UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GERMANO BERTONCELLO

Um Estudo Sobre a Performance de Aplicações Big Data com Deep Learning

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Cláudio Fernando Resin Geyer
Co-orientador: Breno Fanchiotti Zanchetta

Porto Alegre
2018

"O temor do SENHOR é a instrução da sabedoria,
e a humildade precede a honra."
— PROVÉRBIOS 15:33

AGRADECIMENTOS

Agradeço a Deus, e ao meu Senhor Jesus Cristo, o qual se entregou na Cruz para que hoje eu tivesse a vida eterna e a quem eu devo toda honra, glória e louvor para sempre.

Agradeço aos meus pais, Eduardo e Eliane, os quais sempre me deram muito amor, sábios conselhos, consolo em momentos difíceis, educação e disciplina, sem jamais me deixarem faltar nada.

Agradeço aos meus irmãos, Vinícius e Thiago, que compartilharam de muitos momentos vivenciados nessa jornada e que trouxeram, juntamente com os meus pais, o conforto do ambiente familiar.

Agradeço aos meus familiares e meus amigos que estiveram junto comigo ao longo dessa caminhada e que trouxeram sempre muita alegria e companheirismo aos meus dias.

Agradeço ao professor doutor Cláudio Fernando Resin Geyer, que aceitou me orientar nesse trabalho, pela paciência, pela motivação e pelo auxílio prestado sempre que necessário.

Agradeço ao Grupo de Processamento Paralelo e Distribuído, em especial, agradeço ao mestrando Breno Fanchiotti Zanchetta e ao doutorando Kassiano José Matteussi pelos inúmeros esclarecimentos prestados ao longo de todo trabalho.

RESUMO

Deep Learning (DL) e Big Data (BD) convergiram para um paradigma de computação híbrido capaz de unir o processamento dinâmico fornecido pelos modelos DL aliado ao poder do processamento paralelo e distribuído de *frameworks* BD. Este trabalho avalia o impacto causado na performance das aplicações Deep Learning junto ao modelo de programação paralelo e distribuído proveniente de *frameworks* de processamento Big Data. O ambiente de experimentação é controlado e representa um cenário de testes real e é composto por um cluster virtual criado na Microsoft Azure e configurado para suportar o *framework* Apache Spark sob o sistema YARN junto ao sistema de arquivos distribuídos do Hadoop (HDFS). Além disso, para suportar o desenvolvimento das aplicações *TensorFlow* sob o *Apache Spark*, foi utilizado o *framework BigDL*. Os resultados obtidos indicam a eficácia do modelo de processamento Big Data com Deep Learning, constata-se um ganho de desempenho de até 87,4% no ambiente distribuído de ; redução de custodo treinamento de até 41,3%, e perda de precisão dos modelos menor que 5%.

Palavras-chave: Big Data. Deep Learning. Distributed Processing. Apache Spark. TensorFlow.

A Performance Study of Big Data Deep Learning Applications

ABSTRACT

Deep Learning (DL) and Big Data (BD) have converged to a hybrid computing paradigm that merges the dynamic processing provided by DL models compled with the power of parallel and distributed processing of Big Data frameworks. This paper presents the performance evaluation of Big Data Deep Learning applications. Our experimental environment setup was controlled and represents a real test scenario. It is composed of a virtual cluster created in Microsoft Azure and configured to support the Apache Spark framework into the YARN system with the Hadoop Distributed File System (HDFS). In addition, to support the development of TensorFlow applications under Apache Spark, the *framework BigDL* was used. The obtained results indicate the effectiveness of the Big Data processing model with Deep Learning. It was attested that performance with distributed environment was improved up to 87.4%; cost reduces until 41.3%, and loss in accuracy was less than 5% for the testbed.

Keywords: Big Data,Deep Learning, Distributed Processing, Apache Spark, TensorFlow.

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BD	Big Data
BS	Batch size
CNN	Convolutional Neural Network
CPU	Unidade Central de Processamento
DAGs	Directed Acyclic Graphs
DF	DataFrame
DL	Deep Learning
DS	Dataset
FC	Fully-Connected Layer
FNN	Feed-Forward Neural Network
FW	Framework
GPU	Unidade de Processamento Gráfico
GP-GPU	GPUs de Propósito Geral
HDFS	Hadoop Distributed File System
HMR	Hadoop MapReduce
IA	Inteligência Artificial
LSTM	Long-Short Term Memory
ML	Machine Learning
MR	MapReduce
RDD	Resilient Distributed Dataset
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Big Data e Deep Learning (DL) são campos da Ciência da Computação que representam tendências tecnológicas em nível mundial e ambos são tópicos de pesquisa em constante crescimento. A utilização de Big Data (BD) possibilita o processamento massivo de dados, fato que é pouco explorado pelas tecnologias convencionais. Por outro lado, esse fator é considerado um grande desafio para o Deep Learning (DL). Isso porque as aplicações Deep Belief Network (DBN) são complexas e produzem inúmeros atributos e saídas que podem interferir diretamente no tempo de conclusão de uma aplicação (CHEN; LIN, 2014).

Nesse contexto, muitas organizações buscam no processamento de dados a obtenção de informações essenciais para o sucesso de seus negócios. Em geral, as soluções devem ser eficazes, escaláveis e possuir baixo custo. De fato, o modelo de programação paralelo e distribuído MapReduce (MR) e sua implementação mais famosa o Hadoop MapReduce (HMR) possibilitam o processamento massivo de dados necessário para a obtenção da informação. Além disso, pode-se citar inúmeras implementações que seguem o modelo MR e hoje são amplamente utilizadas para o processamento distribuído e em tempo real, tais como: Flink, Storm, Beam, Spark e outros (MATTEUSSI et al., 2018).

O uso da Inteligência Artificial (IA) e de aplicações de Machine e Deep Learning (DL) permitiram a criação de modelos computacionais profundos que são compostos de múltiplas camadas de processamento, que dependem de alto número de cálculos computacionais a fim de classificar e/ou aprender representações sobre um conjunto de dados. Os problemas desses campos englobam classificação, classificação com dados de entrada incompletos, regressão, transcrição, tradução de máquina, saída estruturada, detecção de anomalia, síntese e amostragem, imputação de valores perdidos, remoção de ruídos e estimação de densidade (GOODFELLOW et al., 2016). Também utiliza-se essas técnicas em aplicações de processamento de imagens como reconhecimento facial (TAIGMAN et al., 2014; LAWRENCE et al., 1997; HOWARD et al., 2017; SERMANET et al., 2013; MOLLAHOSSEINI; CHAN; MAHOOR, 2016), detecção e classificação de objetos (HOWARD et al., 2017; SZEGEDY et al., 2015; CIREŞAN et al., 2012) e até mesmo na robótica (GAYA et al., 2016). Em áreas como linguagem natural, redes recorrentes de DL permitiram tradução autônoma de linguagem por máquinas (CHO et al., 2014; BAHDANAU; CHO; BENGIO, 2014; SUTSKEVER; VINYALS; LE, 2014) e até aplicações envolvendo o domínio do som (GRAVES; MOHAMED; HINTON, 2013; VU; WANG,

2016).

Nesse contexto, muitos trabalhos que propõem ferramentas que integram Big Data e DL surgiram (KIM et al., 2016; MORITZ et al., 2015; WANG et al., 2018; GUPTA et al., 2017a; KHUMOYUN; CUI, 2016; YANG et al., 2017; FENG; SHI; JAIN, 2016; LU et al., 2018; AHN; KIM; YOU, 2018) e embora elas empreguem o uso de Big Data para distribuir o processamento de dados com DL, carecem no quesito avaliação de cenários de configurações para DL (aplicações, redes neurais, modelos e datasets) e de performance para Big Data (experimentação local e distribuída).

Neste contexto, este trabalho avalia o impacto causado na performance das aplicações Deep Learning junto ao modelo de programação paralelo e distribuído provenientes de *frameworks* de processamento Big Data. O ambiente de experimentação deste trabalho é controlado e representa um cenário de testes real e é composto por um cluster virtual criado na Microsoft Azure e configurado para suportar o *framework* Apache Spark sob o sistema YARN junto ao sistema de arquivos distribuídos do Hadoop (HDFS). Além disso, para suportar o desenvolvimento das aplicações *TensorFlow* sob o *Apache Spark*, foi utilizado o *framework BigDL* (WANG et al., 2018).

Os experimentos avaliam vários cenários de teste, variáveis de configuração, modelos, conjuntos de dados e métricas a fim de apresentar valiosas descobertas acerca do potencial e das limitações referentes ao paradigma de programação utilizado para o processamento conjunto de DL com BD.

Afora isso, este trabalho apresenta as seguintes contribuições: i) avalia o desempenho e a escalabilidade de um conjunto de aplicações DL ao utilizar modelos de programação paralelos e distribuídos propiciados por *frameworks* Big Data em ambiente real; ii) apresenta uma análise relativa à manutenção da precisão dos modelos DL em meio ao processamento distribuído e; iii) apresenta uma análise de custos de treinamento resultante da análise de performance avaliada.

1.1 Motivação

De acordo com (VENKATESAN; NAM; SHIN, 2018), as principais limitações nas aplicações DL que motivaram a integração com o Spark são:

- Baixa eficiência;
- Limitações na capacidade de suportar bilhões de parâmetros decorrentes do modelo

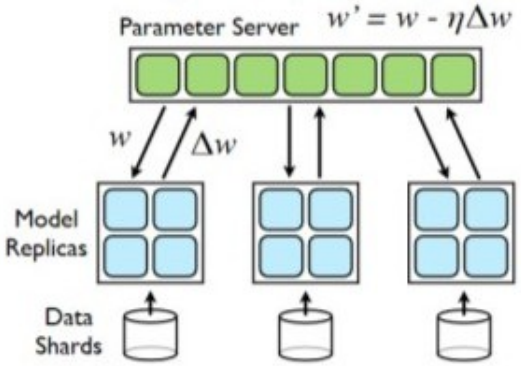
Deep Neural Network(DNN);

- Manipulação massiva de dados;
- Alto custo de manutenção (em casos onde são utilizados hardwares especializados, como Field Programmable Gate Array - FPGA)

A integração do Spark com DL surge como uma solução do tipo *all-in-one* para superar esses desafios citados acima. A qual proporciona redução no tempo de treinamento e baixa taxa de erro do modelo DL (VENKATESAN; NAM; SHIN, 2018). Para isso, Spark distribui os dados e a computação pelo *cluster*, onde cada *worker* node executa o treinamento e/ou a predição sobre o seu próprio *subset* do *dataset* inteiro e Spark agrega os resultados produzidos em cada *worker* para formar um resultado preciso ao final do processo.

DL distribuído apareceu pela primeira vez em (DEAN et al., 2012), onde são apresentados dois modos de paralelismo: paralelismo dos dados e paralelismo do modelo. O paralelismo do modelo é aplicado quando cada nodo *worker* executa uma parte do modelo inteiro.

Figura 1.1: Exemplo de treinamento distribuído



(a) Imagem retirada de (DEAN et al., 2012)

Para que seja obtida uma precisão elevada, os autores (VENKATESAN; NAM; SHIN, 2018) acreditam que o paralelismo dos dados deve ser combinado com o paralelismo de modelo.

1.2 Objetivos

Esta seção apresenta os objetivos do presente trabalho, qualificando-os desde uma visão macro até suas minúcias.

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é avaliar o impacto causado na performance das aplicações Deep Learning junto ao modelo de programação paralelo e distribuído proveniente de *frameworks* de processamento Big Data.

1.2.2 Objetivos Específicos

Para alcançar o objetivo deste trabalho, os seguintes objetivos específicos foram definidos.

1. Compreender o uso do *framework* BigDL;
2. Avaliar o desempenho e a escalabilidade de um conjunto de aplicações DL ao utilizar modelos de programação paralelos e distribuídos propiciados por *frameworks* Big Data;
3. Prover uma análise relativa à manutenção da precisão dos modelos DL em meio aos modelos paralelos e distribuídos propiciados por *frameworks* Big Data;
4. Apresentar uma análise de custos resultante da análise de performance proposta.

1.3 Metodologia

Esta seção possui o objetivo de apresentar a metodologia empregada durante a construção do presente trabalho. Desse modo, quanto à abordagem da mesma, define-se a pesquisa como quantitativa, onde, o tema é estruturado para que os conceitos investigados estejam alinhados com toda a pesquisa, conforme pode-se verificar no Capítulo 2.

Quanto à natureza, esta é uma pesquisa aplicada visto a necessidade da exploração prática de requisitos específicos apresentados nas áreas de Big Data e Deep Learning, conforme pode-se verificar no Capítulo 3. Espera-se ainda que os resultados obtidos fomentem e contribuam para futuras pesquisas na área que envolvam a solução de um novo problema ou de problemas pré-existentes. Elimina-se assim, as dificuldades técnicas e científicas encontradas e superadas durante o processo de desenvolvimento e investigação deste trabalho.

Levando em conta os objetivos desta pesquisa, pode-se classificar a mesma como

uma pesquisa exploratória. Desse modo, cada um dos objetivos propostos na Secção 1.2.1 visa validar uma parte do objetivo geral deste trabalho.

Por fim, os procedimentos adotados para o desenvolvimento desta pesquisa possuem origem experimental. De fato, as avaliações propostas seguem um planejamento rigoroso, bem definido e de cunho exploratório. Além disso, para conduzir a avaliação dos resultados, os experimentos foram definidos com base em procedimentos bem estruturados para tornar a coleta de dados clara e objetiva. Finalmente, a análise dos dados (numérica) foi desenvolvida através de procedimentos estatísticos, observe o Capítulo 4.

1.4 Organização do Trabalho

O presente trabalho está organizado dá seguinte maneira. O Capítulo 2 aborda o estado-da-arte utilizado neste trabalho sendo explorados os tópicos com maior relevância para o presente trabalho: *Big Data* e *Deep Learning*. Já o Capítulo 3 apresenta os trabalhos relacionados com este trabalho, bem como apresenta uma discussão que motiva o desenvolvimento deste trabalho. O Capítulo 4 contextualiza a proposta do trabalho, a metodologia de experimentos e os resultados obtidos. Por fim, a conclusão, Capítulo 5 aborda todas as conclusões inerentes às descobertas deste trabalho, bem como lista desafios e trabalhos futuros da área.

2 FUNDAMENTAÇÃO TEÓRICA

Essa seção aborda os tópicos de interesse do presente trabalho. Inicialmente será apresentado o tópico de BD, com suas definições, conceitos, *frameworks* e modelos de processamento. Em seguida, apresenta-se o tópico de DL, onde são explorados os tipos de redes existentes, os modelos mais famosos com suas cargas de trabalho e os frameworks. Por fim, são expostas considerações que incentivam a correlação entre essas duas áreas.

2.1 Big Data

Um dos grandes desafios computacionais da atualidade se concentra em armazenar, manipular e analisar de forma veloz e à um baixo custo um volume muito grande de dados gerados (na ordem de petabytes de dados diários) por sistemas corporativos, serviços e sistemas Web, mídias sociais (MATTEUSSI, 2016). Ainda, BD pode ser conceituado como uma série de técnicas que operam sobre grande volumes de dados a fim de gerenciá-los e extrair informações úteis de dados brutos, sejam eles estruturados ou não (WHITE, 2009).

2.1.1 Características

BD representa um termo empregado para descrever o crescimento, o uso e a disponibilidade das informações. Pode-se definir as principais características de BD pelo conceito dos 5 V's: volume, velocidade, variedade, veracidade e valor (ABBASI; SARKER; CHIANG, 2016).

- Volume: Este aspecto faz menção à necessidade, que uma solução BD possui, de ser capaz de atuar sobre grandes quantidades de dados sem comprometer o funcionamento do sistema. Grandes volumes de dados ou pequenos dados em alto volume de transações devem ser processados e o sistema projetado deve estar pronto para suportar cargas de todos os tamanhos;
- Variedade: Os dados de entrada possuem naturezas diversas, por exemplo, sites de comércio eletrônico web utilizam dados estruturados, ao passo que *logs* de um servidor web são conhecidos como dados semi-estruturados e redes sociais lidam com dados não estruturados como áudio, vídeo, imagens, entre outros tipos;

- Velocidade: Refere-se a quão rápido os dados estão sendo produzidos, bem como o quão rápido os mesmos devem ser tratados para atender a demanda;
- Veracidade: O sistema BD deve possuir mecanismos que garantam a autenticidade do dado, bem como garantir que estes dados sejam consistentes.
- Valor: Este tópico delimita que a solução BD deve gerar valor agregado para soluções do mundo real, seja ele um valor simbólico como uma tendência, abstrato como uma possível tomada de decisão, imaterial como análise de sentimentos ou monetário quando resulta em tomadas de decisões que aumentam os lucros de um setor produtivo.

A seguir será apresentado o modelo MapReduce, *framework* para o processamento paralelo e distribuído, bem como o uso do mesmo para o processamento DL (DL).

