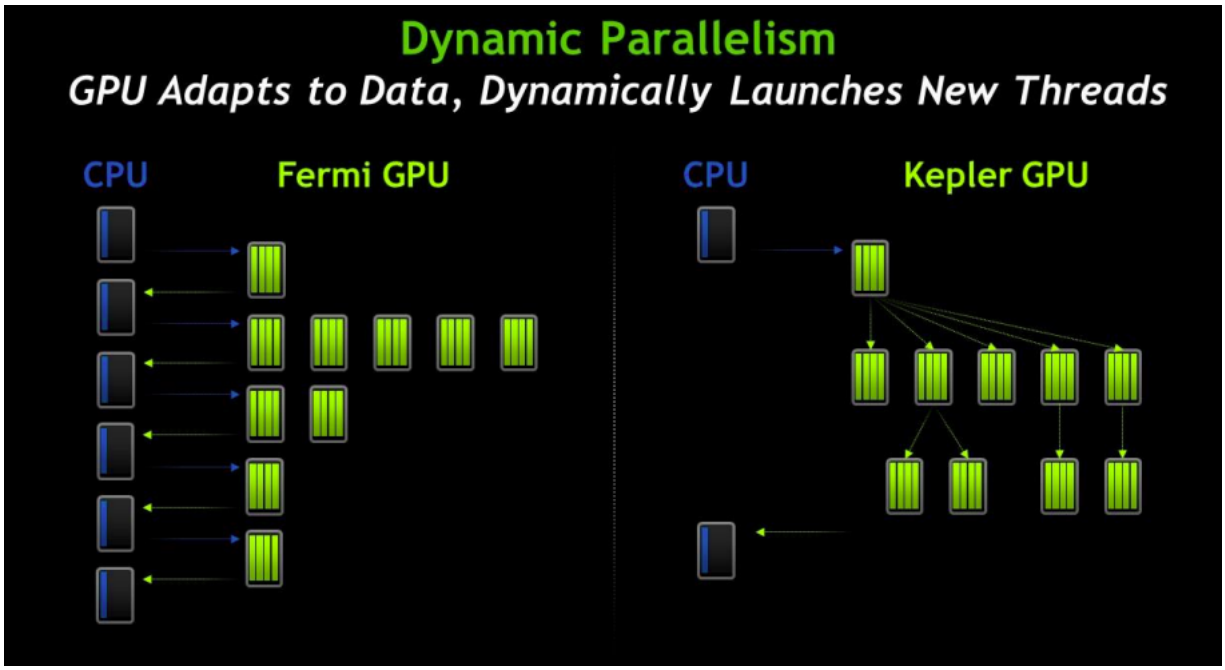
A tecnologia *Dynamic Parallelism* exige um novo tipo de gerenciamento da grade da *GPU*. O *GMU* (*Grid Management Unit*) pode pausar o envio de novas filas e suspendê-las até que o sistema esteja pronto para execução, essa tecnologia fornece a flexibilidade para permitir tempos de execução poderosos, como a tecnologia *Dynamic Parallelism*.

Figura 11 – Tecnologia *Hyper-Q*



Fonte: NVIDIA Corporation (2014b)

Figura 12 – Tecnologia *Dynamic Parallelism*

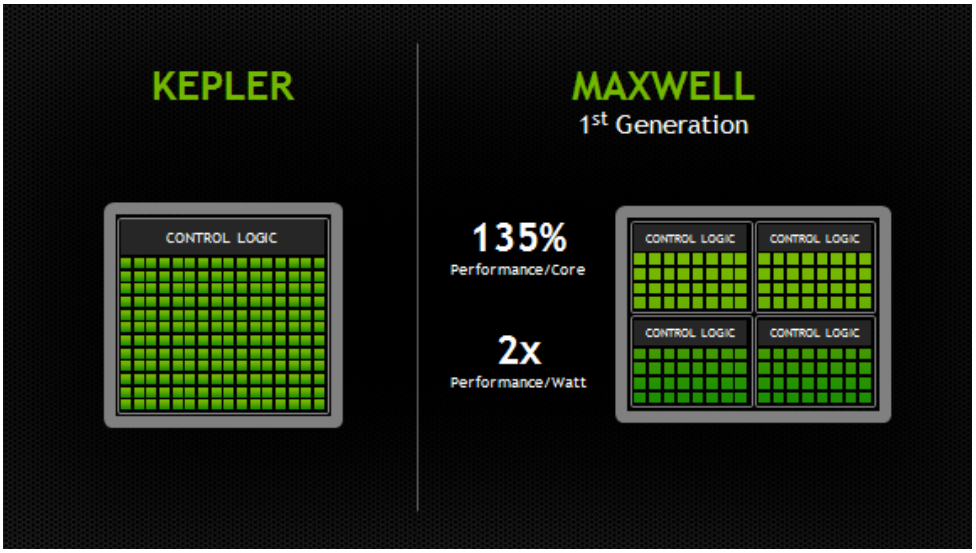


Fonte: NVIDIA Corporation (2014b)

2.2.2 Microrquitetura *Maxwell*

Maxwell é uma microarquitetura de *GPU* desenvolvida pela *NVIDIA* como sucessora da microarquitetura *Kepler*. A arquitetura Maxwell foi introduzida nos modelos posteriores da série *GeForce 700* e também é usada na série *Quadro Mxxx*, todas são fabricadas com o processo de *28 nm da* TSMC (*Taiwan Semiconductor Manufacturing Company, Limited*). A microarquitetura oferece um salto em eficiência energética comparada a microarquitetura *Kepler*, ela possui aproximadamente o dobro de eficiência (HARRIS, 2014a).

Figura 13 – Evolução da microarquitetura *Kepler* para a microarquitetura *Maxwell*



Fonte: Harris (2014a)

A maior eficiência energética dessa microarquitetura se deve ao novo projeto de *Streaming Multiprocessor*, que na microarquitetura *Maxwell* recebeu a sigla *SMM*. O *SMM* possui uma nova organização de caminhos de dados e um planejador de instruções aprimorado que fornece um desempenho por núcleo 40% maior e, em geral, o dobro da eficiência da *GPU Kepler GK104*. Outra melhoria do *SMM* é que ele fornece 64 KB de memória compartilhada dedicada por *SM* - ao contrário das microarquiteturas *Fermi* e *Kepler*, que particionaram os 64 KB de memória entre o *cache L1* e a memória compartilhada (HARRIS, 2014a) (HARRIS, 2014b).

2.2.3 Microarquitetura *Pascal*

A Microarquitetura de *GPU NVIDIA Pascal* foi desenvolvida com foco em *HPC* (*High Performance Computing*) e aplicações de *Big Data*. De acordo com NVIDIA Corporation (2017a), a microarquitetura *Pascal* trouxe algumas inovações tecnológicas com relação a antiga geração de microarquitetura *GPU* (a microarquitetura *NVIDIA Maxwell*):

- 150 bilhões de transistores construídos com a tecnologia *FinFET*[1] de 16 nanômetros, isso garante uma melhoria na eficiência energética
- Aumento em até doze vezes na velocidade de processamento de algoritmos de treinamento de *deep learning* e de até sete vezes na velocidade do processamento de algoritmos de inferência.
- Introdução da tecnologia *NVIDIA NVLink* que aumenta a velocidade bidirecional entre *GPUs*. Essa tecnologia foi projetada para dimensionar aplicativos em várias *GPUs*, fornecendo uma aceleração de 5X na largura de banda de interconexão em comparação a *GPUs* com microarquiteturas anteriores. A introdução dessa tecnologia visa reduzir o gargalo existente em sistemas computacionais *multi-GPU*
- Introdução da tecnologia *CoWoS* (*Chip-on-Wafer-on-Substrate*) com *HBM2* que multiplica em até três vezes a largura de banda de memória. Essa tecnologia unifica processador e dados em um único pacote para oferecer maior eficiência computacional.
- Novas instruções de ponto flutuante de 16 *bits* que entregam uma performance de 21 *teraflops* e novas instruções inteiras de 8 *bits* que entregam uma performance de 47 *teraflops*. A inserção dessas novas intruções visam um melhor desempenho de algoritmos de inteligência artificial.

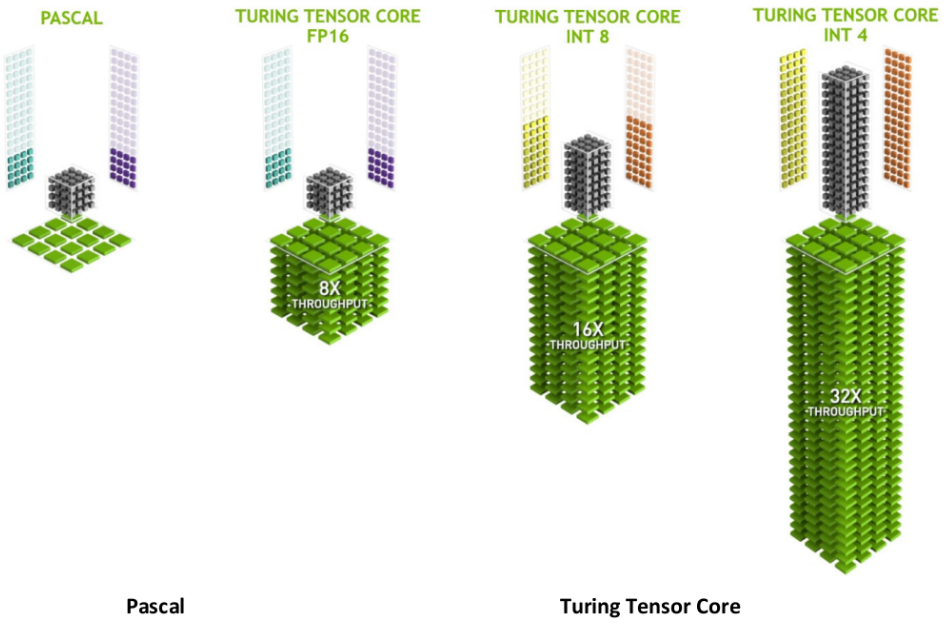
2.2.4 Microarquitetura *Turing*

Turing é uma microarquitetura de *GPU* desenvolvida pela *NVIDIA* como sucessora da microarquitetura *Pascal*, no caso de usuários domésticos, e da microarquitetura *Volta*, no caso de usuários profissionais. Foi lançada em Setembro de 2018 através da série *Quadro RTX* (uso em profissional) e posteriormente através da série *GeForce RTX 20 Series* (uso doméstico) (SMITH, 2018b) (SMITH, 2018a).

A microarquitetura *Turing* traz a segunda geração dos *Tensor Cores* - tecnologia introduzida na microarquitetura *Volta*, são unidades de processamento projetadas especificamente para executar as operações de *tensor* / matriz que são as principais operações usadas na *Deep Learning* e *Deep Inference*. De acordo com NVIDIA Corporation (2020b) "Os *Tensor Cores* permitem computação de precisão mista, adaptando dinamicamente os cálculos para acelerar o rendimento e preservar a precisão". NVIDIA Corporation (2020b) afirma também que a segunda geração oferecem recursos para todas as cargas de trabalho, em suas palavras "Desde acelerações de 10 vezes no treinamento de *AI* com o *Tensor Float 32* (*TF32*), uma nova e revolucionária precisão, a reforços de 2,5 vezes para computação de alto desempenho com *floating point 64 (FP64)*". (NVIDIA CORPORATION, 2018b).

[1] Tipo de estrutura *MOSFET double-gate* auto alinhado (Hisamoto et al., 2000)

Figura 14 – *Tensor Cores*



Fonte: NVIDIA Corporation (2018b), página 16

Na Figura 14 é ilustrado o funcionamento do *Tensor Core* em seus diferentes modos de precisão, os modos *INT8* e *INT4* foram inseridos na microarquitetura *Turing*, enquanto o modo *FP16* já existia na microarquitetura *Volta*. Antes da existência dos *Tensor Cores*, ao receber um *tensor* - estrutura de dado que se assemelha a uma matriz de N dimensões -, os dados eram divididos em partes 2-D para serem processados. A introdução dos *Tensor Cores* possibilita o processamento simultâneo dos dados de um *tensor*, cada modo de precisão permite uma quantidade maior ou menor de processamento simultâneo de dados, o que pode diminuir a taxa de transferência e, consequentemente, melhorar o desempenho de algoritmos de *Deep Learning* e *Deep Inference*.

2.3 Bibliotecas e *APIs* para Redes Neurais Convolucionais

Nessa seção são apresentadas algumas das principais bibliotecas de aprendizagem profunda que utilizam Redes Neurais Convolucionais. Essas bibliotecas foram utilizadas em alguns experimentos.

2.3.1 *CNTK*

O *Microsoft Cognitive Toolkit* (*CNTK*) (SEIDE; AGARWAL, 2016) é um kit de ferramentas de código aberto para aprendizado profundo distribuído de nível comercial desenvolvido pela *Microsoft Research*. Ele descreve as redes neurais como uma série de etapas computacionais através de um grafo direcionado (CHRISBASOGLU, 2018). O CNTK suporta diferentes

arquiteturas de *DNN*, como *Feedforward*, *CNN*, *RNN*, *LSTM* e *Sequence-to-Sequence NN*.

A interface *CNTK* suporta diferentes *APIs* de várias linguagens, como *Python*, *C++*, *C#*, *BrainScript*, tanto na execução da *GPU* (*CUDA*) quanto na execução da *CPU*. De acordo com seus desenvolvedores, foi escrito em *C++* de uma maneira eficiente, onde ele remove cálculos duplicados nos passos para frente e para trás, usa a memória mínima necessária e reduz a realocação de memória reutilizando-os (SHATNAWI et al., 2018).

Uma outra característica é o aproveitamento dos recursos de alta velocidade quando usado com *GPUs* e a plataforma *Microsoft Azure* - serviço de computação em nuvem desenvolvido pela *Microsoft* (CHRISBASOGLU, 2018).

2.3.2 *Keras*

Keras (CHOLLET et al., 2015) é uma Interface de Programação de Aplicações (*API*, do inglês *Application Programming Interface*) de alto nível escrita em *Python* para auxiliar na implementação de algoritmos de *deep learning* e *deep inference* permitindo experimentações rápidas. Foi desenvolvido como parte da pesquisa do projeto *ONEIROS* (Open-ended Neuro-Electronic Intelligent Robot Operating System)), e o seu autor principal é o engenheiro da *Google LLC*, *François Chollet* (KERAS TEAM, 2020). A *API* possui métodos com implementações de otimizadores, funções de ativação e diferentes tipos de camadas de redes neurais (GOOGLE LLC, 2020e).

Diferente de outras bibliotecas apresentadas nesta seção, a *API Keras* precisa de alguma outra biblioteca sendo executada como *back-end*. A *API* é capaz de rodar em cima das bibliotecas *TensorFlow*, *Microsoft Cognitive Toolkit*, *Theano*, ou *PlaidML* e pode funcionar em ambiente *Python* e em ambiente *R*. Porém, a partir de 2017, a equipe da biblioteca *TensorFlow* da *Google LLC* passou a contribuir oficialmente com o desenvolvimento da *API* e, a partir da versão 2.0 da biblioteca *TensorFlow*, a *Google LLC* passou a oferecer a *API Keras* como um conjunto de métodos nativos da *TensorFlow* (GOOGLE LLC, 2020e). A *Microsoft* também passou a oferecer suporte oficial ao uso *API* a partir da versão 2.0 da biblioteca *CNTK*.

2.3.3 *PyTorch*

O *Pytorch* (PASZKE et al., 2017) é desenvolvido pelo *PyTorch core team*, uma equipe formada por varias organizações como: *Nvidia*, *Facebook Open Source*, *ParisTech*, *Twitter*, *Universite Pierre et Marie Curie*, *University of Oxford*, *Stanford University*, *Uber*, entre outras. O foco do desenvolvimento é produzir um *framework* para tensores e redes neurais dinâmicas em *Python* com forte aceleração de GPU (PYTORCH CORE TEAM, 2018).

É uma biblioteca construída para ser profundamente integrado ao *Python*, diferente do *TensorFlow* que é uma ligação do *Python* em uma estrutura monolítica de *C/C++*. Duas de suas principais características são: computação de tensor (como *Numpy*) com forte aceleração

de GPU e redes neurais profundas construídas em um sistema de diferenciação automática no modo reverso que permite alterar a maneira como a rede neural se comporta arbitrariamente com *overhead* (PYTORCH CORE TEAM, 2018) (GOOGLE BRAIN TEAM, 2018b).

O *PyTorch* foi integrado a bibliotecas de aceleração, como *Intel MKL* e *NVIDIA* (*CuDNN*, *NCCL*) para maximizar a velocidade. No núcleo, os *back-ends* de CPU e de Tensor de GPU e de Rede Neural são gravados como bibliotecas independentes com uma *API C99*. Ele pode ser usada como um substituto do *Numpy* para melhor utilização do poder da GPU (PYTORCH CORE TEAM, 2018).

2.3.4 *TensorFlow*

Tensorflow (ABADI et al., 2015) é uma biblioteca de código aberto para computação numérica originalmente desenvolvido pela equipe *Google Brain*. Possui uma arquitetura flexível que permite uma fácil implantação em diferentes arquiteturas (CPU, *GPU* e *TPUs*), por isso é usada em *desktops*, *clusters* e dispositivos móveis. Oferece suporte para *machine learning* e *deep learning* podendo também ser utilizada em vários outros domínios científicos.

Escrita a partir de uma outra biblioteca de *Deep Learning* chamada *DistBelief*, o *Tensorflow* é implementada com base em grafos direcionados. Nesses grafos, os nós representam as operações matemáticas e as arestas representam o fluxo de dados entre os nós, o que torna o *Tensorflow* utilizável em qualquer domínio em que a computação possa ser modelada como um grafo de fluxo (PARVAT et al., 2017) (SHATNAWI et al., 2018).

O *TensorFlow* é desenvolvido como um *Python API* na linguagem *C/C++* buscando alcançar um excelente desempenho (PARVAT et al., 2017). Está disponível para *Windows*, *Linux*, *Mac OS* e em plataformas móveis e embarcadas como *AndroidOS* e *Raspberry* (ABADI et al., 2015) (PARVAT et al., 2017).

3

Estado da Arte

Nesse capítulo apresentamos o estado da arte sobre *benchmarking* de bibliotecas de Redes Neurais Convolucionais. Para situar o estado da arte realizamos um mapeamento sistemático seguindo o método de pesquisa de Petersen, Vakkalanka e Kuzniarz (2015).

3.1 Método de Pesquisa

Este mapeamento aplicou as diretrizes da *SLM* (*Systematic Literature Mapping*), abordadas em Petersen, Vakkalanka e Kuzniarz (2015) para pesquisar, selecionar, revisar e sintetizar os *benchmarkings* de bibliotecas para aprendizagem profunda de publicações acadêmicas relevantes até o ano de 2018. Seguindo a *SLM*, esta seção traz os passos necessários para condução deste estudo. Para tanto faz-se necessário a apresentação do objetivo, questões de pesquisa, da estratégia de busca e da metodologia (critérios) utilizada para a seleção dos artigos de estudos primários. Além disso, cada subseção seguinte detalha cada um dos passos no processo de mapeamento sistemático realizado no presente estudo.

3.1.1 Objetivo do Mapeamento

Seguindo a formalização de definição de objetivo do modelo *GQM* (*Goal Question Metric*), proposto por Basili, Caldiera e Rombach (1994), o objetivo pode ser escrito como: **Analisar** publicações científicas na área de Redes Neurais Convolucionais, **com a finalidade de** identificar *benchmarkings* realizados em bibliotecas **com relação a** tempo de execução e impacto no *hardware* **do ponto de vista** de pesquisadores, cientistas e desenvolvedores que trabalham com redes neurais convolucionais **no contexto de** arquiteturas paralelas, sistemas heterogêneos e computadores *single-board*.

3.1.2 Questão de Pesquisa

De acordo com Petersen, Vakkalanka e Kuzniarz (2015), a questão de pesquisa é primeira etapa da fase de planejamento, onde deve-se determinar o que se está procurando e definir quais resultados se deve alcançar com o mapeamento sistemático. As questões apresentadas a seguir foram definidas para nortear a pesquisa e traçar um perfil das publicações existentes na literatura especializada, a saber

(Q1) - Quais os estudos comparativos e de avaliação de desempenho de bibliotecas para Redes Neurais Convolucionais existentes na literatura?

(Q2) - Quais são os *datasets* mais utilizados?

(Q3) - Quais são as bibliotecas mais avaliadas nas publicações acadêmicas?

(Q4) - Quais são as arquiteturas de Redes Neurais Convolucionais mais utilizadas?

(Q5) - Quais são as arquiteturas computacionais mais utilizadas nos estudos encontrados?

(Q6) - Quais são as métricas mais utilizadas para a avaliação de desempenho das bibliotecas?

Essas perguntas são a base para a construção da *string* de busca e para a definição dos critérios de inclusão e exclusão, servindo para especificar os pontos a serem observados e ponderados na condução da pesquisa, visando a obtenção de resultados que sejam como um resposta a essas questões.

3.1.3 Escopo e Restrições de Pesquisa

Com o objetivo de assegurar a viabilidade da pesquisa, foi definido um escopo para a mesma, que pode ser descrito por meio da definição de critérios de seleção de fontes e algumas restrições. Para a seleção das fontes de pesquisa, foram definidos os seguintes critérios:

- Disponibilidade para consulta web;
- Disponibilidade através do portal de periódicos da Capes (https://www.periodicos.capes.gov.br);
- Disponibilidade de artigos na íntegra por meio do domínio da UFS ou a partir da utilização da mecanismos de busca *Google* e/ou *Google Scholar*;
- Disponibilidade de artigos em inglês, uma vez que é o idioma adotado pela grande maioria das conferências e periódicos nacionais e internacionais relacionados com tema de pesquisa.

3.1.4 Seleção de Fontes

Para a realização da busca por estudos relevantes, as bases eletrônicas alvo desse estudo foram as descritas a seguir na tabela I:

Tabela 1 – Bases Utilizadas

Base	URL
ACM Digital Library	https://dl.acm.org/
IEEEXplore	https://ieeexplore.ieee.org/
Science Direct	https://www.sciencedirect.com/
Scopus	https://www.scopus.com/
Springer	https://www.springer.com
Web of Science	https://www.webofknowledge.com/

3.1.5 Identificação de Palavras-Chaves e Sinônimos

Baseado na questão de pesquisa, duas principais palava-chave são inicialmente identificadas, a saber, *deep learning library*[1] e *benchmarking*. Além disso, possíveis variações como sinônimos ou formas do singular/plural são consideradas.

Como forma de abranger sinônimos (e com base em alguns artigos encontrados previamente) foram adicionados os termos: *deep learning framework*, *deep learning libraries*, *deep learning tools* e *deep learning software tools* como formas de sinônimos para *deep learning library*; *Performance Evaluation* e *Performance Comparison* para *benchmarking*.

3.1.6 Criação de *String* de Busca

A *string* de busca é um procedimento que deve ser adaptado para os motores de busca específicos, com o objetivo de produzir um retorno mais aproximado do ideal para a pesquisa. Uma *string* ineficaz pode trazer um grande número de falsos positivos (PETERSEN; VAKKALANKA; KUZNIARZ, 2015). O resultado da *string* é apresentado a seguir:

> *(benchmarking* ***OR*** *"performance evaluation"* ***OR*** *"performance comparison")* ***AND*** *("deep learning library"* ***OR*** *"deep learning libraries"* ***OR*** *"deep learning framework"* ***OR*** *"deep learning tools"* ***OR*** *"deep learning software tools")*

As principais palavras-chave foram conectadas usando o operador lógico *AND*. Por sua vez, as possíveis variações e sinônimos foram conectados usando o operador lógico *OR*.

[1] Como as bibliotecas disponíveis para Redes Neurais Convolucionais são construídas para suportar outros tipos de redes neurais profundas, optamos por utilizar a expressão *deep learning* ao invés da expressão *convolutional neural network*.

3.1.7 Seleção de Estudos Primários

De acordo com Kitchenham (2004), devem ser seguidos critérios de inclusão e exclusão para os artigos que são retornados pela *string* de busca. Além disso, os critérios de inclusão e exclusão dos estudos primários são os que vão nortear os pesquisadores na seleção dos estudos que foram coletados das fontes de pesquisas, além do que determina o rigor da pesquisa e impossibilita os viesses dos pesquisadores no momento da seleção. Foi definido o seguinte critério de inclusão:

CI1 - Podem ser selecionadas publicações que apresentam estudos comparativos de bibliotecas de *Deep Learning*;

CI2 - Podem ser selecionadas publicações que apresentam uma avaliação de desempenho de bibliotecas de *Deep Learning* em algum sistema computacional;

Em paralelo aos critérios de inclusão, foram definidos critérios de exclusão, descritos a seguir:

CE1 - Não serão selecionadas publicações que não satisfaçam os critérios de inclusão;

CE2 - Não serão selecionadas publicações em que o idioma não seja o inglês;

CE3 - Não serão selecionadas publicações de artigos duplicados;

CE4 - Não serão selecionadas publicações em que as bibliotecas utilizadas não estejam disponíveis gratuitamente na *Internet*;

CE5 - Não serão selecionadas publicações em que o conteúdo disponha apenas de conceitos;

CE6 - Não serão selecionadas publicações que não usem Redes Neurais Convolucionais para a avaliação das bibliotecas.

3.1.8 Processo de Seleção Preliminar (1^o Filtro)

Foram selecionados artigos que apresentem informações no título e no *abstract* relacionado à questão de pesquisa principal.

3.1.9 Processo de Seleção Final (2^o Filtro)

Como a leitura de duas informações (título e *abstract*) não são sucientes para identicar se o estudo é realmente relevante para a pesquisa realizada, torna-se necessário realizar a leitura completa dos estudos que restaram do primeiro ltro. Dessa forma, esta fase do mapeamento, tem como objetivo fazer uma análise mais apurada dos estudos, identicando e extraindo dados também de acordo com os critérios de inclusão e exclusão descritos anteriormente.

3.2 Resultados

Na Tabela 2 é ilustrado o número de trabalhos selecionados em cada etapa de seleção. No final foram selecionados 12 trabalhos, sendo 9 da plataforma *IEEEXplore*, 1 da plataforma *Scopus* e 2 da plataforma *SpringerLink*

Tabela 2 – Documentos Encontrados

Base	Analisados	1^o Filtro	2^o Filtro
ACM DL	7	0	0
IEEE Xplore	27	15	9
Science Direct	14	0	0
Scopus	56	8	1
SpringerLink	211	2	2
Web of Science	8	0	0
Total	323	24	**12**

Nesta seção é apresentada a análise dos resultados obtidos a partir da extração de informações dos 12 trabalhos para responder às questões de pesquisa do presente estudo. A seguir são apresentados os resultados relacionando-os com as questões de pesquisa apresentadas na subseção 3.1.2 deste capítulo.

3.2.1 Estudos Encontrados na Literatura

(Q1) - Quais os estudos comparativos e de avaliação de desempenho de bibliotecas para Redes Neurais Convolucionais existentes na literatura?

Asaadi e Chapman (2017) compararam o desempenho de algumas bibliotecas em uma configuração de *cluster* de *High-Performance Computing (HPC)* regular e sua compatibilidade com a arquitetura de *cluster*. Os autores também estudaram o suporte para recursos específicos de *HPC* fornecidos por cada uma das bibliotecas através de um conjunto de experimentos de comparação de bibliotecas de aprendizagem profundo, as bibliotecas utilizadas foram o *TensorFlow*, o *CNTK* e o *Caffe2* executando uma rede *Inception V3* sobre os *datasets CIFAR-10*, *ImageNet* e *Flowers*. Os resultados de desempenho mostraram queo *CNTK* é a biblioteca com maior compatibilidade para *HPC* entre os três, além dos resultados de desempenho foram observados alguns conflitos de design entre as bibliotecas e a tradicional cadeia de ferramentas *HPC*.

Du et al. (2018) focam no desempenho do treinamento paralelo distribuído em *clusters* de *CPU* de sistemas de supercomputadores, utilizando recursos no sistema do *Tianhe-2*. Os autores avaliaram as bibliotecas *Caffe, TensorFlow* e *BigDL* nas redes neurais *AutoEncoder*, *LeNet*, *ResNet-50* e *AlexNet* treinando os *datasets MNIST* e *CIFAR-10*. As métricas de desempenho utilizadas incluem o tempo do *forward* e *backward propagation*, o tempo de comunicação, o tempo de carregamento de dados, a acurácia e o aumento de velocidade. Os resultados

dos experimentos mostram que o *Caffe* é o melhor em termos de escalabilidade, enquanto o *BigDL* é o melhor em termos de eficiência computacional e o *TensorFlow* possui a biblioteca e funcionalidades mais completas.

Fonnegra, Blair e Díaz (2017) avaliaram e compararam as bibliotecas *TensorFlow*, *Theano* e *Torch*. A comparação é realizada através da implementação de arquiteturas convolucionais e recorrentes para classificar imagens de dois *datasets*: *MNIST* e *CIFAR-10*. As arquiteturas utilizadas foram a *LeNet* e a *LSTM*. Para avaliar o desempenho, os autores calcularam o *forward time* (tempo de execução do gradiente). Como resultados concluiram que *Torch* exige o menor tempo computacional para cada iteração nas configurações de *CPU* e *GPU*, ou seja, ele reporta o menor consumo de tempo para treinamento e computação de gradiente para configurações de *CPU* e *GPU*, porém também reporta o maior tempo de teste. Para o processamento de RNAs com núcleos *LSTM*, o *Tensorflow* foi definitivamente mais rápido que o Theano em todos os casos, exceto pelo tempo de teste da tarefa de reconhecimento *CIFAR-10*. A avaliação não leva em consideração a utilização e a temperatura do *hardware*.

No artigo de Kim et al. (2017) são analisadas as características de desempenho de cinco bibliotecas populares de aprendizagem profunda: *Caffe*, *CNTK*, *TensorFlow*, *Theano* e *Torch* utilizando uma rede *AlexNet* sobre o *dataset ImageNet*. Os autores também sugeriram possíveis métodos de otimização para aumentar a eficiência dos modelos *CNN* construídos pelas bibliotecas. Diferente de outros trabalhos encontrados no mapeamento sistemático, os autores apontam que as características de desempenho indicam que a diferença entre as bibliotecas é causada principalmente por *backends*, especialmente algoritmos de convolução nas camadas de convolução. Apenas ajustando as opções fornecidas pelas bibliotecas, os autores conseguiram um aumento de velocidade de 2X sem modificar o código fonte.

Kruchinin et al. (2015) apresentam uma análise comparativa de algumas bibliotecas de aprendizagem profunda populares e disponíveis gratuitamente: *Caffe, Pylearn2, Torch* e *Theano*. Os autores executam uma rede *MLP* e uma rede *CNN*, cujo o modelo não é detalhado, para treinar o *dataset MNIST*. São calculados o tempo de processamento e a acurácia, como também são medidas a usabilidade e a flexibilidade de cada uma das bibliotecas. Como conclusão, as bibliotecas *Caffe* e *Torch* foram consideradas as mais adequadas para treinar o *dataset MNIST*.

Lin et al. (2018) realizaram um estudo comparativo avaliando as bibliotecas *CNTK*, MXNet, *Neon*, *PyTorch* e *TensorFlow*. As avaliações foram realizadas em *GPUs* e *CPUs* executando as *CNNs ResNet-20*, *ResNet-32* e *IDSIA* para o aprendizado e inferência no *dataset German Traffic Sign Recognition Benchmark* (*GTSRB*). Foram calculados o tempo e a acurácia na inferência e o tempo de treinamento na *CPU* e na *GPU*. Os autores concluiram que o *Neon* e o *MXNet* oferecem a melhor precisão de velocidade e precisão de treinamento em geral para todos os nossos casos de teste, enquanto o *Tensorflow* está sempre entre as bibliotecas com as maiores precisões de inferência.

Liu et al. (2018) apresentaram considerações de projeto, métricas e desafios para o

desenvolvimento de um *benchmark* eficaz para *softwares* de aprendizagem profunda e ilustra algumas observações através de um estudo comparativo de três bibliotecas de *Deep Learning*: *TensorFlow, Caffe* e *Torch*. Os experimentos consistiram na execução de uma rede *LeNet* para aprendizagem e inferência nos *datasets MNIST* e *CIFAR-10*, os autores executaram a rede *LeNet* com dois algoritmos de otimização diferente: *ADAM* e *SGD*. Os resultados ilustraram que essas bibliotecas são otimizadas com suas configurações padrão, porém a configuração padrão otimizada em um conjunto de dados específico pode não funcionar efetivamente para outros conjuntos de dados com relação ao desempenho do tempo de execução e à precisão do aprendizado.

Shams et al. (2017) avaliaram o desempenho de três bibliotecas diferentes, *Caffe*, *TensorFlow* e *Apache SINGA*, em diferentes ambientes de hardware incluindo a ativação e desativação de nós de *CPU* e *GPU*. As bibliotecas foram avaliadas através do dimensionamento e do cálculo do tempo de execução das redes *LeNet*, *GoogleNet*, *AlexNet*, *VGG-19*. Os *datasets* utilizados foram o *ImageNet*, o *MNIST* e o *CIFAR-10*. Através dos experimentos os autores concluíram que o *Apache SINGA* pode apresentar boa escala, mas deixa muito a desejar em termos de tempo de treinamento em comparação com o *Caffe* e o *TensorFlow*. Com relação ao tempo de computação, os autores concluíram que o *Caffe* é mais rápido que outras bibliotecas examinadas e está superando o *TensorFlow*. Redes maiores, especialmente o *VGG-19*, mostram as duas bibliotecas muito mais próximos no tempo, indicando que o desempenho do *TensorFlow* sofre com redes menores como o *LeNet*. Quanto a escalabilidade, as três bibliotecas escalonam com múltiplos *GPUs* em um nó e escalam com múltiplos nós linearmente, enquanto o *Caffe* supera o *TensorFlow*.

Shatnawi et al. (2018) realizaram um estudo comparativo entre três bibliotecas *open source* para *deep learning*: *TensorFlow*, *CNTK* e *Theano*. A avaliação realizada no trabalho leva em consideração o desempenho em *CPU* e *GPU* utilizando redes neurais convolucionais (*CNN*) e os *datasets MNIST* e *CIFAR-10* medindo o tempo de processamento de acordo com o número de *threads* utilizados. Os resultados foram os seguintes: em relação aos conjuntos de dados de reconhecimento de imagem (*MNIST* e *CIFAR-10*) o *CNTK* apresentou melhor desempenho sobre *Tensorflow* e *Theano* em termos de *multithreading* de *GPU* e *CPU*, mas no processamento do *CIFAR-10* utilizando 8,16 e 32 threads em CPU O *Tensorflow* foi mais rápido que o *CNTK*, o *Theano* revelou ser mais demorado do que as outras bibliotecas. Os autores não informaram as arquiteturas de *CNN* utilizadas.

Shi et al. (2016) realizaram um estudo comparativo entre vários bibliotecas de aprendizagem profunda incluindo *Caffe*, *MXNet*, *CNTK*, *TensorFlow* e *Torch*. O estudo considera três tipos de redes neurais, incluindo redes *Perceptron* multicamadas, redes neurais convolucionais (*AlexNet* e *ResNet-50*) e redes neurais recorrentes (*LSTM-32* e *LSTM-64*), sendo executados em duas plataformas de *CPU* e três plataformas de *GPU*. A comparação é feita através do tempo de execução e da taxa de convergência. Foram utilizados conjuntos de dados sintéticos (os autores

não informaram o *dataset* utilizado) para medir o desempenho do tempo de execução e conjuntos de dados do mundo real para medir a taxa de convergência em seus experimentos. Não foram avalidadas a utilização e a temperatura do *hardware*.

Shi, Wang e Chu (2018) avaliaram o desempenho de quatro bibliotecas de aprendizagem profunda distribuídas (*Caffe-MPI, CNTK, MXNet* e *TensorFlow*) em ambientes de uma única *GPU, multi-GPU* e de vários nós. Os autores utilizaram as *CNNs AlexNet*, *GoogleNet* e *ResNet-50* para treinar o *ImageNet*. A avaliação foi realizada através do cálculo do tempo de processamento e mostrou algumas lacunas de desempenho entre as quatro bibliotecas. Segundo os autores, existem métodos sub-ótimos que poderiam ser otimizados para melhorar o desempenho nas estruturas avaliadas em termos de E/S, *cuDNN* invocando, comunicação de dados de *GPUs* intra-nós e *GPUs* inter-nós.

Xu, Shi e Chu (2017) avaliaram sistematicamente o impacto do contêiner *docker* no desempenho de aplicações de aprendizagem profunda. Primeiro, os autores compararam o desempenho dos componentes do sistema (E/S, *CPU* e *GPU*) em um contêiner do *docker* e no sistema *host* e comparamos os resultados para ver se há alguma diferença, posteriormente compararam o desempenho das bibliotecas encapsuladas no *docker* com o desempenho sem o uso do contêiner. Os autores concluíram que o contêiner *docker* não causa desvantagens visíveis durante a execução das bibliotecas de aprendizado profundo, ou seja, encapsular a ferramenta de aprendizado profundo em um contêiner é uma solução viável. Foram utilizadas as bibliotecas *Caffe*, *CNTK*, *MXNet*, *TensorFlow* e *Torch* executando as redes *AlexNet* e *ResNet-50* para treinar o *dataset CIFAR-10*.