2.1.2 O Modelo MapReduce

O modelo de programação paralelo e distribuído MapReduce (MR) foi construído pela primeira vez na linguagem de programação Lisp e foi popularizado mais tarde pelo Google (MATTEUSSI et al., 2017). Basicamente, ele efetua o processamento de grandes volumes de dados por meio da execução de várias operações em paralelo de *map* e *reduce*.

As operações de *map* recebem uma porção de dados do arquivo de entrada a ser processado e geram um conjunto de tuplas intermediárias no formato chave-valor. Em seguida, as tuplas geradas são agrupadas com base em suas chaves. Já as funções de *reduce* recebem uma chave como entrada e uma lista composta por todos os valores gerados pelas fases de *map* para cada uma das chaves recebidas, gerando uma saída reduzida dos dados que tiveram chaves de mesmo índice. É importante salientar que ambas as funções são interligadas pela fase de comunicação e sincronização, denominada *shuffle* (DEAN; GHEMAWAT, 2004). Como estudo de caso simplista, supõe-se que uma frase em uma aplicação de contagem de palavras, como "o rato roeu o queijo", passa pela fase *map* gerando as seguintes tuplas em que a palavra é a chave e à sua ocorrência atribui-se o valor um(1): <"o",1>, <"rato",1>, <"roeu",1>, <"o",1>, <"quejo",1>. A fase de *reduce* une as ocorrências similares: <"o",2>, <"rato",1>, <"roeu",1>, <"quejo",1>.

2.1.3 Frameworks Para Processamento Paralelo e Distribuído

Essa seção apresenta dois modelos distintos de *framework*. O primeiro deles, o *Hadoop MapReduce* (HMR) é considerada a implementação mais famosa do modelo MR. Já o Apache Spark foi criado com o objetivo aprimorar o processamento de grandes volumes de dados aproveitando o uso de memória para reduzir a latência de processamento. Embora compartilhem o mesmo modelo, suas finalidades e as aplicações são implementadas de maneiras distintas, acompanhadas por diferentes otimizações.

2.1.3.1 Hadoop MapReduce

O *framework Hadoop MapReduce* é desenvolvido na linguagem de programação Java e seu código é *Open-Source*. HMR implementa o modelo MR e inclui um escalonador para efetuar o gerenciamento de múltiplas aplicações MR de forma simultânea, semelhante ao uso de *batch jobs* em um *cluster*. Em linhas gerais, um *job* (trabalho) MR é constituído por múltiplas tarefas que são divididas em funções *map/reduce* escalonadas sobre um *cluster* Hadoop (MATTEUSSI; ROSE, 2015).

Além disso, o HMR é composto tanto pelo modelo MR quanto pelo sistema de arquivos distribuídos, chamado *Hadoop Distributed File System* (HDFS) que, por sua vez fornece resiliência e alta taxa de transferência para o acesso aos dados (SOUSA; MOREIRA; MACHADO, 2009). O HDFS é responsável por efetuar o armazenamento dos dados de entrada utilizadas pelas aplicações a serem processadas. O armazenamento é composto por blocos de tamanho 64 MB (padrão), 128 MB ou 256 MB distribuídos entre os múltiplos nodos do *cluster*. Os dados são replicados no HDFS conforme um fator de replicação definido pelo programador, sendo o padrão 3 réplicas.

2.1.3.2 Apache Spark

O Apache Spark[1] representa uma implementação alternativa ao HMR e possui a finalidade de melhorar o uso de recursos computacionais. Dentre as inúmeras características que o Spark oferece, o processamento em memória, conhecido como *Resilient Distributed Dataset* (RDD) se destaca. O Spark é até 10 vezes mais rápido ao se comparar aplicações com uso intensivo de disco e até 100 vezes para aplicações com uso intensivo de memória (SINGH; REDDY, 2015; SOUZA et al., 2018).

[1] https://spark.apache.org/

O RDD utiliza um modelo de persistência elástica para fornecer a flexibilidade para o processamento paralelo sobre um conjunto de dados em memória, nos discos ou em ambos. Quando utilizado somente o processamento em memória, o paradigma RDD é altamente eficiente para aplicações em tempo-real que, possuem características iterativas. Além disso, caso o acesso ao HDFS seja necessário, o RDD contém informações suficientes para não depreciar o desempenho das aplicações (ZHANG et al., 2015). Esses fatores removem o alto custo de acesso aos dados em discos em todas as etapas, como ocorre com HMR.

Embora o uso do RDD não represente problemas para o processamento em lote, para análise em tempo-real isso representa uma barreira. O RDD possui o fluxo de processamento conhecido como (*lazy evaluation* - avaliação lenta), ou seja, a execução de transformações não é acionada até que exista uma ação que precise de resultados (não é iterativa). Para resolver esse problema o Spark também oferece uma API conhecida como DataFrame (DF).

Conceitualmente, DF são equivalentes a uma tabela em um banco de dados relacional, ou a um DataFrame em R[2] ou Python[3]. Os dados podem ser obtidos de diversas fontes, tais como: arquivos de dados estruturados em tabelas, banco de dados diversos e RDDs existentes (ARMBRUST et al., 2015). Ainda, a computação do Spark é baseada em Directed Acyclic Graphs (DAGs), ou seja, são criadas várias tarefas para processamento. Assim, o escalonador do Spark divide o RDD em estágios que recebem tarefas que sofrem sucessivas transformações em paralelo.

2.2 Deep Learning

Essa seção contextualiza *DL* e apresenta as principais definições que explicam o funcionamento e estado-da-arte desse campo.

A curiosidade dos programadores de adquirir inteligência sobre forma de um programa autônomo sempre foi o sonho para o desenvolvimento da área de IA. Dentro de várias tentativas, DL pode ser mencionada como uma abordagem muito promissora desse conjunto no espectro de aplicações do mundo real.

Segundo Goodfellow (GOODFELLOW et al., 2016), nos primórdios das pesquisas do campo de IA, o campo de DL apropriou-se rapidamente de problemas do mundo

[2]https://www.rdocumentation.org/packages/base/versions/3.5.1/topics/data.frame
[3]https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.DataFrame.html

real que podem ser descritos segundo o pensamento: problemas que são proceduralmente custosos ou intelectualmente difíceis para humanos, mas que sejam passíveis de serem executados por uma máquina - problemas que podem ser descritos por uma lista de transformações matemáticas formais. Por exemplo, detecção de faces é uma tarefa automática para seres humanos, sendo capazes de identificar faces em humanos, animais e até em objetos como carros ou tomadas. Contudo, é muito difícil ao ser humano passar procedimentos analíticos que façam um algoritmo identificar potenciais rostos em pessoas e objetos.

Em contraponto, Goodfellow (GOODFELLOW et al., 2016) ainda argumenta que a solução para essa gama de problemas intuitivos exclusivamente para humanos é permitir que algoritmos neurais aprendam por meio da experiência, i.e. tentativa e erro com criação de modelos. Dessa forma, espera-se que as máquinas entendam o mundo real como uma hierarquia de conceitos, onde cada conceito pode ser definido com base em suas relações com os conceitos mais simples - gerados em camadas anteriores.

De fato, a possibilidade de extrair conceitos unida com a formação de novos conceitos não poderia ter se tornado real sem a introdução de camadas em diferentes dimensões. À cada camada se dá a tarefa de aprender uma tarefa específica que é passada para as camadas posteriores. Em análise de caso, seria como uma rede neural nos seguintes moldes: Primeira camada identifica cores e pixels, segunda camada identifica contornos e linhas, terceira camada identifica formas e bordas e quarta camada identifica entidade rosto.

O campo do *DL* se desenvolveu muito rapidamente, mas pode ser definido com base em suas redes fundamentais mais simples. Dentro dos modelos mais famosos estão: *Feed-Forward*(*FNN*), Redes Neurais Convolucionais (*Convolutional Neural Networks, CNN*), Redes Neurais Recorrentes(*RNN*) e *LSTM*(*Long-Short Term Memory*). Esses tipos de redes serão discutidos na próxima subseção, bem como os principais casos de sucesso de implementações dessas redes na literatura.

2.2.1 Definições

DL como grande área da IA, permite a criação de modelos computacionais profundos que são compostos de múltiplas camadas de processamento, que dependem de alto número de cálculos computacionais a fim de classificar e/ou aprender representações sobre um conjunto de dados. Dessa forma, espera-se generalizar a solução de DL feita

sobre um conjunto de dados (*dataset*) para outro conjunto mais amplo e não conhecido, a fim de obter outras instâncias de resultado do mesmo modelo sem grandes alterações nas taxas de acerto de predição.

Essa tarefa pode ser alcançada por meio de procedimentos de cálculos computacionais procedurais, também conhecidos como treinamentos, que visam descobrir os melhores parâmetros que consigam descobrir uma função de classificação ou representação de um conjunto de dados. O treinamento geralmente é a parte mais custosa em termos de recursos computacionais de todos os procedimentos de DL. Pode ser explicado como o ato de ajustar os parâmetros do modelo, ou pesos, durante múltiplas passagens a fim de obter a melhor combinação de pesos que resolvam o problema.

Após esse procedimento, o modelo deve passar por um conjunto único e reduzido de dados similares ao de treinamento a fim de validar esse treinamento. Essa etapa ocorre pois o treinamento do modelo transcorre bem para o universo de dados contidos no conjunto de treinamento(*dataset*) provido no treinamento, mas esse modelo pode ter uma performance diferente quando lhe for apresentado um dado desconhecido. Portanto, com o propósito de adequar o modelo para dados não vistos, separa-se parte do dataset original antes do treinamento para ser alimentado ao modelo somente na etapa de validação.

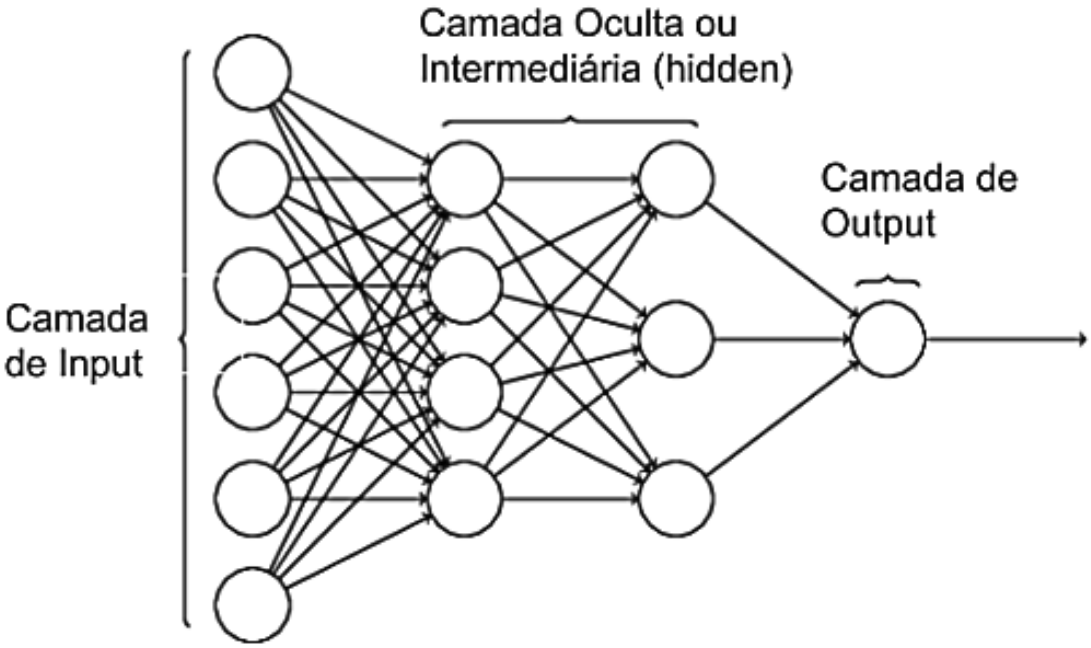
Finalmente, o modelo passa por uma etapa denominada de teste, onde se alimentam algumas instâncias aleatórias dos dados para verificar se ele está realizando a tarefa de forma satisfatória. Passando dessa etapa com sucesso, o modelo neural de DL está pronto para ser utilizado sem restrições para o domínio que foi designado. Caso não apresente taxas de acerto satisfatórias, o modelo pode ser reajustado para mais treinamentos ou validações a fim de alcançar o padrão almejado.

Dentro dos domínios de modelos de DL, encontram-se várias implementações importantes para solução de problemas do mundo real. Geralmente essa soluções se aglutinam em grandes grupos de aplicações que recebem um nome específico graças à principal operação ou otimização que representa a maior contribuição ao campo da IA. Dentre os mais notáveis tipos de redes neurais de DL, encontram-se:

1. FNN: com o nome de Feed-Forward Neural Networks, este grupo pode ser considerado o tipo mais simples de rede neural do DL, porém também representam uma das mais vitais. Esse modelo consiste de vários *perceptrons* (representação computacional de neurônios (ROSENBLATT, 1958)) dispostos em camadas paralelas, formando assim um grafo totalmente conexos entre duas camadas subsequentes como demonstra a figura 2.1. Cada neurônio executa cálculos que visam extrair

características por meio de variáveis de peso, que auxiliem na descoberta de uma função que mapeia o dataset. Estes tipos de redes continuam sendo a base mais confiável de todos os tipos de redes do DL, graças à sua alta capacidade de extração de características dos dados.

Figura 2.1: Modelo de Redes Feed Forward Network

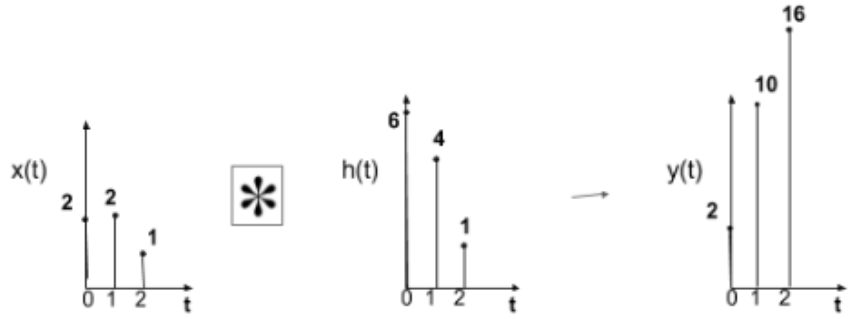


2. CNN: Convolutional Neural Networks foram as principais redes neurais que colocaram o DL em evidência. Essas redes utilizam as redes FNN, com o diferencial que acrescentam múltiplas camadas de convolução anteriormente. Dessa forma, conseguem extrair informações valiosas antes de alimentar esses novos conceitos às camadas FNN.

   Essas operações de convolução são amplamente conhecidas no campo de estudo de visão computacional, mas sua integração às redes neurais possibilitou a redução da carga de trabalho das camadas FNN, dessa forma reduzindo drasticamente o tempo necessário para essas camadas treinarem o modelo. São amplamente utilizadas no contexto de processamento de imagens e vídeos. Em alto nível, a convolução pode ser explicada como um somatório iterativo que ocorre entre produtos de valores discretos de dois conjuntos de dados. De forma simplista, funciona de forma que o primeiro conjunto de dados vai iterando como se lê - da esquerda para a direita - enquanto o segundo conjunto de dados é utilizado da direita para a esquerda. A figura 2.2 demonstra um exemplo de convolução a fim de clarear a ocorrência dessa

operação.

Figura 2.2: Modelo de Operação de Convolução usado nas Redes Convolucionais

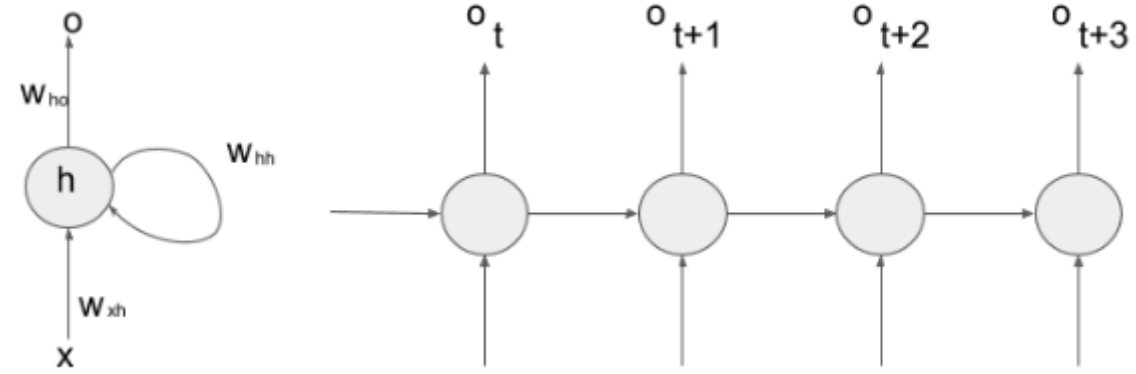


Iteração 1: Pega-se o x(0) e o primeiro do inverso de h(t), ou seja, o valor 1 (h(2))
Retorna-se o resultado do cálculo 2.1 = 2
Iteração 2: Pega-se o x(0) e o x(1) e multiplica-se cada um desses com o h(2) e h(1) e depois se soma.
Retorna-se o resultado do cálculo 2.1 + 2.4 = 10
Iteração 3: Pega-se o x(0), o x(1), x(2) e multiplica-se por h(2), h(1), h(0) seguindo da soma.
Retorna-se o resultado do cálculo 2.1 + 2.4 + 1.6 = 16

3. RNN: as Recurrent Neural Networks, ou Redes Recorrentes, introduziram um conceito temporal às redes neurais que as antecederam. Sua unidade básica é uma célula RNN que está evidenciada à parte esquerda da figura 2.3. A mesma se liga à várias outras células RNNs, propagando os seus respectivos resultados de computações entre si por passos temporais - como evidencia a parte direita da figura 2.3. Dessa forma, as redes RNN possibilitaram que certas computações em um dado instante pudessem facilitar a computação em um instante próximo, ou vice-versa. Essas redes ficaram muito conhecidas por sucesso em aplicações de textos, linguagem natural, traduções, sintaxes, ontologias e até aplicações de som.

   Embora tenham facilitado o processamento em cadeias temporais, as redes RNN apresentam um problema intrinsecamente ligado à grandes dependências de probabilidades: o problema do desaparecimento do gradiente(*vanishing gradient*). Este problema ocorre, em parte, dado que cada célula anexa valores probabilísticos aos dados em processamento. Contudo, a união de probabilidades por mais de três células RNNs consecutivas gera frequentemente probabilidades muito baixas. Estas probabilidades baixas sofrem desaparecimentos indesejados durante as etapas de cálculo de gradiente, considerando que valores tão pequenos quanto se queira são confundidos com zero para cálculo de propagação do erros e reajuste dos pesos.
4. LSTM: Long-Short Term Memory podem ser consideradas um sub-campo de RNN, porém a principal contribuição dessa abordagem foi a distinção de duas cargas em

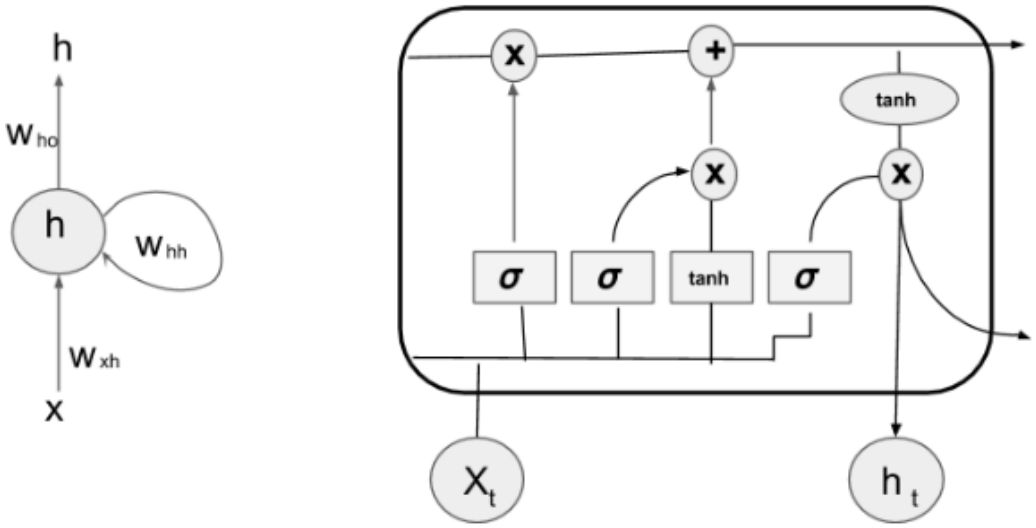
Figura 2.3: Modelo de Redes Neurais Recorrentes



cada célula: Uma carga de curta duração e outra carga de alta duração (HOCHREITER; SCHMIDHUBER, 1997). Para entender a motivação disto, precisa-se entender que durante o processamentos de grandes sequências encadeadas de palavras é corriqueiro que a aplicação recaia em um problema de desaparecimento de gradiente(vanishing gradient) (BENGIO; SIMARD; FRASCONI, 1994).

Conforme explicou-se no item de redes RNN, o algoritmo passava a interpretar o gradiente do sinal próximo de zero como zero e a informação se perdia. Com a introdução dos sinais de curta e longa duração, conseguiu-se mitigar esse problema e garantir que probabilidades, que eram pequenas para longas sequências, se tornassem processáveis de fato. Pode-se comparar essa técnica como uma realimentação periódica do sinal, que possui certa similaridade com a técnica de blocos residuais, que será explicada na próxima seção. Dessa forma, o sinal de longa duração auxilia os cálculos de longas dependências, enquanto o sinal de curta duração efetua os cálculos de trechos de composição mais curtos, que por sua vez retroalimentam o sinal de longa duração. Como exemplo abstrato, pode-se pensar em uma aplicação LSTM de predição de palavras em um discurso. Se a rede contém o trecho "O gato subiu na", ela é capaz de predizer com maior chance que a próxima palavra deve ser "árvore"e não "céu", pois a célula que processa a palavra "na"identifica probabilisticamente que a próxima palavra deve ser feminina e a segunda célula identifica alta probabilidade em longas dependências entre as palavras "gato"e "árvore". Nesse caso, a quarta célula transmite o sinal curto e a segunda célula transmite o sinal de longa duração. É notório, porém, que essas redes possuem uma complexidade maior para processamento, levando mais tempo médio para treinamento do que os outros tipos de redes.

Figura 2.4: Modelo de célula RNN (à esquerda) comparado à modelo de célula LSTM (à direita)



2.2.1.1 Modelos FNN

Durante a evolução dos modelos de DL, os modelos mais primitivos faziam adaptações dos neurônios primitivos reprogramáveis introduzidos em 1958 pelo Perceptron (ROSENBLATT, 1958). Com múltiplas camadas e com a introdução da técnica de retropropagação, ou *backpropagation* (RUMELHART; HINTON; WILLIAMS, 1986), os perceptrons passaram a desenvolver o treinamento em menor unidades de tempo de forma a permitir reajustes para aumentar a precisão relativa.

Com o encadeamento de várias unidades de neurônios Perceptron, deu-se origem às redes Multi-Layer Peceptron(MLP). Essas redes são utilizadas até os dias atuais pois suas camadas ocultas são boas extratoras de características(*features*) do modelo que as acompanham, i.e. um modelo de outro tipo pode ser acompanhado de uma ou mais camadas MLP. Outro ponto digno de nota é que as redes MLP também recebem a denominação de Fully-Connected(FC) quando são consideradas camadas de uma rede maior.

Os estudos desse tipo de rede são bem limitados em DL, uma vez que o foco da área migrou rapidamente para o domínio de imagem e texto, beneficiados pela introdução das redes CNN e RNN, como será visto mais adiante.

A grande menção de trabalhos de sucesso de redes MLP é o trabalho de reconhecimento de caracteres de código de endereçamento postal por reconhecimento de dígitos (LECUN et al., 1989). Este trabalho é dado como o precursor do modelo que é chamado de MNIST nos dias atuais. O MNIST é um modelo simples com duas camadas FC, porém hoje é implementado no domínio CNN, com duas camadas prévias de convolução

(LECUN et al., 1998), que processam e aprendem a identificar dígitos de zero à nove escritos à mão. Trata-se de um modelo bastante utilizado nos dias atuais como introdutor aos ensinamentos de DL. Outro modelo que ganha bastante destaque junto ao MNIST é o CIFAR-10, que visa classificar um dataset em 10 classes específicas com operações similares ao modelo MNIST.

Embora os modelos sejam muito primitivos, o mais importante desses dois trabalhos consta nos datasets (LECUN; CORTES; BURGES, 2010; KRIZHEVSKY; NAIR; HINTON, 2014), pois são casos extremamente conhecidos que servem para testar de forma simples modelos bem complexos. Por essa razão, esse trabalho optou por utilizar desses modelos e datasets.

Seguindo os modelos FNN, vieram os modelos com camadas de convolução, que demonstraram uma adaptação imensa às aplicações de DL em imagem quando postas junto às redes FNN. Esses modelos serão explicados a seguir.

2.2.1.2 Modelos CNN

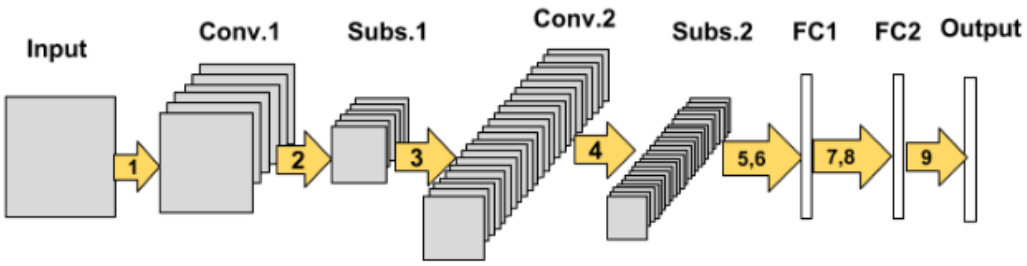
No campo das redes CNN, este trabalho apresenta os modelos LeNet, AlexNet, VGG-16, ResNet e InceptionV1 (LECUN et al., 1989; KRIZHEVSKY; SUTSKEVER; HINTON, 2012; SIMONYAN; ZISSERMAN, 2014; HE et al., 2016; SZEGEDY et al., 2016).

O modelo **LeNet** pode ser considerado um divisor de águas em relação aos modelos de DL. Esta rede implementou uma ordem de operações de convolução, ativação e *downsampling* que possibilitaram um grande aumento na precisão na tarefa de reconhecimento de dígitos escritos à mão (LECUN et al., 1989).

Seguindo uma ordem específica:

1. 6 convoluções 2D com filtro de dimensão 5x5.
2. Ativação ReLU e Max Pooling.
3. 16 convoluções 2D com filtro de tamanho 5x5.
4. Ativação ReLU e Max Pooling, com Normalização(*Flatten*).
5. Camada MLP com 120 neurônios.
6. Ativação ReLU.
7. Camada MLP com 84 neurônios.
8. Ativação ReLU.
9. Camada MLP com 10 neurônios.

Figura 2.5: Arquitetura do Modelo LeNet



O LeNet adquiriu assim, a incrível façanha de implementar um dos primeiros casos de sucesso de rede neural convolucional como é conhecida nos dias atuais. Apesar de ser considerado um modelo inovador, o LeNet foi inicialmente treinado usando apenas datasets pequenos de dígitos escritos à mão, como o MNIST dataset (LECUN; CORTES; BURGES, 2010).

O modelo **AlexNet** (KRIZHEVSKY; SUTSKEVER; HINTON, 2012) sucede o LeNet em contribuições ao apresentar um modelo CNN que alcança maiores taxas de precisão. Entre as melhorias, a rede AlexNet possui mais profundidade, i.e. maior número de camadas de convolução. Além disso, este modelo opera sobre um dataset de tamanho grande e com grande número de classes: o dataset ILSVRC 2012, também conhecido como ImageNet (RUSSAKOVSKY et al., 2015).

Esta rede também optou por adicionar um dropout de 0.5, ao passo que o LeNet não introduzia essa técnica. Além disso, o AlexNet aumentou consideravelmente o número de neurônios na camada Fully Connected: Seu modelo conta com 8192 neurônios na FC, enquanto seu predecessor LeNet conta com apenas 204 unidades.

Por fim, AlexNet também contribuiu com duas particularidades:

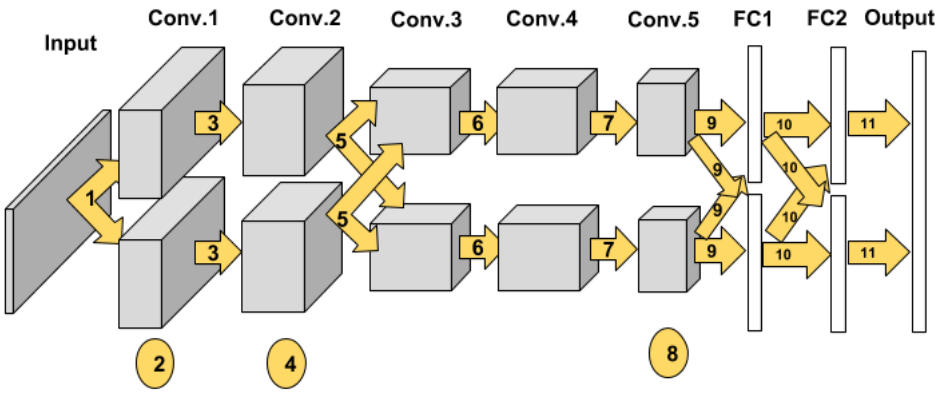
- O resultado da convolução da primeira camada é dividido em dois grupos, que continuam o treinamento sem mútua interferência até se encontrarem na primeira camada FC;
- A terceira camada de convolução dessa rede conta com filtros de convolução colaborativos, i.e. os filtros do primeiro grupo compartilham-se com os filtros do segundo grupo buscando aprimorar a convolução daquela e apenas daquela camada.

Seguindo uma ordem específica, as operações que ocorrem no treinamento do modelo AlexNet são:

1. 96 convoluções 2D com filtro de dimensão 11x11. Ocorre uma separação em dois grupos de 48 resultados.

2. Ativação ReLU e Max Pooling.
3. 256(128x2) convoluções 2D com filtro de tamanho 5x5.
4. Ativação ReLU e Max Pooling.
5. 384(192x2) convoluções 2D com filtro colaborativo de dimensão 3x3. Aqui é a etapa onde os filtros da segunda camada de convolução interagem entre si.
6. 384(192x2) convoluções 2D com filtro de dimensão 3x3.
7. 256(128x2) convoluções 2D com filtro de dimensão 3x3.
8. Ativação ReLU e Max Pooling, seguidos de Dropout.
9. Camada MLP com 4096(2048x2) neurônios, seguido de Dropout.
10. Camada MLP com 4096(2048x2) neurônios, seguido de Dropout.
11. Camada MLP com 1000 neurônios.

Figura 2.6: Arquitetura do Modelo AlexNet



Como a rede AlexNet utilizou de forma bem sucedida o ImageNet, a mesma consolidou uma tendência forte em direção a aumentar-se o número de camadas de convolução, testando inclusive tamanhos diferentes de filtro de convolução e realizando uma colaboração pontual nos mesmos. Contudo, muito embora a rede tenha aumentado a performance em competições de imagem, não foi possível inferir exatamente qual foi a causa do aumento no sucesso da rede até aquele ponto. Inclusive, o aumento do número de neurônios na camada MLP foi reportada pelo trabalho como um fator decisivo ao aumento de precisão, quando na verdade, trabalhos futuros como o ResNet e o Inception concluíram que cerca de mil unidades já eram o suficiente para extrair as informações essenciais. Mesmo assim, as contribuições de AlexNet perduram no tempo, de forma que ainda consta como uma arquitetura de CNN bastante usada.

A grande contribuição do **VGG-16** (SIMONYAN; ZISSERMAN, 2014) se dá pela

escolha de projeto pela redução do tamanho dos filtros de convolução. Dessa forma, os projetistas da rede conseguiram aguçar a extração de conceitos nas camadas de convolução sem prejudicar a precisão do modelo, mas também garantiram uma redução nos parâmetros intermediários de treinamento. Isto abriu caminhos para a adição de mais camadas de convolução ao modelo.

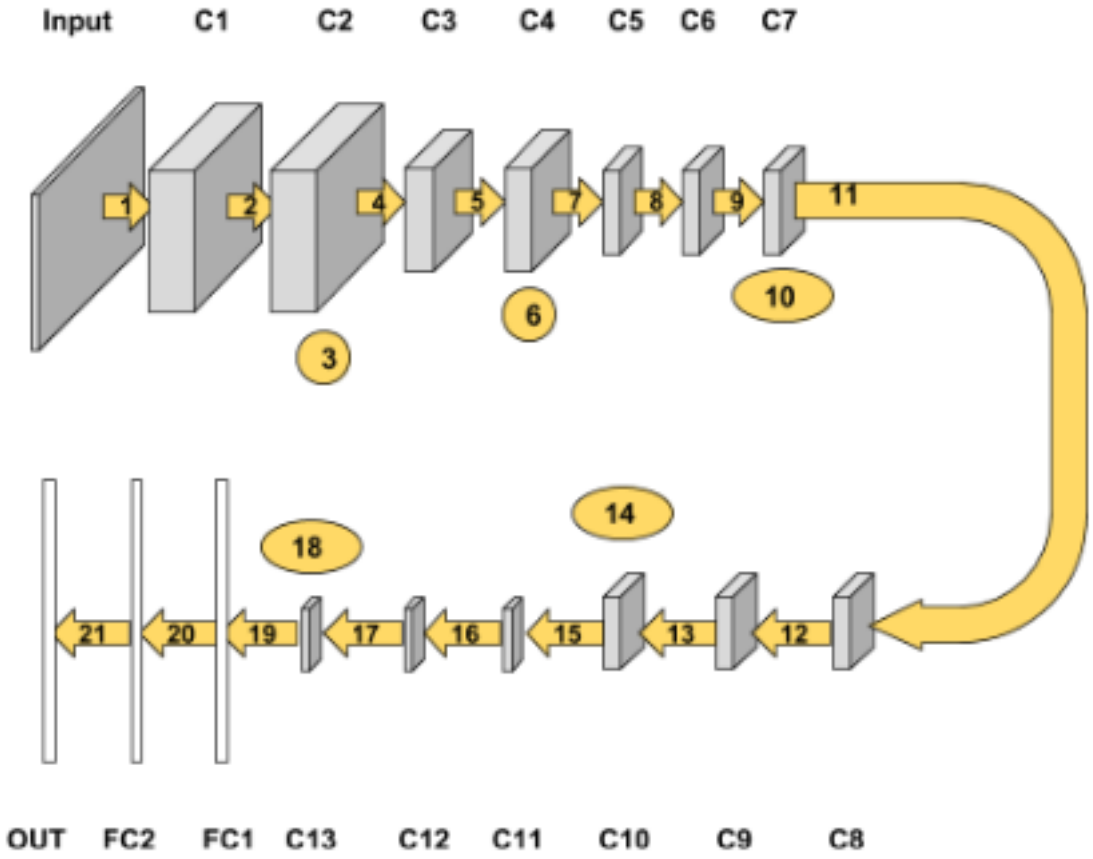
Assim como seu predecessor, o VGG utiliza o ImageNet (RUSSAKOVSKY et al., 2015) dataset para treinamento e também optou por manter um alto número de neurônios na camada FC: 9192 ao total. Este modelo foi o último trabalho renomado a usar tantos neurônios na FC, por razões que serão explicadas mais adiante.