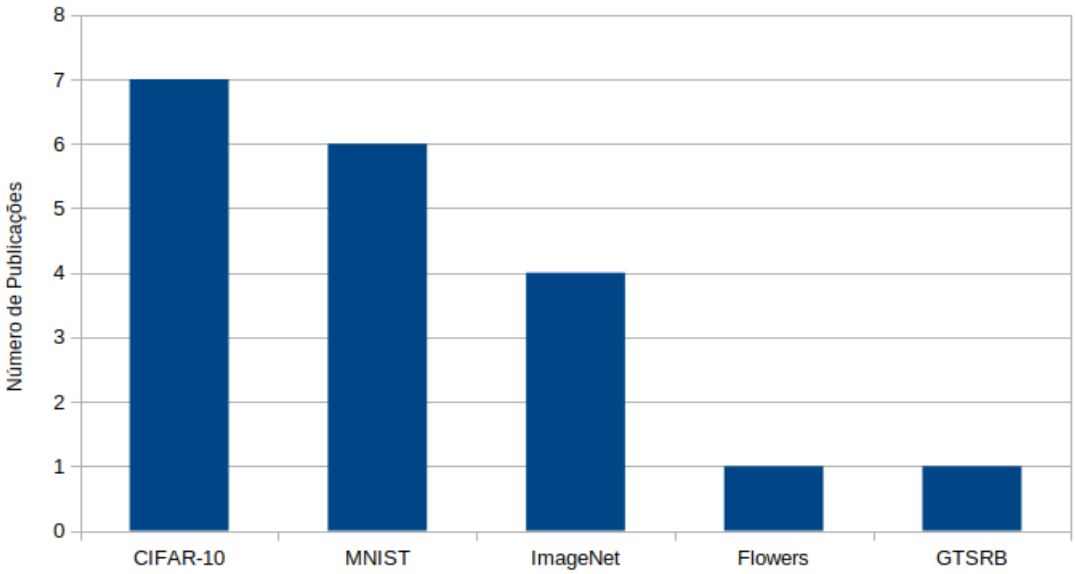
3.2.2 *Datasets*

(Q2) - Quais são os *datasets* mais utilizados?

Essa questão de pesquisa visa identificar os *benchmarks* utilizados durante a avaliação de desempenho das bibliotecas.

Nos documentos selecionados verificou-se que o *dataset* mais utilizado como *benchmark* nos documentos encontrados é o *CIFAR-10* utilizado em 7 trabalhos, seguida do *dataset MNIST* utilizado em 6 trabalhos e o *dataset ImageNet* utilizado em 4 trabalhos. Na Figura 15 é ilustrado graficamente o número de publicações em que cada biblioteca foi avaliada e na Tabela 3 são mostrados os trabalhos que avaliaram cada uma das bibliotecas.

Figura 15 – *Datasets* Utilizados como *Benchmark*



Fonte: Autoria Própria

Tabela 3 – *Datasets* Utilizados como *Benchmark*

Datasets	Trabalhos que Utilizaram
CIFAR-10	(SHATNAWI et al., 2018), (FONNEGRA; BLAIR; DíAZ, 2017), (LIU et al., 2018) (ASAADI; CHAPMAN, 2017), (DU et al., 2018), (SHAMS et al., 2017) (XU; SHI; CHU, 2017)
Flowers	(ASAADI; CHAPMAN, 2017)
GTSRB	(LIN et al., 2018)
ImageNet	(ASAADI; CHAPMAN, 2017), (SHAMS et al., 2017), (KIM et al., 2017) (SHI; WANG; CHU, 2018)
MNIST	(SHATNAWI et al., 2018), (FONNEGRA; BLAIR; DíAZ, 2017), (LIU et al., 2018) (KRUCHININ et al., 2015), (DU et al., 2018), (SHAMS et al., 2017)

Os *datasets* utilizados nos trabalhos encontrados foram:

- ***CIFAR-10***: O conjunto de dados *CIFAR-10* (*Canadian Institute For Advanced Research*) (KRIZHEVSKY; NAIR; HINTON, 2010) é uma coleção de imagens que são comumente usadas para treinar aprendizado de máquina e algoritmos de visão computacional. Contém 60.000 imagens coloridas de 32x32 em 10 classes diferentes, essas classes representam aviões, carros, pássaros, gatos, cervos, cães, sapos, cavalos, navios e caminhões. Existem 6.000 imagens de cada classe.
- ***Flowers***: O *dataset Flowers* (NILSBACK; ZISSERMAN, 2009) é um banco de dados de flores que contém 6150 imagens de treinamento e 1020 imagens de teste divididos em 102 espécies de flores.

- ***GTSBR***: O *dataset GTSBR The German Traffic Sign Recognition Benchmark dataset* (STALLKAMP et al., 2012) é um conjunto de dados multi-classes de sinalização de tráfego disponível publicamente com mais de 50.000 imagens de sinais de trânsito alemães em 43 classes. Os dados foram considerados na segunda etapa do benchmark alemão de reconhecimento de sinais de trânsito realizado no *IJCNN 2011*.

- ***ImageNet***: O projeto *ImageNet* (DENG et al., 2009) é um banco de dados visual projetado para uso em pesquisa de *software* de reconhecimento de objeto visual. Ele possui mais de 14 milhões imagens foram anotadas a mão pelo projeto para indicar quais objetos são retratados e em pelo menos um milhão de imagens, caixas delimitadoras também são fornecidas. O *ImageNet* contém mais de 20.000 categorias com uma categoria típica, como "balão"ou "morango", consistindo em várias centenas de imagens.

- ***MNIST***: O *dataset MNIST* (*Modified National Institute of Standards and Technology database*) (LECUN; CORTES, 2010) é um banco de dados de dígitos manuscritos que possui um conjunto de treinamento de 60.000 exemplos e um conjunto de teste de 10.000 exemplos. É um subconjunto de um conjunto maior disponível no *NIST*. É um bom *dataset* para experimentos de aprendizado e métodos de reconhecimento de padrões em dados do mundo real.

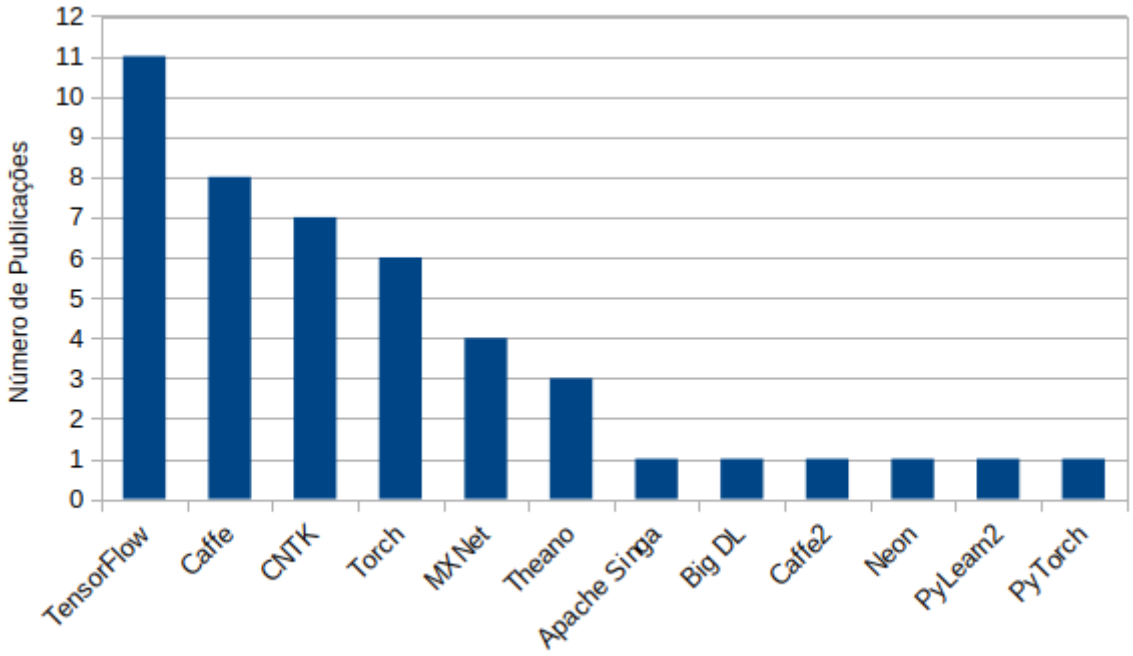
3.2.3 Bibliotecas Utilizadas

(Q3) - Quais são as bibliotecas mais utilizadas nos *benchmarkings*?

Essa questão de pesquisa visa identificar as bibliotecas que já foram avaliadas e levar em consideração as deficiências já encontradas em outros trabalhos.

Nos documentos selecionados verificou-se que a biblioteca mais avaliada foi a biblioteca *TensorFlow* avaliada em 11 trabalhos, seguida da biblioteca *Caffe* avaliada em 8 trabalhos e a *Microsoft CNTK* avaliada em 7 trabalhos. Na Figura 16 é ilustrado graficamente o número de publicações em que cada biblioteca foi avaliada e na Tabela 4 são mostrados os trabalhos que avaliaram cada uma das bibliotecas.

Figura 16 – Bibliotecas Utilizadas nos *Benchmarkings*



Fonte: Autoria Própria

Tabela 4 – Bibliotecas de *Deep Learning*

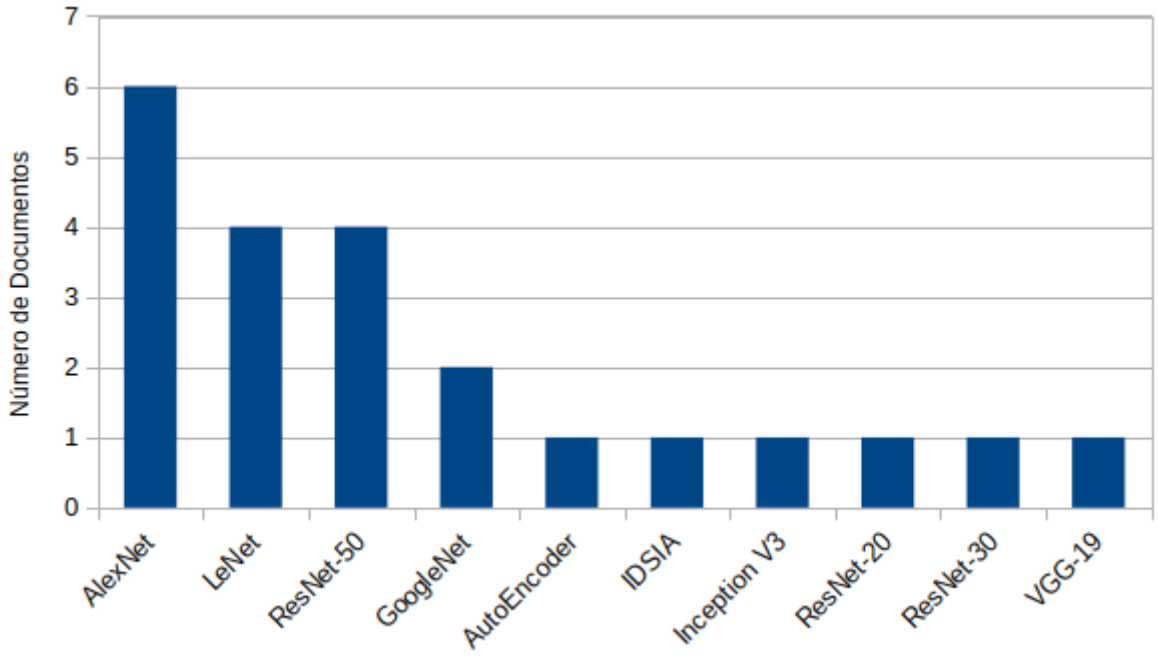
Biblioteca	Trabalhos que Utilizaram
Apache SINGA	(SHAMS et al., 2017),
BigDL	(DU et al., 2018)
Caffe	(SHAMS et al., 2017), (SHI et al., 2016), (LIU et al., 2018), (KIM et al., 2017), (XU; SHI; CHU, 2017), (SHI; WANG; CHU, 2018), (KRUCHININ et al., 2015), (DU et al., 2018)
Caffe2	(ASAADI; CHAPMAN, 2017)
CNTK	(SHI et al., 2016), (SHATNAWI et al., 2018), (LIN et al., 2018), (ASAADI; CHAPMAN, 2017), (KIM et al., 2017), (XU; SHI; CHU, 2017), (SHI; WANG; CHU, 2018)
MXNet	(SHI et al., 2016), (LIN et al., 2018), (XU; SHI; CHU, 2017), (SHI; WANG; CHU, 2018)
Neon	(LIN et al., 2018)
PyLearn2	(KRUCHININ et al., 2015)
PyTorch	(LIN et al., 2018)
TensorFlow	(SHATNAWI et al., 2018), (FONNEGRA; BLAIR; DíAZ, 2017), (LIN et al., 2018), (LIU et al., 2018), (ASAADI; CHAPMAN, 2017), (SHAMS et al., 2017), (KIM et al., 2017), (DU et al., 2018), (XU; SHI; CHU, 2017), (SHI; WANG; CHU, 2018), (SHI et al., 2016)
Theano	(SHATNAWI et al., 2018), (FONNEGRA; BLAIR; DíAZ, 2017), (KIM et al., 2017)
Torch	(SHI et al., 2016), (FONNEGRA; BLAIR; DíAZ, 2017), (LIU et al., 2018), (KIM et al., 2017), (XU; SHI; CHU, 2017), (KRUCHININ et al., 2015)

3.2.4 Arquiteturas de Redes Neurais Convolucionais

(Q4) - Quais são as arquiteturas de Redes Neurais Convolucionais mais utilizadas? Essa questão de pesquisa, assim como a questão de pesquisa Q2, visa identificar os *benchmarks* utilizados durante a avaliação de desempenho das bibliotecas.

Nos documentos selecionados verificou-se que a arquitetura *CNN* mais utilizada nos estudos foi a rede *AlexNet* utilizada em 6 trabalhos, seguida das redes *LeNet* e *ResNet-50* utilizadas, cada uma, em 4 trabalhos. Na Figura 17 é ilustrado graficamente o número de publicações em que cada *CNN* foi utilizada e na Tabela 5 são mostrados os trabalhos que utilizaram cada uma das *CNNs*.

Figura 17 – Arquiteturas *CNN*



Fonte: Autoria Própria

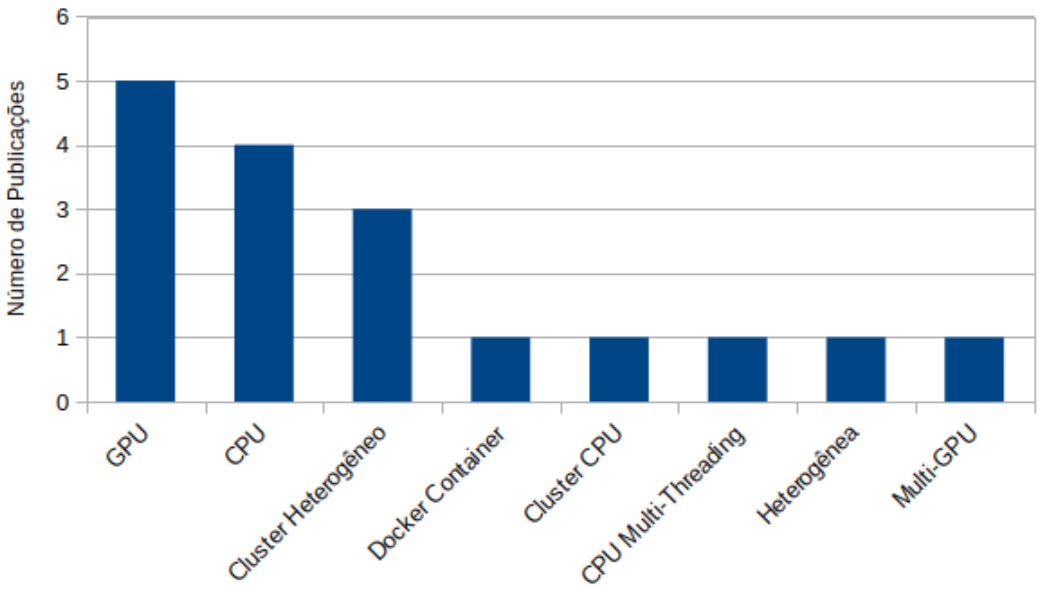
Tabela 5 – Arquiteturas *CNN*

Arquitetura	Trabalhos que Utilizaram
AlexNet	(SHI et al., 2016), (SHAMS et al., 2017), (KIM et al., 2017), (XU; SHI; CHU, 2017), (SHI; WANG; CHU, 2018), (DU et al., 2018)
AutoEncoder	(DU et al., 2018)
LeNet	(FONNEGRA; BLAIR; DíAZ, 2017), (LIU et al., 2018), (SHAMS et al., 2017), (DU et al., 2018)
GoogleNet	(SHAMS et al., 2017), (SHI; WANG; CHU, 2018)
IDSIA	(LIN et al., 2018)
Inception V3	(ASAADI; CHAPMAN, 2017)
ResNet-20	(LIN et al., 2018),
ResNet-32	(LIN et al., 2018)
ResNet-50	(SHI et al., 2016), (SHI; WANG; CHU, 2018), (XU; SHI; CHU, 2017), (DU et al., 2018)
VGG-19	(SHAMS et al., 2017)

3.2.5 Arquiteturas Computacionais

(Q5) - Quais são as arquiteturas computacionais mais utilizadas nos estudos encontrados?

Figura 18 – Arquiteturas Computacionais



Fonte: Autoria Própria

Nos documentos selecionados verificou-se que a arquitetura computacional mais utilizada nos estudos foi as plataformas *GPU* utilizada em 5 trabalhos, seguida da plataforma *CPU* utilizada em 4 trabalhos e a plataforma *cluster heterogêneo* utilizada em 3 trabalhos. Na Figura

18 é ilustrado graficamente o número de publicações em que cada *CNN* foi utilizada e na Tabela 6 são mostrados os trabalhos que utilizaram cada uma das *CNNs*.

Tabela 6 – Arquiteturas Computacionais

Arquiteturas Computacionais	Trabalhos que Utilizaram
Docker Container	(XU; SHI; CHU, 2017)
Cluster CPU	(DU et al., 2018)
Cluster Heterogêneo	(ASAADI; CHAPMAN, 2017), (SHAMS et al., 2017), (SHI; WANG; CHU, 2018)
CPU	(SHI et al., 2016), (FONNEGRA; BLAIR; DíAZ, 2017) (LIN et al., 2018) (LIU et al., 2018)
CPU Multi-threading	(SHATNAWI et al., 2018)
GPU	(SHI et al., 2016), (SHATNAWI et al., 2018), (LIN et al., 2018) (FONNEGRA; BLAIR; DíAZ, 2017), (LIU et al., 2018)
Heterogênea	(KRUCHININ et al., 2015)
Multi-GPU	(KIM et al., 2017)

As arquiteturas computacionais encontradas no mapeamento foram classificadas seguindo os seguintes critério:

- ***Docker Container***: Avaliação de desempenho das bibliotecas em um *Docker Container* visando verificar o impacto do uso do *docker*.
- ***Cluster CPU***: Plataforma com vários nós de *CPU*.
- ***Cluster Heterogêneo***: Plataforma com vários nós de *CPU* e *GPU*.
- ***CPU***: Plataforma utilizando somente *CPU* sem levar em consideração o número de *threads utilizados*.
- ***CPU Multi-threading***: Plataforma utilizando somente *CPU* levando em consideração o número de *threads* utilizados.
- ***GPU***: Plataforma com aceleração *GPU* levando em consideração apenas o desempenho na *GPU*, ou seja, não analisa o processamento da *CPU*.
- **Heterogênea**: Plataforma que utiliza *GPU* e *CPU* conjuntamente levando em consideração o desempenho de ambos.
- ***Multi-GPU***: Plataforma que utiliza várias *GPUs* em um único nó.

Os trabalhos que utilizam a arquitetura computacional classificada como *GPU* realiza a avaliação de desempenho com a *CPU*, mas não medem o impacto da *CPU* no desempenho. Isso pode ser considerada uma ameaça à validade do experimento.

3.2.6 Métricas de Desempenho

(Q6) - Quais são as métricas mais utilizadas para a avaliação de desempenho das bibliotecas?

Essa questão de pesquisa visa identificar os critérios de avaliação de desempenho das bibliotecas.

Para simplificar o conteúdo das tabelas, foram adotas as seguintes siglas para as métricas: Acurácia (Ac), *Batch Size* (BS), Número de Épocas (NE), Número de *GPUs* (NGPU), Número de Nós de *CPU* (NNCPU), Número de nós de *GPU* (NNGPU), Número de *Threads* da *CPU* (NT), Paralelismo de Dados (PD), *SpeedUp* (SU), Tempo de execução do processo de *Backward* (TB), Tempo de Comunicação entre diferentes positivos em paralelo (TC), Tempo de execução do processo de *Forward* (TF), Tempo de computação do Gradiente (TG), Tempo de Inferência (TI), Tempo de Iteração Única (TIU), Tempo Médio de Carregamento de Dados (TMCD), Tempo de transferência de *CPU* para *GPU* (TCPUGPU), Tempo de Treinamento (TT).

Como cada trabalho encontrado no mapeamento utiliza métricas diferentes, os resultados não foram ilustrados por gráfico e sim, apenas, na Tabela 7 na qual são expostos os resultados de todas as questões de pesquisa. O critério mais frequente para a avaliação é o tempo de execução, porém não há um consenso sobre qual parte da execução medir, Fonnegra, Blair e Díaz (2017) e Kim et al. (2017) medem o tempo de todo o processo de treinamento com um número alto de épocas, já Shi, Wang e Chu (2018) medem o tempo de execução do processo de *backward* e *forward* separadamente. A acurácia obtida utilizando funções de diferentes bibliotecas também é uma métrica frequente nos trabalhos.

3.3 Análise Comparativa

Na Tabela 7 é mostrado um comparativo entre todos os trabalhos mapeados levando em consideração os *datasets*, bibliotecas, modelo de *CNN* e arquitetura computacional utilizada.

Nenhum trabalho explora arquiteturas de baixa potência, como também não realizam testes estatísticos para validar os seus resultados.

Poucos trabalhos exploram a escalabilidade, porém Shams et al. (2017) exploraram a escalabilidade das bibliotecas levando em consideração tanto a *CPU* quanto a *GPU* por meio de experimentos com diferentes números de nós ativos em um *cluster* heterogêneo.

Tabela 7 – Análise Comparativa dos Trabalhos Mapeados

Trabalho	*Dataset*	Biblioteca	*CNN*	Arquitetura	Métricas
Asaadi e Chapman (2017)	*CIFAR-10*, *ImageNet*, *Flowers*	*TensorFlow*, *CNTK*, *Caffe2*	*Inception V3*	*cluster* Heterogêneo	NE/Erro, TT(s)/NE
Du et al. (2018)	*MNIST*, *CIFAR-10*	*Caffe*, *TensorFlow*, *BigDL*	*AutoEncoder*, *LeNet*, *ResNet-50*, *AlexNet*	*cluster CPU*	Ac/PD, TT(s)/PD, TC(s)/PD, SU/PD, TMCD(s)/PD, TIU(s)/PD
Fonnegra, Blair e Díaz (2017)	*MNIST*, *CIFAR-10*	*TensorFlow*, *Theano*, *Torch*	*LeNet*	*CPU, GPU*	TT(s), TI(s), TG(s)
Kim et al. (2017)	*ImageNet*	*Caffe*, *CNTK*, *TensorFlow*, *Theano*, *Torch*	*AlexNet*	*Multi-GPU*	TB(ms), TS(ms), Operações/Camada
Kruchinin et al. (2015)	*MNIST*	*Caffe*, *Pylearn2*, *Torch*, *Theano*	Não Informado	Heterogênea	TT(min), Ac
Lin et al. (2018)	*GTSRB*	*CNTK*, *MXNet*, *Neon*, *PyTorch*, *TensorFlow*	*ResNet-20*, *ResNet-50*, *IDSIA*	*CPU, GPU*	TT(s), Ac
Liu et al. (2018)	*MNIST*, *CIFAR-10*	*TensorFlow*, *Caffe*, *Torch*	*LeNet*	*CPU, GPU*	Ac, TT(s), TI(s)
Shams et al. (2017)	*ImageNet*, *CIFAR-10*, *MNIST*	*Caffe*, *TensorFlow*, *Apache SINGA*	*LeNet*, *GoogleNet*, *AlexNet*, *VGG-19*	*Cluster* Heterogêneo	TT(s)/BS, TT(s)/NGPU, TT(s)/NNGPU, TT(s)/NNCPU
Shatnawi et al. (2018)	*CIFAR-10*, *MNIST*	*TensorFlow*, *CNTK*, *Theano*	Não Informado	*CPU Multi-threading, GPU*	TT(s),TT(s)/NT
Shi et al. (2016)	Não Informado	*Caffe*, *MXNet*, *CNTK*, *TensorFlow*, *Torch*	*AlexNet*, *ResNet-50*	*CPU, GPU*	TT(s)/NT, TT(s)/BS
Shi, Wang e Chu (2018)	*ImageNet*	*Caffe*, *CNTK*, *MXNet*, *TensorFlow*	*AlexNet*, *GoogleNet*, *ResNet-50*	*cluster* Heterogêneo	SU/NGPU, TB(s), TF(s), TG(s), TIU(s), TCPUGPU(s)
Xu, Shi e Chu (2017)	*CIFAR-10*	*Caffe*, *CNTK*, *MXNet*, *TensorFlow*, *Torch*	*AlexNet*, *ResNet-50*	*Docker*	TT(s)/BS

3.4 Conclusão do Mapeamento

Nesse capítulo foi apresentado um mapeamento sistemático para situar o estado da arte. Ao todo foram selecionados 12 trabalhos que avaliam o desempenho de bibliotecas de *deep learning* utilizando alguma Rede Neural Convolucional. Além de visar identificar os trabalhos, o mapeamento também fez um levantamento sobre os *datasets* utilizados, as bibliotecas, as arquiteturas de *CNN* utilizadas e arquiteturas computacionais como forma de identificar os *benchmarks*.

É possível concluir que há alguns *datasets* já consolidados como *benchmarks* como o *MNIST* e o *CIFAR-10* utilizados na maioria dos trabalhos. Quanto as arquiteturas de *CNN* utilizadas há a predominância das arquiteturas clássicas como a *LeNet* e a *AlexNet* citadas na subseção 2.1.6.

Quanto as arquiteturas computacionais, há uma deficiência quanto a avaliação de desempenho das bibliotecas em plataformas computacionais de baixa potência, nenhum trabalho mapeado explorou esse tipo de arquitetura. A maioria dos trabalhos exploram as *GPUs* sejam em *clusters* ou em plataformas mais simples com somente um nó, porém grande parte dos experimentos possuem uma ameaça à validade. Ao avaliar o desempenho em *GPUs*, muitos autores não levam em consideração o impacto causado pela *CPU*, é o caso dos trabalhos que utilizaram as arquiteturas rotuladas apenas como *GPU* na subseção 3.2.5.

Os autores dos trabalhos selecionados priorizaram a avaliação de desempenho das bibliotecas mantidas por empresas consolidadas como o *TensorFlow*, mantida pela *Google Brain Team*, e a *CNTK*, mantida pela *Microsoft*. Algumas outras bibliotecas, como *PyTorch*, são pouco estudadas.

Quanto ao desempenho das bibliotecas testadas nos trabalhos mapeados, os resultados variaram de acordo com a arquitetura computacional, o *dataset* utilizado e os critérios utilizados. A biblioteca *Caffe* apresentou uma boa escalabilidade em *clusters* com nós de *CPU* (FONNEGRA; BLAIR; DíAZ, 2017), a biblioteca *CNTK* apresentou boa compatibilidade em ambiente *HPC* (ASAADI; CHAPMAN, 2017), no trabalho de Du et al. (2018) a biblioteca *Torch* apresentou um bom desempenho no processo de treinamento, mas um desempenho ruim no processo de inferência. Outras bibliotecas como *TensorFlow* e *BigDL* também se destacaram nas avaliações de desempenho. Os diferentes resultados obtidos nos trabalhos selecionados apontam para a necessidade de padronização dos critérios de avaliação de desempenho das bibliotecas, como também demonstram a necessidade de uma maior rigidez no uso do paradigma experimental.

Um das principais deficiências encontradas nos trabalhos é a pouca exploração da escalabilidade das bibliotecas nos experimentos. Uma outra deficiência é que todos os trabalhos mapeados não utilizam testes estatísticos para validar suas comparações e nem apontam possíveis ameaças à validade.

4

Estudo do Impacto da *API Keras* Sobre o Desempenho das Bibliotecas

Nesse capítulo é descrito o estudo experimental realizado para avaliar o impacto da ferramenta *Keras* sobre o desempenho de bibliotecas. Os resultados desse estudo servem como base para a implementação dos códigos que são utilizados nos estudos experimentais de comparação de desempenho entre as bibliotecas apresentados nos próximos capítulos.

4.1 Motivação do Estudo Experimental

A realização deste estudo experimental foi motivada devido a necessidade, sentida durante a implementação dos modelos, de investigar a forma mais viável de implementar modelos *CNNs* considerando o tempo de execução dos algoritmos e a facilidade de desenvolvimento.

A partir do lançamento do *TensorFlow 2*, a *Google LLC* passou a estimular os desenvolvedores a utilizarem a *API* de alto nível *Keras* para implementar modelos de Redes Neurais com a biblioteca *TensorFlow*, essa forma de implementação substitui a forma de implementação "clássica"que utiliza prioritariamente métodos e funções de mais baixo nível da própria biblioteca *TensorFlow*. No *site* da biblioteca *TensorFlow*, a *API Keras* é descrita de acordo com a afirmação: "Essa é uma *API* de alto nível para criar e treinar modelos [...], [a *API*] facilita o uso do *TensorFlow* **sem sacrificar a flexibilidade e o desempenho**"(GOOGLE LLC, 2020c). Apesar desta afirmação, não foram encontrados estudos experimentais na literatura que avaliam o impacto da *API Keras* no desempenho da biblioteca *TensorFlow*[1]. A facilitação do uso das bibliotecas proporcionada pela *API Keras* é tratada no Apêndice A através do exemplo da *CNN LeNet-5*.

Caso os resultados do estudo demonstrassem que o uso da *API Keras* não acarreta prejuízos no desempenho das bibliotecas, os modelos de *CNN* seriam implementados com as

[1] Também não foram encontrados estudos experimentais sobre o impacto do uso da *API Keras* na biblioteca *CNTK* - biblioteca que também foi avaliada nessa dissertação

funcionalidades de alto nível da *API*[2]; caso contrário, os modelos seriam implementados sem a utilização das funcionalidades de alto nível da *API Keras*. Essas alternativas foram elaboradas considerando que se o uso de funcionalidades de alto nível da *API Keras* não apresentam prejuízos no desempenho das bibliotecas, então é preferível para os desenvolvedores - com exceção de algumas situações que exijam programação de baixo nível - implementar os modelos *CNN* utilizando as funcionalidades da *API Keras*.

4.2 Definição e Planejamento

Nesta Seção, seguindo a metodologia descrita em Wohlin et al. (2012), é apresentado o planejamento do experimento controlado proposto. As subseções apresentadas a seguir apresentam o objetivo e o planejamento do experimento (contexto, variáveis dependentes e independentes, hipóteses, objetos de análise, projeto do experimento e instrumentação).

4.2.1 Definição do Objetivo

O objetivo deste experimento é avaliar (segundo tempo de execução) o impacto do uso da *API Keras* no desempenho das bibliotecas *TensorFlow* e *CNTK* em um ambiente computacional acelerado por *GPU* verificando o comportamento nas fases de treinamento e teste de uma *CNN*.

Para avaliar o impacto do uso da *API Keras* nas diferentes arquiteturas computacionais, foi realizado um experimento *In Vitro* - realizado em ambiente fechado com condições controladas - levando em consideração duas versões diferentes da biblioteca *TensorFlow* (versão 1.15 e versão 2.2) e uma versão da biblioteca *CNTK* (versão 2.7).

Seguindo a formalização de definição de objetivo do modelo *GQM* (*Goal Question Metric*), proposto por Basili, Caldiera e Rombach (1994), o objetivo pode ser reescrito como: **Analisar** a execução de códigos implementados **com** o auxílio da *API Keras* (utilizando *TensorFlow* e *CNTK* como *back-end*) e a execução de códigos implementados **sem** o auxílio da *API Keras* - implementação direta com os métodos disponíveis nativamente nas bibliotecas *CNTK* e *TensorFlow* - **com a finalidade de** compará-los **com relação a** tempo de execução, **do ponto de vista** de pesquisadores, cientistas e desenvolvedores que trabalham com *deep learning* **no contexto de** redes neurais convolucionais *LeNet-5* executando em arquiteturas computacionais aceleradas por *GPU*.

4.2.2 Planejamento

Seleção de Contexto: O experimento foi *In Vitro*. Em cada experimento, para cada um dos códigos, foram executados os algoritmos de treinamento (aprendizado) e de teste de uma

[2] Com exceção dos modelos implementados com a biblioteca *PyTorch*, pois esta biblioteca não apresenta suporte para a *API Keras*.

CNN LeNet-5 processando o conjunto de dados *MNIST*, o que permitiu verificar o impacto do uso da *API Keras* no desempenho do *TensorFlow*. Os experimentos foram realizados utilizando a ferramenta *Google Colab*[3].

Variáveis Dependentes: Tempo de execução da fase de treinamento (s), tempo de execução da fase de teste (s).

Variáveis Independentes: Compilação do código, capacidade de paralelização da biblioteca, complexidade dos algoritmos contidos nas bibliotecas (*TensorFlow* e *Keras*), ambiente de execução (*hardware* e *software* utilizado no sistema do *Google Colab*).

Formulação de Hipótese: Podemos resumir as questões de pesquisa do experimento em duas: O tempo de execução da fase de treinamento do código implementado com o auxílio da *API Keras* é semelhante ao tempo de execução da fase de treinamento do código implementado sem o auxílio da biblioteca *Keras*? O tempo de execução da fase de teste do código implementado com o auxílio da *API Keras* é semelhante ao tempo de execução da fase de teste do código implementado sem o auxílio da biblioteca *Keras*? Cada uma dessas questões será respondida para cada uma das versões do *TensorFlow* utilizada (versão 1.15 e versão 2.2) e para a biblioteca *CNTK*.

Todos os questionamentos são respondidos a partir de dados extraídos da execução de uma mesma *CNN*. Para a primeira questão pode-se considerar o tempo de execução da fase de treinamento da *CNN* (μ^{CPU}), para a segunda questão pode-se usar utilizar o tempo de execução da fase de treinamento da CNN durante a fase de teste (τ^{CPU}). Cada uma das medidas deve ser feita para cada um dos códigos: a implementação utilizando a *API Keras* e a implementação utilizando apenas os métodos nativos das bibliotecas *TensorFlow* e *CNTK*. Neste contexto, as seguintes hipóteses podem ser verificadas (a saber, MN = implementação utilizando métodos nativos das bibliotecas, K = *Keras*):

Hipótese 1 (Para cada uma das bibliotecas incluindo as duas versões da TensorFlow

H0: Não há diferenças estatística entre o tempo de execução dos códigos durante a fase de treinamento, ou seja, $(\mu^i_{MN} = \mu^i_K)$

H1: Existem diferenças estatísticas entre o tempo de execução dos códigos durante a fase de treinamento, ou seja, $(\mu^i_{MN} \neq \mu^i_K)$

Hipótese 2 (Para cada uma das bibliotecas incluindo as duas versões da TensorFlow)

H0: Não há diferenças estatística entre o tempo de execução dos códigos durante a fase de teste, ou seja, $(\tau^i_{MN} = \tau^i_K)$

H1: Existem diferenças estatísticas entre o tempo de execução dos códigos durante a fase de teste, ou seja, $(\tau^i_{MN} \neq \tau^i_K)$

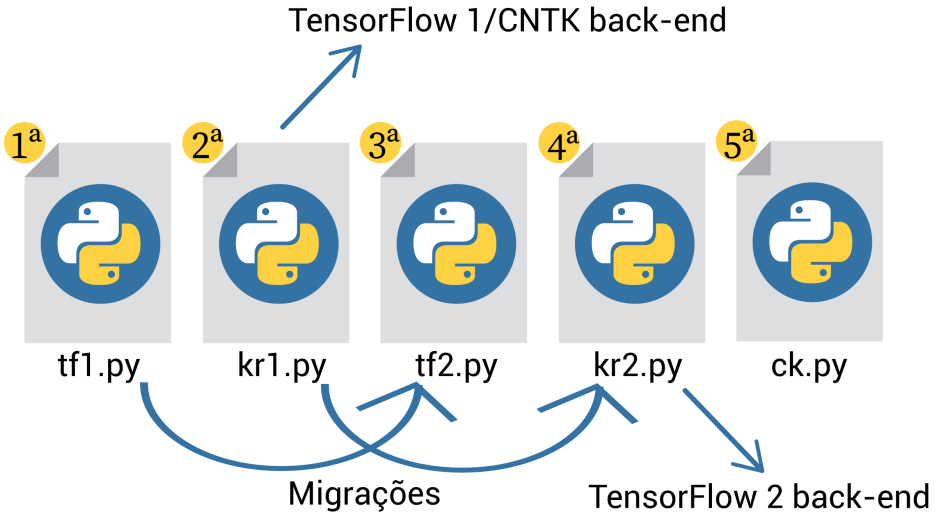
[3] O *Google Colab* (*Colaboratory*) é um serviço em nuvem para disseminar o ensino e a pesquisa em aprendizado de máquina. O serviço oferece ambientes opcionais de computação acelerada, incluindo *GPU* (GOOGLE LLC, 2020a)

Seleção de Objetos: O experimento utilizou um conjunto de dados de imagens, o *Modified National Institute of Standards and Technology database* (*MNIST database*) (LECUN; CORTES, 2010). O *MNIST* é um grande conjunto de dados de dígitos manuscritos comumente utilizado para treinar sistemas de sistemas de processamento de imagens e possui as seguintes características:

- Imagens de 28 x 28 *pixels*.
- 10 classes, uma classe para cada dígito.
- Subconjunto com 60000 dados de treinamento.
- Subconjunto com 10000 dados de teste.