As ordens das operações de treinamento do VGG-16 são:

1. 64 convoluções 2D com filtro de dimensão 3x3.
2. 64 convoluções 2D com filtro de dimensão 3x3.
3. Ativação ReLU e Max Pooling.
4. 128 convoluções 2D com filtro de dimensão 3x3.
5. 128 convoluções 2D com filtro de dimensão 3x3.
6. Ativação ReLU e Max Pooling.
7. 256 convoluções 2D com filtro de dimensão 3x3.
8. 256 convoluções 2D com filtro de dimensão 3x3.
9. 256 convoluções 2D com filtro de dimensão 3x3.
10. Ativação ReLU e Max Pooling.
11. 512 convoluções 2D com filtro de dimensão 3x3.
12. 512 convoluções 2D com filtro de dimensão 3x3.
13. 512 convoluções 2D com filtro de dimensão 3x3.
14. Ativação ReLU e Max Pooling.
15. 512 convoluções 2D com filtro de dimensão 3x3.
16. 512 convoluções 2D com filtro de dimensão 3x3.
17. 512 convoluções 2D com filtro de dimensão 3x3.
18. Ativação ReLU e Max Pooling.
19. Camada MLP com 4096(2048x2) neurônios.
20. Camada MLP com 4096(2048x2) neurônios.
21. Camada MLP com 1000 neurônios.

Reconhecem-se nesse modelo algumas colaborações importantes, tais como: a

Figura 2.7: Arquitetura do Modelo VGG-16



manutenção do *Dropout* como fator de redução do tempo de treinamento da rede mais profunda, aumento considerável na profundidade, consolidação da importância de *kernels* de tamanhos pequenos e também o alto número de neurônios na camada FC, que ajudou trabalhos futuros a entender melhor sobre a profundidade da real contribuição da camada FC e suas limitações.

Outro ponto que deve ser percebido é que a partir deste trabalho ocorrem amplificações gigantescas no número de camadas de convolução e em suas formas. Por isso, a partir deste ponto, as convoluções são relatadas pelo número de camadas e não pelo número de operações, visto que os modelos a seguir superam com facilidade seus antecessores em número de operações.

O modelo **ResNet** (HE et al., 2016) também introduziu mais profundidade aos modelos de CNN do estado-da-arte, mas sua grande colaboração não vem exclusivamente por conta disto. Os autores perceberam que modelos pequenos de arquitetura CNN conseguiam aprender funções próximas do ótimo para a resolução de problemas reais, i.e. funções identidade que poderiam ser exportadas como melhores soluções para uma ins-

tância muito específica de problema.

Com o tempo, passou-se a acreditar que a combinação dessas funções identidade conseguiria-se chegar à uma solução ótima que englobaria um maior conjunto de aplicações. Contudo, experimentações revelaram que a integração das mesmas não ocorre de forma boa (HE et al., 2016). Ficou evidente que modelos identidade não se adaptavam bem com outros, uma vez que necessitariam de certo número mínimos de passagem de treinamento ou ajuste para funcionamento próprio. Uma vez encadeadas, essas funções seriam modificadas umas em relação às outras mesmo que por pouco tempo de treinamento.

Isto motivou o grupo de ResNet a introduzir o conceito de Blocos Residuais. Sua ideia consiste em pegar partes do modelo composto que sabidamente introduzem funções próximas à identidade e realimentar a entrada de cada neurônio daquele bloco à saída do mesmo. Os desenvolvedores argumentam que essa técnica reduz o tempo para que cada bloco aproxime-se do mapeamento de sua função identidade em relação à descoberta da melhor combinação global de todas para todas as funções identidade do modelo completo.

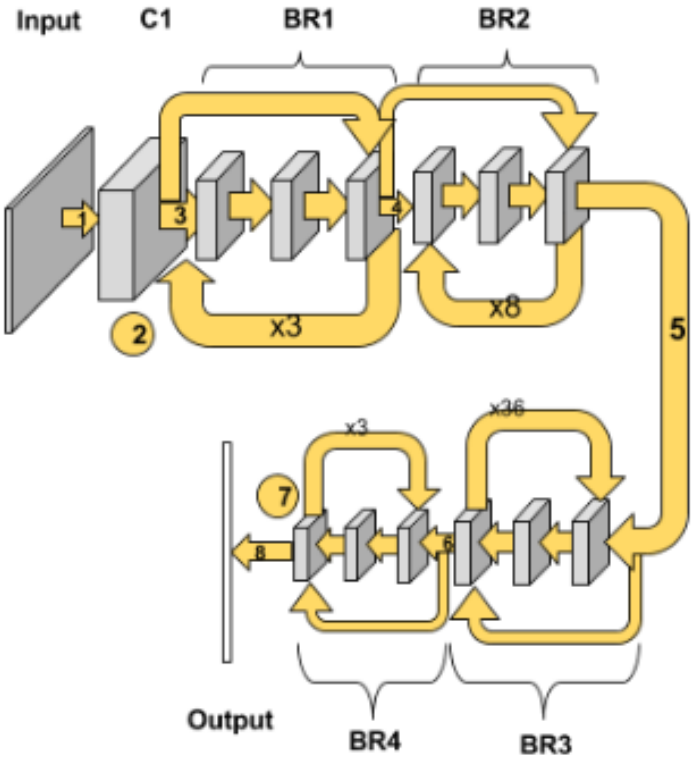
Dessa forma, o segundo neurônio de convolução daquele bloco residual processaria a saída do primeiro, ao passo que sendo o último do bloco, teria sua à saída adicionada a entrada do primeiro neurônio. Dessa forma, reduziu-se o impacto do processamento daquele bloco para blocos posteriores em relação ao dado de entrada. Com essa nova técnica, os desenvolvedores de ResNet descobriram que as várias funções identidade unidas em uma grande rede tendiam à se aproximar de forma melhor às funções identidade desejadas. Isto consiste de uma grande contribuição às redes convolucionais, pois permitiu o aumento exacerbado de profundidade desses modelos, garantindo assim maior extração de conceitos complexos, aumentando a precisão.

1. Primeira camada de convolução com filtro 7x7.
2. Ativação ReLU e Max Pooling.
3. Primeiro bloco residual com três repetições de conjunto de convolução: 1x1, 3x3 e 1x1.
4. Segundo bloco residual com oito repetições de conjunto de convolução: 1x1, 3x3 e 1x1.
5. Terceiro bloco residual com trinta e seis repetições de conjunto de convolução: 1x1, 3x3 e 1x1.
6. Quarto e último bloco residual com três repetições de conjunto de convolução: 1x1,

3x3 e 1x1.

7. Ativação ReLU e Average Pooling.
8. Camada FC com 1000 neurônios.

Figura 2.8: Arquitetura do Modelo ResNet



Como primeiro grande ponto a ser observado pela arquitetura de ResNet, nota-se que os criadores do modelo optaram pelo uso de mesma combinação de convoluções em todos os blocos residuais: 1x1, 3x3 e 1x1. Essa tendência de manter os filtros de convolução pequenos vêm do modelo VGG, porém diverge do mesmo ao optar por uma redução drástica no número de neurônios da FC, mudando de 9192 para 1024. A escolha desse tamanho é fundamentada em resultados *a posteriori*, que indicam que a partir de cerca de 1000 neurônios o modelo não usufrui de ganhos reais em precisão. Portanto, fica evidenciado a limitação da camada FC, que é corroborada pelo modelo a seguir.

Uma rede que ficou muito conhecida no estado-da-arte e ainda é amplamente utilizada é a arquitetura **Inception** (SZEGEDY et al., 2016). Desenvolvida pelo Google, também pode ser chamada de **GoogLeNet**, pois é baseada na estrutura simples da LeNet, com uma série otimizações. Todos os modelos até este ponto focaram no aumento da profundidade de convoluções, alternando ou não os tamanhos dos *kernels* das operações, realizando modificações na quantidade de neurônios FC e otimizações específicas.

A genialidade de GoogLeNet se encontra no fato de que manteve os neurônios da FC em quantidades baixas, assim como o ResNet, mas focou em encapsular múltiplas convoluções paralelas em pequenos blocos de execução que denominou como Inception. A partir desta arquitetura, qualquer cientista de dados pode combinar quantias de convoluções em paralelo ao adaptar um Inception. Dessa forma, o modelo ganhou notoriedade com o uso de múltiplos filtros de convolução assimétricos durante a execução do Inception em si, garantindo assim a extração de conceitos simples e complexos em estágios avançados de convolução.

O modelo Inception, embora largo, pode ser sumarizado fielmente com um conjunto de operações:

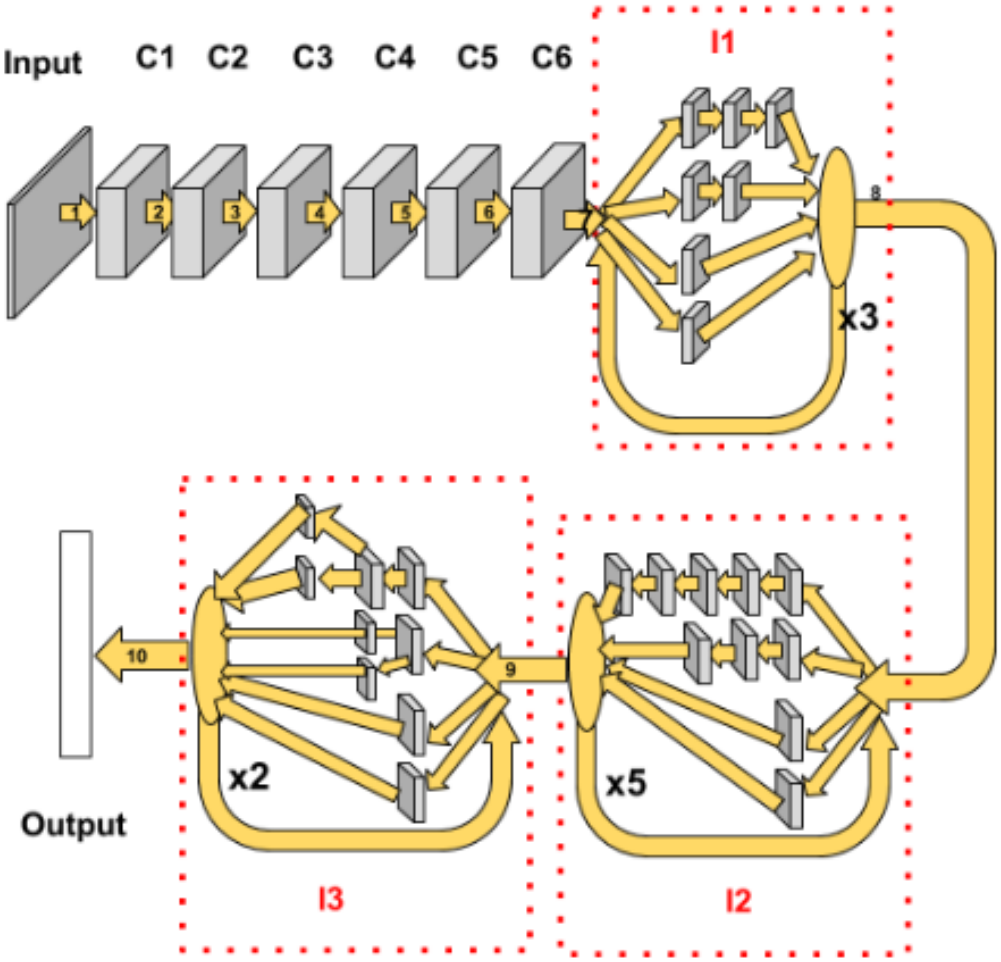
1. Camada de convolução com *kernel* de convolução 3x3
2. Camada de convolução com *kernel* de convolução 3x3
3. Camada de convolução com *kernel* de convolução 3x3
4. Camada de convolução com *kernel* de convolução 3x3
5. Camada de convolução com *kernel* de convolução 3x3
6. Camada de convolução com *kernel* de convolução 3x3
7. Três repetições do primeiro módulo Inception(I1), dos quais estão:
    - Primeiro fluxo com três convoluções consecutivas, de *kernels* 1x1, 3x3, 3x3.
    - Segundo fluxo com duas convoluções consecutivas, de *kernels* 1x1, 3x3.
    - Terceiro fluxo com uma convolução de tamanho 1x1.
    - Quarto e último fluxo de convolução de filtro 1x1.
8. Cinco repetições do segundo módulo Inception(I2), contendo:
    - Primeiro fluxo com cinco convoluções consecutivas, de *kernels* 1x1, 1x7, 7x1, 1x7, 7x1.
    - Segundo fluxo com três convoluções consecutivas, de *kernels* 1x1, 1x7, 7x1
    - Terceiro fluxo com uma convolução de tamanho 1x1.
    - Quarto e último fluxo de convolução de filtro 1x1.
9. Duas repetições do terceiro módulo Inception(I3), abrangendo:
    - Primeiro fluxo com quatro convoluções com sub-paralelismo, primeira convolução de *kernel* 1x1, seguida da segunda convolução de 3x3, cujo resultado é distribuído em dois fluxos paralelos de *kernels* 1x3 e 3x1. Finalmente essas

duas se juntam em um filtro concatenador, para serem passados ao segundo fluxo.

- Segundo fluxo com três convoluções e com sub-paralelismo, sendo a primeira convolução de *kernel* 1x1, dividindo-se em dois sub-fluxos de tamanhos 1x3, 3x1.
- Terceiro fluxo com uma convolução de tamanho 1x1.
- Quarto e último fluxo de convolução de filtro 1x1.

10. Camada FC com 1024 unidades.

Figura 2.9: Arquitetura do Modelo Inception



Primeiramente, nota-se que a contribuição mais significante do modelo se dá pela inserção de paralelismo em blocos bem definidos. Porém, é ainda mais digno de nota que o Inception introduza dois níveis de paralelismo: um a nível de bloco de execução para bloco de Inception e outro dentro do próprio bloco, onde dividem-se fluxos em sub-fluxos.

Nota-se também o volume de convoluções quando contam-se as operações incluindo os blocos de Inceptions: 6 convoluções inciais, 7 convoluções no I1 com três repetições(21 iterações), 10 convoluções no I2 por 5 vezes(50 iterações), 9 convoluções no I3 por duas vezes(18 iterações), totalizando 95 operações de convolução por passagem de treinamento.

Trata-se de um modelo com profundidade consideravelmente maior do que seus predecessores, que colaborou em confirmar que o foco em camadas de convolução ainda continham potencial não explorado para aumento de inferência do modelo, ao passo que o tamanho da FC serviria apenas de suporte às camadas de convolução. Portanto, confirmando o que o modelo ResNet também determinou, que a partir de certa quantidade de *neurons* na FC, não há ganho em precisão, mas há ganho em tempo de processamento, levando assim a *overheads* desnecessários.

2.2.1.3 Modelos RNN

No campo das redes RNN, este trabalho explica os modelos Text Classifier em LSTM, Tree-LSTM e Simple Recurrent Neural Network.

Esses modelos são baseados no modelo GloVe -acrônimo de *Global Vectors for Word Representation*- pré treinado para realizar a conversão de palavras em vetores (PENNINGTON; SOCHER; MANNING, 2014). Utilizam-se dessa representação vetorial de palavras para classificar o dataset 20 Newsgroup em 20 categorias diferentes.

A técnica de representação vetorial dos dados pode variar de acordo com a técnica aplicada, porém os tipos mais relatados na literatura são Vector Space Model (VSM), o Vector Representation of Words, Continuous Bag of Words, Word2Vec, Skip-Grams, Global Co-occurrence Statistics-based Word Vector e GloVe.