Como ilustrado na Figura 19, o experimento utilizou cinco códigos com uma rede *CNN LeNet-5* (LECUN et al., 1998) implementada em *Python*. O primeiro código (*tf1.py)* utiliza os métodos nativos da biblioteca *TensorFlow 1.15* sem o auxílio da *API Keras*. O segundo código (*kr1.py*) é implementado em alto nível com a *API Keras*, este código foi executado em duas situações: utilizando a biblioteca *TensorFlow 1.15* como *back-end* e utilizando a biblioteca *CNTK* como *back-end*. O terceiro código (*tf2.py*) foi implementado utilizando os métodos nativos do *TensorFlow 2.2* - desconsiderando os métodos da *API Keras* disponíveis nativamente na versão 2.2 da biblioteca. O quarto código (*kr2.py*) utiliza a *API Keras* com a biblioteca *TensorFlow 2.2* como *back-end*[4]. O quinto e último código (*ck.py*) utiliza os métodos nativos da biblioteca *CNTK*.

Figura 19 – Códigos utilizados



Fonte: Autoria Própria

[4] A diferença da implementação do quarto código para o segundo código é que no quarto é utilizado a versão da *API Keras* disponibilizada pela *Google LLC* junto do pacote da biblioteca *TensorFlow 2.2*

Os quatro primeiro códigos utilizados são modificações de códigos desenvolvidos por Gazar (2018), as modificações foram realizadas com a finalidade de deixá-los com os mesmos parâmetros e para criar uma versão de cada código que fosse compatível com o *TensorFlow 2*, esse processo é explicado na Subseção 4.3.1. O quinto código foi implementado a partir da modificação dos tutoriais disponíveis no repositório oficial do CNTK no *GitHub* (MICROSOFT CORPORATION, 2019). Os códigos utilizados durante o experimento estão disponíveis no repositório de Florencio (2020a).

Projeto do Experimento: o projeto dos experimentos pode ser resumido nas seguintes etapas:

1. Preparar os ambientes de execução;
2. Criação dos códigos para a biblioteca *TensorFlow 1.15* a partir de modificações dos códigos de Gazar (2018);
3. Criação dos códigos para a biblioteca *TensorFlow 2.2* a partir da migração dos códigos criados na etapa anterior. Essa etapa foi realizada seguindo as recomendações, de uso de ferramentas e de métodos, disponíveis em Google LLC (2020d);
4. Criação dos códigos para a biblioteca *CNTK* a partir de modificações dos códigos de Microsoft Corporation (2019);
5. Medir 100 vezes o tempo de execução do algoritmo durante a fase de treinamento para cada um dos códigos;
6. Medir 100 vezes o tempo de execução do algoritmo durante a fase de teste para cada um dos códigos;
7. Aplicar testes estatísticos para análise das hipóteses;

Instrumentação 1 (Para os códigos executados com a biblioteca *TensorFlow*): Os *softwares* utilizados foram o ambiente do *Google Colab Python* na versão 3.6, a biblioteca *TensorFlow* na versão 1.15 e na versão 2.2, o *CUDA* na versão 10.1, o *Driver NVIDIA* versão 418.67. O conjunto de *hardware* utilizado possui uma *CPU Intel Xeon 2.00 GHZ* com apenas um *core* que conta com duas *threads*, possui uma memória *RAM* de 13 GB, um *HD* de 34 GB e uma *GPU NVIDIA Tesla P4*. As especificações da *GPU* estão na Tabela 9, essas informações foram retiradas do seu *datasheet* (NVIDIA CORPORATION, 2019c) e do seu *product brief* (NVIDIA CORPORATION, 2017b).

Instrumentação 2 (Para os códigos executados com a biblioteca *CNTK*): Os *softwares* utilizados foram o sistema operacional *Ubuntu 18.04 LTS*, *Virtualenv* versão 20.0.21, ambiente *Python* na versão 3.6, o *CUDA* na versão 10.2, o *Driver NVIDIA* versão 440.59, o *Open MPI*

Tabela 8 – Especificações da *GPU NVIDIA Tesla P4*

Informação Técnica	Valor
Microarquitetura	*Pascal*
NVIDIA Cuda Cores	2560
Mémoria	8 GB
Single-Precision Performance	5,5 *TeraFLOPS*
Operações com Valores Inteiros	22 *TOPS* (*Tera-Operations* por segundo)
Velocidade de Memória	192 GB/s
GPU Clock - Base	885 MHz
GPU Clock - Maximum Boost	1531 MHz (1131 MHz *Default*)
GPU Clock - Idle	405 MHz

versão 2.1.1, *CNTK* versão 2.7. O *hardware* utilizado é um *notebook Acer Aspire A515-51G-C97B* com mémoria *RAM* 8GB (2x4GB) 2133MHz *DDR4*, processador *8th Generation Intel Core i5-8250U Quad Core (6MB Cache, up to 3.4GHz)*, disco rígido 1TB 5400rpm e *GPU NVIDIA GeForce MX130*. As especificações da *GPU* estão disponíveis na Tabela 9.

Tabela 9 – Especificações da *GPU NVIDIA GeForce MX130*

Tecnical Information	Value
Microarquitetura	*Maxwell*
NVIDIA Cuda Cores	384
Memória	2GB *GDDR5*
Velocidade de Memória	4000 MHz
Clock do *Core*	1122-1242 (*Boost*) MHz
Barramento de Memória	64 bit

4.3 Operação do Experimento

4.3.1 Preparação

A preparação do experimento teve como primeira tarefa a seleção de um conjunto de dados, o *MNIST dataset*, e a seleção do modelo de *CNN*. A escolha do *dataset* se deu por conta da popularidade do *MNIST Dataset* (como mostrado na Subseção 3.2.2), pela sua disponibilidade, pelo seu tamanho e por ser um *dataset* muito utilizado em experimentos de desenvolvimento de *CNNs* (LECUN; CORTES, 2010).

A segunda tarefa foi a seleção do modelo de *CNN* que seria utilizada. A *LeNet-5* foi utilizada pois é um dos modelos clássicos mais indicados para treinar o *dataset MNIST* como mostrado em LeCun e Cortes (2010), Tsang (2018) e Lecun et al. (1998). A *CNN LeNet-5* e o *MNIST dataset* são utilizados em todos os experimentos dessa dissertação.

A terceira tarefa foi a seleção dos ambientes. O ambiente *Google Colab* foi escolhido para executar os códigos com a biblioteca *TensorFlow* por ser gratuito, ter uma boa disponibilidade, praticidade (todo o *software* utilizado já estava pré-instalado), e por fornecer a possibilidade de aceleração por *GPU*. A princípio, o *Google Colab* também seria utilizado como ambiente de execução dos códigos com a biblioteca *CNTK*, porém o ambiente não permite utilizar a *API Keras* com a biblioteca *CNTK* como *back-end*, por isso foi selecionado o *PC* com a *GPU NVIDIA Geforce MX130* e o sistema operacional *Ubuntu 18.04 LTS*.

Foi modificado um código com o modelo *LeNet-5*, disponibilizado por Gazar (2018), que utiliza os métodos nativos da *TensorFlow* para codificação da *CNN* e um outro código, também disponibilizado por Gazar (2018), que utiliza os métodos da *API Keras* para codificação da *LeNet-5*; este último, utilizado para ser executado com as bibliotecas *TensorFlow 1.15* e *CNTK* como *back-end*. Esses dois códigos foram migrados, posteriormente, para o padrão da *TensorFlow 2* através do processo descrito em Google LLC (2020d) criando mais dois códigos como ilustrado na Figura 19. Os parâmetros utilizados foram Número de Épocas = 10 e *Batch Size* = 100, esses dois parâmetros influenciam no tempo de execução, por isso a padronização é necessária.

Por último foi implementado um código com o modelo *LeNet-5* através dos métodos nativos da biblioteca *CNTK*, os parâmetros foram configurados com os mesmos valores dos demais códigos.

4.3.2 Execução

As etapas da execução seguiram:

1. Execução do código preparado para utilização da biblioteca *TensorFlow 1.15* **sem** o uso do *Keras* que mede 100 vezes o tempo de execução do treinamento e 100 vezes o tempo de execução do teste;

2. Execução do código preparado para utilização da biblioteca *TensorFlow 1.15* **com** o uso do *Keras* que mede 100 vezes o tempo de execução do treinamento e 100 vezes o tempo de execução do teste;

3. Repete as etapas anteriores utilizando os códigos preparado para usar a biblioteca *TensorFlow 2.2* e depois com o códigos preparados para usar a biblioteca *CNTK 2.7*;

Observações sobre a execução:

- Ao armazenar os dados de tempo de execução, o primeiro valor foi ignorado a fim de evitar *outliers* ocasionados pela primeira chamada das funções das *bibliotecas*.

4.3.3 Coleta de Dados

Para coletar os dados de tempo foi utilizado a função *time.time()* da biblioteca *time* da linguagem *Python*. Para medir o tempo de execução do treinamento, o tempo começa a ser contado antes da chamada do procedimento de treinamento, a contagem é finalizada logo após a finalização do procedimento. Para medir o tempo de execução da inferência, a biblioteca *time.time()* foi aplicada da mesma forma, porém sobre o procedimento de teste.

Durante a execução foram coletadas as seguintes amostras dependentes:

- 100 amostras de tempo de execução do treinamento utilizando a biblioteca *TensorFlow 1.15* como *back-end*.
- 100 amostras de tempo de execução do teste utilizando a biblioteca *TensorFlow 1.15* como *back-end*.
- 100 amostras de tempo de execução do treinamento utilizando a biblioteca *TensorFlow 2.2* como *back-end*.
- 100 amostras de tempo de execução do teste utilizando a biblioteca *TensorFlow 2.2* como *back-end*.
- 100 amostras de tempo de execução do treinamento utilizando a biblioteca *CNTK 2.7* como *back-end*.
- 100 amostras de tempo de execução do teste utilizando a biblioteca *CNTK 2.7* como *back-end*.

4.3.4 Apresentação e Validação dos Dados

Os gráficos *boxplot* são plotados através do método *boxplot* da biblioteca *Seaborn*. Esse método possui o parâmetro *whis* configurado com o valor 1.5. Esse parâmetro define o limite inferior e o limite superior das hastes do *boxplot*, ou seja, os limites ficaram configurados da seguinte maneira:

$$LimiteInferior = PrimeiroQuartil - 1,5 * (TerceiroQuartil - PrimeiroQuartil)$$

$$LimiteSuperior = TerceiroQuartil + 1,5 * (TerceiroQuartil - PrimeiroQuartil)$$

Como forma de validar os dados e averiguar estatisticamente as hipóteses levantadas, o teste estatístico *Kolmogov-Smirnov* (*KS*) foi utilizado inicialmente para testar se as métricas adquiridas possuíam uma distribuição de probabilidade gaussiana (normal). A partir do resultado deste teste - que mostrou que todos os dados coletados não possuíam uma distribuição normal - foi utilizado o *Teste de Wilcoxon Pareado* para análise das hipóteses apresentadas.

O *Teste de Wilcoxon Pareado* foi selecionado para análise das hipóteses apresentadas pois ele é um teste não-paramétrico utilizado para comparar se as medidas de posição de duas amostras são iguais no caso em que as amostras são dependentes.

A seção subsequente apresenta, portanto, o resultado do teste *KS* e os resultados obtidos pelo *Teste de Wilcoxon Pareado* tendo em pauta as hipóteses levantadas.

4.4 Resultados

Como mencionado ao fim da seção anterior, o teste estatístico *Kolmogorov-Smirnov* foi utilizado para verificação da normalidade dos dados, para tanto, um nível de confiança de 95% foi aplicado. A partir do resultados identificou-se que para todas as variáveis dependentes, a distribuição de probabilidade não é normal, pois os *p-values* retornados (medida que indica a probabilidade do conjunto avaliado seguir a distribuição normal), foram próximos a zero (menor que 10^{-10}).

Como também mencionado anteriormente, o teste utilizado para análise das hipóteses foi o *Teste de Wilcoxon Pareado* com um nível de confiança de 95%, analisaremos separadamente os seus resultados na Subseção 4.4.1 e na Subseção 4.4.2.

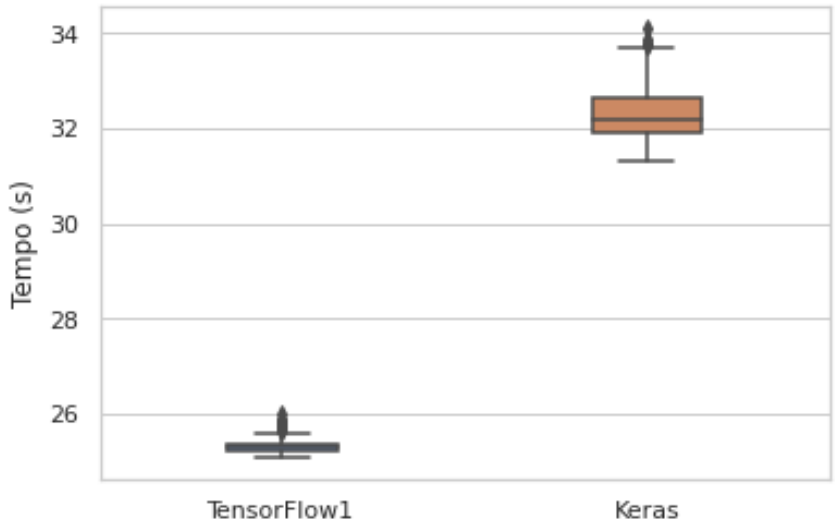
4.4.1 Análise e Interpretação *TensorFlow 1.15*

Nessa subseção são analisados os dados de execução dos códigos que utilizam a biblioteca *TensorFlow 1.15* como *back-end*.

4.4.1.1 Tempo de execução da fase de treinamento (Hipótese 1)

O resultado do *Teste de Wilcoxon Pareado* para o tempo de execução de treinamento utilizando GPU retornou um *p-value* de $3.896559845095909x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 20, que possui gráfico do tempo em segundos para cada um dos códigos, mostra com clareza um menor tempo de execução da fase de treinamento para o código que **não utiliza** a *API Keras*.

Figura 20 – Tempo de execução do treinamento (s) com a biblioteca TensorFlow 1.15

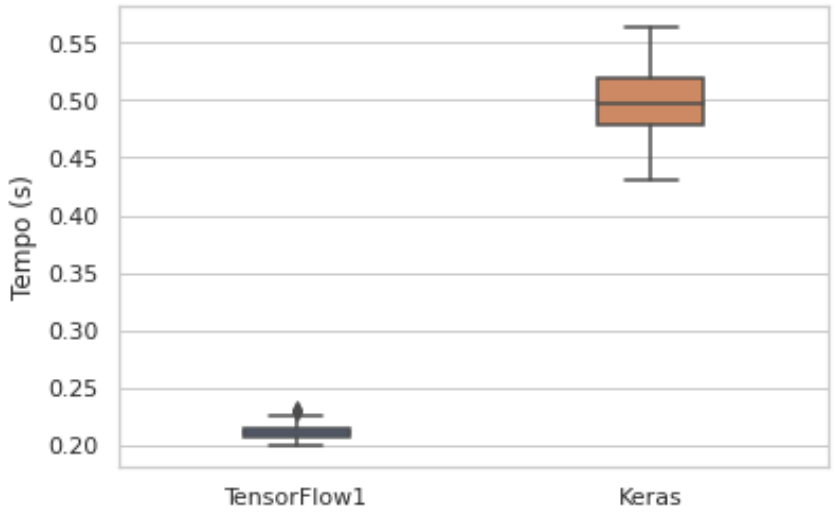


Fonte: Autoria Própria

4.4.1.2 Tempo de execução da fase de teste (Hipótese 2)

O resultado do *Teste de Wilcoxon Pareado* para o tempo de execução de teste retornou um *p-value* de $3.896559845095909x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 21, que possui gráfico do tempo em segundos para cada um dos códigos, mostra com clareza um menor tempo de execução da fase de treinamento para o código que **não utiliza** a *API Keras*.

Figura 21 – Tempo de execução de teste (s) com a biblioteca TensorFlow 1.15



Fonte: Autoria Própria

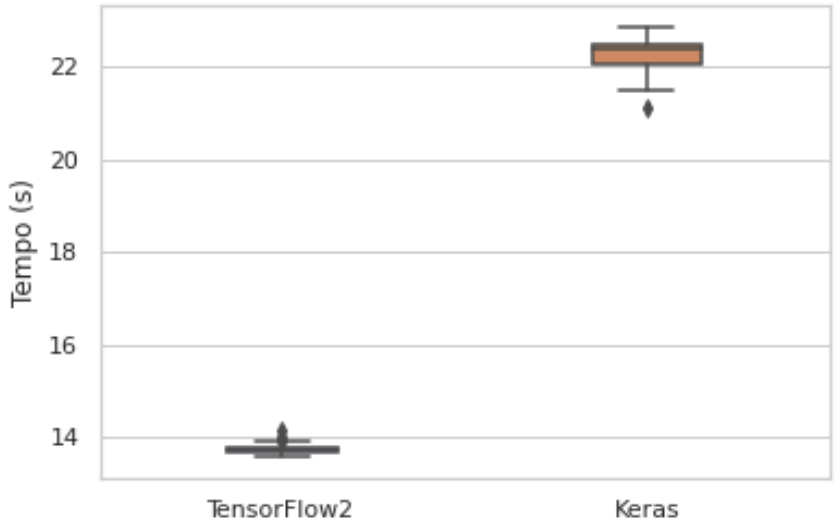
4.4.2 Análise e Interpretação *TensorFlow 2.2*

Nessa subseção são analisados os dados de execução dos códigos que utilizam a biblioteca *TensorFlow 2.2* como *back-end*.

4.4.2.1 Tempo de execução da frase de treinamento (Hipótese 1)

O resultado do *Teste de Wilcoxon Pareado* para o tempo de execução de treinamento retornou um *p-value* de $3.896559845095909x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 22, que possui gráfico do tempo em segundos para cada um dos códigos, mostra com clareza um menor tempo de execução da fase de treinamento para o código que **não utiliza** a *API Keras*.

Figura 22 – Tempo de execução do treinamento (s) com a biblioteca TensorFlow 2.2

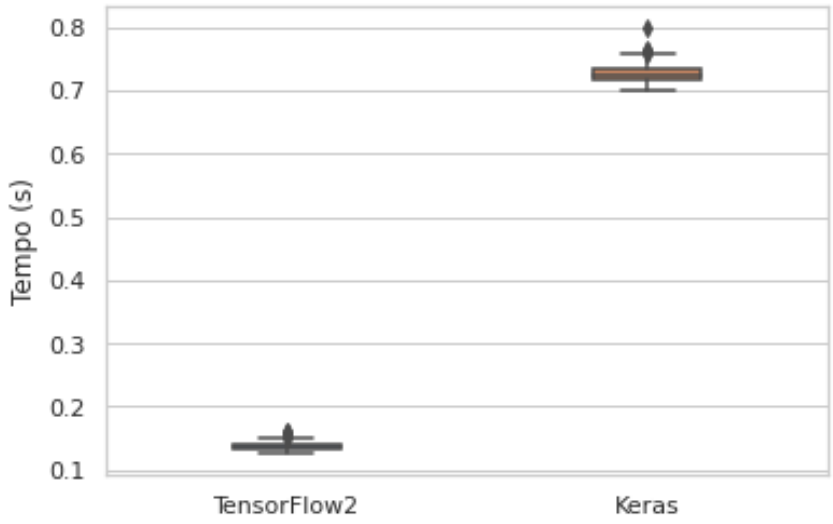


Fonte: Autoria Própria

4.4.2.2 Tempo de execução da fase de teste (Hipótese 2)

O resultado do *Teste de Wilcoxon Pareado* para o tempo de execução de teste retornou um *p-value* de $3.896559845095909x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 23, que possui gráfico do tempo em segundos para cada um dos códigos, mostra com clareza um menor tempo de execução da fase de treinamento para o código que **não utiliza** a *API Keras*.

Figura 23 – Tempo de execução de teste (s) com a biblioteca *TensorFlow 2.2*



Fonte: Autoria Própria

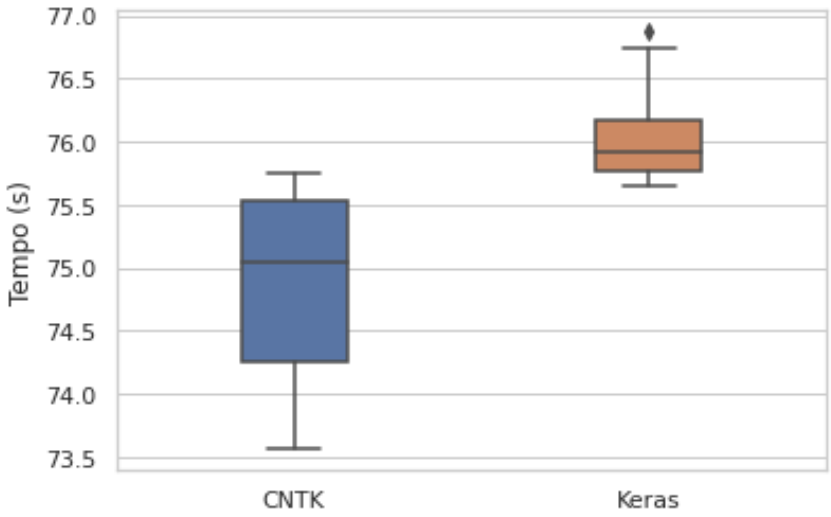
4.4.3 Análise e Interpretação *CNTK 2.7*

Nessa subseção são analisados os dados de execução dos códigos que utilizam a biblioteca *CNTK 2.7* como *back-end*.

4.4.3.1 Tempo de execução da fase de treinamento (Hipótese 1)

O resultado do *Teste de Wilcoxon Pareado* para o tempo de execução de treinamento retornou um *p-value* de $4.813701823147399x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 24, que possui gráfico do tempo em segundos para cada um dos códigos, mostra com clareza um menor tempo de execução da fase de treinamento para o código que **não utiliza** a *API Keras*.

Figura 24 – Tempo de execução do treinamento (s) com a biblioteca *CNTK 2.7*

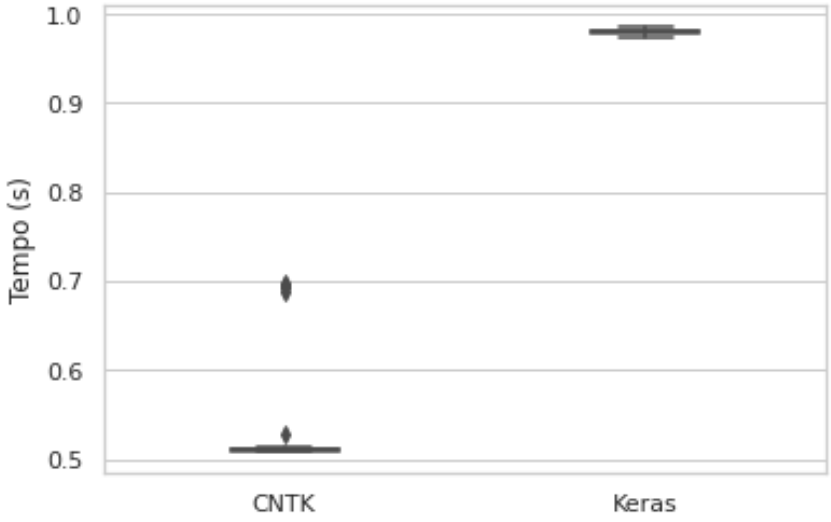


Fonte: Autoria Própria

4.4.3.2 Tempo de execução da fase de teste (Hipótese 2)

O resultado do *Teste de Wilcoxon Pareado* para o tempo de execução de teste retornou um *p-value* de $3.896559845095909x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 25, que possui gráfico do tempo em segundos para cada um dos códigos, mostra com clareza um menor tempo de execução da fase de treinamento para o código que **não utiliza** a *API Keras*.

Figura 25 – Tempo de execução de teste (s) com a biblioteca *CNTK 2.7*



Fonte: Autoria Própria

4.4.4 Ameaças à Validade

Foram utilizados testes estatísticos como forma de mitigar vieses relacionados à conclusão a respeito das hipóteses estabelecidas (**validade de conclusão**). O primeiro foi o teste de *Kolmogorov-Smirnov*, utilizado para verificação da normalidade dos dados. Esta etapa foi necessária para a escolha do teste seguinte, relacionado à comparação dos algoritmos a serem utilizados, que, sendo a hipótese nula rejeitada pelo teste *KS*, utilizou-se um teste não-paramétrico para amostras dependentes, o *Teste de Wilcoxon Pareado*. Desta forma, pode-se ter uma conclusão estatisticamente satisfatória, evitando a avaliação de desempenho apenas pela média dos dados amostrados.

Quanto a **validade interna**, o ambiente não foi totalmente monitorado e controlado, pois o *Google Colab* é um serviço em nuvem mantido e controlado pela *Google Colab*. Consequentemente, não houve um monitoramento de todos os processos executados pelo sistema operacional durante a execução dos códigos, alguns processos podem ter influenciado ocasionalmente na execução dos códigos e causado *outliers*, isso foi mitigado para os tempos de execução eliminando a coleta da primeira amostra. A possibilidade de sobrecargas nos servidores da *Google LLC* também é considerada uma ameaça à validade, pois pode interferir no tempo de execução dos códigos executados. Para o outro ambiente de execução (PC utilizado para a execução dos códigos com a biblioteca *CNTK*), todos os *softwares* não essenciais para o funcionamento do sistema operacional - incluindo a interface gráfica - foram desativados para reduzir a interferência no processamento dos algoritmos.

Apenas uma arquitetura de *CNN* foi utilizada, a *CNN LeNet-5*, o que faz com que o experimento tenha uma baixa variabilidade de *benchmarks* constituindo uma ameaça à **validade externa**. Outra ameaça à validade externa é que apenas um modelo de *GPU* foi utilizado para processamento dos códigos, uma maior variedade de modelos de *GPU* com diferentes microarquiteturas poderia mitigar os vieses.

Os códigos utilizados podem ter erros ocasionados por falha humana constituindo uma ameaça à **validade de construção**. Para mitigar essa ameaça, os códigos utilizados foram construídos a partir de modificações de códigos já utilizados pela comunidade de desenvolvedores e cientistas e construídos implementando uma *CNN* clássica (*LeNet-5*) treinando e testando o *MNIST dataset*.

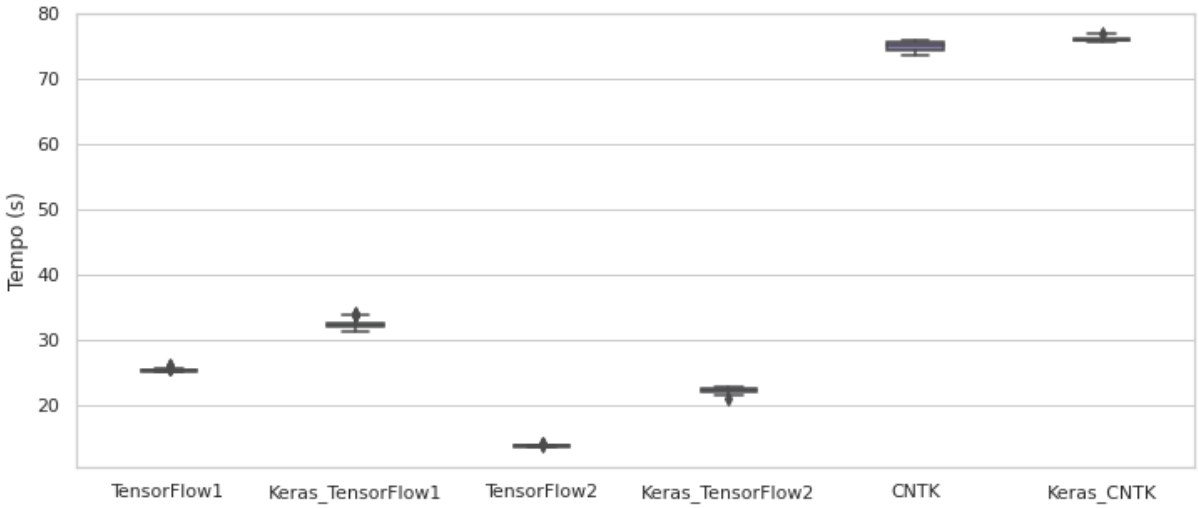
4.5 Considerações Finais do Capítulo

Através dos resultados do experimento, é possível concluir que os modelos implementados com o auxilio da *API Keras* apresentam um desempenho inferior aos códigos implementados sem o auxílio da *API*. Observando as Figuras 26 e 27 [5], essas conclusões podem ser tiradas

[5] Os gráficos das Figuras 26 e 27 são gráficos gerais que ilustram diferentes cenários em uma mesma imagem, os gráficos foram plotados para facilitar a visualização geral dos experimentos e não devem ser utilizados para uma

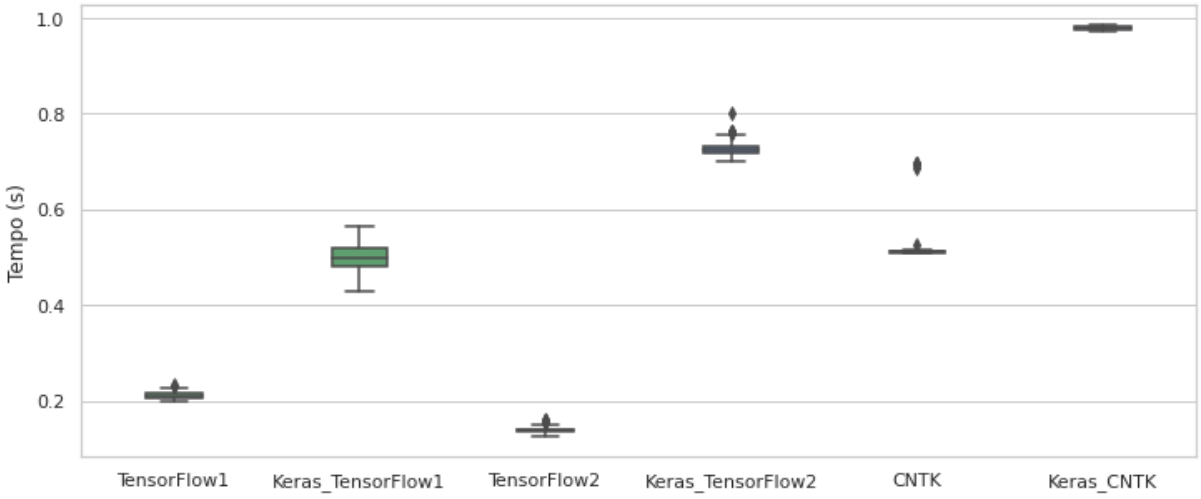
tanto para as duas versões da biblioteca *TensorFlow*, quanto para a biblioteca *CNTK*. Ou seja, os resultados refutam a afirmação presente em Google LLC (2020c), no qual o autor afirma que a *API Keras* facilita o uso do *TensorFlow* sem sacrificar a flexibilidade e o desempenho.

Figura 26 – Tempo de execução de treinamento (s) - Gráfico Geral



Fonte: Autoria Própria

Figura 27 – Tempo de execução de teste (s) - Gráfico Geral



Fonte: Autoria Própria

Nesse experimento não são investigadas as possíveis causas da diferença de desempenho, porém algumas hipóteses podem ser levantadas. É de conhecimento da ciência da computação que ao acrescentar uma *API* de alto nível a uma pilha de *software*, a execução pode ser consideravelmente afetada, esse pode ser o motivo pelo qual o desempenho dos códigos implementados com o auxilio da *API Keras* apresentou o desempenho inferior na maioria dos casos. No entanto,

comparação direta de valores.

a remodelagem de qualquer componente da pilha de *software* (incluindo o interpretador) pode impactar positivamente ou negativamente o desempenho de códigos implementados com as bibliotecas e a *API Keras*. Para levantar outras teses é necessário um experimento que meça outras variáveis como o nível de utilização da *GPU* e da *CPU* em cada caso.

Destaca-se que a *API Keras* e as bibliotecas *TensorFlow* e *CNTK* são ferramentas de alto nível e estão em contínuo desenvolvimento. Portanto, atualizações futuras de qualquer ferramenta de *software* utilizada nesse estudo podem alterar consideravelmente o desempenho das bibliotecas e da *API Keras*.

Uma vez que os resultados deste estudo mostraram que implementações com a *API Keras* impactam negativamente no desempenho das bibliotecas, os códigos utilizados no estudo comparativo apresentado nos próximos capítulos foram implementados sem as funcionalidades de alto nível da *API Keras*. O Experimento (*Maxwell*) foi realizado no *PC* enquanto os outros experimentos foram realizados.

5

Metodologia Experimental

Nesse capítulo é descrita a metodologia experimental dos quatro experimentos realizados: Análise de Desempenho na Microarquitetura *Maxwell* (Experimento 1), Análise de Desempenho na Microarquitetura *Kepler* (Experimento 2), *Pascal* (Experimento 3) e Análise de Desempenho na Microarquitetura *Turing* (Experimento 4).

O Experimento 1 foi realizado em um *Desktop* e os demais experimentos foram executados no *Google Colab* (GOOGLE LLC, 2020b). Em todos os experimentos houveram restrições que resultaram em uma seleção diferente de variáveis dependentes. As duas versões da biblioteca *TensorFlow* utilizadas ao longo da dissertação apresentaram incompatibilidade com a *GPU NVIDIA MX 130* presente no *notebook* utilizado, portanto durante o Experimento 1 só foram avaliadas as bibliotecas *CNTK* e *PyTorch*.

O ambiente *Google Colab* possui uma interface baseada na interface *Jupyter* e não permite a execução simultânea de mais de um terminal, ou seja, não foi possível analisar a utilização do *hardware* ao mesmo tempo em que se executavam as bibliotecas. Outro problema do *Google Colab* é a falta de informação detalhada sobre os processos executados: ao executar o *software NVIDIA System Management Interface* (*nvidia-smi*), o *software* não detalha os processos executados na *GPU* e nem a sua taxa de utilização; portanto, foram medidas a temperatura e a potência momentânea da *GPU* como forma de verificar a sua utilização. Ao executar o comando *top* [1], o sistema apresenta uma imprecisão na impressão de valor da taxa de utilização do processo *PyThon*, o gerenciador sempre imprime na tela apenas um dos dois valores: 0% ou 6,7%; portanto, foi medido a taxa de utilização da *CPU* considerando todos os processos. Por último, o ambiente *Google Colab* não permite a instalação do *software Sensors* utilizado no Experimento 1 para medir a temperatura da *CPU*.

A escolha das bibliotecas avaliadas e dos *benchmarks* foi feita depois dos resultados do mapeamento sistemático apresentado no Capítulo 3. O mapeamento mostrou que o *dataset* mais

[1] *top* é um comando nativo dos sitemas *Linux* para executar o gerenciador de tarefas.

utilizado é o *CIFAR-10* seguido do *MNIST*, como o *MNIST* é um *dataset* menor e, consequentemente, exige um menor tempo de execução para ser treinado e testado. A *CNN* selecionada foi a *LeNet-5*, por ser uma arquitetura clássica muito usada para treinar o *dataset MNIST*.

Quanto as bibliotecas, inicialmente seriam selecionadas as três bibliotecas mais utilizadas nos estudos de avaliação de desempenho mais a biblioteca *PyTorch* que apresentou um bom desempenho no estudo preliminar (FLORENCIO et al., 2019). O mapeamento mostrou que as três bibliotecas mais utilizadas são a *TensorFlow*, *Caffe* e *CNTK*, porém a biblioteca *Caffe* tem uma funcionalidade distinta das outras, ela funciona como um modulo externo a linguagem e não como uma biblioteca *PyThon*, por isso a biblioteca *Caffe* não foi avaliada junto com as outras.

Ao lançar a biblioteca *TensorFlow 2*, a *Google* manteve o suporte para a biblioteca *TensorFlow 1*. Cada uma das versões da biblioteca *TensorFlow* possuem funcionalidades e métodos diferentes, por isso ambas as versões foram avaliadas neste estudo.

5.1 Definição e Planejamento

Nesta Seção, seguindo a metodologia descrita em Wohlin et al. (2012), é apresentado o planejamento do experimento controlado proposto. As subseções apresentadas a seguir apresentam o objetivo e o planejamento do experimento (contexto, variáveis dependentes e independentes, hipóteses, objetos de análise, projeto do experimento e instrumentação).

5.1.1 Definição de Objetivo

O objetivo deste experimento é avaliar (segundo tempo de execução, temperatura e escalabilidade) as bibliotecas *TensorFlow 1*, *TensorFlow 2 PyTorch* e *CNTK*, verificando seus comportamentos no aprendizado e inferência de redes neurais convolucionais.

Para comparar o desempenho das bibliotecas nas diferentes microarquiteturas de *GPU*, foram realizados quatro experimentos *In Vitro* - realizados em ambiente fechado com condições controladas.

Seguindo a formalização de definição de objetivo do modelo GQM (*Goal Question Metric*), proposto por Basili, Caldiera e Rombach (1994), o objetivo pode ser reescrito como: **Analisar** as bibliotecas de Redes Neurais Convolucionais *TensorFlow* (nas duas versões suportadas), *PyTorch* e *CNTK* **com a finalidade de** compará-los **com relação a** tempo de execução e utilização do hardware **do ponto de vista** de pesquisadores, cientistas e desenvolvedores que trabalham com *deep learning* **no contexto de** redes neurais convolucionais *LeNet-5* executando em ambientes computacionais heterogêneos contendo *CPU* e *GPUs* com diferentes microarquiteturas (*Kepler*, *Maxwell*, *Pascal* e *Turing*).

5.1.2 Planejamento

Seleção de Contexto: Os quatro experimentos foram *In Vitro*. Em cada experimento, para cada uma das bibliotecas foram executados os algoritmos de treinamento (aprendizado) e de teste (conjunto de inferências) de uma *CNN LeNet-5* processando o conjunto de dados *MNIST*, o que permitiu verificar o comportamento das bibliotecas em sistemas híbridos com *GPUs* que utilizam *CUDA* e *CPUs*. No primeiro experimento foi utilizado o processamento do *notebook Acer Aspire 5 A515-51G-C97B*. Nos demais experimentos foi utilizado o ambiente *Google Colab* (GOOGLE LLC, 2020b).

Variáveis Dependentes - Experimento 1: Tempo de execução da fase de treinamento (s), Tempo de execução da fase de teste (s), temperatura da *CPU* (°C) durante a fase de treinamento, temperatura da *CPU* (°C) durante a fase de teste, temperatura da *GPU* (°C) durante a fase de treinamento, temperatura da *GPU* (°C) durante a fase de teste, taxa de utilização da *CPU* (%) durante a fase de treinamento,taxa de utilização da *CPU* (%) durante a fase de teste, taxa de utilização da *GPU* (%) durante a fase de treinamento, taxa de utilização da *GPU* (%) durante a fase de teste, taxa de utilização da memória principal (%) durante a fase de treinamento, taxa de utilização da memória principal (%) durante a fase de teste, taxa de utilização da memória da *GPU* durante a fase de treinamento e taxa de utilização da memória da *GPU* durante a fase de teste [2].

Várias Dependentes - Experimentos 2, 3 e 4: Tempo de execução da fase de treinamento (s), tempo de execução da fase de teste (s), temperatura da *GPU* (°C) durante a fase de treinamento, temperatura da *GPU* (°C) durante a fase de treinamento, temperatura da *GPU* (°C) durante a fase de teste, potência utilizada pela *GPU* (W) durante a fase de treinamento, potência utilizada pela *GPU* (W) durante a fase de teste, taxa de utilização da *CPU* durante a fase de treinamento (%), taxa de utilização da *CPU* durante a fase de teste (%), taxa de utilização da memória principal durante a fase de treinamento (%), taxa de utilização da memória principal durante a fase de teste (%), taxa de utilização da memória da *GPU* durante a fase de treinamento (%) e taxa de utilização da memória da *GPU* durante a fase de teste(%).

Variáveis Independentes: Compilação do código, capacidade de paralelização das bibliotecas, complexidade dos algoritmos contidos nas bibliotecas, propriedades termodinâmicas dos materiais do *hardware*, temperatura ambiente, sistema de refrigeração do *hardware* e ambiente de execução (*hardware* e *software*).

Formulação de Hipótese - Experimento 1: As questões de pesquisa para este experimento são: As execuções dos diferentes códigos apresentam tempo de execução semelhantes? As execuções dos diferentes códigos geram temperaturas semelhantes da *GPU*? As execuções

[2] No Experimento 1 foi possível verificar a influência de cada tarefa individualmente na taxa de utilização do *hardware*, portanto as taxas de utilização coletadas são somente da execução do ambiente *Python* ignorando automaticamente todos os outros processos executados pelo sistema operacional. O mesmo não pode ser feito nos demais experimentos.

dos diferentes códigos geram temperaturas semelhantes da *CPU*? As execuções dos diferentes códigos geram taxas semelhantes de utilização da *GPU*? As execuções dos diferentes códigos geram taxas semelhantes de utilização na *CPU*? As execuções dos diferentes códigos apresentam a mesma taxa de utilização da memória principal? As execuções dos diferentes códigos apresentam a mesma taxa de utilização da memória da *GPU*? Cada uma dessas questões deve ser respondida tanto para a fase de treinamento da *CNN*, quanto para a fase de teste.

Todos os questionamentos são respondidos a partir de dados extraídos da execução de uma mesma *CNN*. Para a primeira questão pode-se considerar o tempo de treinamento da rede (μ^{lrn}) e o tempo da fase de teste (μ^{test}), para a segunda questão pode-se usar a temperatura da *GPU* durante a fase de treinamento (τ^{lrn}) e durante a fase de teste (τ^{test}), para a terceira questão pode-se usar a temperatura da *CPU* durante a fase de treinamento (ψ^{lrn}) e durante a fase de teste (ψ^{test}), para a quarta questão pode-se utilizar a taxa de utilização da *GPU* durante a fase de treinamento (ω^{lrn}) e durante a fase de teste (ω^{test}), para a quinta questão, pode-se utilizar a média da taxa de utilização da *CPU* durante a fase de treinamento (ϕ^{lrn}) e durante a fase de teste (ϕ^{test}), para a sexta questão pode-se utilizar a taxa de utilização da memória principal durante a fase de treinamento (ϑ^{lrn}) e durante a fase de teste (ϑ^{test}) e, para a ultima questão, pode-se utilizar a taxa de utilização da memória da *GPU* durante a fase de treinamento (κ^{lrn}) e durante a fase de teste (κ^{test}). Cada uma das medidas deve ser feita para cada uma das bibliotecas: *CNTK* e *PyTorch*. Neste contexto, as seguintes hipóteses podem ser verificadas (a saber, Ck = *CNTK* e PT = *PyTorch*):

Hipótese 1 (Para as fases de treinamento e de teste)

H0: Não há diferença estatística entre os tempos de execução dos códigos executados, $(\mu^i_{CK} = \mu^i_{PT})$

H1: Há diferença entre os tempos de execução dos códigos executados, ou seja, $(\mu^i_{CK} \neq \mu^i_{PT})$

Hipótese 2 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as temperaturas da *GPU* medidas durante a execução de cada código, ou seja, $(\tau^i_{CK} = \tau^i_{PT})$

H1: Há diferença estatística entre as temperaturas da *GPU* medidas durante a execução de cada código, ou seja, $(\tau^i_{CK} \neq \tau^i_{PT})$

Hipótese 3 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as temperaturas da *CPU* medidas durante a execução de cada código, ou seja, $(\psi^i_{CK} = \psi^i_{PT})$

H1: Há diferença estatística entre as temperaturas da *CPU* medidas durante a execução de cada código, ou seja, $(\psi^i_{CK} \neq \psi^i_{PT})$

Hipótese 4 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as taxas de utilização da *GPU* medidas durante a execução de cada código, ou seja, $(\omega^i_{CK} = \omega^i_{PT})$

H1: Há há diferença estatística entre as taxas de utilização da *GPU* medidas durante a execução de cada código, ou seja, $(\omega^i_{CK} \neq \omega^i_{PT})$

Hipótese 5 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as taxas de utilização da *CPU* medidas durante a execução de cada código, ou seja, $(\phi^i_{CK} = \phi^i_{PT})$

H1: Há diferença estatística entre as taxas de utilização da *CPU* medidas durante as execução de cada código, ou seja, $(\phi^i_{CK} \neq \phi^i_{PT})$

Hipótese 6 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as taxas de utilização da memória principal medidas durante a execução de cada código, ou seja, $(\vartheta^i_{CK} = \vartheta^i_{PT})$

H1: Há diferença estatística entre as taxas de utilização da memória principal medidas durante a execução de cada código, ou seja, $(\vartheta^i_{CK} \neq \vartheta^i_{PT})$

Hipótese 7 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as taxas de utilização da memória *GPU* medidas durante a execução de cada código, ou seja, $(\kappa^i_{CK} = \kappa^i_{PT})$

H1: Há diferença estatística entre as taxas de utilização da memória da *GPU* medidas durante a execução de cada código, ou seja, $(\kappa^i_{CK} \neq \kappa^i_{PT})$

Formulação de Hipótese - Experimentos 2, 3 e 4: As questões de pesquisa para este experimento são: As execuções dos diferentes códigos apresentam tempo de execução semelhantes? As execuções dos diferentes códigos geram temperaturas semelhantes da *GPU*? As execuções dos diferentes códigos apresentam taxas semelhantes de consumo de energia da *GPU*? As execuções dos diferentes códigos geram taxas semelhante de utilização na *CPU*? As execuções dos diferentes códigos geram taxas semelhantes de utilização da memória principal? As execuções dos diferentes códigos geram taxa semelhantes de utilização da memória da *GPU*? Cada uma dessas questões devem ser respondidas tanto para a fase de treinamento da *CNN*, quanto para a fase de teste.

Todos os questionamentos são respondidos a partir de dados extraídos da execução de uma mesma *CNN*. Para a primeira questão pode-se considerar o tempo de treinamento da rede (μ^{lrn}) e o tempo da fase de teste (μ^{test}), para a segunda questão pode-se usar a temperatura da *GPU* durante a fase de treinamento (τ^{lrn}) e durante a fase de teste (τ^{test}), para a terceira questão pode-se usar a potência da *GPU* durante a fase de treinamento (ν^{lrn}) e durante a fase de teste (ν^{test}), para a quarta questão, pode-se utilizar a a taxa de utilização da *CPU* durante a fase de treinamento (ϕ^{lrn}) e durante a fase de teste (ϕ^{test}), para a quinta questão pode-se utilizar a taxa

de utilização da memória principal durante a fase de treinamento (ϑ^{lrn}) e durante a fase de teste (ϑ^{test}) e, para a ultima questão, pode-se utilizar a taxa de utilização da memória da *GPU* durante a fase de treinamento (κ^{lrn}) e durante a fase de teste (κ^{test}). Cada uma das medidas deve ser feita para cada uma das bibliotecas: *CNTK*, *PyTorch*, *TensorFlow 1* e *TensorFlow 2*. Neste contexto, as seguintes hipóteses podem ser verificadas (a saber, Ck = *CNTK*, PT = *PyTorch*, TF1 = *TensorFlow 1* e TF2 = *TensorFlow2*):

Hipótese 1 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre os tempos de execução dos códigos executados, $(\mu^i_{CK} = \mu^i_{PT} = \mu^i_{TF1} = \mu^i_{TF2})$

H1: Há diferença entre os tempos de execução dos códigos executados, ou seja, $(\mu^i_{CK} \neq \mu^i_{PT} \neq \mu^i_{TF1} \neq \mu^i_{TF2})$

Hipótese 2 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as temperaturas da *GPU* medidas durante a execução de cada código, ou seja, $(\tau^i_{CK} = \tau^i_{PT} = \tau^i_{TF1} = \tau^i_{TF2})$

H1: Há diferença estatística entre as temperaturas da *GPU* medidas durante a execução de cada código, ou seja, $(\tau^i_{CK} \neq \tau^i_{PT} \neq \tau^i_{TF1} \neq \tau^i_{TF2})$

Hipótese 3 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as potências da *GPU* medidas durante a execução de cada código, ou seja, $(\nu^i_{CK} = \nu^i_{PT} = \nu^i_{TF1} = \nu^i_{TF2})$

H1: Há diferença estatística entre as potências da *GPU* medidas durante a execução de cada código, ou seja, $(\nu^i_{CK} \neq \nu^i_{PT} \neq \nu^i_{TF1} \neq \nu^i_{TF2})$

Hipótese 4 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as taxas de utilização da *CPU* medidas durante a execução de cada código, ou seja, $(\phi^i_{CK} = \phi^i_{PT} = \phi^i_{TF1} = \phi^i_{TF2})$

H1: Há diferença estatística entre as taxas de utilização da *CPU* medidas durante as execução de cada código, ou seja, $(\phi^i_{CK} \neq \phi^i_{PT} \neq \phi^i_{TF1} = \phi^i_{TF2})$

Hipótese 5 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as taxas de utilização da memória principal medidas durante a execução de cada código, ou seja, $(\vartheta^i_{CK} = \vartheta^i_{PT} = \vartheta^i_{TF1} = \vartheta^i_{TF2})$

H1: Há diferença estatística entre as taxas de utilização da memória principal medidas durante a execução de cada código, ou seja, $(\vartheta^i_{CK} \neq \vartheta^i_{PT} \neq \vartheta^i_{TF1} \neq \vartheta^i_{TF2})$

Hipótese 6 (Para as fases treinamento e de teste)

H0: Não há diferença estatística entre as taxas de utilização da memória *GPU* medidas durante a execução de cada código, ou seja, $(\kappa^i_{CK} = \kappa^i_{PT} = \kappa^i_{TF1} = \kappa^i_{TF2})$

H1: Há diferença estatística entre as taxas de utilização da memória da *GPU* medidas durante a execução de cada código, ou seja, $(\kappa^i_{CK} \neq \kappa^i_{PT} \neq \kappa^i_{TF1} \neq \kappa^i_{TF2})$

Seleção de Objetos: O experimento utilizou um conjunto de dados de imagens, o *Modified National Institute of Standards and Technology database* (*MNIST database*) (LECUN; CORTES, 2010). O *MNIST* é um grande conjunto de dados de dígitos manuscritos comumente utilizado para treinar sistemas de sistemas de processamento de imagens e possui as seguintes características:

- Imagens de 28 x 28 *pixels*.
- 10 classes, uma classe para cada dígito.
- Subconjunto com 60000 dados de treinamento.
- Subconjunto com 10000 dados de teste.

Quanto aos códigos, foram utilizados cinco códigos com uma rede *CNN LeNet-5* (LECUN et al., 1998) implementada em *Python*. O primeiro código utiliza os métodos nativos da biblioteca *TensorFlow 1.15*. O segundo código foi implementado utilizando os métodos nativos do *TensorFlow 2.2* - desconsiderando os métodos da *API Keras* disponíveis nativamente na versão 2.2 da biblioteca. O terceiro código utiliza os métodos nativos da biblioteca *CNTK*. O último código foi implementado com os métodos nativos da biblioteca *PyTorch*.

Os dois primeiros códigos utilizados são modificações de códigos desenvolvidos por Gazar (2018), as modificações foram realizadas com a finalidade de deixá-los com os mesmos parâmetros e para criar uma versão de cada código que fosse compatível com o *TensorFlow 2*. O terceiro código foi implementado a partir da modificação dos tutoriais disponíveis no repositório oficial do *CNTK* no *GitHub* (MICROSOFT CORPORATION, 2019). E o quinto código foi implementado a partir da modificação de exemplos disponíveis no repositório oficial do *PyTorch* no *GitHub* (PYTORCH CORE TEAM, 2018). Os códigos utilizados durante o experimento estão disponíveis no repositório de Florencio (2020b).

Projeto dos Experimentos: o projeto de cada experimento pode ser resumido nas seguintes etapas:

1. Preparar o ambiente de execução;
2. Criação dos códigos para a biblioteca *TensorFlow 1.15* a partir de modificações dos códigos de Gazar (2018);
3. Criação dos códigos para a biblioteca *TensorFlow 2.2* a partir da migração dos códigos criados na etapa anterior. Essa etapa foi realizada seguindo as recomendações, de uso de ferramentas e de métodos, disponíveis em Google LLC (2020d);

4. Criação dos códigos para a biblioteca *CNTK* a partir de modificações dos códigos de Microsoft Corporation (2019);
5. Criação dos códigos para a biblioteca *PyTorch* a partir de modificações dos códigos de PyTorch Core Team (2018);
6. Implementar *scripts* para automatizar a execução da coleta de informações da *CPU*, *GPU* e memória (apenas para o Experimento 1);
7. Medir 100 vezes o tempo de execução de cada uma das fases (treinamento e teste);
8. Coletar 100 vezes as informações do gerenciamento de tarefas na *CPU* e memória principal durante ambas as fases (treinamento e teste) através do gerenciador de tarefas nativo do sistema operacional;
9. Coletar 100 vezes as informações do gerenciamento de tarefas na *GPU* durante ambas as fases (treinamento e teste) através do *software NVIDIA System Management Interface*;
10. Coletar 100 vezes a temperatura da *CPU* durante a execução de cada um das fases (treinamento e teste) através do *software Sensors* (apenas para o Experimento 1);
11. Gerar gráficos *boxplot* para apresentar os dados coletados;
12. Aplicar testes estatísticos para análise das hipóteses;

Instrumentação - Experimento 1: Os *softwares* utilizados foram o sistema operacional *Ubuntu 18.04 LTS*, *Virtualenv* versão 20.0.21, ambiente *Python* na versão 3.6, o *CUDA* na versão 10.2, o *Driver NVIDIA* versão 440.59, o *cuDNN* na versão 7.5, o *Open MPI* versão 2.1.1, *CNTK* versão 2.7. O *hardware* utilizado é um *notebook Acer Aspire A515-51G-C97B* com mémoria *RAM* 8GB (2x4GB) 2133MHz *DDR4*, processador *8th Generation Intel Core i5-8250U Quad Core (6MB Cache, up to 3.4GHz)*, disco rígido 1TB 5400rpm e *GPU NVIDIA GeForce MX130*. As especificações da *GPU* estão disponíveis na Tabela 10.

Tabela 10 – Especificações da *GPU NVIDIA GeForce MX130*

Informação Técnica	Valor
Microarquitetura	*Maxwell*
NVIDIA Cuda Cores	384
Memória	2GB *GDDR5*
Velocidade de Memória	4000 MHz
GPU Clock	1122-1242 (*Boost*) MHz
Barramento de Memória	64 bit

Instrumentação - Experimento 2: Os *softwares* utilizados foram o ambiente do *Google Colab Python* na versão 3.6, a biblioteca *TensorFlow* na versão 1.15 e na versão 2.2, a biblioteca

CNTK na versão 2.7, a biblioteca *PyTorch* na versão 1.5, o *CUDA* na versão 10.1, o *Driver NVIDIA* versão 418.67, o *Open MPI* na versão 2.1.1 e o *NVIDIA System Management Interface* versão 440.82. O conjunto de *hardware* possui uma *CPU Intel Xeon 2.30GHz* com apenas um *core* que conta com duas *threads*, possui uma memória *RAM* de 13 GB, um *HD* de 34 GB e e uma *GPU NVIDIA Tesla K80*. A *GPU NVIDIA Tesla K80* é uma placa que combina duas *GPUs NVIDIA GK210* projetadas para atuar conjuntamente, porém o sistema do *Google Colab* permite apenas o uso de uma das *GPU NVIDIA GK210*. Na Tabela 11 são ilustradas as especificações da *NVIDIA GK210*, essas informações foram retiradas do *datasheet* da *Tesla K80* (NVIDIA CORPORATION, 2014a), do *Board Specification* da *Tesla K80* (NVIDIA CORPORATION, 2015) e o *whitepaper* de apresentação das *NVIDIA GK210* (NVIDIA CORPORATION, 2014b).

Tabela 11 – Especificações da *GPU NVIDIA GK210*

Informação Técnica	Valor
Microarquitetura	*Kepler*
NVIDIA Cuda Cores	2496
Memória	12 GB *GDDR5*
Largura de Banda da Memória	240 GB/s
Memory Clock	1253 MHz
GPU Clock - Base	560 MHz
GPU Clock - Boost	824 MHz
Potência Máxima	149 W

Instrumentação - Experimento 3: Os *softwares* utilizados foram o ambiente do *Google Colab Python* na versão 3.6, a biblioteca *TensorFlow* na versão 1.15 e na versão 2.2, a biblioteca *CNTK* na versão 2.7, a biblioteca *PyTorch* na versão 1.5, o *CUDA* na versão 10.1, o *Driver NVIDIA* versão 418.67, o *Open MPI* na versão 2.1.1 e o *NVIDIA System Management Interface* (*NVIDIA SMI*) versão 440.82. O conjunto de *hardware* possui uma *CPU Intel Xeon 2.20GHz* com apenas um *core* que conta com duas *threads*, possui uma memória *RAM* de 13 GB, um *HD* de 34 GB e e uma *GPU NVIDIA Tesla P4*. As especificações da *GPU* estão na Tabela 12, essas informações foram retiradas do seu *datasheet* (NVIDIA CORPORATION, 2019c) e do seu *product brief* (NVIDIA CORPORATION, 2017b).

Instrumentação - Experimento 4: Os *softwares* utilizados foram o ambiente do *Google Colab Python* na versão 3.6, a biblioteca *TensorFlow* na versão 1.15 e na versão 2.2, a biblioteca *CNTK* na versão 2.7, a biblioteca *PyTorch* na versão 1.5, o *CUDA* na versão 10.1, o *Driver NVIDIA* versão 418.67, o *Open MPI* na versão 2.1.1 e o *NVIDIA System Management Interface* versão 440.82. O conjunto de *hardware* possui uma *CPU Intel Xeon 2.00GHz* com apenas um *core* que conta com duas *threads*, possui uma memória *RAM* de 13 GB, um *HD* de 32 GB e e uma *GPU NVIDIA Tesla T4*. As especificações da *GPU* estão na Tabela 13, essas informações foram retiradas do seu *datasheet* (NVIDIA CORPORATION, 2019b) e do seu *product brief* (NVIDIA CORPORATION, 2020a).

Tabela 12 – Especificações da *GPU NVIDIA Tesla P4*

Informação Técnica	Valor
Microarquitetura	*Pascal*
NVIDIA Cuda Cores	2560
Mémoria	16 GB *GDDR5*
Largura de Banda da Memória	192 GB/s
Single-Precision Performance	5,5 *TeraFLOPS*
Operações com Valores Inteiros	22 *TOPS* (*Tera-Operations* por segundo)
GPU Clock - Base	885 MHz
GPU Clock - Maximum Boost	1531 MHz (1131 MHz *Default*)
GPU Clock - Idle	405 MHz
Potência Máxima	250 W

Tabela 13 – Especificações da *GPU NVIDIA Tesla T4*

Informação Técnica	Valor
Microarquitetura	*Turing*
NVIDIA Turing Tensor Cores	320
NVIDIA Cuda Cores	2560
GPU Clock - Base	585 MHz
GPU Clock - Maximum Boost	1590 MHz
Potência Máxima	70 W
Single-Precision Performance	8,5 *TeraFLOPS*
Mixed-Precision Performance	65 *TeraFLOPS*
Operações *INT8*	130 *TOPS*
Operações *INT4*	260 *TOPS*
Mémoria	16 GB *GDDR6*
Largura de Banda da Memória	320 GB/s
Memory Clock - Máximo	5001 MHz

5.2 Operação do Experimento

5.2.1 Preparação

A preparação do experimento teve como primeira tarefa a seleção de um conjunto de dados, o *MNIST dataset*, e a seleção do modelo de *CNN*. A escolha do *dataset* se deu por conta da popularidade do *MNIST Dataset* (como mostrado na Subseção 3.2.2), pela sua disponibilidade, pelo seu tamanho e por ser um *dataset* muito utilizado em experimentos de desenvolvimento de *CNNs* (LECUN; CORTES, 2010).

A segunda tarefa foi a seleção do modelo de *CNN* que seria utilizada. A *LeNet-5* foi utilizada pois é um dos modelos clássicos mais indicados para treinar o *dataset MNIST* como mostrado em LeCun e Cortes (2010), Tsang (2018) e Lecun et al. (1998). A *CNN LeNet-5* e o *MNIST dataset* são utilizados em todos os experimentos dessa dissertação.

A terceira tarefa foi a seleção dos ambientes. O ambiente *Google Colab* foi escolhido para execução dos experimentos 2, 3 e 4 por ser gratuito, ter uma boa disponibilidade, praticidade (vários *softwares* utilizados já estavam pré-instalados), e por disponibilizar modelos de *GPU* com três microarquiteturas diferentes (*Kepler*, *Pascal* e *Turing*). A princípio, o *Google Colab* também seria utilizado como ambiente de execução para o Experimento 1, porém o ambiente não disponibiliza *GPUs* com microarquitetura *Maxwell*, por isso foi selecionado o *PC* com a *GPU NVIDIA Geforce MX130* e o sistema operacional *Ubuntu 18.04 LTS*.

A *CNN LeNet-5* foi implementada de quatro formas diferentes, cada uma utilizando cada uma biblioteca diferente. As *CNNs* foram implementadas a partir de modificações de códigos disponíveis em repositórios abertos. Os valores dos parâmetros utilizados na implementação dos algoritmos de treinamento são ilustrados na Tabela 14. Os parâmetros Tamanho do *Batch*, Número de *Steps* e Número de Épocas influenciam no tempo de execução, por isso a padronização é necessária.

Tabela 14 – Parâmetros da *CNN LeNet-5* Utilizada

Parâmetros	Valor
Tamanho do *Batch*	100
Número de Épocas	10
Número de *Steps*	600
Taxa de Aprendizagem	0.001

Foi criado um *script* para a automatização da coleta de dados, o *script* foi programado para coletar informações do gerenciador de tarefas do sistema *Linux* e do *software NVIDIA System Management Interface*. Esse *script* somente é usado no Experimento 1, pois o *Google Colab* não permite a execução de mais de um terminal simultaneamente.

5.2.2 Execução

No Experimento 1, o *script* criado para automatização da coleta de dados é executado de maneira simultânea a execução dos códigos. O código da *CNN* é executado em um terminal e o *script* em outro terminal.

As etapas da execução para o Experimento 1 seguiram:

1. Execução do código preparado para utilização da biblioteca *TensorFlow 1.15* que mede 100 vezes o tempo de execução do treinamento e 100 vezes o tempo de execução do teste;

2. Execução do código preparado para utilização da biblioteca *TensorFlow 1.15* em paralelo a execução do *script* que coleta 100 vezes as informações sobre a *CPU*, a memória principal, a *GPU* e a memória da *GPU* durante a fase de treinamento e a fase de teste;

3. Repete as etapas anteriores utilizando os códigos preparado para usar a biblioteca *TensorFlow 2.2*, depois com os códigos preparados para usar a biblioteca *PyTorch 1.5* e depois com o códigos preparados para usar a biblioteca *CNTK 2.7*;

As etapas da execução para os Experimentos 2, 3 e 4 seguiram:

1. Execução do código preparado para utilização da biblioteca *TensorFlow 1.15* que mede 100 vezes o tempo de execução do treinamento e 100 vezes o tempo de execução do teste;
2. Execução do código preparado para utilização da biblioteca *TensorFlow 1.15* que coleta 100 vezes as informações da utilização da *GPU* e de sua memória durante a fase de treinamento e durante a fase de teste;
3. Execução do código preparado para utilização da biblioteca *TensorFlow 1.15* que coleta 100 vezes as informações da utilização da *CPU* e da memória principal durante as duas fases (treinamento e teste);
4. Repete as etapas anteriores utilizando os códigos preparado para usar a biblioteca *TensorFlow 2.2*, depois com os códigos preparados para usar a biblioteca *PyTorch 1.5* e depois com o códigos preparados para usar a biblioteca *CNTK 2.7*;

Observações sobre a execução:

- Ao armazenar os dados de tempo de execução, o primeiro valor foi ignorado a fim de evitar *outliers* ocasionados pela primeira chamada dos métodos das *bibliotecas*.

5.2.3 Coleta de Dados

Para coletar os dados de tempo foi utilizado a função *time.time()* da biblioteca *time* da linguagem *Python*. Para medir o tempo de execução do aprendizado, o tempo começa a ser contado antes da chamada do procedimento de treinamento, a contagem é finalizada logo após a finalização do procedimento. Para medir o tempo de execução da inferência, a biblioteca *time.time()* foi aplicada da mesma forma, porém sobre o procedimento de classificação.

A coleta da temperatura e da taxa de utilização da CPU foram realizada usando o *Sensors* versão 3.4.0, já a coleta da temperatura e da taxa de utilização da *GPU* foram realizadas usando o *NVIDIA System Management Interface*. As medições foram feitas várias vezes durante a execução de cada procedimento (treinamento e teste), podendo ter algum dado coletado pouco tempo antes do inicio da execução da fase ou pouco tempo depois da finalização da fase.

Para cada código executado foram coletadas as seguintes amostras dependentes:

- 100 amostras de tempo de execução da fase de treinamento.

- 100 amostras de tempo de execução da fase de teste.
- 100 amostras de temperatura da *GPU* (°C) durante a fase de treinamento.
- 100 amostras de temperatura da *GPU* (°C) durante a fase de teste.
- 100 amostras de potência da *GPU* (W) durante a fase de treinamento (apenas para os Experimentos 2, 3 e 4).
- 100 amostras de potência da *GPU* (W) durante a fase de teste (apenas para os Experimentos 2, 3 e 4).
- 100 amostras de temperatura da *CPU* (°C) durante a fase de treinamento (apenas para o Experimento 1).
- 100 amostras de temperatura da *CPU* (°C) durante a fase de teste (apenas para o Experimento 1).
- 100 amostras de taxa de utilização da *GPU* (%) durante a fase de treinamento (apenas para o Experimento 1),
- 100 amostras de taxa de utilização da *GPU* (%) durante a fase de teste (apenas para o Experimento 1),
- 100 amostras de taxa de utilização da *CPU* (%) durante a fase de treinamento,
- 100 amostras de taxa de utilização da *CPU* (%) durante a fase de teste,
- 100 amostras de taxa de utilização da memória principal (%) durante a fase de treinamento,
- 100 amostras de taxa de utilização da memória principal (%) durante a fase de teste,
- 100 amostras de taxa de utilização da memória da *GPU* (MB) durante a fase de treinamento,
- 100 amostras de taxa de utilização da memória da *GPU* (MB) durante a fase de teste,

5.2.4 Apresentação e Validação dos Dados

Os gráficos *boxplot* são plotados usando o método *boxplot* da biblioteca *Seaborn*. Esse método possui o parâmetro *whis* configurado com o valor 1.5. Esse parâmetro define o limite inferior e o limite superior das hastes do *boxplot*, ou seja, os limites ficaram configurados da seguinte maneira:

$$LimiteInferior = PrimeiroQuartil - 1,5 * (TerceiroQuartil - PrimeiroQuartil)$$

$$LimiteSuperior = TerceiroQuartil + 1,5 * (TerceiroQuartil - PrimeiroQuartil)$$

Como forma de validar os dados e averiguar estatisticamente as hipóteses levantadas, o teste estatístico *Kolmogov-Smirnov* (*KS*) foi utilizado inicialmente para testar se as métricas adquiridas possuiam uma distribuição de probabilidade gaussiana (normal). A partir do resultado deste teste (que mostrou que os dados não apresentavam uma distribuição normal), foi utilizado o *Teste de Wilcoxon Pareado* e o *Teste de Friedman* para análise das hipóteses apresentadas.

O *Teste de Wilcoxon Pareado* foi selecionado para análise das hipóteses que envolvem apenas duas populações, pois ele é um teste não-paramétrico utilizado para comparar se as medidas de posição de duas populações são iguais no caso em que as amostras são dependentes. O *Teste de Friedman* foi selecionado para análise das hipóteses que envolvem mais de duas populações, pois ele é um teste não-paramétrico utilizado para comparar se as medidas de posição de mais de duas populações são iguais no caso em que as amostras são dependentes.

6

Resultados dos Experimentos

Esse capítulo descreve os resultados e apresenta discussões sobre os quatro experimentos realizados: Análise de Desempenho na Microarquitetura *Maxwell* (Experimento 1), Análise de Desempenho na Microarquitetura *Pascal* (Experimento 2), Análise de Desempenho na Microarquitetura *Volta* (Experimento 3) e Análise de Desempenho na Microarquitetura *Turing* (Experimento 4).

Como mencionado ao fim do capítulo anterior, o teste estatístico *Kolmogorov-Smirnov* foi utilizado para verificação da normalidade dos dados, para tanto, um nível de confiança de 95% foi aplicado. A partir do resultados identificou-se que para todas as variáveis dependentes, a distribuição de probabilidade não é normal, pois os *p-values* retornados (medida que indica a probabilidade do conjunto avaliado seguir a distribuição normal), foram próximos a zero (menor que 10^{-10}).

Como também mencionado no capítulo anterior, os testes utilizado para análise das hipóteses foram: o *Teste de Wilcoxon Pareado* com um nível de confiança de 95% nos casos em que é necessário analisar apenas duas populações e o *Teste de Friedman* com o nível de confiança de 95% nos casos em que é necessário analisar mais de duas populações.

6.1 Análise de Desempenho na Microarquitetura *Maxwell*

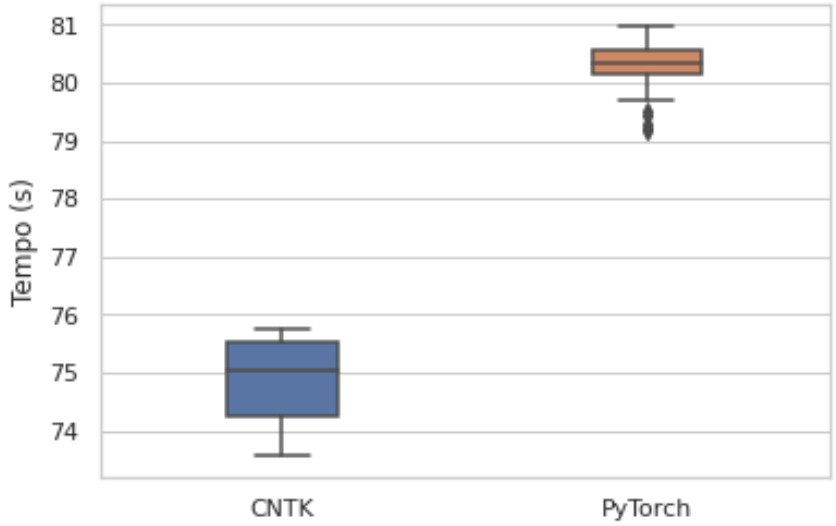
Nessa seção são analisados os dados da execução dos códigos no ambiente com *GPU* com a microarquitetura *Maxwell*.

6.1.1 Tempo de execução (Hipótese 1)

O resultado do *Teste de Wilcoxon Pareado* para o tempo de execução de treinamento retornou um *p-value* de $3,896559845095909x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 28, que

possui gráfico do tempo em segundos para cada um dos códigos, mostra com clareza um menor tempo de execução da fase de treinamento para o código que utiliza a biblioteca *CNTK*.

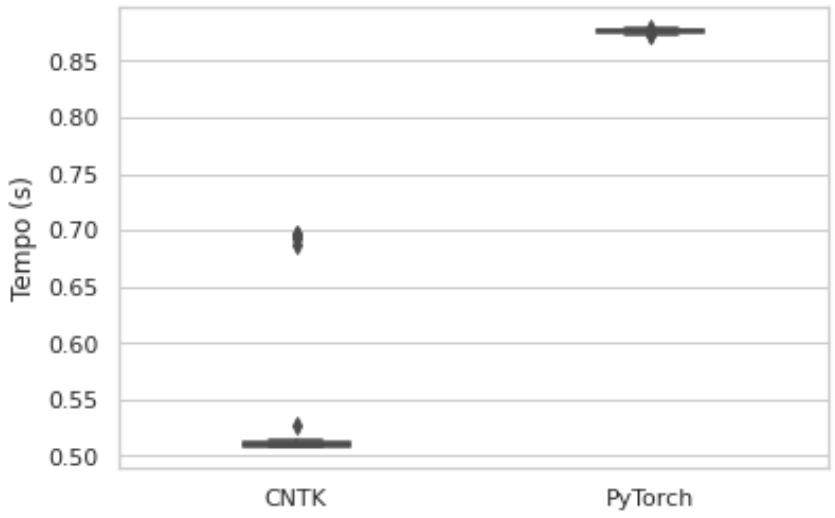
Figura 28 – Tempo de execução do treinamento (s) com a microarquitetura *Maxwell*



Fonte: Autoria Própria

O resultado do *Teste de Wilcoxon Pareado* para o tempo de execução de teste retornou um *p-value* de $3,896559845095909x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 29, que possui gráfico do tempo em segundos, mostra com clareza um menor tempo de execução da fase de treinamento para o código que utiliza a biblioteca *CNTK*.