Este trabalho também optou por executar as redes Text Classifier em CNN a fim de validar os resultados da execução do Text Classifier em LSTM. Os resutlados dessa comparação serão demonstrados no próximo capítulo.

2.2.2 Frameworks

Dentre os frameworks mais famosos de DL, se encontram:

- **TensorFlow**: é um *framework open-source* de *DL* desenvolvido inicialmente por parte do grupo de projeto Google Brain, que especializou-se em aperfeiçoar esta

ferramenta (ABADI et al., 2016). Sem sombra de dúvidas é a ferramenta mais famosa de *DL* e muito embora sua hegemonia no campo seja algo discutível, é um *framework* bastante confiável, pois conta com: ótima documentação de sua *API*, incorporação de otimizações de *frameworks* diferentes, grande equipe de desenvolvimento, suporte, manutenção e larga comunidade de usuários.

- **Caffe**: desenvolvido e mantido pela equipe de Berkeley Vision and Learning Center (BVLC) e também contando com uma comunidade digna de nota, Caffe (JIA et al., 2014) surgiu como uma grande opção de ambiente de programação em DL durante o crescimento da área. Suas principais virtudes residem na facilidade de escrever modelos, graças à uma estrutura própria de descrição similar à um arquivo XML e sua implementação simples em C++, que garantiu por muito tempo menor exaustão de recursos do que em relação a outros *frameworks*. Havia, no entanto certas limitações quanto à compatibilidade de bibliotecas gráficas que o modelo podia incorporar e quanto ao espectro de tipos de redes que poderia desenvolver, observando-se, por exemplo, que modelos RNN eram muito limitados e não havia suporte a redes LSTM.
- **Theano**: desde novembro de 2017, com o lançamento de sua versão 1.0, o Theano (BASTIEN et al., 2012) parou de ser desenvolvido pelo grupo MILA. O trabalho de Theano facilitou a escrita de modelos DL, enquanto introduzia várias otimizações que foram absorvidas por grandes ferramentas do campo, entre as melhorias estão: manifestação de modelos como expressões matemáticas, execução transparente em GPUs, reescrita de grafos computacionais para otimização de uso de memória e performance e diferenciação automática de alta-ordem. Muito embora se encontre sem continuação, o software é mantido com documentação e versão estável para a comunidade.
- **Torch/Torch7/PyTorch**: Torch (COLLOBERT; BENGIO; MARIÉTHOZ, 2002) foi introduzido com uma implementação mista de LuaJIT com C/CUDA com ênfase em maximização de GPUs. Sendo sucedido pela introdução de Torch7 e PyTorch que são duas adaptações que visam otimizar o Torch para outras linguagens de programação.
- **CNTK**: O Computational Network Tolkit, ou CNTK (SEIDE; AGARWAL, 2016), desenvolvido pela Microsoft, consiste em um *deep-learning toolkit* de código aberto para treinamento e avaliação de redes neurais. Microsoft utiliza o CNTK no desenvolvimento de modelos de análise de discurso utilizados na Cortana e web ranking.

Suporta redes feed-forward, convolucionais e de recorrência.

- **Chainer**: Chainer (TOKUI et al., 2015) é um framework que, segundo os autores, faz parte de uma segunda geração de frameworks de DL. Ele provê um meio mais fácil e objetivo de implementar arquiteturas de redes neurais mais complexas. Utiliza como linguagem de programação o Python e implementa diversos métodos de otimização, tais como: SGD, AdaGrad, Adam e RMSprop. Mas o principal aspecto do Chainer está na relação estreita entre a definição do modelo e seu treinamento.

Dentre as ferramentas aqui descritas, relata-se que o framework TensorFlow continua sendo o mais usado, sendo inclusive referenciado novamente ao longo deste trabalho. As razões para tamanho uso constam no fato de que boa parte das otimizações dos outros frameworks foram incorporadas pelo TensorFlow, bem como no fato de que esta ferramenta conta com uma equipe de desenvolvimento e suporte grande.

2.2.3 DL *Pipelines* Para Apache Spark

O Spark possibilita ao desenvolvedor distribuir a alta demanda computacional exigida pelo DL em um cluster de maneira paralela e distribuída (GUPTA et al., 2017b). Para tornar isso possível, pode-se optar alguma das seguintes ferramentas: Keras (CHOLLET et al., 2015) & PySpark, TensorFlowOnSpark (YANG et al., 2017), BigDL (WANG et al., 2018) e DeepSpark (KIM et al., 2016). Além disso, o Spark também oferece as seguintes bibliotecas:

ML library (MLlib): é formado por algoritmos de aprendizagem e utilitários estatísticos comuns. Entre as suas principais funcionalidades, incluem-se: Classificação, regressão, agrupamento, filtragem colaborativa, otimização e redução de dimensionalidade. Esta biblioteca foi especialmente projetada para simplificar *pipelines* ML em ambientes de grande escala. Nas versões mais recentes do Spark, a biblioteca MLlib foi dividida em dois pacotes: MLlib com compilação sobre de RDDs e ML com compilação sobre do DataFrames para a construção de *pipelines* (MENG et al., 2016).

Spark GraphX: é o sistema de processamento de grafos em Spark. Graças a este motor, os usuários podem visualizar, transformar e se juntar de forma intercambiável, tanto em grafos como em coleções. Ele também permite expressar a computação de grafos usando a abstração de Pregel (MALEWICZ et al., 2010).

2.3 Resumo

Este capítulo apresentou o estado da arte em BD e ML. Foram apresentados aspectos técnicos e científicos que levam à necessidade da convergência de soluções que unam ambas as áreas. Nesse contexto, o próximo capítulo ira apresentar o atual estado da arte para o processamento BD atrelado à Inteligência Artificial propiciado pelo DL.

3 ESTADO DA ARTE

Este capítulo apresenta o estado da arte relacionado ao processamento de aplicações DL sobre *frameworks* BD, sendo destacadas as características dessa abordagem e suas principais vantagens.

3.1 Trabalhos Relacionados

Como citado na capítulo anterior, as áreas de BD e DL convergiram na busca por novas e mais eficazes soluções que realizem extração de informações a fim de obter valor dos dados produzidos a cada instante. A busca pela utilização dessas duas ferramentas em conjunto é um tópico pouco abordado na literatura, dado o quão recente esses campos são, porém alguns trabalhos tentam abordar tentativas iniciais de como fazê-la. Os trabalhos que realizam essas tentativas formam o escopo desta monografia e são discutidos presentemente.

O *framework* **DeepSpark** (KIM et al., 2016) consiste em 3 componentes principais: o Apache Spark, um intercambista de parâmetros (*parameter exchanger*) para SGD assíncrono e uma *engine* com suporte a GPU. Apesar do Spark fornecer uma camada de paralelismo de dados efetiva, a alta demanda por comunicação proveniente do SGD representa um alto custo. Outro fator que dificulta a execução desta integração reside no fato de que o Dataset Distribuído e Resiliente (RDD) do Spark provê operações assíncronas limitadas entre os *masters* e *workers* nativamente. De modo a resolver esses gargalos, foi implementado um método assíncrono SGD com um *custom parameter exchanger* sobre o ambiente Spark. Adicionalmente, foi projetada uma variante assíncrona de um *elastic averaging stochastic gradient descent* (EASGD) com objetivo de adaptar o período com que os parâmetros são atualizados, aliviando assim a quantidade de comunicação necessária. DeepSpark permite uma flexibilidade em seu uso, aceitando tanto os *frameworks* Tensorflow como Caffe para o treinamento das redes neurais. Neste trabalho, executaram-se os experimentos tanto em configuração *standalone* como em um ambiente distribuído, mas em ambos os casos foi utilizado GPGPU no processo de treinamento. Os experimentos demonstraram ganhos expressivos quando comparados ao Caffe "puro" e o CaffeOnSpark.

Outra ferramenta que é relatada nesse domínio é o **SparkNet** (MORITZ et al., 2015). O mesmo surgiu como uma opção de fácil utilização, mesmo para situações en-

volvendo restrições de *hardware* e banda de rede. SparkNet também possui interface para leitura de RDDs e uma interface Scala para interação com o *framework* Caffe que possibilita uma fácil paralelização de modelos Caffe com poucas alterações nos modelos originais. Ele apresenta uma comparação do método de paralelização do SGD desenvolvido para o SparkNet em relação ao algoritmo Naive Bayes. O artigo do SparkNet descreve as limitações de cada modelo em um cenário sem *overhead* de comunicação. Nos experimentos utilizam 2 conjuntos diferentes de configurações de máquinas. No primeiro avaliam SparkNet em *clusters* de 3, 5 e 10 nós, onde cada nó possui uma GPU. No segundo, avaliam o SparkNet onde cada nó possui 4 GPUs, sendo um *cluster* com 3 nós e outro com 6 nós. Em todos os casos de teste o gráfico também apresenta o desempenho do Caffe em uma execução *single-node*. Além disso, os autores quantificaram o *speedup* atingido pelo SparkNet em função do tamanho do *cluster*, frequência de comunicação e *overhead* na comunicação do *cluster*. Eles demostraram que é possível de se atingir uma boa performance sem que o SGD seja sincronizado durante cada iteração e atualização do mesmo.

Talvez o *framework* mais digno de nota dessa integração seja o **BigDL** (WANG et al., 2018), desenvolvido pela Intel para atuar sobre o Apache Spark de maneira a unificar todo o complexo *pipelines* que envolve as aplicações reais de DL com BD. Além disso, o *framework* dispõe de uma execução mais intuitiva do ponto de vista do usuário. O BigDL usa *Resilient Distributed Dataset* (RDD) provido pelo Spark e modela os elementos básicos dos dados usados nas computações das redes neurais como um *N-dimensional array*. Além disso, cada *record* utilizado no modelo de treinamento e predição é modelado como um *Sample* que é constituído de *input feature*, que pode ser um ou mais *N-dimensional arrays*, e um rótulo opcional. Na construção do modelo, assim como outros *frameworks* de DL, o BigDL usa representação por fluxo de dados (dataflow), onde cada vértice no grafo representa uma camada de rede neural. Pode ser definido especificamente pelo usuário qual método de otimização será utilizado, tal como SGD, AdaGrid, Adam, entre outros. O BigDL também permite que os usuários utilizem modelos pré-treinados no Caffe, Keras, TensorFlow, Torch ou BigDL. Esta ferramenta realiza o treinamento paralelo dos dados usando pequenos lotes (*mini-batch*) de SGD síncrono. Para a sincronização dos parâmetros, BigDL usa uma solução semelhante a AllReduce, mas que imita uma arquitetura com servidor de parâmetros. Também é importante salientar que é possível se executar o BigDL localmente numa JVM sem o Spark.

O artigo (GUPTA et al., 2017a) investiga soluções para extrair o máximo de infor-

mações partindo de características pré-existentes no *dataset* (DS) de entrada para ganhar em acurácia no modelo final, além de resolver o desequilíbrio nas classes de DS oriundos do mundo real. O *framework* apresentado no artigo pretende utilizar as vantagens do processamento de BD oferecidos pelo Spark, juntamente com as vantagens da utilização de DL em grandes *datasets*, utilizando, para tal, a técnica de Aprendizado em Cascata (Cascade Learning). Para efetuar isso, sua estrutura foi designada basicamente por três estágios: BD análise, *Cascading* e DL. Para o primeiro estágio são utilizados algoritmos de regressão, providos pela biblioteca MLLib do Spark, que implementam algoritmos de regressão. Os dados pré-processados passam por esses algoritmos criando um modelo de regressão que apresenta a probabilidade de cada *data-point* pertencer a uma classe binária. No segundo, são adicionadas as probabilidades advindas do primeiro estágio com o DS original formando o conhecimento (*'knowledge'*), que servirá como entrada para o terceiro estágio. No último estágio, o *knowledge* é usado para treinar uma arquitetura com *multi-layer perceptron* (MLP). Obteve-se ganho de acurácia nos dois únicos *datasets* testados, mas o *framework* apresenta dificuldades quando a classificação não é binária no primeiro estágio.

O trabalho (KHUMOYUN; CUI, 2016) apresenta um *framework* de DL distribuído e baseado em Spark cuja arquitetura consiste basicamente em 3 componentes principais, que são: o *Master*, o *Parameter Server* e os *Data Shards*. Todos esses serviços foram definidos sobre o Apache Spark. O *Master* executa o papel de *Spark Driver* e é responsável pela manutenção, coordenação e escalonamento do processo de treinamento. Ele inicializa os parâmetros do *Parameter Server* e as camadas das redes neurais, que residem no HDFS. O *Parameter Server* é particionado de maneira que cada partição é responsável pelos parâmetros de uma camada de rede neural. Já nos *Data Shards* é onde acontece todo processo de treinamento do modelo de fato. Cada *Data Shard* recebe um parte do DS e aplica o algoritmo SGD sobre os dados, treinando na sua própria réplica do modelo. Após isso, cada um destes atualiza os parâmetros em suas partições correspondentes no *Parameter Server*. O processo continua até que se tenha passado por todos *Data Points*. O *framework* apresenta um ganho linear no treinamento conforme o aumento do número de nós e uma redução exponencial na taxa de erros conforme o aumento do número de iterações.

Além dessas propostas, existem o **TensorflowOnSpark** (YANG et al., 2017) e o **CaffeOnSpark**(FENG; SHI; JAIN, 2016), desenvolvidos pela Yahoo, que são soluções que habilitam o Tensorflow e Caffe a executarem em *clusters* Apache Spark. Elas

suportam paralelismo de modelo e/ou dados, funções próprias dos *frameworks* e Tensorboard (para o TensorflowOnSpark). O TensorflowOnSpark é implementado utilizando-se de uma biblioteca reduzida e adaptada de TensorFlow chamada de Slim (SILBERMAN, 2017), que é bastante utilizada a fim de facilitar a implementação e testes de modelos no TensorFlow. Contudo, a ferramenta é um projeto dissidente do TensorFlow que não possui suporte confiável e que dificulta o entendimento das implementações de TensorflowOnSpark. Toda a implementação de TensorflowOnSpark reside na criação de uma classe Spark fundamental denominada TfNode, que encapsula os modelos TensorFlow e facilita a passagem dos mesmos para a execução distribuída em Spark.

Um levantamento sobre os principais *pipelines* de *frameworks* de DL sobre BD é apresentado em (LU et al., 2018). **DLoBD** se detém em analisar os resultados de modelos que exploram o paralelismo dos dados, focando no CaffeOnSpark, TensorflowOnSpark e BigDL, e avaliando secundariamente o MMLSpark e o CNTKOnSpark, com o objetivo de avaliar como os *pipelines* destes *frameworks* de DL sobre BD estão sendo projetados e porque necessitam de comunicação de alto desempenho em suas camadas internas. Também avalia tipos de gargalos enfrentados e outros fatores. Ele compara principalmente performance, escalabilidade, utilização de recursos e impacto de protocolos de comunicação. Utiliza-se dos datasets MNIST[1], CIFAR-10[2] e ImageNet[3], além de dois *clusters* com configurações diferentes para efetuar experimentos. Após isso, ele faz um análise mais aprofundada do TensorflowOnSpark, apresentando o funcionamento da comunicação entre as camadas por dentro do *framework* e o *overhead* nas camadas do YARN e do Spark. Como conclusão, discorre sobre os ganhos que uma comunicação baseada em RDMA proporciona, principalmente no que tange cargas de trabalho com DL, e ressalta a falta de *benchmarks* para esse tipo de *pipeline* de *frameworks*.

Em (AHN; KIM; YOU, 2018), efetuou-se uma avaliação de desempenho de BD distribuído sobre um *cluster* YARN usando HiBench *suite benchmark* e TensorflowOnSpark. Foram medidos tempo de execução, vazão, uso de CPU, uso de memória e vazão do disco segundo diversas cargas de trabalho divididas em 3 categorias: micro, SQL e ML. Como resultado ratificaram a boa escalabilidade proporcionada pelo Spark, constatando também uma degradação da performance quando o TensorflowOnSpark usa o HDFS ao invés do LFS (Local File System) para gerenciar os dados e verificando uma redução de 2.23x no tempo de treinamento e 1.22x no tempo de predição quando utilizado cenário de

[1] http://yann.lecun.com/exdb/mnist/
[2] https://www.cs.toronto.edu/ kriz/cifar.html
[3] http://www.image-net.org/

3 nodos. Como trabalhos futuros destacam pesquisa em métodos para redução de custos em plataformas de ML/DL quando aplicadas em BD *stream*.

3.2 Discussão

Os trabalhos acima citados se destacam e são importantes por apresentarem contribuições importantes para a integração das áreas de BD e DL. Além disso, reforçam o potencial estudo e implementação de novas soluções.

A Tabela 3.1 apresenta a comparação entre os trabalhos descritos neste capítulo. Para demonstrar a diferença entre os mesmo foram definidas cinco métricas (acurácia, erro da fase de teste, tempo de treinamento, integrações SGD até atualização e pontuação F1), métodos de otimização utilizados e datasets dos experimentos.