Figura 29 – Tempo de execução do teste (s) com a microarquitetura *Maxwell*

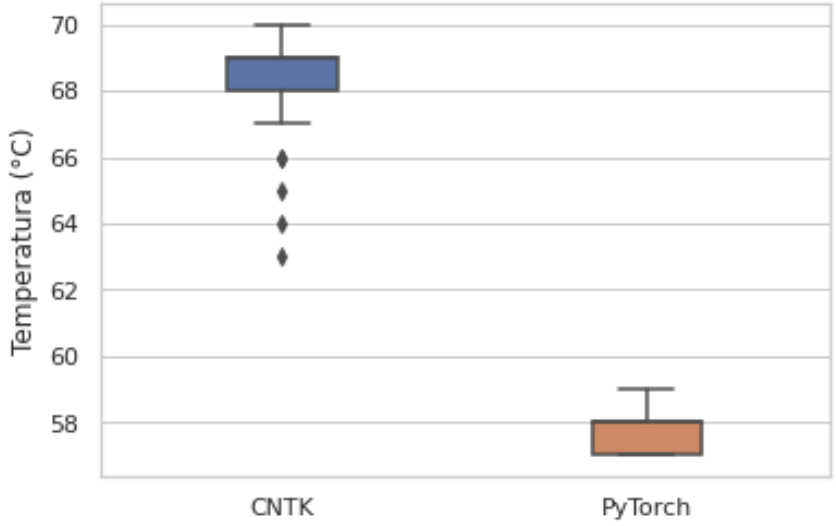


Fonte: Autoria Própria

6.1.2 Temperatura da *GPU* (Hipótese 2)

O resultado do *Teste de Wilcoxon Pareado* para a temperatura da *GPU* durante a fase de treinamento retornou um *p-value* de $4,314439986678757x10^{-19}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 30, que possui gráfico da temperatura em °C, mostra com clareza uma menor temperatura durante a fase de treinamento do código que utiliza a biblioteca *PyTorch*.

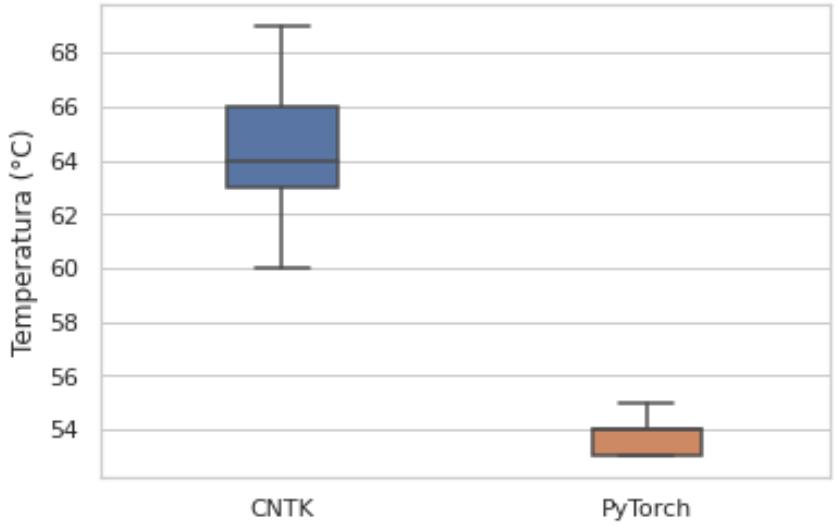
Figura 30 – Temperatura da *GPU* durante a fase de treinamento (°C) com a microarquitetura *Maxwell*



Fonte: Autoria Própria

O resultado do *Teste de Wilcoxon Pareado* para a temperatura da *GPU* durante a fase de teste retornou um *p-value* de $3,286316162455075x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 31, que possui gráfico da temperatura em °C, mostra com clareza uma menor temperatura durante a fase de teste do código que utiliza a biblioteca *PyTorch*.

Figura 31 – Temperatura da *GPU* durante a fase de teste (°C) com a microarquitetura *Maxwell*

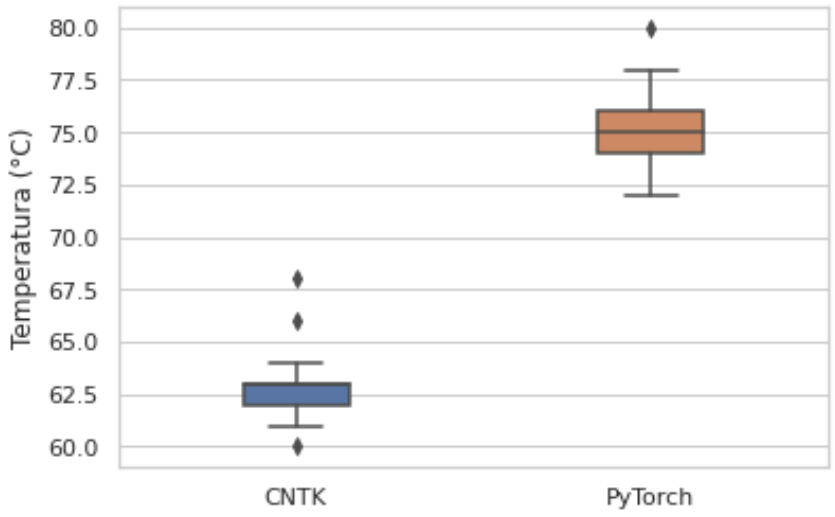


Fonte: Autoria Própria

6.1.3 Temperatura da *CPU* (Hipótese 3)

O resultado do *Teste de Wilcoxon Pareado* para a temperatura da *CPU* durante a fase de treinamento retornou um *p-value* de $2,1794635721189843x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 32, que possui gráfico da temperatura em °C, mostra com clareza uma menor temperatura durante a fase de treinamento do código que utiliza a biblioteca *CNTK*.

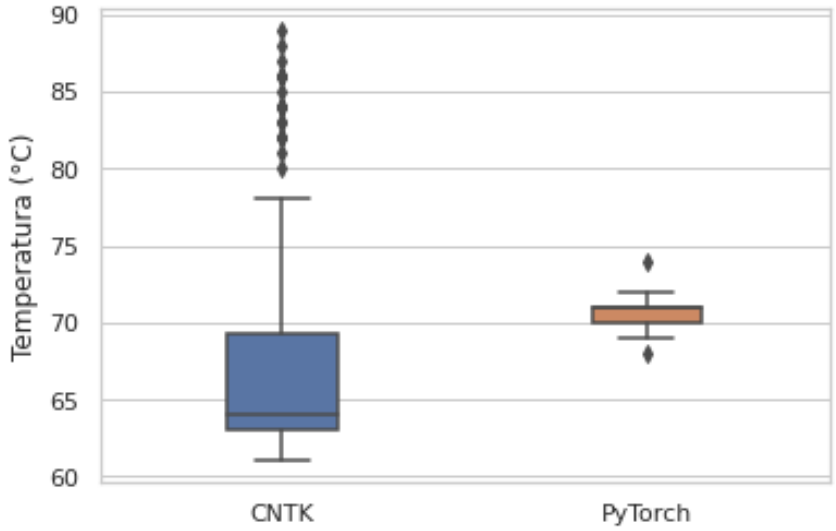
Figura 32 – Temperatura da *CPU* durante a fase de treinamento (°C) com a microarquitetura *Maxwell*



Fonte: Autoria Própria

O resultado do *Teste de Wilcoxon Pareado* para a temperatura da *CPU* durante a fase de teste retornou um *p-value* de $0,03350765396578013$, portanto a hipótese H0 foi rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 33, que possui gráfico da temperatura em °C, mostra com clareza uma menor temperatura durante a fase de teste do código que utiliza a biblioteca *CNTK*.

Figura 33 – Temperatura da *CPU* durante a fase de teste (°C) com a microarquitetura *Maxwell*



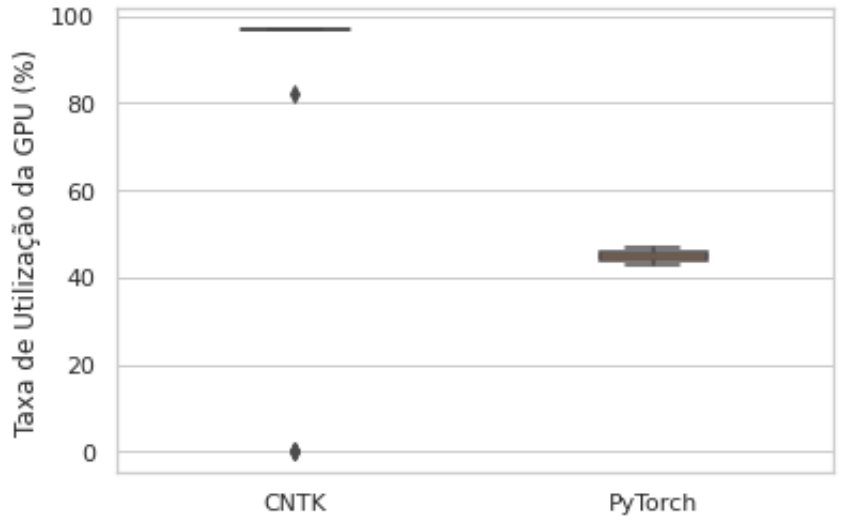
Fonte: Autoria Própria

6.1.4 Taxa de utilização da *GPU* (Hipótese 4)

O resultado do *Teste de Wilcoxon Pareado* para a taxa de utilização da *GPU* durante a fase de treinamento retornou um *p-value* de $4,757109284092181x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 34, que possui gráfico do taxa de utilização da *GPU* em porcentagem, mostra com clareza uma menor taxa de utilização durante a fase de treinamento do código que utiliza a biblioteca *PyTorch*.

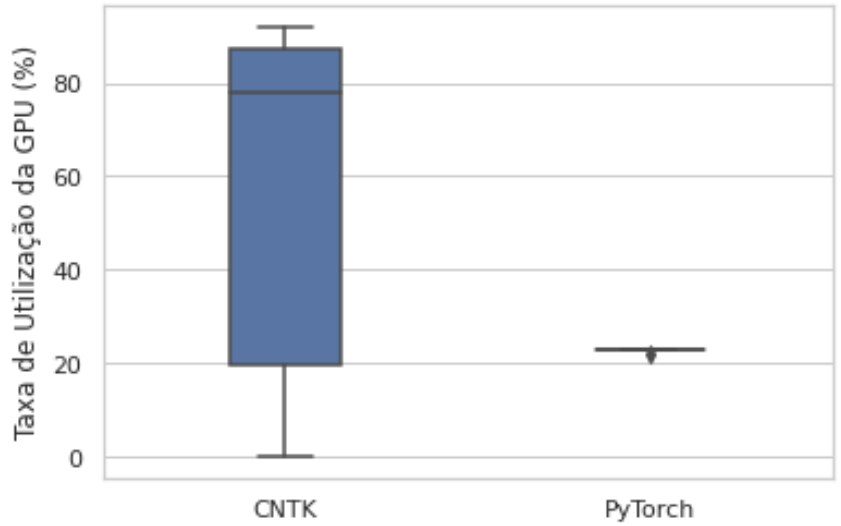
O resultado do *Teste de Wilcoxon Pareado* para a taxa de utilização da *GPU* durante a fase de teste retornou um *p-value* de $6,6158174533594015x10^{-12}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 35, que possui gráfico do taxa de utilização da *GPU* em porcentagem, mostra com clareza uma menor taxa de utilização durante a fase de teste do código que utiliza a biblioteca *PyTorch*.

Figura 34 – Taxa de utilização da *GPU* durante a fase de treinamento (%) com a microarquitetura *Maxwell*



Fonte: Autoria Própria

Figura 35 – Taxa de utilização da *GPU* durante a fase de teste (%) com a microarquitetura *Maxwell*

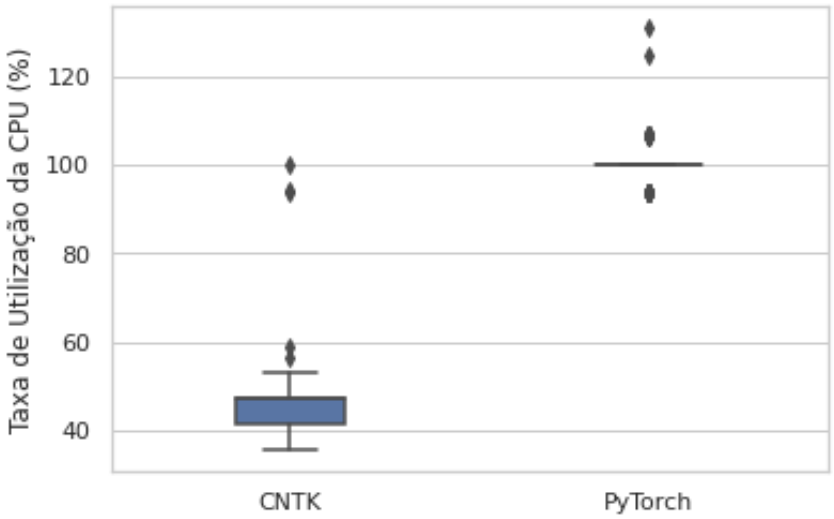


Fonte: Autoria Própria

6.1.5 Taxa de utilização da *CPU* (Hipótese 5)

O resultado do *Teste de Wilcoxon Pareado* para a taxa de utilização da *CPU* durante a fase de treinamento retornou um *p-value* de $4,902654854306188x10^{-18}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 36, que possui gráfico do taxa de utilização da *CPU* em porcentagem, mostra com clareza uma menor taxa de utilização durante a fase de treinamento do código que utiliza a biblioteca *CNTK*.

Figura 36 – Taxa de utilização da *CPU* durante a fase de treinamento (%) com a microarquitetura *Maxwell*

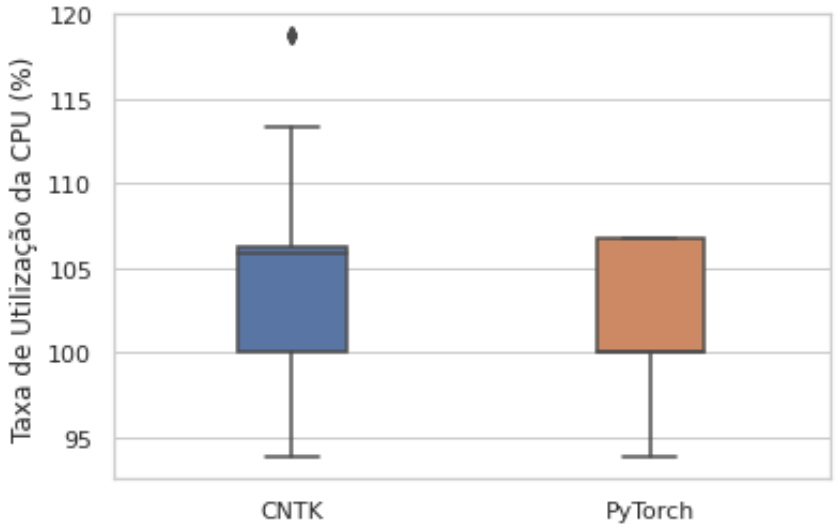


Fonte: Autoria Própria

O resultado do *Teste de Wilcoxon Pareado* para a taxa de utilização da *CPU* durante a fase de teste retornou um *p-value* de $9,448992188408321x10^{-5}$, portanto a hipótese H0 foi rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 37, que possui gráfico do taxa de utilização da *CPU* em porcentagem, não mostra com tanta clareza qual código possui uma menor taxa de utilização, porém a média da taxa de utilização da *CPU* durante a execução do código implementado com a biblioteca *CNTK* é $104,232\%$ enquanto a média da taxa de utilização durante a execução do código implementado com a biblioteca *PyTorch* é $100,626\%$.

É importante ressaltar que somente foi medida a taxa de utilização da execução do *PyThon*, ignorando todas as outras tarefas. O gerenciador de tarefas do sistema *Linux* apresenta os dados de cada tarefa considerando que cada núcleo do computador tem um valor máximo de taxa de utilização de 100%. Quando a tarefa utiliza mais de um núcleo, a taxa de utilização pode ultrapassar a taxa de 100%.

Figura 37 – Taxa de utilização da *CPU* durante a fase de teste (%) com a microarquitetura *Maxwell*



Fonte: Autoria Própria

6.1.6 Taxa de utilização da Memória Principal (Hipótese 6)

A taxa de utilização da memória principal permaneceu constante em todas as execuções, por isso os dados não foram apresentados em gráficos, mas sim apresentados na Tabela 15.

Tabela 15 – Taxa de Utilização da Memória Principal - Microarquitetura *Maxwell*

	Treino	Teste
CNTK	14,9 %	12,8 %
PyTorch	21,0 %	20,6 %

Em todas as ocasiões, a biblioteca *PyTorch* apresenta uma taxa de utilização da memória principal maior do a biblioteca *CNTK*.

6.1.7 Taxa de Utilização da Memória da *GPU* (Hipótese 7)

A taxa de utilização da memória da *GPU* permaneceu constante em todas as execuções, por isso os dados não foram apresentados em gráficos, mas sim apresentados na Tabela 16. Nesse caso, a taxa de utilização é apresentada em *Mega Bytes*.

Tabela 16 – Taxa de Utilização da Memória *GPU* - Microarquitetura *Maxwell*

	Treino	Teste
CNTK	220 MB	191 MB
PyTorch	445 MB	445 MB

Em todas as ocasiões, a biblioteca *PyTorch* apresenta uma taxa de utilização da memória da *GPU* maior do a biblioteca *CNTK*.

6.1.8 Análise geral do Experimento 1

Em alguns gráficos *boxplot* gerados a partir dos dados coletados no Experimento 1 é possível verificar a presença de *outliers*.

Na Figura 29 que ilustra o gráfico do tempo de execução da fase de teste são apresentados alguns *outliers*, é possível que esses *outliers* tenham sido ocasionados pela intervenção de outros processos, porém essa hipótese não foi testada durante a execução do experimento e os *outliers* não foram removidos sendo contabilizados durante a aplicação dos testes estatísticos.

Nos gráficos de temperatura, os *outliers* são ocasionados pela rápida mudança de temperatura do *hardware*, como na Figura 30 que a temperatura da *GPU* durante a execução da fase de treinamento do código implementado com a biblioteca *CNTK* cresce rapidamente em consequência de um aumento abrupto na taxa de utilização da *GPU*, esse aumento pode ser visualizado no gráfico da Figura 34. Isso também é notável ao analisarmos a temperatura da *CPU* durante a fase de treino, ilustrado na 32, e a taxa de utilização da *CPU* durante a mesma fase, ilustrado na 36.

Já os *outliers* da temperatura do *hardware* (*CPU* e *GPU*) durante a fase de teste e os *outliers* - ilustrados nas Figuras 31 e 33 - ocasionados pelo motivo contrário: uma rápida queda na taxa de utilização de ambas as unidades de processamento - as taxas de utilização ilustradas na Figura 35 e Figura 37. Este fenômeno não pode ser observado somente visualizando os gráficos, assim como a subida repentina da taxa de utilização, pois um gráfico *boxplot* não ilustra o comportamento ascendente ou descendente dos dados, porém, assim como descrito na Subseção 5.2.2, a fase de teste é realizada logo depois da fase de treinamento - que exige um maior poder computacional. Ou seja, há uma queda abrupta da taxa de utilização do *hardware* e, consequentemente, da temperatura.

Em ambas as fases, a quantidade de *outliers* nos dados coletados sobre a temperatura do *hardware* são maiores do que a quantidade de *outliers* presentes nos dados coletados sobre a taxa de utilização do *hardware*. Isso ocorre porque - em consequência das propriedades termodinâmicas dos materiais do *hardware*, temperatura ambiente, sistema de refrigeração do *hardware*- a capacidade de variação de temperatura do *hardware* em um intervalo de tempo é menor do que a capacidade de variação da taxa de utilização do *hardware*.

O código implementado com a biblioteca *CNTK* apresentou menor tempo de execução nas duas fases e apresentou uma maior taxa de utilização da *GPU* e da *CPU* na fase de inferência. Já o código implementado com *PyTorch* ocupou uma quantidade maior de memória principal e mémoria da *GPU*. Ou seja, os dados indicam que o melhor desempenho da biblioteca *CNTK*, neste caso, se deve a melhor capacidade de utilização do processamento da *GPU*, porém deve se

levar em consideração outros critérios.

A biblioteca *PyTorch* utiliza mais memória, principalmente memória da *GPU* que em ambas as fases ultrapassou a taxa de utilização de memória da biblioteca *CNTK*. A primeira suposição levantada é de que a baixa utilização da *GPU* é uma deficiência da biblioteca *PyTorch* com relação a biblioteca *CNTK*, porém outra suposição é que a maior quantidade de memória utilizada resulta em um maior tempo de carregamento e descarregamento da memórias principal e da memória *GPU* prejudicando seu desempenho. Ou seja, há dois fatores que podem influenciar na deficiência de desempenho com relação a tempo de execução da biblioteca *PyTorch* se comparada a biblioteca *CNTK*.

A Tabela 17 apresenta a média de todas as populações do Experimento 1 com o intervalo de confiança.

Tabela 17 – Tabela Geral do Experimento 1 - Microarquitetura Maxwell

Variável	*CNTK*	*PyTorch*
Tempo de Treinamento (s)	74,916721 $\pm 0,124260$	80,306269 $\pm 0,070628$
Tempo de Teste (s)	0,517951 $\pm 0,007148$	0,876513 $\pm 0,000218$
Temperatura da *GPU* Treino (°C)	68,37 $\pm 0,23$	57,84 $\pm 0,12$
Temperatura da *GPU* Teste (°C)	64,44 $\pm 0,44$	53,73 $\pm 0,11$
Temperatura da *CPU* Treino (°C)	62,63 $\pm 0,23$	75,10 $\pm 0,24$
Temperatura da *CPU* Teste (°C)	68,47 $\pm 1,67$	70,76 $\pm 0,19$
Utilização da *GPU* Treino (%)	93,91 $\pm 3,34$	45,16 $\pm 0,24$
Utilização da *GPU* Teste (%)	57,73 $\pm 7,23$	22,99 $\pm 0,02$
Utilização da *CPU* Treino (%)	47,27 $\pm 1,89$	100,18 $\pm 1,08$
Utilização da *CPU* Teste (%)	104,23 $\pm 1,22$	100,63 $\pm 0,85$
Memória Principal Treino (%)	14,9	21,0
Memória Principal Teste (%)	12,8	20,6
Memória *GPU* Treino (MB)	220	445
Memória *GPU* Teste (MB)	191	445

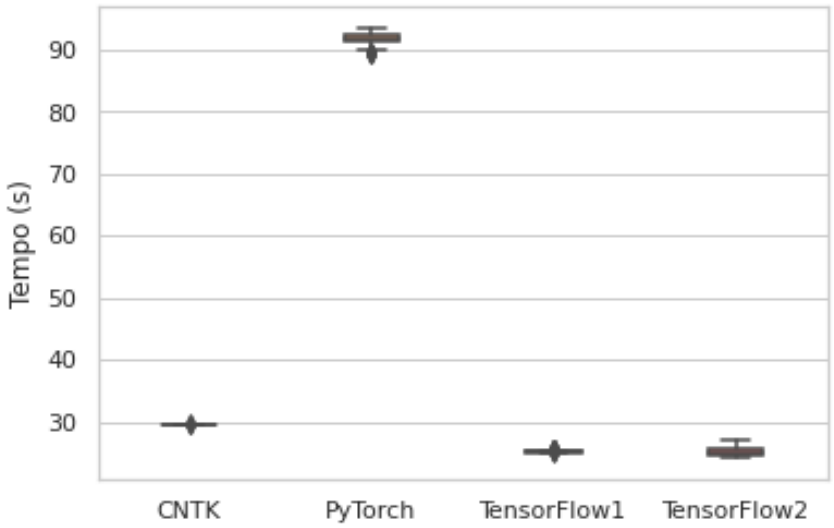
6.2 Análise de Desempenho na Microarquitetura *Kepler*

Nessa seção são analisados os dados da execução dos códigos no ambiente com *GPU* com a microarquitetura *Kepler*.

6.2.1 Tempo de execução (Hipótese 1)

O resultado do *Teste de Friedman* para o tempo de execução de treinamento retornou um *p-value* de $2,302782349861235x10^{-58}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 38, que possui gráfico do tempo em segundos, mostra com clareza um menor tempo de execução da fase de treinamento para os códigos que utilizam a bibliotecas *TensorFlow 1* e *TensorFlow 2*.

Figura 38 – Tempo de execução do treinamento (s) com a microarquitetura *Kepler*



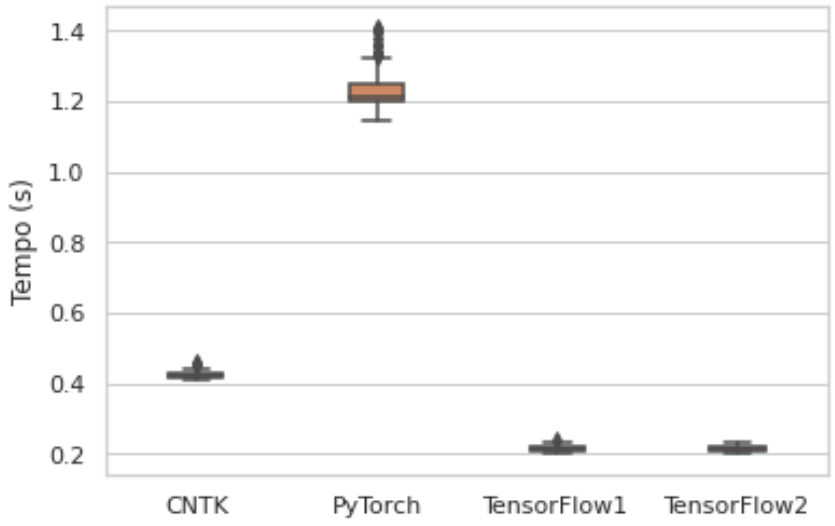
Fonte: Autoria Própria

Como na Figura 38 não foi possível identificar uma diferença entre o tempo de execução do código implementado com a biblioteca *TensorFlow 1* e o tempo de execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $0,22093576367931422$, ou seja, se considerarmos a hipótese de que não há diferença estatística entre as duas populações, a hipótese não poderá ser rejeitada.

O resultado do *Teste de Friedman* para o tempo de execução de teste retornou um *p-value* de $2,6580414271985574x10^{-58}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 39, que possui gráfico do tempo em segundos para cada um dos códigos, mostra com clareza um menor tempo de execução da fase de treinamento para os códigos que utilizam as bibliotecas *TensorFlow 1* e *TensorFlow 2*

Como na Figura 39 não foi possível identificar uma diferença entre o tempo de execução do código implementado com a biblioteca *TensorFlow 1* e o tempo de execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $0,48304202468022994$, ou seja, se considerarmos a hipótese de que não há diferença estatística entre as duas populações, a hipótese não poderá ser rejeitada.

Figura 39 – Tempo de execução do teste (s) com a microarquitetura *Kepler*

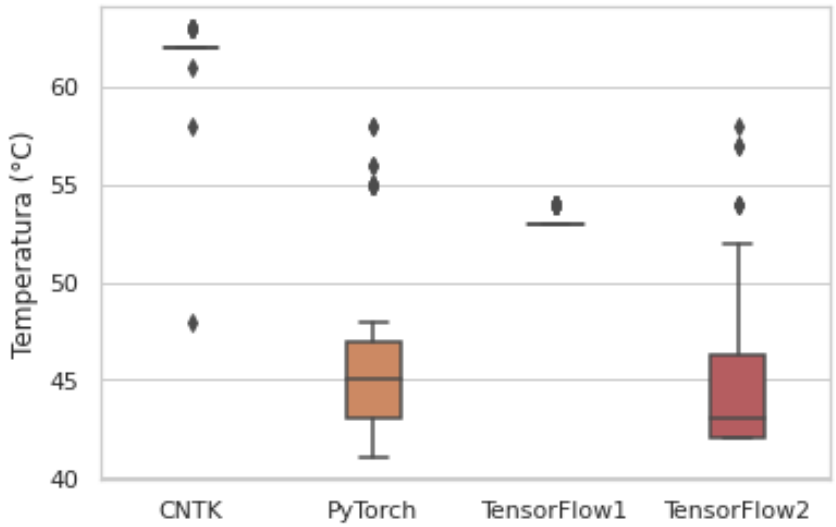


Fonte: Autoria Própria

6.2.2 Temperatura da *GPU* (Hipótese 2)

O resultado do *Teste de Friedman* para a temperatura da *GPU* durante a fase de treinamento retornou um *p-value* de $7,658683463606682x10^{-52}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 40, que possui gráfico de temperatura em graus *Celsius*, mostra com clareza temperatura da *GPU* durante a fase de treinamento para os códigos que utilizam as bibliotecas *TensorFlow 2* e *PyTorch*.

Figura 40 – Temperatura da *GPU* durante fase de treinamento (°C) com a microarquitetura *Kepler*

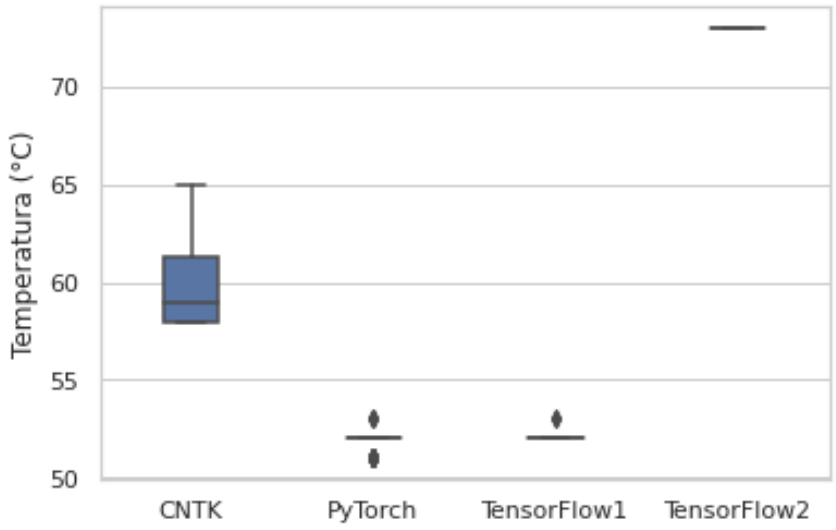


Fonte: Autoria Própria

Como na Figura 40 não foi possível identificar uma diferença entre a temperatura da *GPU* durante a execução do código implementado com a biblioteca *PyTorch* e temperatura da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $0,04985222934585293$, ou seja, há diferença estatística entre as duas populações. A temperatura da *GPU* durante a execução do código implementado com *TensorFlow 2* apresenta uma média de 44,67°C e a temperatura da *GPU* durante a execução do código implementado com *PyTorch* apresenta uma média de 46,10°C.

O resultado do *Teste de Friedman* para a temperatura da *GPU* durante a fase de teste retornou um *p-value* de $1,5155142407508852x10^{-52}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 41, que possui gráfico de temperatura em graus *Celsius*, mostra com clareza temperatura da *GPU* durante a fase de treinamento para os códigos que utilizam as bibliotecas *TensorFlow 1* e *PyTorch*.

Figura 41 – Temperatura da *GPU* durante a fase de teste (°C) com a microarquitetura *Kepler*



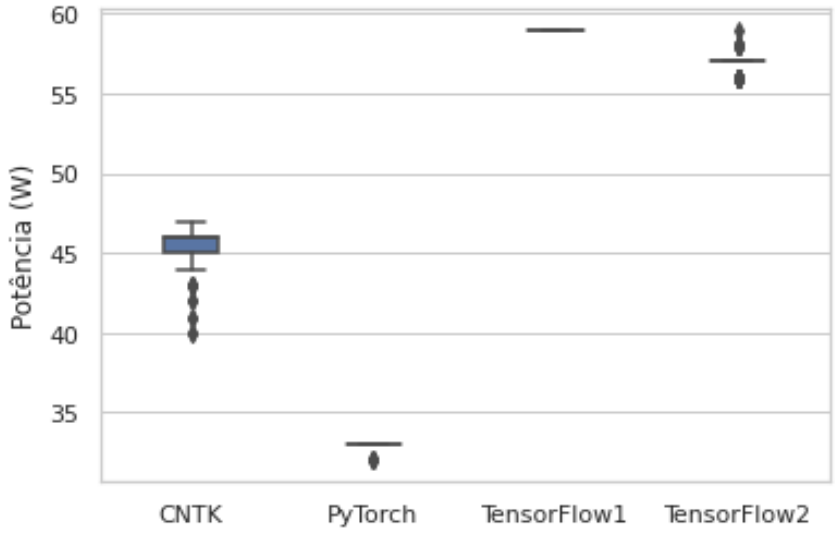
Fonte: Autoria Própria

Como na Figura 41 não foi possível identificar uma diferença entre a temperatura da *GPU* durante a execução do código implementado com a biblioteca *PyTorch* e temperatura da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $5,699411623331837x10^{-5}$, ou seja, há diferença estatística entre as duas populações. A temperatura da *GPU* durante a execução do código implementado com *TensorFlow 1* apresenta uma média de 52,02°C e a temperatura da *GPU* durante a execução do código implementado com *PyTorch* apresenta uma média de 51,84°C.

6.2.3 Potência da *GPU* (Hipótese 3)

O resultado do *Teste de Friedman* para a potência da *GPU* durante a fase de treinamento retornou um *p-value* de $1,1537515410447051x10^{-52}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 42, que possui potência em *Watts* para cada um dos códigos, mostra com clareza uma menor potência da *GPU* durante a fase de treinamento para o código que utiliza a biblioteca *PyTorch* e uma maior potência para o código implementado com a biblioteca *TensorFlow 1*.

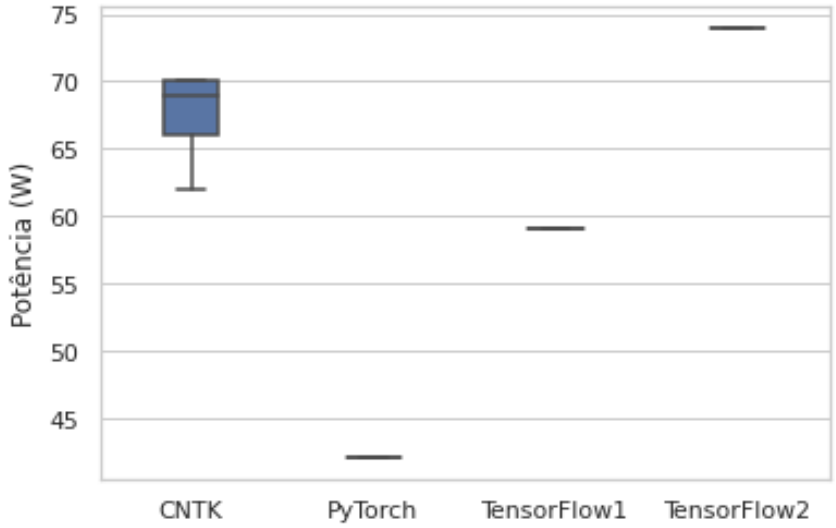
Figura 42 – Potência da *GPU* durante a fase de treinamento (W) com a microarquitetura *Kepler*



Fonte: Autoria Própria

O resultado do *Teste de Friedman* para a potência da *GPU* durante a fase de teste retornou um *p-value* de $9,948758346327588x10^{-65}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 43, que possui potência em *Watts* para cada um dos códigos, mostra com clareza uma menor potência da *GPU* durante a fase de teste para o código que utiliza a biblioteca *PyTorch* e uma maior potência para o código implementado com a biblioteca *TensorFlow 2*.

Figura 43 – Potência da *GPU* durante a fase de teste (W) com a microarquitetura *Kepler*



Fonte: Autoria Própria

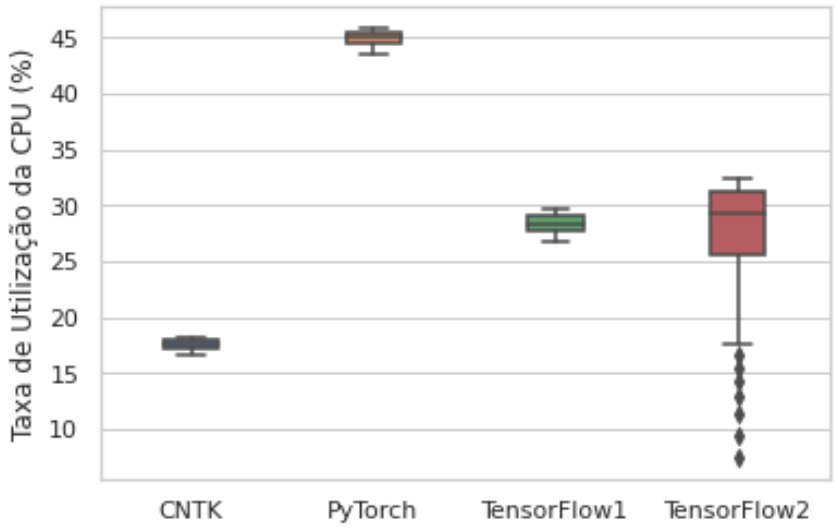
6.2.4 Taxa de Utilização da *CPU* (Hipótese 4)

O resultado do *Teste de Friedman* para a taxa de utilização da *CPU* durante a fase de treinamento retornou um *p-value* de $9,41494841443219x10^{-56}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 44, que possui taxa de utilização da *CPU* em porcentagem, mostra com clareza uma menor taxa de utilização da *CPU* durante a fase de treinamento para o código que utiliza a biblioteca *CNTK* e uma maior taxa de utilização da *CPU* para o código implementado com a biblioteca *PyTorch*.