Tabela 3.1: Trabalhos Relacionados

Ref.	**Accuracy**	**Test Loss**	**Training Time**	**Iterações SGD até atualização**	**F1**	**Método de Otimização**	**Dataset**
(KIM et al., 2016)	Sim	Sim	Sim	Sim	Não	Asynchronous EASGD	ImageNet
(MORITZ et al., 2015)	Sim	Não	Sim	Sim	Não	SGD Modificado	ImageNet
(GUPTA et al., 2017a)	Sim	Não	Não	Não	Sim	Cascade Learning	H-1B Visa Applications, Cardiac Arrhythmia
(KHUMOYUN; CUI, 2016)	Sim	Não	Não	Sim	Não	Downpour SGD	Set of tweets
(WANG et al., 2018)	Sim	Sim	Sim	Sim	Não	SGD, AdamOpt e outros	ImageNet, MNIST, CIFAR-10, Text
(LU et al., 2018)	Sim	Não	Sim	Não	Não	Não consta	MNIST, CIFAR-10, ImageNet
(AHN; KIM; YOU, 2018)	Não	Não	Sim	Não	Não	Não Consta	MNIST, CIFAR-10
Proposta	**Sim**	**Não**	**Sim**	**Não**	**Não**	**SGD, AdamOpt e outros**	**CIFAR-10, MNIST, 20NewsGroup/ GloVe-6B Stanford Sentiment Trebank**

Embora seja uma métrica muito importante na validação de modelos de DL, a acurácia -ou precisão- foi desconsiderada no trabalho (AHN; KIM; YOU, 2018), o que é incoerente com trabalhos de DL, pois esta métrica é extremamente simples de ser incorporada à extração no código DL. De forma similar, os trabalhos (GUPTA et al., 2017a; GUPTA et al., 2017b) deixaram de coletar a simples métrica de tempo de treinamento, contrariando o senso comum de se buscar métricas simples, como fez a maioria dos trabalhos deste estado-da-arte.

A métrica que relaciona acurácia e relevância -F1-*score*- foi utilizada somente no (GUPTA et al., 2017a). Por outro lado, a métrica *Test Loss* só foi utilizada em dois de sete trabalhos do estado-da-arte. Isso demonstra que os trabalhos deste domínio, que é bem recente, estão procurando suas contribuições em métricas que os primeiros trabalhos não procuraram incorporar.

Nenhum dos trabalhos citados nesta tabela convergiu no método de otimização utilizado em seus testes, sendo que dois trabalhos em específico sequer chegam a mencioná-los. Em questão de DS, o ImageNet foi mais utilizado, seguido pelo MNIST e CIFAR-10. Este dado não representa surpresa alguma, pois conforme explicado na seção 2.2.1.1, a grande maioria dos modelos do estado-da-arte de DL é treinado em cima do DS ImageNet, mas também realiza treinamentos adaptados para MNIST e CIFAR-10, dado a simplicidade de se treinar nestes datasets.

Por fim, é importante salientar que, grande parte dos artigos deste estado-da-arte tendem a apresentar superficialmente os *frameworks* desenvolvidos, omitindo suas avaliações de resultados, bem como as aplicações executadas e os ambientes suportados neles. Desse modo, eles retratam muito brevemente a capacidade que possuem, além de revelarem uma baixa flexibilidade nas configurações dos treinamentos, como, por exemplo, na permutação de modelos e *datasets*. Também dificultam o usuário, pois muitas não possuem códigos abertos ao público, nem canais de comunicação de suporte e algumas das que possuem estas qualidades não atualizam suas implementações com frequência.

Esta monografia, portanto, baseou-se em uma busca detalhada na literatura para confeccionar este estudo abrangente do estado-da-arte para facilitar o desenvolvimento de experimentos nesse domínio. Tais experimentos serão explicados a seguir.

4 PROPOSTA

O estado da arte, apresentado no Capítulo 3 demonstra a tendência em adequar o uso de aplicações Deep Learning aos modelos Big data. De fato, a convergência entre às áreas é complexa e custosa por diversos fatores, como por exemplo a escalabilidade necessária para o processamento de dados de complexos *pipelines* ligados à IA. Desse modo, ao passo que os modelos, aplicações e *frameworks* são desenvolvidos e otimizados para o processamento distribuído, fatores como o desempenho e escalabilidade das soluções são postos à prova. Embora o estado da arte apresente estudos que demonstrem avaliação de desempenho (KIM et al., 2016; MORITZ et al., 2015; WANG et al., 2018; GUPTA et al., 2017a; KHUMOYUN; CUI, 2016; YANG et al., 2017; FENG; SHI; JAIN, 2016; LU et al., 2018; AHN; KIM; YOU, 2018), estes estudos não demonstram o real impacto que o processamento distribuído pode de fato causar em aplicações DL.

Levando em conta estes fatores, o presente trabalho possui o objetivo de avaliar o impacto causado na performance das aplicações DL junto ao modelo de programação paralelo e distribuído provenientes de *frameworks* de processamento BD. Para conduzir tal investigação, optou-se pelo uso do *framework* BigDL (WANG et al., 2018).

4.1 BigDL

O BigDL é um *framework* de DL distribuído para plataformas de BD criado e mantido pela Intel®. O mesmo é implementado como uma biblioteca sobre o Apache Spark, ao qual permite que usuários escrevam suas aplicações de DL em larga escala como aplicações Spark.

O *framework* BigDL oferece uma representação dos dados comum aos *frameworks* de DL e que ao mesmo tempo é suportada pelos sistemas de BD. Para suportar o processamento paralelo e distribuído o BigDL utiliza o *framework* Apache Spark de maneira otimizada.

Ao realizar a sincronização dos parâmetros de modo síncrono, o BigDL suprime a ineficiência do SparkNet na coleta e sincronização dos pesos das camadas das redes neurais, além do fato de que o BigDL supera a inabilidade com que o TensorflowOnSpark (YANG et al., 2017) e o CaffeOnSpark (FENG; SHI; JAIN, 2016) usam o Spark. Estes dois frameworks usam a camada de orquestração para alocar recursos do cluster a fim de executar o trabalho TensorFlow ou Caffe distribuído nas máquinas alocadas. Contudo,

a limitação reside no ambiente de execução: a carga de trabalho TensorFlow ou Caffe continua sendo executada fora do *framework* de Big Data e possui poucas interações com os *pipelines* de processamento analítico dessa solução. O BigDL propõe assim um middleware que opera simultaneamente nas duas ferramentas, mas que também encapsula a programação de DL no BD (WANG et al., 2018).

O design adotado pelo *framework* BigDL pode ser observado na Tabela 4.1. Nesta tabela é possível verificar quais são os tipos de rede, aplicações, domínios e datasets oferecidos pela ferramenta.

Tabela 4.1: Conjunto de Experimentos

Tipo	**Modelo**	**Dataset**
CNN	VGG	CIFAR10
CNN	ResNet	CIFAR10
CNN	LeNet	MNIST
CNN	Text Classifier (CNN)	20 NewsGroup/GloVe-6B
RNN	Text Classifier (LSTM)	20 NewsGroup/GloVe-6B
RNN	Simple Recurrent Neural Network	Tiny Shakespeare Texts
RNN	Tree-LSTM	Stanford Sentiment Treebank

Além destas combinações de modelo com dataset, três pontos referentes aos conteúdos da ferramenta BigDL são dignos de nota:

1. O BigDL contém um exemplo de rede do tipo AutoEncoder (AE), porém este trabalho não realizou uso do mesmo, visto que os demais trabalhos do estado-da-arte não possuem o mesmo nos seus domínios;
2. Graças ao grande tamanho do *dataset* ImageNet (aproximadamente 150 GB), este trabalho passou por inconsistências nos testes que levaram a erros de execução de treinamento. Por essa razão, delimitou-se o escopo de testes sem este dataset;
3. O modelo *Simple* RNN não foi implementado com as mesmas métricas dos outros modelos da ferramenta BigDL, por isso não foi considerado para o conjunto experimental.

Ainda, sobre os datasets do BigDL utilizados por esta monografia, pode-se destacar:

- CIFAR-10: É formado por um conjunto de imagens RGB com tamanho 32x32 separadas em 10 classes distintas. O conjunto é sub-divido em 50 mil imagens para treinamento e 10 mil imagens para testes;

- MNIST: Consiste em 70 mil imagens de dígitos escritos à mão em preto e branco, que foram centralizadas e tiveram seu tamanho normalizado e fixado em 28x28. Está classificado em 10 Classes diferentes, separado em um conjunto de treinamento composto por 60 mil imagens e um conjunto de testes composto por 10 mil imagens;
- 20NewsGroup: É uma coleção de aproximadamente 20 mil documentos de grupos de discussão separados em 20 diferentes tópicos;
- Stanford Sentiment Treebank: *Dataset* contendo críticas sobre filmes que foram coletadas originalmente do *website rottentomatoes.com* e publicadas por (PANG; LEE, 2005). É constituído por aproximadamente 12 mil sentenças e 215 mil frases únicas.

Por fim, neste trabalho procurou-se extrair a capacidade do BigDL em diferentes cenários de configuração e como as mesmas impactam sobre as execuções de diferentes tipos de redes neurais e até mesmo em cada modelo especificamente.

A análise de BD utilizando DL requer técnicas de processamento distribuídas utilizando múltiplos computadores. Essa asserção é confirmada pois este cenário contém muitos parâmetros a serem ajustados e muitos dados a serem aprendidos, dos quais requerem tempos de computação consideráveis para realizarem seus aprendizados (LEE; SHIN; SONG, 2017).

Apresenta-se dessa forma, uma relação do ganho que um ambiente distribuído trás para uma aplicação e/ou um modelo que executará sobre o BigDL. A seguir é apresentada a metodologia de experimentação utilizada por este trabalho.

4.2 Metodologia de Avaliação

Este trabalho avalia o conjunto de aplicações apresentado na Tabela 4.1 sobre os cenários de configurações listados na Tabela 4.2.

Tabela 4.2: Configuração Cluster

Quantidade de Nodos	**Cores**	**Mémoria(GB)**	***Armazenamento(TB)***
4 Nodos	16	112	4
8 Nodos	32	228	8
12 Nodos	48	336	12

Considerando a Tabela 4.2, propõe-se neste trabalho a avaliação do *Speed-UP* e eficiência da escalabilidade, bem como se procura mensurar o impacto na precisão dos

modelos DL ao utilizar o paradigma de processamento paralelo e distribuído, fornecido pelo *framework* Big Data - Spark. As métricas avaliadas no presente trabalho são: tempo em segundos (s) e precisão em modo (%).

Quanto ao conjunto de testes, a terminologia design de (JAIN, 1990) revela o número de experimentos, bem como as combinações a nível de fatores e numero de réplicas de cada experimento. O ambiente de testes de modelos e datasets pode ser descrito como a combinação de 5 modelos com 4 datasets em 6 possibilidades únicas de teste. Além disso, avaliou-se cada configuração standalone e em mais 3 cenários de configuração de cluster 4.2, gerando assim 24 configurações de execução distintas.

Embora (JAIN, 1990) apresente o número de experimentos ideal igual a 30 repetições a fim de garantir interdependência dos experimentos. Esta monografia reduziu o escopo dos testes para 10 repetições para cada configuração de ambiente, dado o alto custo do serviço de nuvem, tempo de treinamento elevado e baixo desvio padrão apresentado nas execuções. Em contra-partida, este trabalho certificou-se da interdependência de experimentos e garantiu-se de que as amostragens pertencessem ao intervalo de confiança I.C. de 95%). Ao total foram executados 240 experimentos.

4.2.1 Especificação do Ambiente

Esta subseção apresenta as especificações de software e *hardware* empregados neste trabalho.

4.2.1.1 Software

O cluster utilizou Sistema Operacional Ubuntu 16.04.5 LTS, sendo configurado o Apache Spark v2.2.0 em conjunto com o HDFS, provido pelo Hadoop v2.6.3 e o YARN para o gerenciamento de recursos. Sobre o sistema foram instalados e configurados Java OpenJDK 8 181, Scala 2.11.8 e Python 2.7.15. Após isso, foram instaladas as bibliotecas NumPy v1.15.4 e Six v1.11.0 em todos nodos. Por fim, o cluster BigDL versão 0.7.1 foi clonado do seu repositório oficial no GitHub e então compilado para o cluster pré-configurado.

[0]https://github.com/intel-analytics/BigDL.git

4.2.1.2 Infraestrutura

Inicialmente cogitou-se a utilização de GP-GPUs a fim de otimizar a carga de trabalho do TensorFlow no framework BigDL, contudo, o trabalho (DAI; LIU; WANG, 2017) realizou testes com configurações de BigDL em GP-GPUs e Intel® Xeon® CPU *clusters*, concluindo que as otimizações de BigDL para processamento em CPU, aliado ao uso intensivo da biblioteca Math Kernel Library (MKL) presente nos processadores Intel® Xeon®, inviabilizam ganhos no processamento em conjunto com GPU.

Por essa razão, o ambiente utilizado para execução dos experimentos foi um *cluster* CPU instanciado na Microsoft Azure, seguindo as configurações apresentadas na Tabela 4.2. Os nós computacionais pertencem a família D12v2 de configuração, observam-se as especificações na Tabela 4.3.

Tabela 4.3: Tabela de Configurações de Hardware

Componente	**Especificação**
Processador	Intel® Xeon® E5-2673 v3 (Haswell) 4 Core (2.4-3.1 GHz)
Mémoria	28,00 GiB RAM
Armazenamento	SSD 1TB

À medida que os cenários foram executados, o cluster foi reconfigurado com mais nós de computação para suprir os cenários de configuração descritos em 4.2.

4.2.1.3 Parâmetros de Configuração

O comando do Spark responsável pela submissão das tarefas aos worker nodes possui alguns parâmetros responsáveis pela especificação dos recursos que serão posteriormente alocados pelo YARN. Os principais parâmetros que foram ajustados para a execução dos experimentos são:

- Master: define o modo em que o Spark será executado. Pode ser definido como local ou yarn.
- Driver-memory: define o tamanho da mémoria que será destinada ao driver.
- Num-executors: define o número de executores que serão destinados a execução.
- Executor-cores: define o número de cores por executor.
- Executor-memory: define a quantidade de memória alocada para cada executor.
- Class: define o ponto de entrada da aplicação

Finalmente, foi criado um script em Shell para execução automatizada dos experi-

mentos e otimização do tempo de utilização do cluster e consequentemente dos recursos disponíveis na Microsoft Azure. Os resultados obtidos são apresentados a seguir.

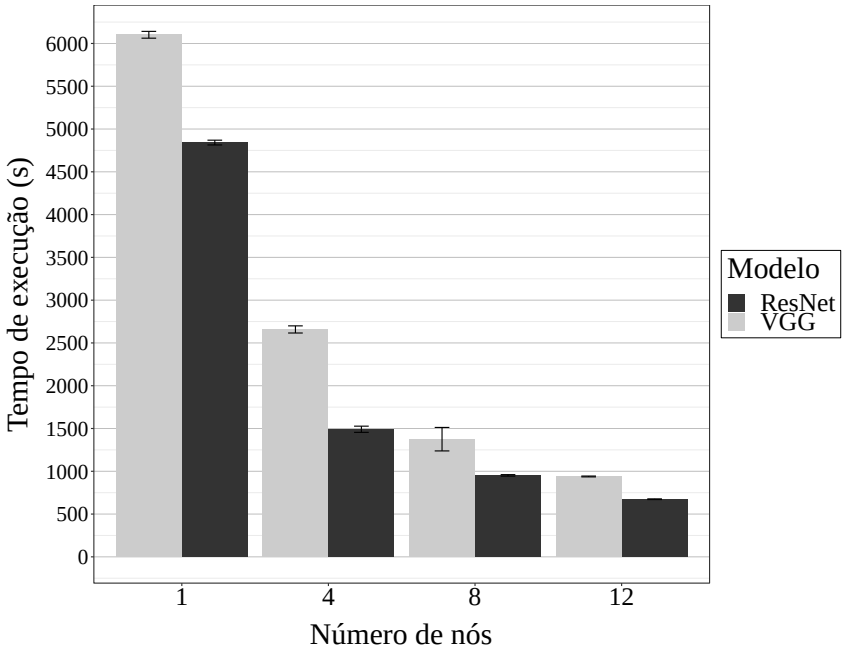
4.3 Resultados

Esta seção apresenta os resultados obtidos neste trabalho, inicialmente são apresentados os dados obtidos quanto a avaliação de desempenho das aplicações e modelos DL sobre BD, em seguida apresenta-se o impacto que ocorre na precisão dos modelos decorrente do processamento paralelo e distribuído e, por fim, apresenta-se uma análise de custo quanto a execução dos modelos sobre a nuvem.

4.3.1 Análise de Desempenho

A Figura 4.1 apresenta o desempenho dos modelos ResNet e VGG executados sobre o mesmo dataset, CIFAR-10. O modelo VGG obteve uma redução de 84,6% no tempo necessário para o treinamento quando comparada a execução em 12 worker nodes com a execução standalone, enquanto para o modelo ResNet a redução foi de 86,1%.

Figura 4.1: Comparativo para Rede CNN, Modelos ResNet e VGG, Dataset CIFAR-10

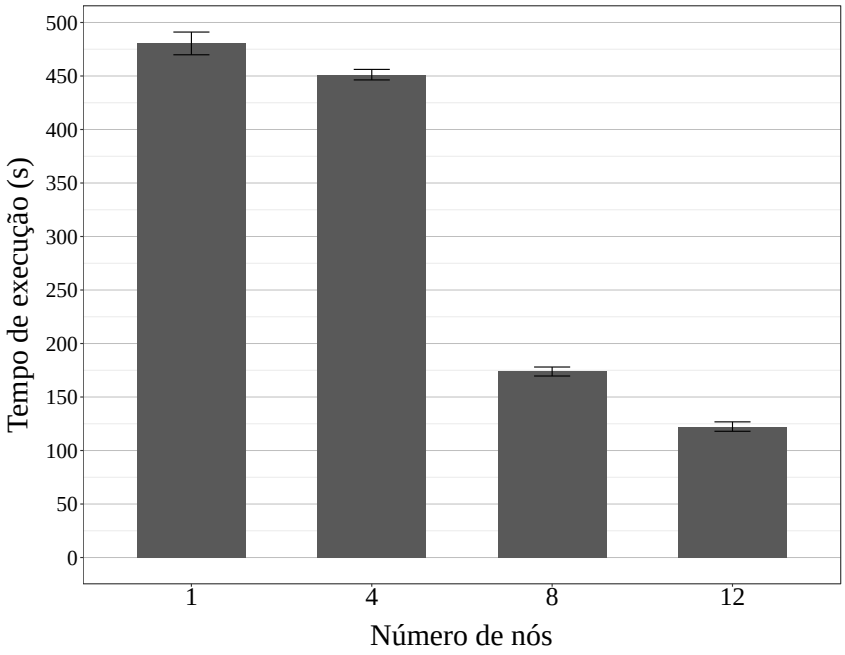


Embora o ganho de performance seja expressivo para ambos os modelos, é importante ressaltar que o modelo ResNet possui desempenho 20% superior ao VGG quando

comparado o cenário standalone e 28% após a execução distribuída com 12 nós de computação. Isso acontece porque a construção da réplica do modelo ResNet utilizando a técnica do bloco residual é mais rápida em cada worker node do que a construção da réplica do VGG.

Ao observar a Figura 4.2 pode-se notar o desempenho do modelo LeNet executado sobre o dataset MNIST. O modelo Lenet obteve uma redução de 74,5% no tempo necessário para o treinamento quando comparada a execução em 12 worker nodes com a execução standalone.

Figura 4.2: Rede CNN, Modelo LeNet - Dataset MNIST



Verifica-se pela figura 4.2 que a redução no tempo de treinamento não é expressiva quando executado sobre 4 nós. Isso ocorrer porque a carga de trabalho não é processada de forma rápida suficiente quando processada com poucos nós de computação. Ainda, embora o paralelismo propicie o ganho de desempenho, o uso intensivo de CPU e memória podem expressar gargalos durante a execução.

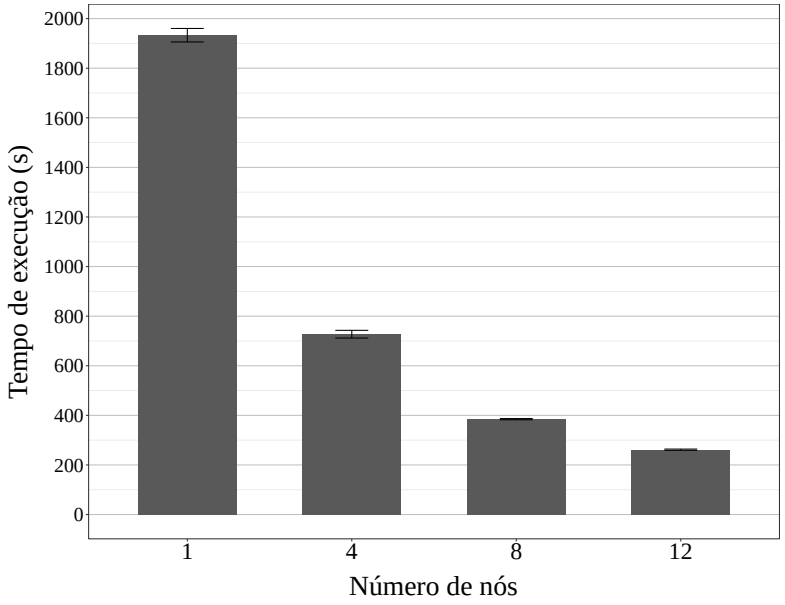
Isso fica evidente ao distribuir a aplicação para 8 e 12 nós, aonde o ganho de desempenho é significativamente maior. Nesse caso, o nível de paralelismo empregado comporta a distribuição de carga e aumenta a vazão dos dados, diminuindo o tempo de execução do modelo.

A Figura 4.3 retrata o desempenho do modelo CNN para Text Classifier executado sobre o dataset 20NewsGroup/GloVe-6B. O modelo obteve uma redução de 86,5% no tempo necessário para o treinamento quando comparada a execução em 12 worker nodes

com a execução standalone.

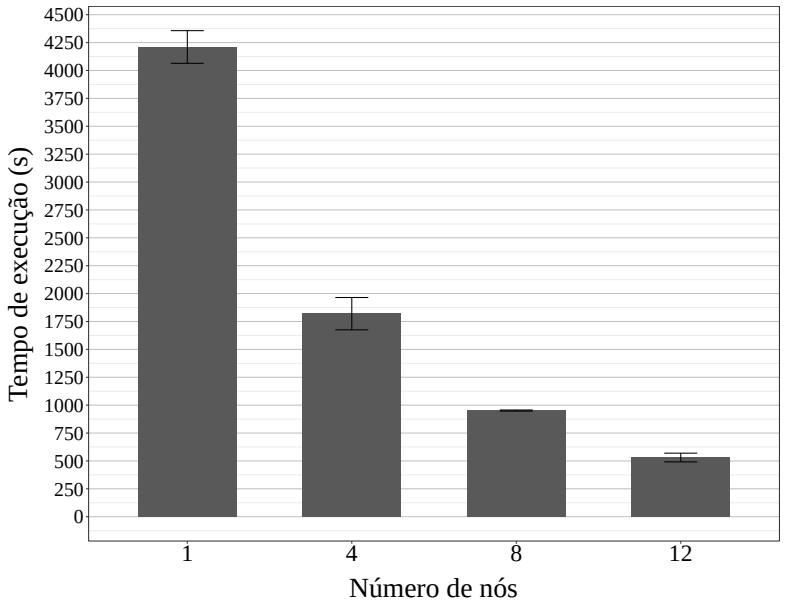
Nesse caso, pode-se evidenciar como o uso do paralelismo favoreceu o desempenho das aplicações. Além disso, a distribuição da carga de trabalho possivelmente minimizou o uso de recursos (cpu e memória), evitando possíveis cenários de interferência, possibilitando assim a alta vazão dos dados.

Figura 4.3: Rede CNN, Modelo Text Classifier (CNN) - Dataset 20NewsGroup/GloVe-6B



A Figura 4.4 apresenta o desempenho do modelo LSTM para Text Classifier executado sobre o dataset 20NewsGroup/GloVe-6B. O modelo obteve uma redução de 87,4% no tempo necessário para o treinamento quando comparada a execução em 12 worker nodes com a execução standalone.

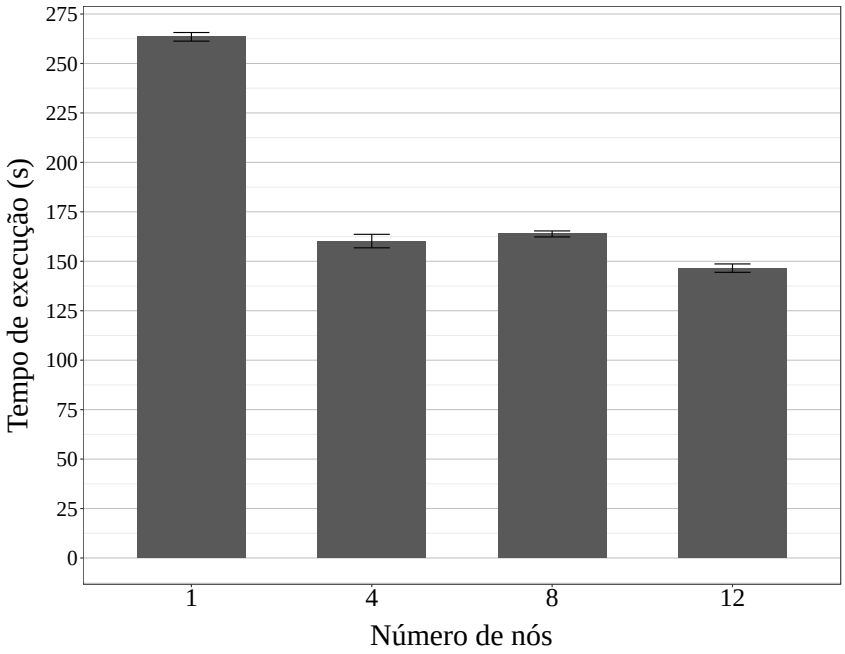
Figura 4.4: Rede RNN, Modelo Text Classifier (LSTM) - Dataset 20NewsGroup/GloVe-6B



A Figura 4.5 apresenta o desempenho do modelo Tree-LSTM executado sobre o dataset Stanford Sentiment Treebank. O modelo obteve uma redução de 44,5% no tempo necessário para o treinamento quando comparada a execução em 12 worker nodes com a execução standalone. Além disso, pode-se notar que esse modelo não apresentou ganhos de desempenho significativos entre as execuções distribuídas, conforme pode ser visto na tabela 4.5.

Somado a isso, como o dataset é relativamente pequeno, consequentemente a carga de trabalho em cada nó também é pequena, de modo que as demais tarefas do treinamento que não se relacionam especificamente com o tamanho do subset do dataset presente no worker node demoram aproximadamente o mesmo tempo independentemente do número de nodos. Isso pode representar o comportamento intermitente observados no desempenho da aplicação com 4, 8 e 12 nós de computação.

Figura 4.5: Rede RNN, Modelo Tree-LSTM - Dataset Stanford Sentiment Treebank



Em resumo, a Tabela 4.4 apresenta os tempos de execução de cada um dos modelos. Além disso, pode-se observar o tempo total necessário para executar todas as aplicações em modo *standalone* e distribuído.

Tabela 4.4: Tempo de Treinamento (s)

Tipo	**Modelo**	**Dataset**	**Local**	**4 Nós**	**8 Nós**	**12 Nós**
CNN	ResNet	CIFAR10	4841	1491	952	673
CNN	VGG	CIFAR10	6101	2657	1375	939
CNN	LeNet	MNIST	480	315	173	122
CNN	Text Classifier	20NewsGroup/GloVe-6B	1933	727	384	261
RNN	Text Classifier	20NewsGroup/GloVe-6B	4210	1819	951	530
RNN	Tree-LSTM	Stanford Sentiment Treebank	263	160	163	146
Tempo total	-	-	17828	7169	3998	2671

De modo complementar, na Tabela 4.5 pode-se verificar a análise de desempenho levando em conta o Speedup e a eficiência das aplicações em relação a execução distribuída com 1, 4, 8, 12 nós dos modelos DL.

Tabela 4.5: Speedup e Eficiência

	Speedup			**Eficiência**			
Modelo	***4-Nós***	***8-Nós***	***12-Nós***	***Standalone***	***4-Nós***	***8-Nós***	***12-Nós***
ResNet	3,24	5,08	7,19	1	0,81	0,64	0,60
VGG	2,29	4,43	6,49	1	0,57	0,55	0,54
LeNet	1,43	2,77	3,93	1	0,36	0,35	0,33
TC-CNN	2,65	5,03	7,40	1	0,66	0,63	0,62
TC-LSTM	2,31	4,42	7,94	1	0,58	0,55	0,66
TreeLSTM	1,64	1,61	1,80	1	0,41	0,20	0,15

Considerando os dados apresentados na Tabela 4.5 podemos verificar que os speedups são significativos (super lineares) e muito similares para os modelos ResNet, VGG, TextClassifier-CNN e TextClassifier-LSTM, enquanto são mais baixos para TreeLSTM e LeNet. Em geral, o paralelismo resulta em ganhos expressivos para os modelos devido ao fato de distribuir a carga de processamento entre os nós de computação. Contudo, alguns modelos que são mais complexos, seja pelo número de camadas, número de neurônios nas camadas FC, ou relação entre os termos, como no LSTM, juntamente com as configurações específicas dos hiper-parâmetros, podem ter seu desempenho prejudicado.

Analisando os ganhos apresentados especificamente por cada modelo, observa-se que o overhead da comunicação, sincronização e demais operações decorrentes da execução distribuídas não são compensadas posteriormente pela distribuição do treinamento no modelo LeNet e muito menos no modelo TreeLSTM. Para os modelos ResNet, VGG, TextClassifier-CNN e TextClassifier-LSTM os ganhos na aceleração do treinamento são expressivos e justificam a escalabilidade dos modelos.

Por fim, a eficiência dos modelos quando paralelizados é satisfatória. Ainda, esse resultado nos indica que a carga de trabalho é um fator limitante para o ganho de desempenho, ou seja, ao momento que se descobrir uma equação ideal para o escalonamento de nós versus distribuição de carga de trabalho o desempenho pode ser otimizado. O modelo ResNet por exemplo, com 8 e 12 nós apresentou resultado semelhante, mas inferior ao obtido com 4 nós. Isso ocorre justamente pelo paralelismo desnecessário, Ao contrário, o modelo TC-LSTM apresenta ganho de desempenho constante devido ao processamento intensivo nos nós de computação, possivelmente decorrentes de uma carga de trabalho maior.

4.3.2 Uma Análise Quanto à Precisão dos Modelos

Os resultados apresentados nas Tabelas 4.4 e 4.5 da Seção 4.3.1 indicam a eficiência quanto a execução distribuída dos modelos DL. Entretanto, embora o ganho de desempenho seja expressivo, as propriedades dos modelos deve ser mantida. Nesse contexto, a precisão de um modelo se torna um fator crucial a ser analisada. Essa é a propriedade essencial para distinguir em meio a análise de um conjunto de dados, quais imagens representam um cachorro ou um ser humano. A Tabela 4.6 apresenta a precisão original de cada modelo em comparação a um cenário distribuído.

Tabela 4.6: Precisão dos Modelos DL (%)

Modelo	Dataset	Local	4 Nós	8 Nós	12 Nós	Variação
ResNet	CIFAR10	84,14	83,53	82,28	80,124	-4,77
VGG	CIFAR10	67,90	60,65	54,12	45,91	-32,38
LeNet	MNIST	98,50	98,16	97,27	95,64	-2,9
TC (CNN)	20NewsGroup/GloVe-6B	85,64	85,63	84,50	84,22	-1,66
TC (LSTM)	20NewsGroup/GloVe-6B	29,46	31,32	24,20	18,31	-37,85
Tree-LSTM	Stanford Sentiment Treebank	46,17	46,56	47,07	45,45	-1,56

O ResNet sofreu perdas médias de precisão na ordem de 0,72%, 1,49%, 2,62% para as três variações entre as quatro configurações ordenadas de local à 12 nós, respectivamente. Dessa forma, o ResNet possuiu um declínio de 4,77% de precisão ao passar de execução local para 12 nós.

Já o modelo VGG demonstrou perda de precisão média de 10,67% entre execução local e 4 nós, perda de 10,76% entre 4 e 8 nós e 15.16% de perda entre 8 e 12 nós. De forma geral, o conjunto de treinamento VGG passou por uma perda de 32,38% quando passou de execução local para configuração de 12 nós, um resultado consternador que será explicado adiante.

LeNet obteve perdas médias de precisão na ordem de 0,34%, 1,23%, 1,67% quando passou de local para 4 nós, de 4 para 8 nós e de 8 para 12 nós, respectivamente. De sua precisão média local para a precisão de 12 nós, sofreu uma perda média de apenas 2,9%.

Text Classifier em CNN manteve o melhor dos resultados entre o conjunto de treinamento, sofrendo perda média de precisão de 0,01% no primeiro intervalo, 1,32% no segundo e 0,33% no terceiro. Ao total, passando de execução local para 12 nós, perdeu modestos 1,66% de precisão média.

Em contraponto, os piores índices de perda de precisão média vieram de Text

Classifier em LSTM. Inicialmente, o mesmo iniciou ganhando 6,31% entre configuração local e 4 nós, passando então a perder 22,02% entre 4 e 8 nós e aumentando ainda mais sua perda em 24,34% entre 8 e 12 nós. De configuração local para 12 nós, perdeu abismais 37,85% de precisão média. As razões para esse comportamento são explicadas mais adiante.

Os experimentos de Tree-LSTM demonstraram ganhos sucessivos de 0,84% e 1,1% quando comparou-se o intervalo de configuração local para 4 nós e 4 para 8 nós, respectivamente. Porém, de 8 para 12 nós, o modelo voltou a registrar perda de 3,44% de precisão média. Desde sua configuração local até com 12 nós, a precisão do treinamento de Tree-LSTM abaixou de 1,56%.