Como na Figura 44 não foi possível identificar uma diferença entre a taxa de utilização da *CPU* durante a execução do código implementado com a biblioteca *TensorFlow1* e a taxa de utilização da *CPU* durante a execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $0,8526950921674088$, ou seja, se considerarmos a hipótese de que não há diferença estatística entre as duas populações, a hipótese não poderá ser rejeitada.

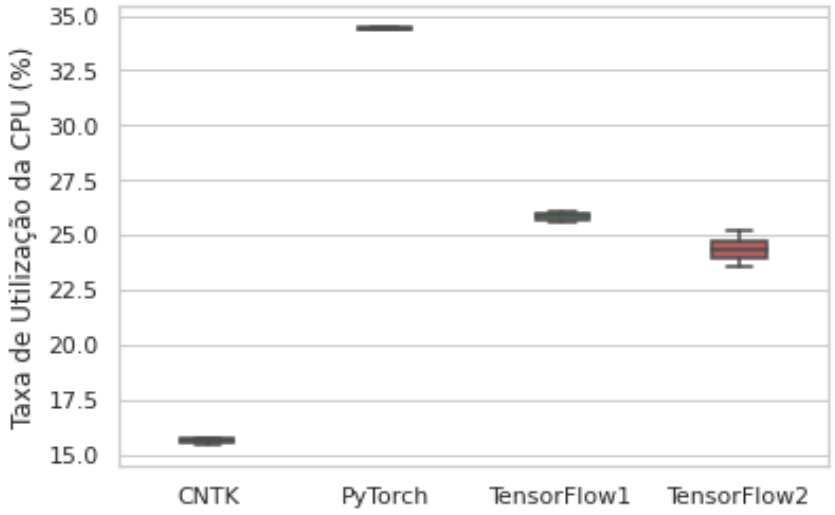
O resultado do *Teste de Friedman* para a taxa de utilização da *CPU* durante a fase de teste retornou um *p-value* de $9,948758346327588x10^{-65}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 44, que possui taxa de utilização da *CPU* em porcentagem, mostra com clareza uma menor taxa de utilização da *CPU* durante a fase de teste para o código que utiliza a biblioteca *CNTK* e uma maior taxa de utilização da *CPU* para o código implementado com a biblioteca *PyTorch*.

Figura 44 – Taxa de utilização da *CPU* durante a fase de treinamento (°C) com a microarquitetura *Kepler*



Fonte: Autoria Própria

Figura 45 – Taxa de utilização da *CPU* durante a fase de teste (%) com a microarquitetura *Kepler*



Fonte: Autoria Própria

6.2.5 Taxa de Utilização da Memória Principal (Hipótese 5)

A taxa de utilização da memória principal permaneceu constante em todas as execuções, por isso os dados não foram apresentados em gráficos, mas sim apresentados na Tabela 18.

O código implementado com a biblioteca *CNTK* apresentou a menor taxa de utilização nas fases de treinamento e de teste. O código implementado com a biblioteca *TensorFlow 1* apresentou a maior taxa de utilização na fase de treino e o código implementado com a biblioteca *PyTorch* apresentou a maior taxa de utilização da memória principal na fase de teste.

Tabela 18 – Taxa de Utilização da Memória Principal - Microarquitetura *Kepler*

	Treino	Teste
CNTK	9,8 %	11,0 %
PyTorch	13,5 %	25,1 %
TensorFlow 1	14,2 %	14,6 %
TensorFlow 2	13,6 %	14,2 %

6.2.6 Taxa de Utilização da Memória da *GPU* (Hipótese 6)

A taxa de utilização da memória da *GPU* permaneceu constante em todas as execuções, por isso os dados não foram apresentados em gráficos, mas sim apresentados na Tabela 19.

Tabela 19 – Taxa de Utilização da Memória da *GPU* - Microarquitetura *Kepler*

	Treino	Teste
CNTK	273 MB	233 MB
PyTorch	341 MB	573 MB
TensorFlow 1	304 MB	304 MB
TensorFlow 2	571 MB	303 MB

O código implementado com a biblioteca *CNTK* apresentou a menor taxa de utilização da memória da *GPU* nas fases de treinamento e de teste. O código implementado com a biblioteca *TensorFlow 2* apresentou a maior taxa de utilização durante a fase de treinamento e o código implementado com a biblioteca *PyTorch* apresentou a maior taxa de utilização da memória da *GPU* durante a fase de teste.

6.2.7 Análise Geral do Experimento 2

Na análise do Experimento 2 não aprofundaremos a discussão sobre *outliers*, pois esta discussão é semelhante para todos os experimentos tratados no presente capítulo, a discussão deste ponto é feita na Subseção 6.1.8. Porém devemos tratar aqui sobre a potência da *GPU* - variável dependente medida nos Experimentos 2, 3 e 4 - que sofre influência direta da taxa de utilização da *GPU*, os *outliers* presentes nos gráficos das Figuras 42 e 43 possivelmente são causados pela rápida mudança na taxa de utilização da *GPU* - variável não medida nesse experimento por limitações já tratadas anteriormente - e por oscilações do sistema de alimentação de energia do *hardware*.

O gráfico da 39 e o gráfico da Figura 39 junto com os testes estatísticos aplicados mostram que os códigos que utilizam as bibliotecas *TensorFlow 1* e *TensorFFlow 2* apresentam o menor tempo de execução em ambas as fases e o código que utiliza a *biblioteca PyTorch* apresentou o menor tempo de execução em ambas as fases.

Por limitações do ambiente *Google Colab*, já expostas no Capítulo 5, não foi possível obter a taxa de utilização da *GPU*, porém foi possível obter a medida de potência momentânea e a temperatura da *GPU*, medidas que estão relacionadas a taxa de utilização da *GPU*. Nas Figuras 40, 41, 42 e 35 é possível observar que, apesar da biblioteca *TensorFlow 1* e *TensorFlow 2* apresentarem o mesmo tempo de execução, eles utilizam a *GPU* de maneiras diferentes. A biblioteca *TensoFlow 1* consome mais potência e eleva mais a temperatura da *GPU* durante a fase de treinamento, já na fase de teste isso se inverte com relação a biblioteca *TensorFlow 2*. Porém os gráficos das Figuras 44 e 45 mostra, que a biblioteca *TensorFlow 1* tem um comportamento muito semelhante com relação a utilização da *CPU*, sendo que a *TensorFlow 2* apresenta uma menor taxa de utilização da *CPU*.

Quanto a utilização da memória, as duas bibliotecas (*TensorFlow 1* e *TensorFlow 2*) apresentaram comportamentos diferentes: a *TensorFlow 2* apresentou uma taxa de utilização da memória da *GPU* durante a fase de treinamento muito acima da taxa de utilização apresentada pela *TensorFlow 1*, na fase de teste, a *TensorFlow 2* apresentou uma taxa de utilização da memória da *GPU* próxima a taxa de utilização apresentada pela *TensorFlow 1*. A taxa de utilização da memória principal da biblioteca *TensorFlow 2* é menor do que a da biblioteca *TensorFlow 1* para as duas fases.

A biblioteca *PyTorch* apresentou o maior tempo de execução em ambas as fases e a biblioteca *CNTK* o segundo maior tempo. A diferença de tempo de execução entre a biblioteca *PyTorch* e a biblioteca *CNTK* foi consideravelmente maior do que a diferença do tempo de execução entre a biblioteca *CNTK* e as bibliotecas *TensorFlow 1* e *TensorFlow 2*. Avaliando os gráficos das Figuras 42, 43 é notado um menor consumo de potência durante a execução da biblioteca *PyTorch* com relação às outras bibliotecas - embora a temperatura ilustrada na Figura 40 não seja menor do que a temperatura da biblioteca *TensorFlow 2*, possivelmente por conta das variáveis independentes - e uma maior taxa de utilização da *CPU* ilustrados nos gráficos das Figuras 44 e 45. Observando as Tabelas 18 e 19, a biblioteca *PyTorch* apresenta a maior taxa de utilização das memória principal e da memória *GPU* durante a fase de teste. Quanto a utilização da memória principal, a biblioteca *PyTorch* apresenta a maior taxa de utilização na fase de teste e segunda maior taxa de utilização na fase de treinamento.

A biblioteca *CNTK* apresentou o segundo maior tempo de execução em ambas as fases, porém o seu desempenho foi próximo ao desempenho das bibliotecas *TensorFlow 1* e *TensorFlow 2*. Ao analisarmos a potência da *GPU* nas Figuras 42 e 43, observa-se que a potência consumida da *GPU* é um pouco maior durante a fase de teste, sendo a segundo maior potência. Durante a fase de treinamento, a potência consumida da *GPU* é menor do que as potências consumidas durante as execuções dos códigos das bibliotecas *TensorFlow 1 e TensorFlow 2*, apesar disso a biblioteca apresentou a maior temperatura da *GPU* durante a fase de treinamento e a segunda maior durante a fase de teste. A execução do código implementado com a biblioteca *CNTK* apresentou a menor taxa de utilização da *CPU* em ambas as fases e a menor taxa de utilização

de memória em todos os casos conforme mostram as Tabelas 19 e 18.

Neste experimento, não dá para atribuir um único fator que explique a diferença e a não diferença entre o desempenho das bibliotecas. Há diferenças de comportamento até mesmo entre as duas bibliotecas que não apresentaram diferença estatística do tempo de execução, porém é possível verificar que, assim como no Experimento 1, a biblioteca *PyTorch* apresentou o maior tempo de execução, menor uso da *GPU* (de acordo com a potência e temperatura medida) e maior taxa de utilização da *CPU*. Ao comparar o comportamento da biblioteca *CNTk* somente com o comportamento da biblioteca *PyTorch*, é possível observar que a relação entre os comportamentos se repetem, ou seja, o *CNTK* apresenta menor tempo de execução, maior da utilização da *GPU* (de acordo com a potência) e menor taxa de utilização da *CPU*.

Ao analisarmos as bibliotecas *TensorFlow 1* e *TensorFlow 2* é difícil levantar hipóteses sobre os motivos do menor tempo de execução em relação as outras bibliotecas, ambas apresentam uma alta potência da *GPU* na fase de treinamento, mas na fase de testes a potência da *GPU* durante a execução da biblioteca *CNTK* é maior. A taxa de utilização de *CPU* de ambas das bibliotecas *TensorFlow 1* e *TensorFlow 2* está entre a taxa de utilização da *CPU* durante a execução da biblioteca *CNTK* e da taxa de utilização da *CPU* durante a execução biblioteca *PyTorch*. Ou seja, essas duas bibliotecas (TensorFlow 1 e TensorFlow 2) apresentam valores de taxa de utilização que indicam um equilíbrio no uso do *hardware*.

A Tabela 20 apresenta a média de todas as populações do Experimento 2 com o intervalo de confiança.

Tabela 20 – Tabela Geral do Experimento 2 - Microarquitetura *Kepler*

Variável	*CNTK*	*PyTorch*	*TensorFlow 1*	*TensorFlow 2*
Tempo de Treinamento (s)	29,460917 ±0,013035	91,707285 ±0,177099	25,138235 ±0,036239	25,055372 ±0,143548
Tempo de Teste (s)	0,422731 ±0.001727	1,232349 ±0,011209	0,215054 ±0.001401	0,214265 ±0,001362
Temperatura da *GPU* Treino (°C)	61,93 ±0,30	46,10 ±0,89	53,19 ±0,08	44,67 ±0,80
Temperatura da *GPU* Teste (°C)	60,07 ±0,48	51,84 ±0,09	52,02 ±0,03	73,00 ±0,00
Potência da *GPU* Treino (W)	45,34 ±0,28	32,94 ±0,05	59,00 ±0,00	56,95 ±0,12
Potência da *GPU* Teste (W)	67,96 ±0,47	42,00 ±0,00	59,00 ±0,00	74,00 ±0,00
Utilização da *CPU* Treino (%)	17,58 ±0,09	44,95 ±0,14	28,36 ±0,17	27,35 ±1,08
Utilização da *CPU* Teste (%)	15,58 ±0,02	34,43 ±0,01	25,85 ±0,03	24,33 ±0,09
Memória Principal Treino (%)	9,8	13,5	14,2	13,6
Memória Principal Teste (%)	11,0	25,1	14,6	14,2
Memória *GPU* Treino (MB)	273	341	304	571
Memória *GPU* Teste (MB)	233	573	304	303

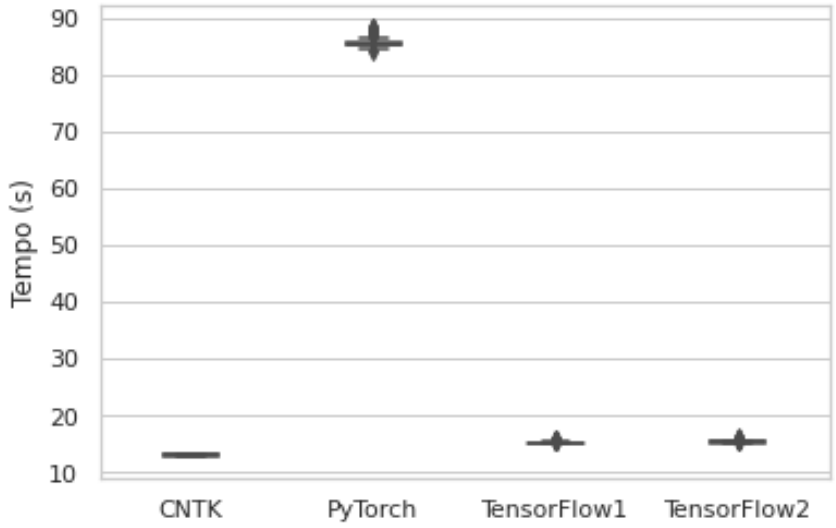
6.3 Análise de Desempenho na Microarquitetura *Pascal*

Nessa seção são analisados os dados da execução dos códigos no ambiente com *GPU* com a microarquitetura *Pascal*.

6.3.1 Tempo de execução (Hipótese 1)

O resultado do *Teste de Friedman* para o tempo de execução de treinamento retornou um *p-value* de $8,608079057809072x10^{-62}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 46, que possui gráfico do tempo em segundos, mostra com clareza um menor tempo de execução do fase de treinamento para o código que utiliza a biblioteca *CNTK*.

Figura 46 – Tempo de execução do treinamento (s) com a microarquitetura *Pascal*



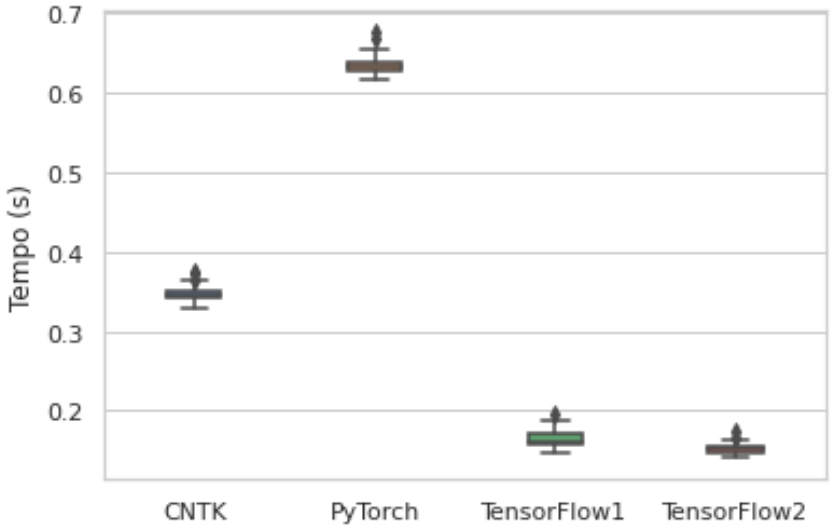
Fonte: Autoria Própria

Como na Figura 46 não foi possível identificar uma diferença entre o tempo de execução do código implementado com a biblioteca *TensorFlow 1* e o tempo de execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $4,081605935918749x10^{-10}$, ou seja, há diferenças estatísticas entre as duas populações. O tempo de execução durante a execução do código implementado com *TensorFlow 1* apresenta uma média de $15,196989$ s e o tempo de execução do código implementado com *TensorFlow 2* apresenta uma média de $15,364843$ s.

O resultado do *Teste de Friedman* para o tempo de execução de teste retornou um *p-value* de $1,3326239988754905x10^{-62}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 47, que possui gráfico do tempo em segundos, mostra com clareza um menor tempo de execução da fase de teste para os códigos que utilizam as bibliotecas *TensorFlow 1* e *TensorFlow 2*.

Como na Figura 47 não foi possível identificar uma diferença entre o tempo de execução do código implementado com a biblioteca *TensorFlow 1* e o tempo de execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado*

Figura 47 – Tempo de execução do teste (s) com a microarquitetura *Pascal*



Fonte: Autoria Própria

retornou um *p-value* de $2,198665592591444x10^{-16}$, ou seja, há diferenças estatísticas entre as duas populações. O tempo de execução durante a execução do código implementado com *TensorFlow 1* apresenta uma média de $0,164300$ s e o tempo de execução do código implementado com *TensorFlow 2* apresenta uma média de $0,151466$ s.

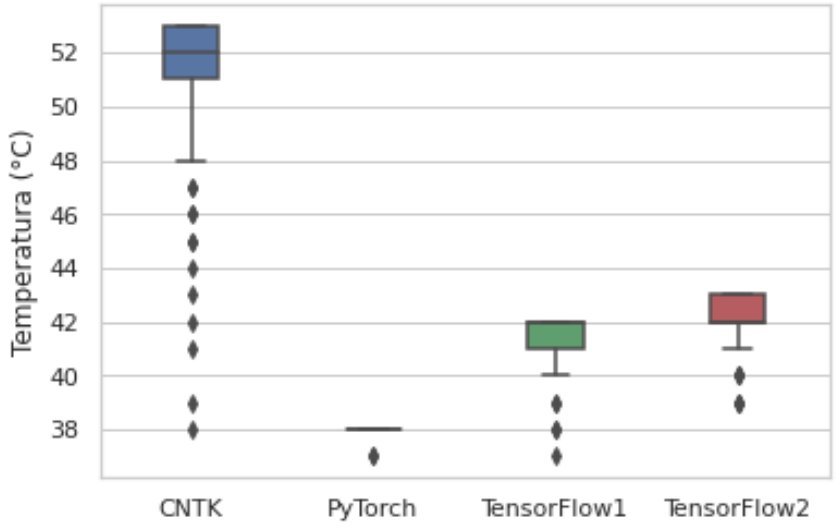
6.3.2 Temperatura da *GPU* (Hipótese 2)

O resultado do *Teste de Friedman* para a temperatura da *GPU* durante a fase de treinamento retornou um *p-value* de $2,185577119887258x10^{-62}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 48, que possui gráfico de temperatura em graus *Celsius*, mostra com clareza uma menor temperatura da *GPU* durante a fase de treinamento para o código que utiliza a biblioteca *PyTorch* e uma maior temperatura da *GPU* durante a fase de treinamento para o código que utiliza a biblioteca *CNTK*.

O resultado do *Teste de Friedman* para a temperatura da *GPU* durante a fase de teste retornou um *p-value* de $1,5155142407508852x10^{-52}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 49, que possui gráfico de temperatura em graus *Celsius*, mostra com clareza temperatura da *GPU* durante a fase de teste para os códigos que utilizam as bibliotecas *TensorFlow 1* e *TensorFlow 2*.

Como na Figura 49 não foi possível identificar uma diferença entre a temperatura da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 1* e temperatura da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de 0.0003114909767673833, ou seja, há

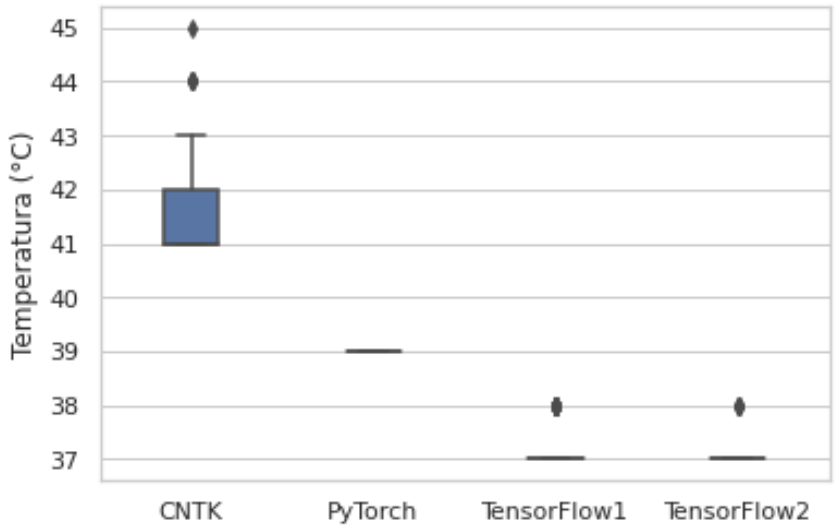
Figura 48 – Temperatura da *GPU* durante a fase de treinamento (°C) com a microarquitetura *Pascal*



Fonte: Autoria Própria

diferença estatística entre as duas populações. A temperatura da *GPU* durante a execução do código implementado com *TensorFlow 1* apresenta uma média de 37,20 °C e a temperatura da *GPU* durante a execução do código implementado com *TensorFlow 2* apresenta uma média de 37,07 °C.

Figura 49 – Temperatura da *GPU* durante a fase de teste (°C) com a microarquitetura *Pascal*

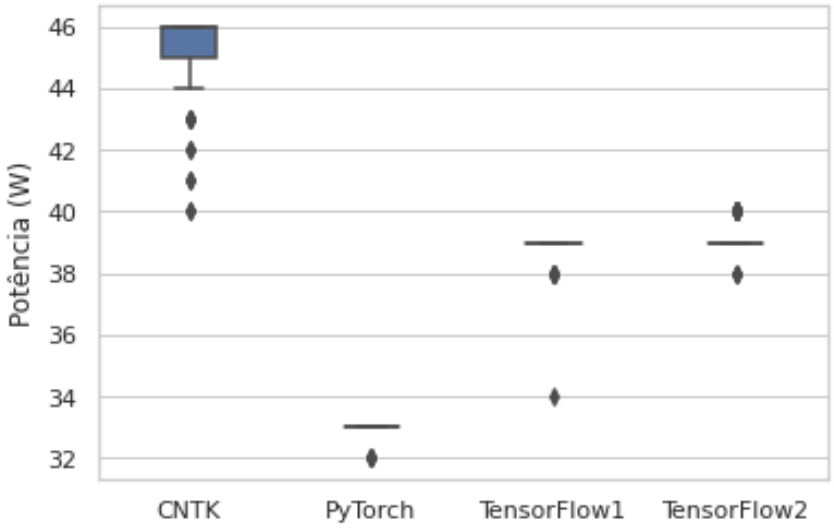


Fonte: Autoria Própria

6.3.3 Potência da *GPU* (Hipótese 3)

O resultado do *Teste de Friedman* para a potência da *GPU* durante a fase de treinamento retornou um *p-value* de $1,4699139808370653x10^{-63}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 50, que possui potência em *Watts* para cada um dos códigos, mostra com clareza uma menor potência da *GPU* durante a fase de treinamento para o código que utiliza a biblioteca *PyTorch* e uma maior potência para o código implementado com a biblioteca *CNTK*.

Figura 50 – Potência da *GPU* durante a fase de treinamento (W) com a microarquitetura *Pascal*



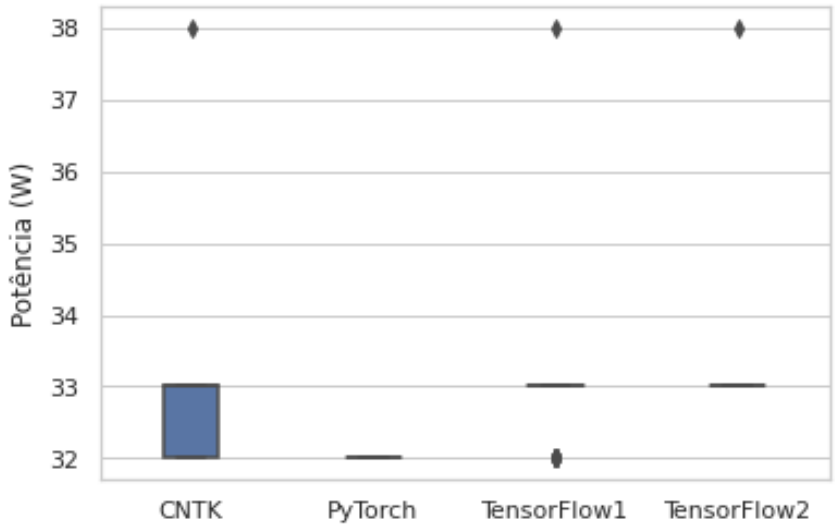
Fonte: Autoria Própria

Como na Figura 50 não foi possível identificar uma diferença entre a potência da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 1* e a potência da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $7,097908330908269x10^{-6}$, ou seja, há diferença estatística entre as duas populações. A potência da *GPU* durante a execução do código implementado com *TensorFlow 1* apresenta uma média de 38,88 W e a potência da *GPU* durante a execução do código implementado com *TensorFlow 2* apresenta uma média de 39,13 W.

O resultado do *Teste de Friedman* para a potência da *GPU* durante a fase de teste retornou um *p-value* de $7,9049074843878x10^{-42}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 51, que possui potência em *Watts* para cada um dos códigos, mostra com clareza uma menor potência da *GPU* durante a fase de teste para o código que utiliza a biblioteca *PyTorch* e uma maior potência para os códigos implementados com as bibliotecas *TensorFlow 1* e *TensorFlow 2*.

Como na Figura 51 não foi possível identificar uma diferença entre a potência da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 1* e a potência da

Figura 51 – Potência da *GPU* durante a fase de teste (W) com a microarquitetura *Pascal*



Fonte: Autoria Própria

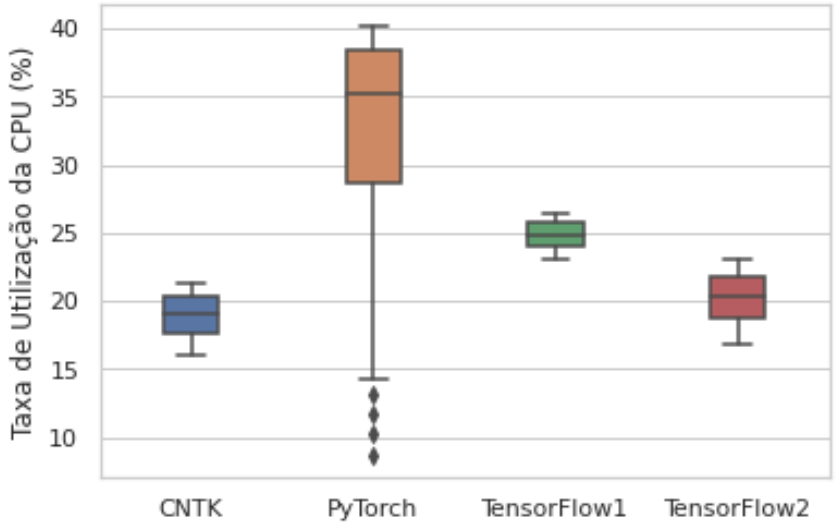
GPU durante a execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $6,334248366623973x10^{-5}$, ou seja, há diferença estatística entre as duas populações. A potência da *GPU* durante a execução do código implementado com *TensorFlow 1* apresenta uma média de 32,89 W e a potência da *GPU* durante a execução do código implementado com *TensorFlow 2* apresenta uma média de 33,05 W.

6.3.4 Taxa de Utilização da *CPU* (Hipótese 4)

O resultado do *Teste de Friedman* para a taxa de utilização da *CPU* durante a fase de treinamento retornou um *p-value* de $1,1778234440926679x10^{-53}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 52, que possui taxa de utilização da *CPU* em porcentagem, mostra com clareza uma menor taxa de utilização da *CPU* durante a fase de treinamento para o código que utiliza a biblioteca *CNTK* e uma maior taxa de utilização da *CPU* para o código implementado com a biblioteca *PyTorch*.

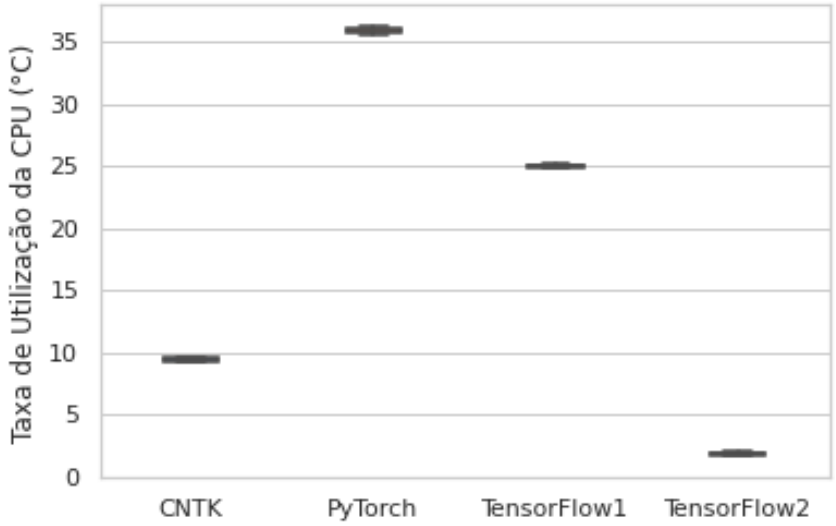
O resultado do *Teste de Friedman* para a taxa de utilização da *CPU* durante a fase de teste retornou um *p-value* de $9.948758346327588x10^{-65}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 52, que possui taxa de utilização da *CPU* em porcentagem, mostra com clareza uma menor taxa de utilização da *CPU* durante a fase de teste para o código que utiliza a biblioteca *TensorFlow 2* e uma maior taxa de utilização da *CPU* para o código implementado com a biblioteca *PyTorch*.

Figura 52 – Taxa de utilização da *CPU* durante a fase de treinamento (°C) com a microarquitetura *Pascal*



Fonte: Autoria Própria

Figura 53 – Taxa de utilização da *CPU* durante fase de teste (%) com a microarquitetura *Pascal*



Fonte: Autoria Própria

6.3.5 Taxa de Utilização da Memória Principal (Hipótese 5)

A taxa de utilização da memória principal permaneceu constante em todas as execuções, por isso os dados não foram apresentados em gráficos, mas sim apresentados na Tabela 21.

O código implementado com a biblioteca *CNTK* apresentou a menor taxa de utilização durante a fase de teste e a maior taxa de utilização durante a fase de treinamento. O código implementado com a biblioteca *TensorFlow 1* apresentou a menor taxa de utilização durante a fase de treino e o código implementado com a biblioteca *PyTorch* apresentou a maior taxa de

Tabela 21 – Taxa de Utilização da Memória Principal - Microarquitetura *Pascal*

	Treino	Teste
CNTK	24,9 %	13,3 %
PyTorch	18,4 %	18,4 %
TensorFlow 1	17,5 %	17,7 %
TensorFlow 2	17,8 %	17,3 %

utilização da memória principal durante a fase de teste.

6.3.6 Taxa de Utilização da Memória da *GPU* (Hipótese 6)

A taxa de utilização da memória da *GPU* permaneceu constante em todas as execuções, por isso os dados não foram apresentados em gráficos, mas sim apresentados na Tabela 22.

Tabela 22 – Taxa de Utilização da Memória da *GPU* - Microarquitetura *Pascal*

	Treino	Teste
CNTK	527 MB	569 MB
PyTorch	741 MB	739 MB
TensorFlow 1	619 MB	619 MB
TensorFlow 2	619 MB	619 MB

O código implementado com a biblioteca *CNTK* apresentou a menor taxa de utilização da memória da *GPU* durante as fases de treinamento e de teste. O código implementado com a biblioteca *PyTorch* apresentou a maior taxa de utilização durante a fase de treinamento e a fase de teste. É importante observar que o código implementado com a biblioteca *TensorFlow 1* e o código implementado com a biblioteca *TensorFlow 2* apresentaram as mesmas taxas de utilização da *GPU* em ambas as fases.

6.3.7 Análise Geral do Experimento 3

Na análise do Experimento 3 não aprofundaremos a discussão sobre *outliers*, pois esta discussão é semelhante para todos os experimentos tratados no presente capítulo, a discussão deste ponto é feita nas Subseções 6.1.8 e 6.2.7.

Neste experimento a biblioteca *CNTK* apresentou o menor tempo de execução durante a fase de treinamento enquanto a biblioteca *TensorFlow 2* apresentou o segundo menor tempo. Três bibliotecas apresentaram tempos de execução próximos (*CNTK*, *TensorFlow 1* e *TensorFlow 2*), enquanto a biblioteca *PyTorch* apresentou o maior tempo de execução, muito acima do tempo de execução das outras bibliotecas. A biblioteca *TensorFlow 2* apresentou o menor tempo de execução e a biblioteca *TensorFlow 1* apresentou o segundo menor tempo.

Ao observarmos as outras variáveis, é possível observar que a biblioteca *CNTK* apresenta uma potência e temperatura de *GPU* (Figuras 50 e 40) alta durante a fase de treinamento, enquanto possui uma taxa de utilização de *CPU* baixa (se comparado ao desempenho das outras bibliotecas). A biblioteca *PyTorch* apresenta o comportamento inverso, potência e temperatura da *GPU* baixas e taxa de utilização da *CPU* alta em ambas as fases. Apesar de haver diferença estatística em todas as variáveis, as bibliotecas *TensorFlow 1* e *TensorFlow 2* apresentaram potência e temperatura da *GPU* próximas em ambas as fases, porém elas apresentam taxas de utilização da *CPU* bem diferentes, a biblioteca *TensorFlow 1* utiliza mais o processamento da *CPU*.

Apesar da biblioteca *PyTorch* apresentar uma baixa potência e temperatura da *GPU*, a biblioteca apresentou a maior taxa de utilização da memória da *GPU* nas duas fases e a maior taxa da utilização da memória principal na fase de teste, esses fatores podem influenciar no tempo de execução. Já a biblioteca *CNTK* aprentou a maior taxa de utilização da memoria principal durante a fase de treino. As bibliotecas *TensorFlow 1* e *TensorFlow 2* apresentaram taxas de utilização de memória semelhantes em todos os casos.

A Tabela 23 apresenta a média de todas as populações do Experimento 3 com o intervalo de confiança.

Tabela 23 – Tabela Geral do Experimento 3 - Microarquitetura *Pascal*

Variável	*CNTK*	*PyTorch*	*TensorFlow 1*	*TensorFlow 2*
Tempo de Treinamento (s)	12,997173 ±0,027720	85,483023 ±0,160050	15,196989 ±0,034099	15,364843 ±0,034438
Tempo de Teste (s)	0,347223 ±0,001687	0,633910 ±0,002027	0,164300 ±0,001808	0,151466 ±0,001049
Temperatura da *GPU* Treino (°C)	51,02 ±0,64	37,98 ±0,03	41,53 ±0,20	42,06 ±0,17
Temperatura da *GPU* Teste (°C)	41,69 ±0,20	39,00 ±0,00	37,20 ±0,08	37,07 ±0,05
Potência da *GPU* Treino (W)	45,29 ±0,28	32,95 ±0,04	38,88 ±0,11	39,13 ±0,08
Potência da *GPU* Teste (W)	32,64 ±0,15	32,00 ±0,00	32,89 ±0,13	33,05 ±0,10
Utilização da *CPU* Treino (%)	18,95 ±0,31	32,30 ±1,56	24,82 ±0,19	20,21 ±0,36
Utilização da *CPU* Teste (%)	9,37 ±0,03	35,92 ±0,04	24,98 ±0,02	1,81 ±0,03
Memória Principal Treino (%)	24,9	18,4	17,5	17,8
Memória Principal Teste (%)	13,3	18,4	17,7	14,3
Memória *GPU* Treino (MB)	527	741	619	619
Memória *GPU* Teste (MB)	569	739	613	619

6.4 Análise de Desempenho na Microarquitetura *Turing*

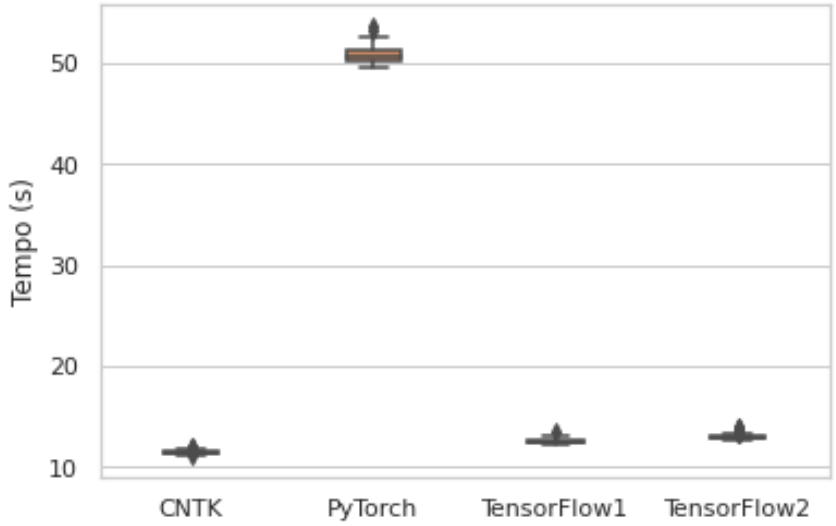
Nessa seção são analisados os dados da execução dos códigos no ambiente com *GPU* com a microarquitetura *Turing*.

6.4.1 Tempo de execução (Hipótese 1)

O resultado do *Teste de Friedman* para o tempo de execução de treinamento retornou um *p-value* de $1,703610931602483x10^{-63}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 54, que possui gráfico do

tempo em segundos, mostra com clareza um menor tempo de execução da fase de treinamento para o código que utiliza a biblioteca *CNTK*.

Figura 54 – Tempo de execução do treinamento (s) com a microarquitetura *Turing*



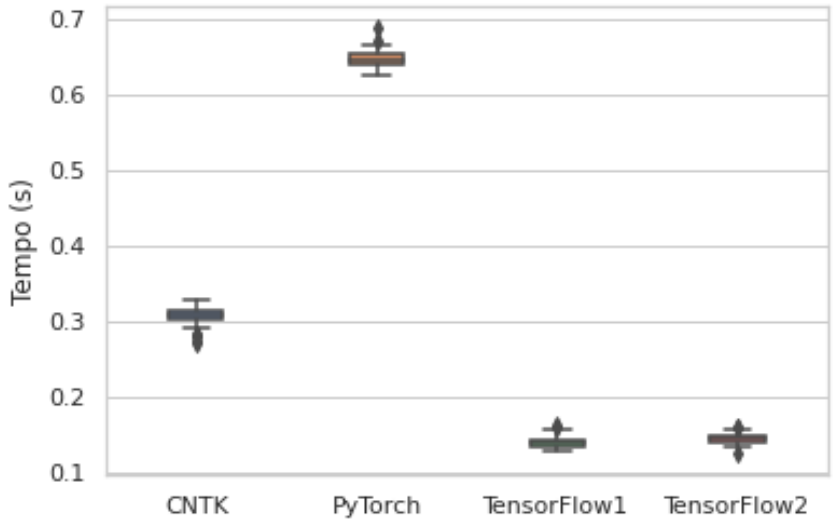
Fonte: Autoria Própria

Como na Figura 54 não foi possível identificar uma diferença entre o tempo de execução do código implementado com a biblioteca *TensorFlow 1* e o tempo de execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $1,3489339428091349x10^{-16}$, ou seja, há diferenças estatísticas entre as duas populações. O tempo de execução durante a execução do código implementado com *TensorFlow 1* apresenta uma média de $12,564867$ s e o tempo de execução do código implementado com *TensorFlow 2* apresenta uma média de $12,988177$ s.

O resultado do *Teste de Friedman* para o tempo de execução de teste retornou um *p-value* de $3,566283985047307x10^{-59}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 55, que possui gráfico do tempo em segundos, mostra com clareza um menor tempo de execução da fase de teste para os códigos que utilizam a bibliotecas *TensorFlow 1* e *TensorFlow 2*.

Como na Figura 55 não foi possível identificar uma diferença entre o tempo de execução do código implementado com a biblioteca *TensorFlow 1* e o tempo de execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $7,164688685993915x10^{-6}$, ou seja, há diferenças estatísticas entre as duas populações. O tempo de execução durante a execução do código implementado com *TensorFlow 1* apresenta uma média de 0.139995 s e o tempo de execução do código implementado com *TensorFlow 2* apresenta uma média de $0,144302$ s.

Figura 55 – Tempo de execução do teste (s) com a microarquitetura *Turing*

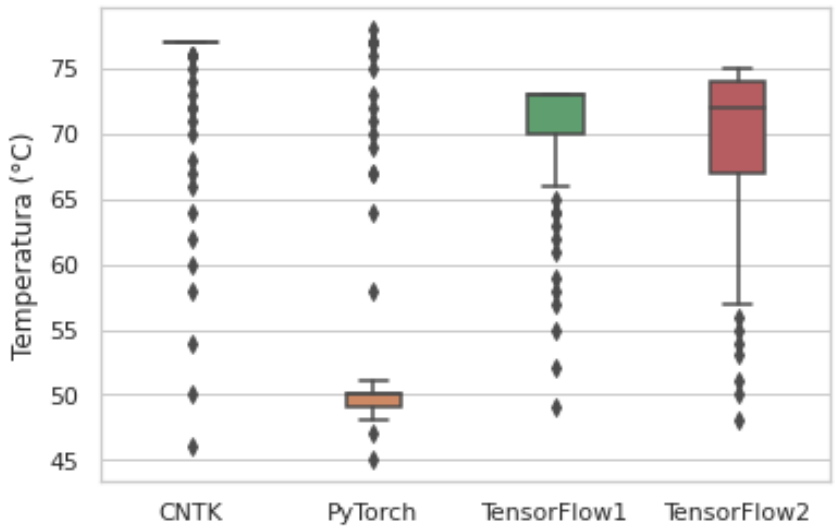


Fonte: Autoria Própria

6.4.2 Temperatura da *GPU* (Hipótese 2)

O resultado do *Teste de Friedman* para a temperatura da *GPU* durante a fase de treinamento retornou um *p-value* de $2,2631117305821824x10^{-50}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 56, que possui gráfico de temperatura em graus *Celsius*, mostra com clareza uma menor temperatura da *GPU* durante a fase de treinamento para o código que utiliza a biblioteca *PyTorch* e uma maior temperatura para o código que utiliza a biblioteca *CNTK*.

Figura 56 – Temperatura da *GPU* durante a fase de treinamento (°C) com a microarquitetura *Turing*

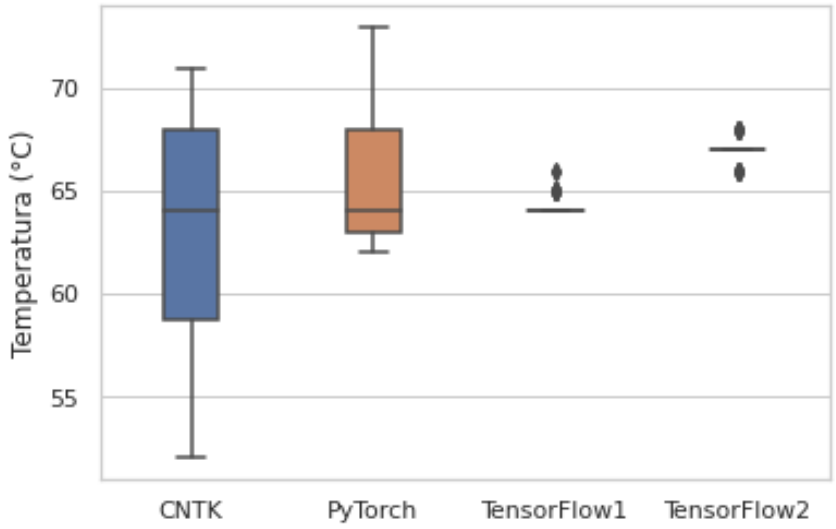


Fonte: Autoria Própria

Como na Figura 56 não foi possível identificar uma diferença entre a temperatura da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 1* e temperatura da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de 0.0001750762462930463, ou seja, há diferença estatística entre as duas populações. A temperatura da *GPU* durante a execução do código implementado com *TensorFlow 1* apresenta uma média de 70,41°C e a temperatura da *GPU* durante a execução do código implementado com *TensorFlow 2* apresenta uma média de 69,43°C.

O resultado do *Teste de Friedman* para a temperatura da *GPU* durante a fase de teste retornou um *p-value* de $9,505414868579314x10^{-15}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 57, que possui gráfico de temperatura em graus *Celsius*, mostra com clareza uma menor temperatura da *GPU* durante a fase de treinamento para os códigos que utilizam as bibliotecas *CNTK*, *TensorFlow 1* e *PyTorch*.

Figura 57 – Temperatura da *GPU* durante a fase de teste (°C) com a microarquitetura *Turing*



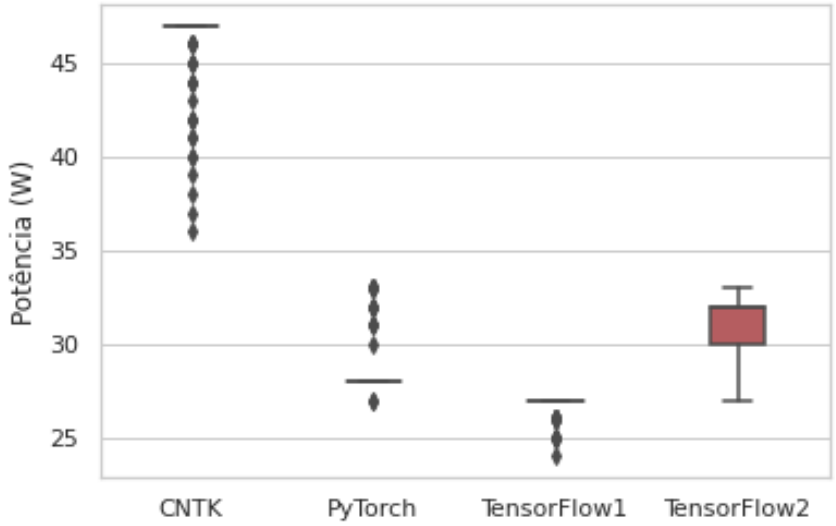
Fonte: Autoria Própria

Como na Figura 57 não foi possível identificar uma diferença entre a temperatura da *GPU* durante a execução do código implementado com a biblioteca *CNTK*, a temperatura da *GPU* durante a execução do código implementado com a biblioteca *PyTorch* e a temperatura da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 1*, foi aplicado o *Teste de Friedman* para verificar se há diferenças estatísticas entre as três populações. O *Teste de Friedman* retornou um *p-value* de 0.9655663763965059, ou seja, se considerarmos a hipótese de que não há diferença estatística entre essas três populações, essa hipótese não pode ser rejeitada.

6.4.3 Potência da *GPU* (Hipótese 3)

O resultado do *Teste de Friedman* para a potência da *GPU* durante a fase de treinamento retornou um *p-value* de $2,3168730892709692x10^{-61}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 58, que possui potência em *Watts*, mostra com clareza uma menor potência da *GPU* durante a fase de treinamento para o código que utiliza a biblioteca *TensorFlow 1* e uma maior potência para o código implementado com a biblioteca *CNTK*.

Figura 58 – Potência da *GPU* durante a fase de treinamento (W) com a microarquitetura *Turing*

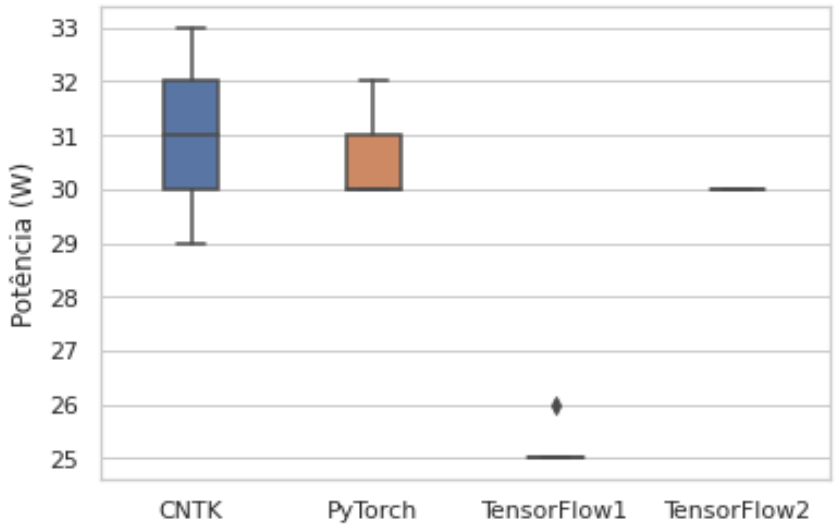


Fonte: Autoria Própria

O resultado do *Teste de Friedman* para a potência da *GPU* durante a fase de teste retornou um *p-value* de $3,48526762742578 66x10^{-48}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 51, que possui potência em *Watts* para cada um dos códigos, mostra com clareza uma menor potência da *GPU* durante a fase de teste para os códigos que utilizam as bibliotecas *CNTK*, *PyTorch* e *TensorFlow* e uma maior potência para o código implementado com a biblioteca *TensorFlow 1*

Como na Figura 59 não foi possível identificar uma diferença entre a potência da *GPU* durante a execução do código implementado com a biblioteca *CNTK*, a potência da *GPU* durante a execução do código implementado com a biblioteca *PyTorch* e a potência da *GPU* durante a execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Friedman* para verificar se há diferenças estatísticas entre as três populações. O *Friedman* retornou um *p-value* de $1,6199808120863737x10^{-12}$, ou seja, há diferença estatística entre as três populações. A potência da *GPU* durante a execução do código implementado com *CNTK* apresenta uma média de 31,34 W, a potência da *GPU* durante a execução do código implementado com *PyTorch* apresenta uma média de 30,58 W e a potência da *GPU* durante a execução do código implementado com *TensorFlow 2* apresenta uma média de 30 W.

Figura 59 – Potência da *GPU* durante a fase de teste (W) com a microarquitetura *Turing*



Fonte: Autoria Própria

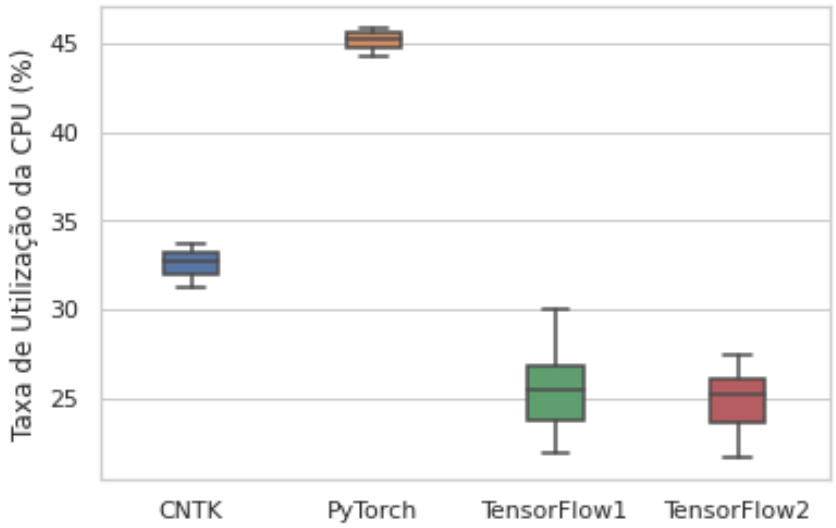
6.4.4 Taxa de Utilização da *CPU* (Hipótese 4)

O resultado do *Teste de Friedman* para a taxa de utilização da *CPU* durante a fase de treinamento retornou um *p-value* de $1,1778234440926679x10^{-53}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 60, que possui taxa de utilização da *CPU* em porcentagem, mostra com clareza uma menor taxa de utilização da *CPU* durante a fase de treinamento para os código que utilizam as bibliotecas *TensorFlow 1* e *TensorFlow 2* e uma maior taxa de utilização da *CPU* para o código implementado com a biblioteca *PyTorch*.

Como na Figura 60 não foi possível identificar uma diferença entre a taxa de utilização da *CPU* durante a execução do código implementado com a biblioteca *TensorFlow1* e a taxa de utilização da *CPU* durante a execução do código implementado com a biblioteca *TensorFlow 2*, foi aplicado o *Teste de Wilcoxon Pareado* para verificar se há diferenças estatísticas entre as duas populações. O *Teste de Wilcoxon Pareado* retornou um *p-value* de $2.0889355790168867x10^{-18}$, ou seja, há diferença estatística entre essas duas populações. A taxa média de utilização da *CPU* durante a execução do código implementado com a biblioteca *TensorFlow 1* é $25,29\%$ e a taxa média de utilização da *CPU* durante a execução do código implementado com a biblioteca *TensorFlow 2* é $24,91\%$

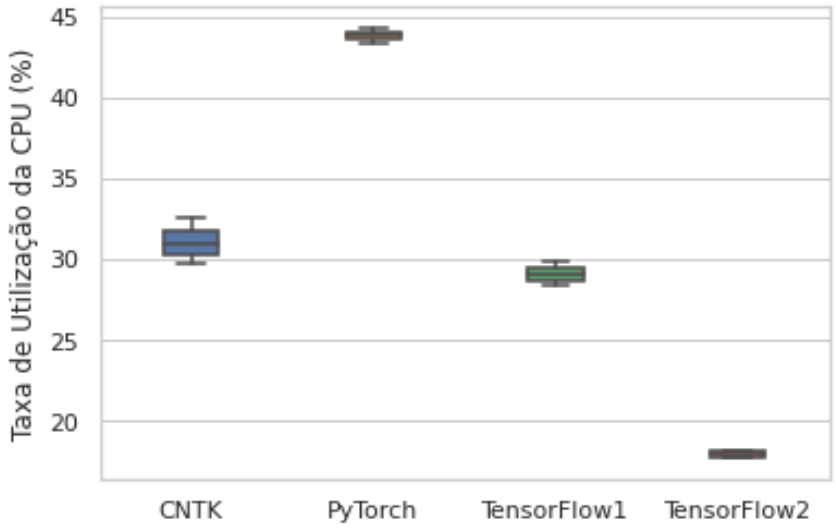
O resultado do *Teste de Friedman* para a taxa de utilização da *CPU* durante a fase de teste retornou um *p-value* de $9,948758346327588x10^{-65}$, portanto a hipótese H0 foi fortemente rejeitada e, consequentemente, a hipótese H1 não foi rejeitada. O gráfico da Figura 61, que possui taxa de utilização da *CPU* em porcentagem, mostra com clareza uma menor taxa de utilização da *CPU* durante a fase de teste para o código que utiliza a biblioteca *TensorFlow 2* e uma maior taxa de utilização da *CPU* para o código implementado com a biblioteca *PyTorch*.

Figura 60 – Taxa de utilização da *CPU* durante a fase de treinamento (%) com a microarquitetura *Turing*



Fonte: Autoria Própria

Figura 61 – Taxa de utilização da *CPU* durante a fase de teste (%) com a microarquitetura *Turing*



Fonte: Autoria Própria

6.4.5 Taxa de Utilização da Memória Principal (Hipótese 5)

A taxa de utilização da memória principal permaneceu constante em todas as execuções, por isso os dados não foram apresentados em gráficos, mas sim apresentados na Tabela 24.

O código implementado com a biblioteca *CNTK* apresentou a menor taxa de utilização da memória principal durante a execução de ambas as fases. O código implementado com a biblioteca *PyTorch* apresentou a maior taxa de utilização da memória principal durante a execução de ambas as fases.

Tabela 24 – Taxa de Utilização da Memória Principal - Microarquitetura *Turing*

	Treino	Teste
CNTK	16,0 %	16,0 %
PyTorch	24,1 %	24,5 %
TensorFlow 1	19,0 %	19,4 %
TensorFlow 2	23,2 %	23,7 %

6.4.6 Taxa de Utilização da Memória da *GPU* (Hipótese 6)

A taxa de utilização da memória da *GPU* permaneceu constante em todas as execuções, por isso os dados não foram apresentados em gráficos, mas sim apresentados na Tabela 25.

Tabela 25 – Taxa de Utilização da Memória da *GPU* - Microarquitetura *Turing*

	Treino	Teste
CNTK	529 MB	489 MB
PyTorch	815 MB	817 MB
TensorFlow 1	459 MB	459 MB
TensorFlow 2	677 MB	677 MB

O código implementado com a biblioteca *TensorFlow 1* apresentou a menor taxa de utilização da memória da *GPU* durante a fase de treinamento e a fase de teste. O código implementado com a biblioteca *PyTorch* apresentou a maior taxa de utilização durante a fase de treinamento e durante a fase de teste. É importante observar que para todos os códigos, a taxa de utilização da memória da *GPU* durante a fase de treinamento é próxima da taxa de utilização da memória da *GPU* durante a fase de teste.

6.4.7 Análise Geral do Experimento 4

Na análise do Experimento 4 não aprofundaremos a discussão sobre *outliers*, pois esta discussão é semelhante para todos os experimentos tratados no presente capítulo, a discussão deste ponto é feita nas Subseções 6.1.8 e 6.2.7.

Neste experimento a biblioteca *CNTK* apresentou o menor tempo de execução durante a fase de treinamento enquanto a biblioteca *TensorFlow 1* apresentou o segundo menor tempo. Três bibliotecas apresentaram tempos de execução próximos (*CNTK*, *TensorFlow 1* e *TensorFlow 2*), enquanto a biblioteca *PyTorch* apresentou o maior tempo de execução, muito acima do tempo de execução das outras bibliotecas. A biblioteca *TensorFlow 1* apresentou o menor tempo de execução e a biblioteca *TensorFlow 2* apresentou o segundo menor tempo. O desempenho das bibliotecas possui semelhanças com o desempenho das bibliotecas nos outros experimentos, porém com as bibliotecas *TensorFlow 1* e *TensorFlow 2* alternando o menor tempo de execução entre as duas.

A biblioteca *CNTK* novamente apresentou uma alta potência e temperatura da *GPU* se comparado as outras bibliotecas, porém apresentou uma taxa de utilização da *CPU* consideravelmente acima das bibliotecas *TensorFlow 1* e *TensorFlow 2* na fase de treinamento e próximo da taxa de utilização da biblioteca *TensorFlow 1* na fase de teste. A biblioteca *PyTorch* novamente apresentou uma baixa potência e temperatura da *GPU* na fase de treinamento - maior apenas que a potência e temperaturas geradas pela execução da *TensorFlow 1* - e uma alta taxa de utilização da *CPU* nas duas fases, porém na fase de teste a *PyTorch* apresentou comportamento próximo ao das outras bibliotecas.

Quanto ao consumo de memória, a biblioteca *PyTorch* apresentou a maior tacha de utilização em todos os casos. A biblioteca *CNTK* apresentou a menor taxa de utilização da memória principal em ambas as fases e apresentou taxas de utilização da memória da *GPU* próximas as taxas apresentadas pela biblioteca *TensorFlow 1*. Essas informações são ilustradas nas Tabelas 24 e 25.

A Tabela 26 apresenta a média de todas as populações do Experimento 4 com o intervalo de confiança.

Tabela 26 – Tabela Geral do Experimento 4 - Microarquitetura *Turing*

Variável	*CNTK*	*PyTorch*	*TensorFlow 1*	*TensorFlow 2*
Tempo de Treinamento (s)	11.475067 ±0.031500	50,819012 ±0,144075	12,564867 ±0,050894	12,988177 ±0,047565
Tempo de Teste (s)	0,307079 ±0,002003	0,647122 ±0,002073	0,139995 ±0,001454	0,144302 ±0,001120
Temperatura da *GPU* Treino (°C)	74,86 ±1,14	53,16 ±1,68	70,41 ±0,96	69,43 ±1,29
Temperatura da *GPU* Teste (°C)	62,97 ±1,10	65,61 ±0,69	64,19 ±0,09	67,00 ±0,09
Potência da *GPU* Treino (W)	45,97 ±0,47	28,60 ±0,31	26,72 ±0,12	31,07 ±0,31
Potência da *GPU* Teste (W)	31,34 ±0,26	30,58 ±0,15	25,01 ±0,02	30,00 ±0,00
Utilização da *CPU* Treino (%)	32,57 ±0,14	45,21 ±0,10	25,29 ±0,37	24,91 ±0,32
Utilização da *CPU* Teste (%)	31,02 ±0,16	43,78 ±0,06	29,13 ±0,09	17,96 ±0,03
Memória Principal Treino (%)	16,0	24,1	19,0	23,2
Memória Principal Teste (%)	16,0	24,5	19,4	23,7
Memória *GPU* Treino (MB)	529	815	459	677
Memória *GPU* Teste (MB)	489	817	459	677

6.5 Ameaças à Validade

Foram utilizados testes estatísticos como forma de mitigar vieses relacionados à conclusão a respeito das hipóteses estabelecidas (**validade de conclusão**). O primeiro foi o teste de *Kolmogorov-Smirnov*, utilizado para verificação da normalidade dos dados. Esta etapa foi necessária para a escolha do teste seguinte, relacionado à comparação dos algoritmos a serem utilizados, que, sendo a hipótese nula rejeitada pelo teste *KS*, utilizou-se dois testes não-paramétrico para amostras dependentes, o *Teste de Wilcoxon Pareado* e o *Teste de Friedman*. Desta forma, pode-se ter uma conclusão estatisticamente satisfatória, evitando a avaliação de desempenho apenas pela média dos dados amostrados.

Quanto a **validade interna**, nos experimentos 2, 3 e 4 o ambiente não foi totalmente monitorado e controlado, pois o *Google Colab* é um serviço em nuvem mantido e controlado

pela *Google Colab*. Consequentemente, não houve um monitoramento de todos os processos executados pelo sistema operacional durante a execução dos códigos, alguns processos podem ter influenciado ocasionalmente na execução dos códigos e causado *outliers*, isso foi mitigado para os tempos de execução eliminando a coleta da primeira amostra. A possibilidade de sobrecargas nos servidores da *Google LLC* também é considerada uma ameaça à validade, pois pode interferir no tempo de execução dos códigos executados. Para o outro ambiente de execução (PC utilizado para a execução do Experimento 1), todos os *softwares* não essenciais para o funcionamento do sistema operacional - incluindo a interface gráfica - foram desativados para reduzir a interferência no processamento dos algoritmos.

Apenas uma arquitetura de *CNN* foi utilizada, a *CNN LeNet-5*, o que faz com que o experimento tenha uma baixa variabilidade de *benchmarks* constituindo uma ameaça à **validade externa**. É importante que posteriormente sejam utilizados outros modelos *CNN* em experimentos semelhantes.

Os códigos utilizados podem ter erros ocasionados por falha humana constituindo uma ameaça à **validade de construção**. Para mitigar essa ameaça, os códigos utilizados foram construídos a partir de modificações de códigos já utilizados pela comunidade de desenvolvedores e cientistas e construídos implementando uma *CNN* clássica (*LeNet-5*) treinando e testando o *MNIST dataset*.

6.6 Considerações Finais do Capítulo

Neste capítulo foram apresentados os resultados dos quatro experimentos. O Experimento 1 realizado em um ambiente computacional acelerado por uma *GPU* com microarquitetura *Maxwell*, o Experimento 2 realizado em um ambiente computacional acelerado por uma *GPU* com microarquitetura *Kepler*, o Experimento 3 realizado em um ambiente computacional acelerado por uma *GPU* com microarquitetura *Pascal* e o Experimento 4 realizado em um ambiente computacional acelerado por uma *GPU* com microarquitetura *Turing*.

Nas Figuras 62 e 63 são ilustrados dois gráficos gerais, um ilustrando a fase de treino e um ilustrando a fase de teste, do tempo de execução[1]. A biblioteca *PyTorch* apresentou o pior desempenho. Como características observadas a partir dos dados de sua execução, a biblioteca possui uma alta utilização da *CPU*, uma baixa utilização da *GPU* e um alta utilização das memórias. É possível que seu baixo desempenho seja consequência da combinação dessas características, a baixa utilização da *GPU* tem como consequência um menor nível de paralelismo.

As bibliotecas *CNTK*, *TensorFlow 1* e *TensorFlow 2* apresentaram desempenhos próximos alternando entre o melhor desempenho. Deve-se destacar que as bibliotecas *TensorFlow*

[1] Os gráficos das Figuras 62 e 63 são gráficos gerais que ilustram diferentes cenários em uma mesma imagem, os gráficos foram plotados para facilitar a visualização geral dos experimentos e não devem ser utilizados para uma comparação direta de valores.

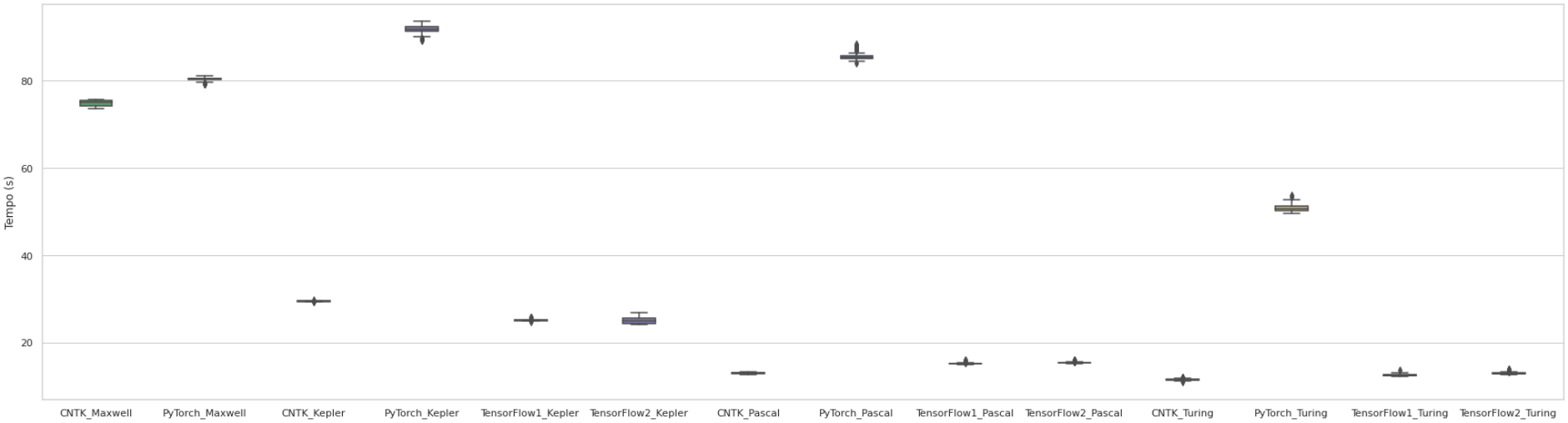
1 e *TensorFlow 2* apresentaram desempenho ainda mais semelhantes entre elas, inclusive no Experimento 2 não houve diferença estatística entre o tempo de execução das duas bibliotecas durante as duas fases (treinamento e teste). Como características observadas a partir dos dados de execução da biblioteca *CNTK* estão a alta utilização da *GPU* e a baixa utilização da *CPU*. Como característica observada a partir dos dados de execução das bibliotecas *TensorFlow 1* e *TensorFlow 2* está a variação da utilização do *hardware* de acordo com a fase e com a microarquitetura, outra observação que deve ser feita é que apesar das duas bibliotecas apresentarem desempenho semelhantes, elas apresentaram comportamentos de utilização do *hardware* diferentes que variaram em cada experimento.

Os resultados sobre tempo de execução da biblioteca *PyTorch* e *TensorFlow 1* contrariam os dados obtidos durante o estudo preliminar (FLORENCIO et al., 2019). Neste estudo a biblioteca *PyTorch* apresentou menor tempo de execução em um ambiente computacional acelerado por uma *GPU* com microarquitetura *Pascal*, porém no estudo preliminar a *CNN* foi treinada por apenas uma época. Pode-se levantar a hipótese de que o tempo de execução da primeira época de treinamento de uma *CNN* implementada com *TensorFlow 1* seja maior do que o tempo de execução da primeira época de treinamento de uma *CNN* implementada com *PyTorch*, porém não se pode descartar a hipótese da existência de erros durante a implementação dos códigos do estudo preliminar, uma vez que é um estudo inicial realizado por autores com pouca experiência com as ferramentas. Quanto a utilização do *hardware* por parte da biblioteca *PyTorch*, o estudo preliminar apresentou resultados que mostram uma baixa utilização da *GPU* e alta utilização da *CPU* coincidindo com os resultados dos experimentos apresentados neste capítulo.

Uma pequena mudança de contexto pode alterar significativamente os dados obtidos neste estudo. Como citado na Subseção 2.3.1, a biblioteca *CNTK* é otimizada para os serviços da *Microsoft Azure* e nenhum dos experimentos foi realizado em um ambiente de execução da *Microsoft*, ou seja, há a possibilidade da biblioteca apresentar desempenho superior às outras bibliotecas em um ambiente do serviço *Microsoft Azure*. Como citado na Subseção 4.5, as bibliotecas são ferramentas de alto nível e estão em contínuo desenvolvimento, portanto atualizações futuras de qualquer ferramenta da pilha de *software* utilizada nesse estudo podem alterar consideravelmente o valor das variáveis.

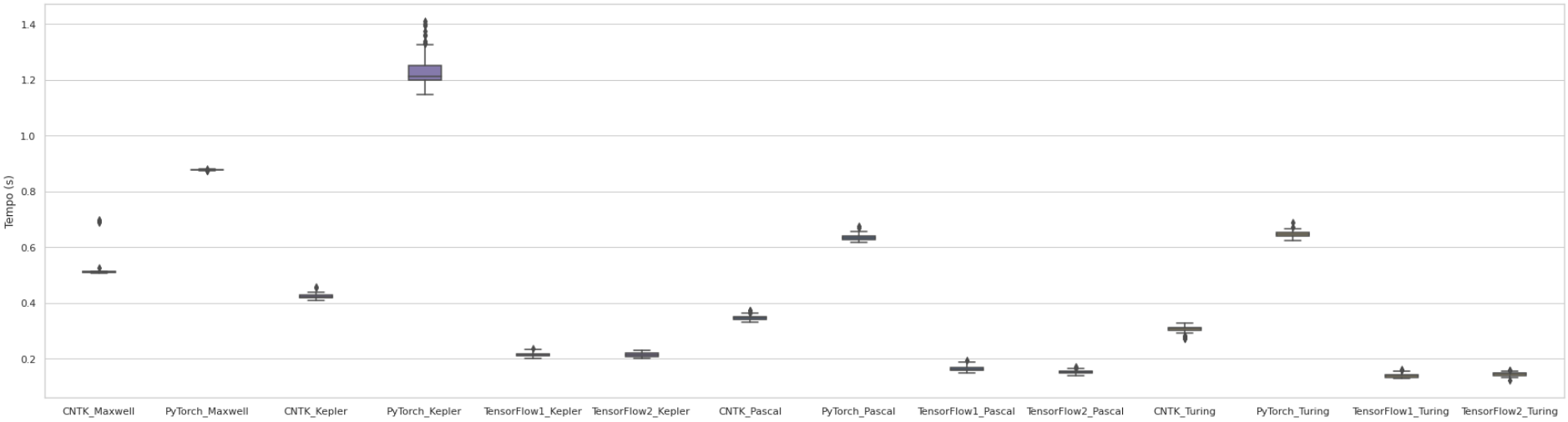
É importante destacar que, como mostrado em Leiserson et al. (2020), a linguagem *Python* não é uma linguagem eficiente se comparada a linguagem *C*. Em uma multiplicação entre duas matrizes de 4096 x 4096, a linguagem *C* pode apresentar um tempo de execução 47 vezes menor do que a linguagem *Python*. Essa relação de desempenho pode ser alterada caso os códigos e/ou outras ferramentas da pilha de *software* sejam alterados para explorar recursos específicos do *hardware* como, por exemplo, a capacidade de paralelização.

Figura 62 – Tempo de execução da fase de treinamento (s) - Gráfico Geral



Fonte: Autoria Própria

Figura 63 – Tempo de execução da fase de teste (s) - Gráfico Geral



Fonte: Autoria Própria

7 Conclusão

Neste capítulo são apresentadas as contribuições desta dissertação e de outros trabalhos desenvolvidos durante o Mestrado, as limitações, trabalhos futuros e considerações finais.

7.1 Contribuições

A principal contribuição deste estudo consiste na condução de quatro experimentos comparando o desempenho de bibliotecas para *CNN* em sistemas computacionais com diferentes microarquiteturas de *GPU* e o levantamento de dafizeram parte do desenvolvimentodos sobre a utilização do *hardware* durante a execução dos códigos implementados com as bibliotecas. Além disso outras contribuições pode ser colocadas:

- Mapeamento sistemático utilizado para identificar e sistematizar os principais *benchmarks* e bibliotecas já avaliadas. Ressalta-se que os resultados foram submetidos e aprovados no *EATIS 2020 10th Euro American Conference on Telematics and Information Systems* sob o título *Convolutional Neural Network Libraries Benchmarking: A Systematic Mapping*.
- Um estudo do impacto da *API Keras* sobre o desempenho das bibliotecas *CNTK*, *TensorFlow 1.15*, cujo resultado mostrou que a *API* impacta negaticamente no desempenho de todas as bibliotecas testadas.
- A formalização de uma metodologia experimental baseada na abordagem *GQM* para comparar desempenho de bibliotecas de *CNN* em diferentes microarquiteturas. Uma formalização da metodologia experimental anterior foi realizada no estudo preliminar Florencio et al. (2019), porém nesta dissertação sofreu algumas mudanças para adaptar aos ambientes de execução.
- Os resultados que mostram um baixo desempenho da biblioteca *PyTorch* com relação as outras bibliotecas testadas.

- Os resultados que evidenciam que deficiências de desempenho de alguma biblioteca pode estar ligada a baixa capacidade de utilização da *GPU* e o alto consumo de memória.

Como consequência destas contribuições, conseguimos responder às questões de pesquisa elaboradas no início do trabalho:

- Q1: Há diferenças de desempenho entre as bibliotecas? Sim. Os experimentos mostraram que a biblioteca *PyTorch* apresenta um desempenho inferior as outras três em todos os casos. As bibliotecas *CNTK*, *TensorFlow 1* e *TensorFlow 2* alternam entre os três melhores desempenhos. No Experimento 2 as bibliotecas *TensorFlow 1* e *TensorFlow 2* não apresentaram diferenças estatísticas no tempo de execução
- Q2: Quais são os *benchmarks* disponíveis para a avaliação de desempenho das bibliotecas? Através do mapeamento sistemático foi possível identificar alguns dos mais usados. Entre os *datasets* foram mais frequentes o *CIFAR-10*, o *MNIST* e o *ImageNet*. Entre as arquiteturas de *CNN* se destacaram a *AlexNet*, *LeNet* e *ResNet-50*.
- Q3: Quais são as causas de uma possível diferença de desempenho entre as bibliotecas? Os experimentos evidenciaram que a biblioteca com pior desempenho *PyTorch* tem uma baixa utilização da *GPU* e um alto consumo de memória.

Ao longo do curso do mestrado, também existiram outras contribuições que não estão dentro do escopo desta dissertação como:

- O desenvolvimento de um estudo experimental preliminar que pode ser visto em Florencio et al. (2019)
- O desenvolvimento de um estudo sobre detecção de intrusão usando um dispositivo de baixa potência (de Almeida Florencio et al., 2018). Esse estudo foi realizado no inicio do mestrado que mostrou a possibilidade de um detector de intrusão utilizando redes *MLP* (*Multi Layer Perceptron*) em dispositivos de baixa potência com a finalidade de detectar intrusões de rede.
- O desenvolvimento de um mapeamento científico e tecnológico sobre Redes Neurais Convolucionais em Sistemas Embarcados. Ressalta-se que os resultados foram submetidos e aprovados no *EATIS 2020 10th Euro American Conference on Telematics and Information Systems* sob o título *Convolutional Neural Network on Embedded System: A technological and scientific mapping*.

7.2 Limitações

Inicialmente os experimentos seriam realizados utilizando o serviço *Google Cloud*, porém a *Google* não liberou o acesso as *GPUs* o que acabou inviabilizando a realização dos experimentos neste serviço, também foram solicitadas cotas na *Amazon AWS* que foram negadas. Portanto o Experimento 1 foi realizado em um *PC* e os demais experimentos foram realizados na plataforma *Google Colab* (GOOGLE LLC, 2020b).

As bibliotecas *TensorFlow 1* e *TensorFlow 2* apresentaram incompatibilidade com a *GPU* presente no *PC*, a *NVIDIA MX130*, o que tornou impossível a execução dos códigos implementados com estas bibliotecas.

Quanto ao ambiente *Google Colab* há dois problemas. Por ser baseado no ambiente *Jupyter*, ele não permite a execução simultânea de dois terminais, ou seja, os comandos para coleta de dados foram inseridos no meio dos códigos. O outro problema é que o *Google Colab* não imprime na tela os dados de processamento de cada tarefa executada na *GPU* e *CPU*, por isso foram utilizados dados gerais do processamento de cada dispositivo (*CPU* e *GPU*).

7.3 Trabalhos Futuros

Há muitos pontos a serem explorados neste tema, por exemplo, a realização de estudos em outras microarquiteturas de *GPU*, é recomendada a realização de outros estudos de desempenho em outras microarquiteturas de *GPU* como as microarquiteturas *Volta* e *Ampere*, não estudadas nesta dissertação.

Existem outras arquiteturas computacionais aprimoradas para o processamento de *CNN* como *FPGA*, *TPU* (*Tensor Processing Unit*) e dispositivos de baixa potência como *Intel Neural Computer Stick* (*NCS*). Como mostrado no Capítulo 3, há uma carência de trabalhos que avaliem o desempenho de biblioteca em dispositivos de baixa potência. Portanto, como trabalhos futuros, recomenda-se avaliar as bibliotecas em outras arquiteturas computacionais. Também é recomendado avaliações de desempenho com outros modelos *CNNs*, avaliações que testem o desempenho dos principais métodos de cada biblioteca e avaliações que utilizem o *batch size* com valores resultantes da potência de 2, 64 e 128.

É importante que trabalhos futuros apresentem cálculos do número de operações do modelo LeNet-5 em *Python* e em outras linguagens de programação de mais baixo nível - como a linguagem *C* - a fim de mostrar o impacto do uso da linguagem *Python* no desempenho da *CNN*.

Trabalhos futuros devem considerar a realização de testes de correlação entre o tempo de execução e a taxa de utilização de algum dos componentes do *hardware*, esses testes poderiam rejeitar ou não hipóteses que podem ser levantadas através dos resultados dos experimentos, por exemplo, se há correlação entre a taxa de utilização da *GPU* com o tempo de execução.

7.4 Considerações Finais

O principal objetivo deste trabalho é avaliar o desempenho das bibliotecas *CNTK*, *PyTorch*, *TensorFlow 1* e *TensorFlow 2*, porém isso exigiu outras etapas como: identificar *benchmarks* e bibliotecas utilizadas em outras avaliações de desempenho de bibliotecas, e verificar a forma mais adequada de implementar os modelos *CNN*. A partir dessas necessidades foram realizados um mapeamento sistemático apresentado no Capítulo 3 e um estudo que avalia o impacto da *API Keras* no desempenho das bibliotecas *CNTK*, *TensorFlow 1* e *TensorFlow 2* apresentado no Capítulo 4. O estudo mostrou que a *API Keras* impacta negativamente no desempenho das bibliotecas.

O resultado do estudo comparativo mostrou que a biblioteca *PyTorch* possui o pior desempenho e evidenciou que as causas para a sua deficiência são o alto uso de memória e a falta a baixa utilização da *GPU*, porém essas deficiências devem ser melhor investigadas em estudos futuros que levem em consideração a correlação de dados.

Referências

ABADI, M. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Software available from tensorflow.org. Acessado em 8 Set. 2018. Disponível em: <https://www.tensorflow.org/>. Citado na página 41.

AGHDAM, E. J. H. a. H. H. *Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification*. 1. ed. [S.l.]: Springer International Publishing, 2017. ISBN 978-3-319-57549-0, 978-3-319-57550-6. Citado na página 33.

AMAZON WEB SERVICE. *AWS and NVIDIA*. Amazon, 2020. Acessado em 23 Jun. 2020. Disponível em: <https://aws.amazon.com/pt/nvidia/>. Citado na página 21.

ASAADI, H.; CHAPMAN, B. Comparative study of deep learning framework in hpc environments. In: *2017 New York Scientific Data Summit (NYSDS)*. [S.l.: s.n.], 2017. p. 1–7. Citado 7 vezes nas páginas 46, 50, 52, 54, 55, 57 e 58.

BAHRAMPOUR, S. et al. Comparative study of caffe, neon, theano, and torch for deep learning. *CoRR*, abs/1511.06435, 2015. Disponível em: <http://arxiv.org/abs/1511.06435>. Citado na página 22.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. In: *Encyclopedia of Software Engineering*. [S.l.]: Wiley, 1994. Citado 5 vezes nas páginas 23, 24, 42, 60 e 76.

BRAGA, A. de P.; CARVALHO, A. P. de Leon F. de; LUDERMIR, T. B. *Redes Neurais Artificiais*: teoria e aplicações. 2nd. ed. Rio de Janeiro, RJ, BRA: LTC - Livros Técnicos e Científicos Editora Ltda., 2012. ISBN 978-85-216-1564-4. Citado na página 28.

BUCK, I. Gpu computing: Programming a massively parallel processor. In: *International Symposium on Code Generation and Optimization (CGO'07)*. [S.l.: s.n.], 2007. p. 17–17. Citado na página 33.

CHOLLET, F. et al. *Keras*. [S.l.]: GitHub, 2015. <https://github.com/fchollet/keras>, note=Acessado em 14 Jun. 2020. Citado na página 40.

CHRISBASOGLU. *The Microsoft Cognitive Toolkit*. 2018. Acessado em 15 Jul. 2018. Disponível em: <https://docs.microsoft.com/en-us/cognitive-toolkit/>. Citado 2 vezes nas páginas 39 e 40.

de Almeida Florencio, F. et al. Intrusion detection via mlp neural network using an arduino embedded system. In: *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*. [S.l.: s.n.], 2018. p. 190–195. Citado na página 129.

DENG, J. et al. ImageNet: A Large-Scale Hierarchical Image Database. In: *CVPR09*. [S.l.: s.n.], 2009. Citado na página 51.

DENG, L. A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA Transactions on Signal and Information Processing*, Cambridge University Press, January 2014. Disponível em: <https://www.microsoft.com/en-us/research/publication/

a-tutorial-survey-of-architectures-algorithms-and-applications-for-deep-learning/>. Citado na página 26.

DERTAT, A. *Applied Deep Learning - Part 4*: Convolutional neural networks. Towards Data Science, 2017. Disponível em: <https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2>. Citado na página 30.

DU, X. et al. Comparative study of distributed deep learning tools on supercomputers. In: VAIDYA, J.; LI, J. (Ed.). *Algorithms and Architectures for Parallel Processing*. Cham: Springer International Publishing, 2018. p. 122–137. ISBN 978-3-030-05051-1. Citado 7 vezes nas páginas 46, 50, 52, 54, 55, 57 e 58.

ELSEVIER B.V. *Scopus*. 2018. Acessado em 1 Dez. 2018. Disponível em: <https://www.scopus.com/>. Citado na página 20.

FAUSETT, L. (Ed.). *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Upper Saddle River, NJ, EUA: Prentice-Hall, Inc., 1994. ISBN 0-13-334186-0. Citado na página 27.

FLORENCIO, F. *The Impact of Using Keras on TensorFlow Performance*: The files. 2020. Acessado em 08 Abr. 2020. Disponível em: <https://github.com/florencioufs/kerastfexperiment>. Citado na página 63.

FLORENCIO, F. *Libraries Performance*: The files. 2020. Acessado em 28 Jun. 2020. Disponível em: <https://github.com/florencioufs/libraries-performance>. Citado na página 81.

FLORENCIO, F. et al. Performance analysis of deep learning libraries: Tensorflow and pytorch. *Journal of Computer Science*, v. 15, n. 6, p. 785–799, 2019. Disponível em: <https://thescipub.com/abstract/10.3844/jcssp.2019.785.799>. Citado 5 vezes nas páginas 23, 76, 125, 128 e 129.

FONNEGRA, R. D.; BLAIR, B.; DíAZ, G. M. Performance comparison of deep learning frameworks in image classification problems using convolutional and recurrent networks. In: *2017 IEEE Colombian Conference on Communications and Computing (COLCOM)*. [S.l.: s.n.], 2017. p. 1–6. Citado 8 vezes nas páginas 47, 50, 52, 54, 55, 56, 57 e 58.

FRANCO, A. da Costa e S. *On deeply learning features for automatic person image re-identification*. Tese (Doutorado) — Programme of Post-graduation in Mechatronics Federal University of Bahia, Salvador, BA, Brazil, 2016. Citado 4 vezes nas páginas 26, 28, 29 e 30.

FUKUSHIMA, K. Self-organization of a neural network which gives position-invariant response. In: *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1979. (IJCAI'79), p. 291–293. ISBN 0-934613-47-8. Disponível em: <http://dl.acm.org/citation.cfm?id=1624861.1624928>. Citado na página 26.

GAZAR, M. *LeNet-5 in 9 lines of code using Keras*. Medium, 2018. Acessado em 08 Abr. 2020. Disponível em: <https://medium.com/@mgazar/lenet-5-in-9-lines-of-code-using-keras-ac99294c8086>. Citado 8 vezes nas páginas 63, 65, 81, 141, 142, 143, 144 e 145.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016. Acessado em 8 Jan. 2019. Disponível em: <http://www.deeplearningbook.org>. Citado 5 vezes nas páginas 19, 20, 26, 28 e 29.

GOOGLE BRAIN TEAM. *A Guide to TF Layers: Building a Convolutional Neural Network | TensorFlow*. 2018. Acessado em 1 Jul. 2018. Disponível em: <https://www.tensorflow.org/tutorials/layers>. Citado 2 vezes nas páginas 29 e 30.

GOOGLE BRAIN TEAM. *TensorFlow*. 2018. Acessado em 1 Mai. 2018. Disponível em: <https://www.tensorflow.org/>. Citado na página 41.

GOOGLE LLC. *Google Trends*. 2018. Acessado em 7 Dez. 2018. Disponível em: <https://trends.google.com.br/>. Citado na página 20.

GOOGLE LLC. *Colaboratory*. 2020. Acessado em 03 Abr. 2020. Disponível em: <https://research.google.com/colaboratory/faq.html>. Citado na página 61.

GOOGLE LLC. *Google Colaboratory*. Google, 2020. Acessado em 11 Jun. 2020. Disponível em: <https://colab.research.google.com/>. Citado 4 vezes nas páginas 24, 75, 77 e 130.

GOOGLE LLC. *Keras overview*: Tensorflow core. 2020. Acessado em 14 Abr. 2020. Disponível em: <https://www.tensorflow.org/guide/keras/overview>. Citado 2 vezes nas páginas 59 e 73.

GOOGLE LLC. *Migrate your TensorFlow 1 code to TensorFlow 2: TensorFlow Core*. 2020. Acessado em 14 Abr. 2020. Disponível em: <https://www.tensorflow.org/guide/migrate>. Citado 3 vezes nas páginas 63, 65 e 81.

GOOGLE LLC. *Module: tf.keras*. 2020. Acessado em 14 Jun. 2020. Disponível em: <https://www.tensorflow.org/api_docs/python/tf/keras>. Citado na página 40.

HARRIS, M. *5 Things You Should Know About the New Maxwell GPU Architecture*. NVIDIA Corporation, 2014. Acessado em 30 Out. 2019. Disponível em: <https://devblogs.nvidia.com/5-things-you-should-know-about-new-maxwell-gpu-architecture/>. Citado na página 37.

HARRIS, M. *Maxwell: The Most Advanced CUDA GPU Ever Made*. NVIDIA, 2014. Acessado em 22 Jun. 2020. Disponível em: <https://developer.nvidia.com/blog/maxwell-most-advanced-cuda-gpu-ever-made/>. Citado na página 37.

HEBB, D. O. *The organization of behavior: A neuropsychological theory*. New York: Wiley, 1949. Hardcover. ISBN 0-8058-4300-0. Citado na página 31.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X. Citado na página 35.

Hisamoto, D. et al. Finfet-a self-aligned double-gate mosfet scalable to 20 nm. *IEEE Transactions on Electron Devices*, v. 47, n. 12, p. 2320–2325, Dec 2000. ISSN 1557-9646. Citado na página 38.

KERAS TEAM. *Keras documentation: About Keras*. 2020. Acessado em 14 Jun. 2020. Disponível em: <https://keras.io/about/>. Citado na página 40.

KIM, H. et al. Performance analysis of cnn frameworks for gpus. In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. [S.l.: s.n.], 2017. p. 55–64. Citado 7 vezes nas páginas 47, 50, 52, 54, 55, 56 e 57.

KITCHENHAM, B. *Procedures for Performing Systematic Reviews*. Department of Computer Science, Keele University, UK, 2004. Citado na página 45.

KRIZHEVSKY, A.; NAIR, V.; HINTON, G. Cifar-10 (canadian institute for advanced research). 2010. Disponível em: <http://www.cs.toronto.edu/~kriz/cifar.html>. Citado na página 50.

KRUCHININ, D. et al. Comparison of deep learning libraries on the problem of handwritten digit classification. In: KHACHAY, M. Y. et al. (Ed.). *Analysis of Images, Social Networks and Texts*. Cham: Springer International Publishing, 2015. p. 399–411. ISBN 978-3-319-26123-2. Citado 5 vezes nas páginas 47, 50, 52, 55 e 57.

KUEBLER, A.; LAKE, J. *NVIDIA Kepler GPU Microarchitecture*. 2020. Acessado em 14 Jun. 2020. Disponível em: <https://poli.cs.vsb.cz/edu/apps/cuda/NVIDIA_Kepler_microarchitecture.pdf>. Citado na página 35.

LECUN, Y. et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, v. 86, n. 11, p. 2278–2324, Nov 1998. ISSN 0018-9219. Citado 7 vezes nas páginas 20, 32, 33, 62, 64, 81 e 84.

LECUN, Y.; CORTES, C. MNIST handwritten digit database. 2010. Disponível em: <http://yann.lecun.com/exdb/mnist/>. Citado 7 vezes nas páginas 32, 33, 51, 62, 64, 81 e 84.

LEISERSON, C. E. et al. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, American Association for the Advancement of Science, v. 368, n. 6495, 2020. ISSN 0036-8075. Disponível em: <https://science.sciencemag.org/content/368/6495/eaam9744>. Citado na página 125.

LIN, Z. et al. Benchmarking deep learning frameworks with fpga-suitable models on a traffic sign dataset. In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. [S.l.: s.n.], 2018. p. 1197–1203. ISSN 1931-0587. Citado 6 vezes nas páginas 47, 50, 52, 54, 55 e 57.

LINDHOLM, E. et al. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, v. 28, n. 2, p. 39–55, March 2008. ISSN 0272-1732. Citado na página 34.

LIU, L. et al. Benchmarking deep learning frameworks: Design considerations, metrics and beyond. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. [S.l.: s.n.], 2018. p. 1258–1269. ISSN 2575-8411. Citado 6 vezes nas páginas 47, 50, 52, 54, 55 e 57.

MARSLAND, S. *Machine Learning: An Algorithmic Perspective, Second Edition*. 2nd. ed. [S.l.]: Chapman & Hall/CRC, 2014. ISBN 1466583282, 9781466583283. Citado 2 vezes nas páginas 31 e 32.

MCCULLOCH, W.; PITTS, W. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, v. 5, p. 127–147, 1943. Citado 3 vezes nas páginas 26, 27 e 28.

MICROSOFT CORPORATION. *CNTK*: Tutorials. 2019. Acessado em 26 Mai. 2020. Disponível em: <https://github.com/microsoft/CNTK/tree/master/Tutorials>. Citado 4 vezes nas páginas 63, 81, 82 e 146.

MUJTABA, H. *NVIDIA Expected to launch Eight New 28nm Kepler GPU's in April 2012*. Wccftech, 2012. Acessado em 21 Jun. 2020. Disponível em: <https://wccftech.com/nvidia-expected-launch-28nm-kepler-gpus-april-2012/>. Citado na página 35.

NILSBACK, M.-E.; ZISSERMAN, A. Delving deeper into the whorl of flower segmentation. *Image and Vision Computing*, 2009. Citado na página 50.

NVIDIA CORPORATION. *NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256*. 1999. Acessado em 2 Jul. 2018. Disponível em: <https://bit.ly/21jWu1e>. Citado na página 33.

NVIDIA CORPORATION. *NVIDIA GeForce® 8800GT*. 2007. Acessado em 30 Jul. 2018. Disponível em: <https://bit.ly/2xa5oHy>. Citado na página 34.

NVIDIA CORPORATION. *NVIDIA Tesla*: Gpu accelerators. 2014. Acessado em 13 Jun. 2020. Disponível em: <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/TeslaK80-datasheet.pdf>. Citado na página 83.

NVIDIA CORPORATION. *NVIDIA's Next Generation CUDA Compute Architecture*: Kepler gk110/210. 2014. Acessado em 13 Jun. 2020. Disponível em: <https://www.techpowerup.com/gpu-specs/docs/nvidia-gk110-210-compute-architecture.pdf>. Citado 2 vezes nas páginas 36 e 83.

NVIDIA CORPORATION. *Tesla K80 GPU Accelerator*: Board specification. 2015. Acessado em 13 Jun. 2020. Disponível em: <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/Tesla-K80-BoardSpec-07317-001-v05.pdf>. Citado na página 83.

NVIDIA CORPORATION. *NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU*. 2017. Acessado em 15 Mar. 2020. Disponível em: <https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/>. Citado na página 37.

NVIDIA CORPORATION. *NVIDIA Tesla P4 GPU Accelerator*: Product brief. 2017. Acessado em 09 Abr. 2020. Disponível em: <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/Tesla-P4-Product-Brief.pdf>. Citado 2 vezes nas páginas 63 e 83.

NVIDIA CORPORATION. *NVIDIA CUDA C Programming Guide*. Version 9.2. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 2018. Disponível em: <https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>. Citado na página 35.

NVIDIA CORPORATION. *NVIDIA Turing GPU Architecture*: Graphics reinvented. 2018. Acessado em 22 Jun. 2020. Disponível em: <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>. Citado 2 vezes nas páginas 38 e 39.

NVIDIA CORPORATION. *Microsoft Azure*. 2019. Acessado em 23 Jun. 2020. Disponível em: <https://www.nvidia.com/pt-br/data-center/gpu-cloud-computing/microsoft-azure/>. Citado na página 21.

NVIDIA CORPORATION. *NVIDIA T4*: Tensor core gpu. 2019. Acessado em 14 Jun. 2020. Disponível em: <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf>. Citado na página 83.

NVIDIA CORPORATION. *NVIDIA Tesla P4*: Inference accelerator. 2019. Acessado em 09 Abr. 2020. Disponível em: <https://images.nvidia.com/content/pdf/tesla/184457-Tesla-P4-Datasheet-NV-Final-Letter-Web.pdf>. Citado 2 vezes nas páginas 63 e 83.

NVIDIA CORPORATION. *NVIDIA T4 70W Low Profile PCIe GPU Accelerator*: Product brief. 2020. Acessado em 14 Jun. 2020. Disponível em: <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-product-brief.pdf>. Citado na página 83.

NVIDIA CORPORATION. *NVIDIA Tensor Cores*: Aceleração sem precedentes para hpc e ai. NVIDIA, 2020. Acessado em 22 Jun. 2020. Disponível em: <https://www.nvidia.com/pt-br/data-center/tensor-cores/>. Citado na página 38.

NYLAND, L.; JONES, S. *Inside Kepler*. GTC On-Demand, 2012. Acessado em 21 Jun. 2020. Disponível em: <http://on-demand.gputechconf.com/gtc/2012/presentations/S0642-GTC2012-Inside-Kepler.pdf>. Citado na página 35.

OWENS, J. D. et al. Gpu computing. *Proceedings of the IEEE*, v. 96, n. 5, p. 879–899, May 2008. ISSN 0018-9219. Citado na página 34.

PARVAT, A. et al. A survey of deep-learning frameworks. In: *2017 International Conference on Inventive Systems and Control (ICISC)*. [S.l.: s.n.], 2017. p. 1–7. Citado na página 41.

PASZKE, A. et al. Automatic differentiation in pytorch. 2017. Citado na página 40.

PETERSEN, K.; VAKKALANKA, S.; KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, v. 64, p. 1 – 18, 2015. ISSN 0950-5849. Disponível em: <https://bit.ly/2KXqSxW>. Citado 3 vezes nas páginas 42, 43 e 44.

PYTORCH CORE TEAM. *PyTorch*: Examples. 2018. Acessado em 13 Jun. 2018. Disponível em: <https://github.com/pytorch/examples/tree/master/mnist>. Citado 2 vezes nas páginas 81 e 82.

PYTORCH CORE TEAM. *PyTorch | About*. 2018. Acessado em 1 Mai. 2018. Disponível em: <https://pytorch.org/about/>. Citado 2 vezes nas páginas 40 e 41.

RASCHKA, S. *Python Machine Learning*. [S.l.]: Packt Publishing, 2015. ISBN 1783555130, 9781783555130. Citado 2 vezes nas páginas 19 e 30.

SEIDE, F.; AGARWAL, A. Cntk: Microsoft's open-source deep-learning toolkit. In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: ACM, 2016. (KDD '16), p. 2135–2135. ISBN 978-1-4503-4232-2. Disponível em: <http://doi.acm.org/10.1145/2939672.2945397>. Citado na página 39.

SHAMS, S. et al. Evaluation of deep learning frameworks over different hpc architectures. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. [S.l.: s.n.], 2017. p. 1389–1396. ISSN 1063-6927. Citado 7 vezes nas páginas 48, 50, 52, 54, 55, 56 e 57.

SHATNAWI, A. et al. A comparative study of open source deep learning frameworks. In: *2018 9th International Conference on Information and Communication Systems (ICICS)*. [S.l.: s.n.], 2018. p. 72–77. Citado 9 vezes nas páginas 19, 22, 40, 41, 48, 50, 52, 55 e 57.

SHI, S.; WANG, Q.; CHU, X. Performance modeling and evaluation of distributed deep learning frameworks on gpus. In: *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*. [S.l.: s.n.], 2018. p. 949–957. Citado 7 vezes nas páginas 49, 50, 52, 54, 55, 56 e 57.

SHI, S. et al. Benchmarking state-of-the-art deep learning software tools. In: *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. [S.l.: s.n.], 2016. p. 99–104. Citado 5 vezes nas páginas 48, 52, 54, 55 e 57.

SILVA, I. N. da; SPATTI, D. H.; FLAUZINO, R. A. *Redes Neurais Artificiais*: para engenharia e ciências aplicadas. 1st. ed. São Paulo, SP, BRA: ArtLiber Editora Ltda., 2010. ISBN 978-85-88098-53-4. Citado 4 vezes nas páginas 27, 28, 30 e 31.

SIMARD, P. Y.; STEINKRAUS, D.; PLATT, J. C. Best practices for convolutional neural networks applied to visual document analysis. In: *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.* [S.l.: s.n.], 2003. p. 958–963. Citado na página 20.

SMITH, R. *NVIDIA Launches Tesla K20 K20X: GK 110 Arrives At Last*. AnandTech, 2012. Acessado em 21 Jun. 2020. Disponível em: <https://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-at-last/3>. Citado na página 35.

SMITH, R. *NVIDIA Announces the GeForce RTX 20 Series: RTX 2080 2080 on Sept. 20th, RTX 2070 in October*. AnandTech, 2018. Acessado em 22 Jun. 2020. Disponível em: <https://www.anandtech.com/show/13249/nvidia-announces-geforce-rtx-20-series-rtx-2080-ti-2080-2070>. Citado na página 38.

SMITH, R. *NVIDIA Reveals Next-Gen Turing GPU Architecture: NVIDIA Doubles-Down on Ray Tracing, GDDR6, amp; More*. AnandTech, 2018. Acessado em 22 Jun. 2020. Disponível em: <https://www.anandtech.com/show/13214/nvidia-reveals-next-gen-turing-gpu-architecture>. Citado na página 38.

STALLKAMP, J. et al. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, n. 0, p. –, 2012. ISSN 0893-6080. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0893608012000457>. Citado na página 51.

TSANG, S.-H. *Review: LeNet-1, LeNet-4, LeNet-5, Boosted LeNet-4 (Image Classification)*. 2018. Acessado em 25 Jul. 2019. Disponível em: <https://bit.ly/32Q9off>. Citado 3 vezes nas páginas 33, 64 e 84.

VERHELST, M.; MOONS, B. Embedded deep neural network processing: Algorithmic and processor techniques bring deep learning to iot and edge devices. *IEEE Solid-State Circuits Magazine*, v. 9, n. 4, p. 55–65, Fall 2017. ISSN 1943-0582. Citado na página 19.

WALTER, D. et al. *Your ML workloads cheaper and faster with the latest GPUs*. Google, 2020. Acessado em 23 Jun. 2020. Disponível em: <https://cloud.google.com/blog/products/

ai-machine-learning/your-ml-workloads-cheaper-and-faster-with-the-latest-gpus?hl=pt-br>. Citado na página 21.

WIKIMEDIA FOUNDATION. *Graphics Processing Unit*. 2018. Acessado em 9 Set. 2018. Disponível em: <https://bit.ly/21jWu1e>. Citado na página 33.

WIPO. *World Intellectual Property Organization*. 2018. Acessado em 7 Dez. 2018. Disponível em: <https://www.wipo.int/>. Citado na página 20.

WIPO. *What is Wipo?* WIPO, 2019. Acessado em 7 Jan. 2019. Disponível em: <https://www.wipo.int/about-wipo/en/>. Citado na página 20.

WOHLIN, C. et al. *Experimentation in Software Engineering*. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435. Citado 2 vezes nas páginas 60 e 76.

XU, P.; SHI, S.; CHU, X. Performance evaluation of deep learning tools in docker containers. In: *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*. [S.l.: s.n.], 2017. p. 395–403. Citado 6 vezes nas páginas 49, 50, 52, 54, 55 e 57.

Apêndices

APÊNDICE A – Sobre a Implementação da *LeNet-5*

Neste apêndice são apresentados trechos dos códigos utilizados nos experimentos do Capítulo 4. O objetivo é comparar a usabilidade da *API* de alto nível *Keras* com a usabilidade dos métodos nativos das bibliotecas *CNTK*, *TensorFlow 1.15* e *TensorFlow 2.2*. Esta comparação tem como métrica a quantidade de linhas necessária para codificar o modelo *LeNet-5* e para codificar as fases de treinamento e inferência.

No Código 1 é apresentado a codificação do modelo *LeNet-5* com os métodos da *API Keras* quando usado com as bibliotecas *TensorFlow 1.15* e *CNTK* como *back-end*. Verifica-se que bastam 11 linhas de código para implementar o modelo e aplicar o método *compile()* que associa o modelo a uma função de erro e a uma função de otimização. Ao converter esse código para o formato utilizado na *TensorFlow 2.2* - formato em que a *API Keras* está integrada a biblioteca *TensorFlow* -, o estilo de escrita é alterada, porém não há alteração no número de linhas. Para ser codificado com os métodos nativos da biblioteca *TensorFlow*, o mesmo modelo exige aproximadamente 30 linhas de códigos para ser implementado como ilustrado no Código 3. Com os métodos nativos da biblioteca *CNTK*, a rede *LeNet-5* exige aproximadamente 11 linhas de código[1], como mostrado no Código 4. Para codificar as funções de erro e de otimização com os métodos nativos das bibliotecas *TensorFlow* e *CNTK* são necessárias mais linhas de código.

Código 1 – Definição da LeNet-5 utilizando a *API Keras*

```
1  model = keras.Sequential()
2  model.add(layers.Conv2D(filters=6, kernel_size=(3, 3), activation='relu',
    ↪  input_shape=(32,32,1)))
3  model.add(layers.AveragePooling2D())
4  model.add(layers.Conv2D(filters=16, kernel_size=(3, 3), activation='relu'))
5  model.add(layers.AveragePooling2D())
6  model.add(layers.Flatten())
7  model.add(layers.Dense(units=120, activation='relu'))
8  model.add(layers.Dense(units=84, activation='relu'))
9  model.add(layers.Dense(units=10, activation = 'softmax'))
10 model.summary()
11 model.compile(loss=keras.losses.categorical_crossentropy,
    ↪  optimizer=keras.optimizers.Adam(), metrics=['accuracy'])
```

Fonte: Gazar (2018)

[1] O número de linhas em qualquer biblioteca e *API* pode variar de acordo com o estilo de escrita do programador.

Código 2 – Definição da LeNet-5 utilizando a *API Keras* com a *TensorFlow 2.2*

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(filters=6, kernel_size=(3, 3), activation='relu',
   ↪  input_shape=(32,32,1)),
  tf.keras.layers.AveragePooling2D(),
  tf.keras.layers.Conv2D(filters=16, kernel_size=(3, 3), activation='relu'),
  tf.keras.layers.AveragePooling2D(),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(units=120, activation='relu'),
  tf.keras.layers.Dense(units=84, activation='relu'),
  tf.keras.layers.Dense(units=10, activation = 'softmax')
 ])
model.compile(loss=tf.keras.losses.categorical_crossentropy, optimizer='adam',
 ↪  metrics=['accuracy'])
```

Fonte: Modificado a partir de Gazar (2018)

Código 3 – Definição da LeNet-5 utilizando métodos nativos da *TensorFlow 1.15*

```
def LeNet(x):
    mu = 0
    sigma = 0.1
    weights = {
        'conv1': tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu,
        ↪  stddev = sigma)),
        'conv2': tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu,
        ↪  stddev = sigma)),
        'fl1': tf.Variable(tf.truncated_normal(shape=(5 * 5 * 16, 120), mean = mu,
        ↪  stddev = sigma)),
        'fl2': tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu, stddev =
        ↪  sigma)),
        'out': tf.Variable(tf.truncated_normal(shape=(84, n_classes), mean = mu,
        ↪  stddev = sigma))
    }
    biases = {
        'conv1': tf.Variable(tf.zeros(6)),
        'conv2': tf.Variable(tf.zeros(16)),
        'fl1': tf.Variable(tf.zeros(120)),
        'fl2': tf.Variable(tf.zeros(84)),
        'out': tf.Variable(tf.zeros(n_classes))
    }
    conv1 = tf.nn.conv2d(x, weights['conv1'], strides=[1, 1, 1, 1], padding='VALID')
    conv1 = tf.nn.bias_add(conv1, biases['conv1'])
    conv1 = tf.nn.relu(conv1)
    conv1 = tf.nn.avg_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
    ↪  padding='VALID')
    conv2 = tf.nn.conv2d(conv1, weights['conv2'], strides=[1, 1, 1, 1],
    ↪  padding='VALID')
    conv2 = tf.nn.bias_add(conv2, biases['conv2'])
    conv2 = tf.nn.relu(conv2)
    conv2 = tf.nn.avg_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
    ↪  padding='VALID')
    fl0 = flatten(conv2)
    fl1 = tf.add(tf.matmul(fl0, weights['fl1']), biases['fl1'])
    fl1 = tf.nn.relu(fl1)
    fl2 = tf.add(tf.matmul(fl1, weights['fl2']), biases['fl2'])
    fl2 = tf.nn.relu(fl2)
    logits = tf.add(tf.matmul(fl2, weights['out']), biases['out'])

    return logits
```

Fonte: Gazar (2018)

Código 4 – Definição da LeNet-5 utilizando métodos nativos da *CNTK*

```
def create_model(features):
    with C.layers.default_options(init = C.layers.glorot_uniform(), activation =
     ↪  C.relu):
            h = features
            h = C.layers.Convolution2D(filter_shape=(5,5), num_filters=6,
             ↪  strides=(1,1),  pad=True, name="first_conv")(h)
            h = C.layers.AveragePooling(filter_shape=(2,2), strides=(2,2),
             ↪  name="first_max")(h)
            h = C.layers.Convolution2D(filter_shape=(5,5), num_filters=16,
             ↪  strides=(1,1), pad=True, name="second_conv")(h)
            h = C.layers.AveragePooling(filter_shape=(2,2), strides=(2,2),
             ↪  name="second_max")(h)
            h = C.layers.Convolution2D(filter_shape=(5,5), num_filters=120,
             ↪  strides=(1,1), pad=True, name="third_conv")(h)
            h = C.layers.Dense(84, activation = C.relu, name="first_dense")(h)
            r = C.layers.Dense(num_output_classes, activation = C.softmax,
             ↪  name="classify")(h)
            return r
```

Fonte: Autoria própria

A *API Keras* possui um método de treinamento já implementado; para treinar um modelo, basta executar esse método como ilustrado no Código 5. O mesmo ocorre com o processo de inferência (neste caso, teste), cuja chamada do método é ilustrado no Código 6.

Código 5 – Treinamento utilizando a *API Keras*

```
model.fit_generator(train_generator,steps_per_epoch=steps_per_epoch, epochs=EPOCHS,
 ↪  validation_data=validation_generator, validation_steps=validation_steps,
 ↪  shuffle=True)
```

Fonte: Gazar (2018)

Código 6 – Inferência utilizando a *API Keras*

```
model.evaluate(test['features'], to_categorical(test['labels']))
```

Fonte: Gazar (2018)

Para codificar o algoritmo de treinamento da rede *LeNet-5* com os métodos nativos da biblioteca *TensorFlow* são necessárias, aproximadamente, 13 linhas de código como mostrado no Código 7 e para codificar o algoritmo de inferência / teste são necessárias, aproximadamente, 9 linhas de código como mostrado no Código 8. Para codificar o algoritmo treinamento da rede *LeNet-5* com os métodos nativos da biblioteca *CNTK* são necessárias, aproximadamente, 3 linhas de código como mostrado no Código 9 e para codificar o algoritmo de inferência / teste são necessárias, aproximadamente, 3 linhas de código como mostrado no Código 10.

Código 7 – Treinamento utilizando métodos nativos da *TensorFlow 1.15*

```
with tf.Session() as session:
    session.run(tf.global_variables_initializer())
    num_examples = len(train['features'])
    for i in range(EPOCHS):
        X_train, y_train = shuffle(train['features'], train['labels'])
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:end]
            session.run(training_operation, feed_dict={x: batch_x, y: batch_y})
        validation_accuracy = evaluate(validation['features'], validation['labels'])
    train_file_writer = tf.summary.FileWriter('logs', session.graph)
    train_file_writer.close()
    saver.save(session, './lenet')
```

Fonte: Gazar (2018)

Código 8 – Inferência utilizando métodos nativos da *TensorFlow 1.15*

```
def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE],
        ↪  y_data[offset:offset+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

Fonte: Gazar (2018)

Código 9 – Treinamento utilizando métodos nativos da *CNTK*

```
for i in range(0, int(num_minibatches_to_train)):
            data=train_reader.next_minibatch(minibatch_size, input_map=input_map)
            trainer.train_minibatch(data)
```

Fonte: Microsoft Corporation (2019)

Código 10 – Inferência utilizando métodos nativos da *CNTK*

```
for i in range(num_minibatches_to_test):
    data = test_reader.next_minibatch(test_minibatch_size, input_map=test_input_map)
    eval_error = trainer.test_minibatch(data)
    test_result = test_result + eval_error
```

Fonte: Microsoft Corporation (2019)

A quantidade aproximada de linhas de código altera dependendo do estilo de escrita do programador e da estratégia de implementação utilizada, principalmente em códigos implementados com os métodos nativos das bibliotecas *TensorFlow* e *CNTK*. A *API Keras* também facilita a codificação da configuração da *CNN*, como a configuração das funções de erro e de otimização.