ResNet possui otimizações chamadas blocos residuais, logo como a tabela de tempo 4.4 revela, necessita de menos épocas para alcançar ou superar o resultado de VGG. Logo, VGG necessita de mais épocas para poder alcançar a mesma *accuracy* que ResNet e mitigar as perdas pelo aumento de paralelismo. Não se efetuou tal teste, pois tentou se ater ao padrão heterogêneo de configuração de testes para não beneficiar experimentos específicos.

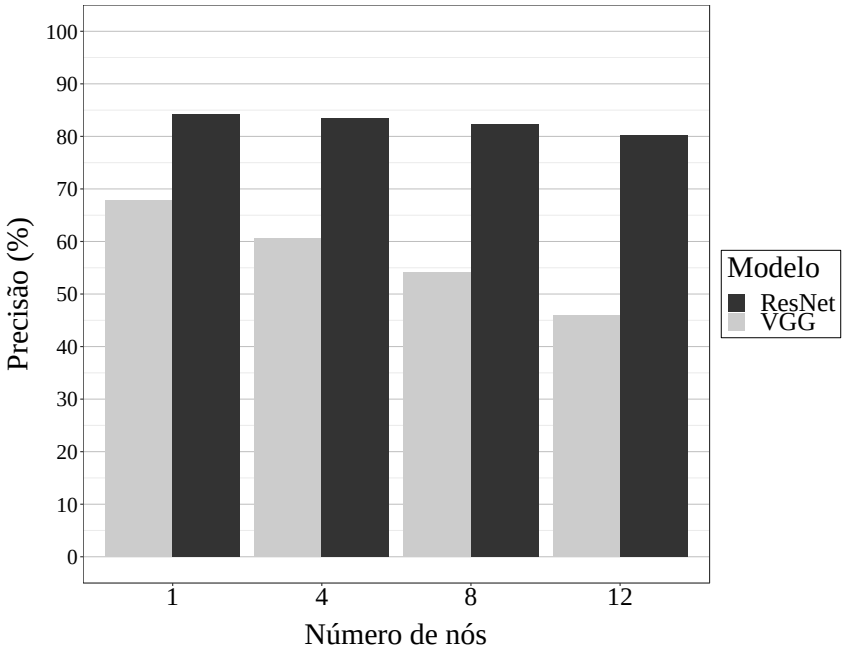
Em geral, pode-se observar que a precisão dos modelos possuí baixa variação e isso potencializa o uso do processamento paralelo e distribuído. Muito embora o comportamento das métricas dos modelos RNN tenham sido coerentes com a tendência do conjunto experimental deste trabalho, é importante perceber que os resultados alcançados nestes modelos, tanto em tempo como em precisão, estão muito aquém dos valores que eram esperados.

Após identificar essa nuance, investigou-se tal comportamento junto à comunidade de desenvolvedores da Intel®. Em resumo, a fonte deste problema é causada por um defeito na API do otimizador, sendo sua razão desconhecida até o momento de escrita deste trabalho. Porém, este problema pode ser contornado pelo uso de qualquer outra função de otimização disponível. Todavia, como os recursos computacionais disponíveis para estes experimentos são limitados e a anomalia só pode ser identificada em etapas de interpretação de resultados, não houve condições de se escolher pela retomada dos experimentos com a verificação da proposta de solução do desenvolvedor.

A Figura 4.6 apresenta um caso onde a computação distribuída pode interferir na precisão dos modelos DL, conforme pode ser analisado na Tabela 4.6. Nesse caso, observa-se que o modelo VGG apresenta redução na precisão obtida conforme o ambiente se torna mais distribuído. O motivo para isso pode ser decorrente das camadas FC com

muitos neurônios do modelo VGG. Então para que cada *worker* node treine sua replica do modelo com camadas FC extremamente densas se faz necessário muitas iterações para que só então essas soluções sejam novamente agregadas na solução final.

Figura 4.6: Comparativo - Modelos ResNet e VGG - Dataset CIFAR-10



Portanto, se o número de iterações não for tal que o modelo consiga ser treinado a tempo em cada *worker* node com o porção do dataset que lhe é devida, a eficiência tende a decair. Isso não ocorre com o ResNet justamente por esse treinamento ocorrer a tempo, justificando o fato da precisão ter se mantido aproximadamente estável para o ResNet.

4.3.3 Análise de Custo

Atualmente, existe a tendência em utilizar ambientes de nuvem para o processamento de aplicações. Pode-se observar ainda que esse comportamento vem crescendo nos últimos anos e que o processamento Big Data, antes efetuado em *clusters* dedicados agora não é mais restritivo devido a evolução das plataformas de nuvem (MATTEUSSI et al., 2018; XAVIER et al., 2016; XAVIER et al., 2015; MATTEUSSI, 2016). Em relação ao custo, este é dado por hora de alocação de uma instância local, que para esse trabalho corresponde a R$1,24/hora, além de serem considerados 2 *Head nodes* (responsáveis pela distribuição da carga de trabalho e tolerância a falhas) e o número de *Worker nodes*

específicos de cada configuração, conforme equação 4.1 abaixo:

$$C(n) = 2 * 1.241 + 1.241 * n \tag{4.1}$$

Sendo assim, o custo total por hora de alocação de uma instância local, de 4 nós, de 8 nós e 12 nós é respectivamente: R$3,72/hora, R$7,44/hora, R$12,41/hora e R$17.37/hora. O custo médio da execução de cada modelo, bem como o custo médio total dos experimentos são estimados a partir dos valores fornecidos pela Microsoft Azure e estão contidos na Tabela 4.7a. É importante ressaltar que os valores não levam em conta o custo de tempo para configurar o ambiente e preparar os experimentos.

Tabela 4.7: Tabela de Análise de Custo por Modelo
(a) Custo Médio de Processamento por Modelo, Tempo de Treinamento (s), Valor por Treino R$ e Redução de Custo Obtida (Ganho) %

Modelo	**Local**	**Valor**	**4 Nós**	**Valor**	**8 Nós**	**Valor**	**12 Nós**	**Valor**	**Ganho**
ResNet	4841	5	1491	3,08	952	3,28	673	3,24	35,2
VGG	6101	6,3	2657	5,49	1375	4,73	939	4,53	28
LeNet	480	0,49	315	0,65	173	0,59	122	0,58	-18
Text Classifier	1933	1,99	727	1,5	384	1,32	261	1,25	37,1
Text Classifier	4210	4,35	1819	3,76	951	3,27	530	2,55	41,3
Tree-LSTM	263	0,27	160	0,33	163	0,56	146	0,70	-159,2
Acumulado	17828	18,43	7169	14,82	3998	13,78	2671	12,80	30,5

A Tabela 4.7a explicita que o processamento paralelo e distribuído, além de beneficiar as aplicações em questões de tempo de processamento, revela também ganhos consistentes de custos operacionais para a grande maioria dos cenários de teste, para alguns modelos o ganho aproximado foi de 40%. Passando de um nó de *worker* para 4 unidades o ganho médio foi de 18.06%. De mesma forma, de 4 para 8 nós ocorre um ganho médio de 8.74%, ao passo que de 8 para 12 ocorre um ganho médio de 7.11%. Desta maneira, o ganho médio geral em relação a todos os modelos do conjunto de experimentos foi de 30,5% quando compara-se o melhor custo médio (12 nós) com o menor custo médio (local).

Por fim, acredita-se que os modelos LeNet e Tree-LSTM não possuem carga de trabalho expressiva que necessite ser processada de forma distribuída. Como resultado, ao distribuir essas aplicações o desempenho não foi considerado eficaz e consequentemente levou ao aumento de custo. Isso evidencia uma correlação alta entre a escalabilidade (número de máquinas virtuais) e a carga de trabalho, interferindo diretamente no custo de

processamento.

4.4 Considerações

Neste capítulo buscou-se discorrer sobre todos os aspectos teóricos e práticos utilizados nos experimentos desta monografia. Ainda, buscou-se abordar as questões metodológicas, especificações de software, hardware e de configuração dos experimentos. Foram também apresentados os resultados dos experimentos, bem como buscou-se avaliar e interpretar tais resultados dando substância à seus significados. Em resumo, os resultados obtidos indicam a eficácia do modelo de processamento Big Data com Deep Learning. O próximo capítulo encerra esta monografia com um fechamento geral dos pontos mais importantes desta pesquisa.

.

5 CONCLUSÃO

Este trabalho avalia o impacto causado na performance das aplicações DL junto ao modelo de programação paralelo e distribuído provenientes de *frameworks* de processamento BD. Inicia revisando a fundamentação teórica abordando os principais conceitos envolvendo as áreas de DL e BD no Capítulo 2. Em seguida, no Capítulo 3 são relatados os principais trabalhos relacionados, destacando as suas principais características. Para finalizar, o Capítulo 4 apresenta a proposta para atingir-se os objetivos e os resultados obtidos ao final dos experimentos. Os resultados obtidos indicam a eficácia do modelo de processamento Big Data com Deep Learning. Constataram-se que o ganho de desempenho utilizando-se ambiente distribuído foi de até 87,4%; redução de custo de até 41,3%, e perda na manutenção da precisão dos modelos menor que 5%.

Revisitando os objetivos traçados no Capítulo 1: o *framework* BigDL é analisado no Capítulo 3 e melhor detalhado na seção 4.1; a avaliação do desempenho e escalabilidade de aplicações DL ao utilizar modelos de programação paralelos e distribuídos propiciados por *frameworks* Big Data é tratado na seção 4.3.1; a análise relativa a manutenção da precisão dos modelos DL em meio aos modelos paralelos e distribuídos propiciados por *frameworks* Big Data é realizada na subseção 4.3.2; e a análise de custos resultante da análise de performance proposta é discutida na subseção 4.3.3.

Para trabalhos futuros, deseja-se amplificar o universo de estudo que este trabalho abordou, incluindo também as ferramentas TensorFlowOnSpark e SparkNet. Além disso, deseja-se adicionar mais modelos e datasets, incluindo o ImageNet e adaptações do mesmo, ao conjunto experimental, traçando métodos e rotinas que possibilitem a qualquer usuário de frameworks de BD com DL testar o seu hardware e avaliar o potencial da distribuição de cargas de trabalho sobre frameworks distintos. Deseja-se implementar um *benchmarking* que una os frameworks de propósito específico referenciados na seção de trabalhos relacionados desse trabalho. Por fim, também deseja-se realizar um estudo da viabilidade da distribuição do treinamento de modo a inferir até onde é viável o paralelismo para cada modelo especificamente.

REFERÊNCIAS

ABADI, M. et al. Tensorflow: a system for large-scale machine learning. In: **OSDI**. [S.l.: s.n.], 2016. v. 16, p. 265–283.

ABBASI, A.; SARKER, S.; CHIANG, R. H. Big data research in information systems: Toward an inclusive research agenda. **Journal of the Association for Information Systems**, v. 17, n. 2, 2016.

AHN, H. Y.; KIM, H.; YOU, W. Performance study of distributed big data analysis in yarn cluster. In: IEEE. **2018 International Conference on Information and Communication Technology Convergence (ICTC)**. [S.l.], 2018. p. 1261–1266.

ARMBRUST, M. et al. Spark sql: Relational data processing in spark. In: **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2015. (SIGMOD '15), p. 1383–1394. ISBN 978-1-4503-2758-9. Available from Internet: <http://doi.acm.org/10.1145/2723372.2742797>.

BAHDANAU, D.; CHO, K.; BENGIO, Y. Neural machine translation by jointly learning to align and translate. **arXiv preprint arXiv:1409.0473**, 2014.

BASTIEN, F. et al. Theano: new features and speed improvements. **arXiv preprint arXiv:1211.5590**, 2012.

BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. **IEEE transactions on neural networks**, v. 5, n. 2, p. 157–166, 1994.

CHEN, X.-W.; LIN, X. Big data deep learning: challenges and perspectives. **IEEE access**, Ieee, v. 2, p. 514–525, 2014.

CHO, K. et al. Learning phrase representations using rnn encoder-decoder for statistical machine translation. **arXiv preprint arXiv:1406.1078**, 2014.

CHOLLET, F. et al. **Keras**. 2015.

CIREŞAN, D. et al. Multi-column deep neural network for traffic sign classification. **Neural networks**, Elsevier, v. 32, p. 333–338, 2012.

COLLOBERT, R.; BENGIO, S.; MARIÉTHOZ, J. **Torch: a modular machine learning software library**. [S.l.], 2002.

DAI, J.; LIU, X.; WANG, Z. Building large-scale image feature extraction with bigdl at jd. com. **Honeywell launches UAV industrial inspection service, teams with Intel on innovative offering." Sept**, 2017.

DEAN, J. et al. Large scale distributed deep networks. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2012. p. 1223–1231.

DEAN, J.; GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In: **Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation, OSDI'04**. Berkeley, CA, USA: USENIX Association, 2004. (OSDI'04), p. 10–10. Available from Internet: <http://dl.acm.org/citation.cfm?id=1251254.1251264>.

FENG, A.; SHI, J.; JAIN, M. **CaffeOnSpark Open Sourced for Distributed Deep Learning on Big Data Clusters**. [S.l.]: Feb, 2016.

GAYA, J. O. et al. Vision-based obstacle avoidance using deep learning. In: IEEE. **Robotics Symposium and IV Brazilian Robotics Symposium (LARS/SBR), 2016 XIII Latin American**. [S.l.], 2016. p. 7–12.

GOODFELLOW, I. et al. **Deep learning**. [S.l.]: MIT press Cambridge, 2016.

GRAVES, A.; MOHAMED, A.-r.; HINTON, G. Speech recognition with deep recurrent neural networks. In: IEEE. **Acoustics, speech and signal processing (icassp), 2013 ieee international conference on**. [S.l.], 2013. p. 6645–6649.

GUPTA, A. et al. A big data analysis framework using apache spark and deep learning. In: IEEE. **Data Mining Workshops (ICDMW), 2017 IEEE International Conference on**. [S.l.], 2017. p. 9–16.

GUPTA, A. et al. A big data analysis framework using apache spark and deep learning. In: **2017 IEEE International Conference on Data Mining Workshops (ICDMW)**. [S.l.: s.n.], 2017. p. 9–16. ISSN 2375-9259.

HE, K. et al. Identity mappings in deep residual networks. In: SPRINGER. **European conference on computer vision**. [S.l.], 2016. p. 630–645.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural computation**, MIT Press, v. 9, n. 8, p. 1735–1780, 1997.

HOWARD, A. G. et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. **arXiv preprint arXiv:1704.04861**, 2017.

JAIN, R. **The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling**. [S.l.]: John Wiley & Sons, 1990.

JIA, Y. et al. Caffe: Convolutional architecture for fast feature embedding. In: ACM. **Proceedings of the 22nd ACM international conference on Multimedia**. [S.l.], 2014. p. 675–678.

KHUMOYUN, A.; CUI, Y. Spark based distributed deep learning framework for big data applications. In: IEEE. **Information Science and Communications Technologies (ICISCT), International Conference on**. [S.l.], 2016. p. 1–5.

KIM, H. et al. Deepspark: A spark-based distributed deep learning framework for commodity clusters. **arXiv preprint arXiv:1602.08191**, 2016.

KRIZHEVSKY, A.; NAIR, V.; HINTON, G. The cifar-10 dataset. **online: http://www. cs. toronto. edu/kriz/cifar. html**, 2014.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2012. p. 1097–1105.

LAWRENCE, S. et al. Face recognition: A convolutional neural-network approach. **IEEE transactions on neural networks**, IEEE, v. 8, n. 1, p. 98–113, 1997.

LECUN, Y. et al. Backpropagation applied to handwritten zip code recognition. **Neural computation**, MIT Press, v. 1, n. 4, p. 541–551, 1989.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, IEEE, v. 86, n. 11, p. 2278–2324, 1998.

LECUN, Y.; CORTES, C.; BURGES, C. Mnist handwritten digit database. **AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist**, v. 2, 2010.

LU, X. et al. Dlobd: A comprehensive study of deep learning over big data stacks on hpc clusters. **IEEE Transactions on Multi-Scale Computing Systems**, IEEE, 2018.

MALEWICZ, G. et al. Pregel: a system for large-scale graph processing. In: ACM. **Proceedings of the 2010 ACM SIGMOD International Conference on Management of data**. [S.l.], 2010. p. 135–146.

MATTEUSSI, K. Um Estudo Sobre a ContenÇão de Disco em Ambientes Virtualizados Utilizando Contêineres e Seu Impacto Sobre Aplicações MapReduce. p. 94, 2016. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS. 2016. pp.

MATTEUSSI, K.; ROSE, C. D. Uma estratégia de alocaç ao dinâmica e preditiva de recursos de I/O para reduzir o tempo de execuç ao em aplicaç oes mapreduce. In: **Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul**. [S.l.: s.n.], 2015. p. 45–47. ISBN 85-88442-74-4.

MATTEUSSI, K. J. et al. Understanding and Minimizing Disk Contention Effects for Data-Intensive Processing in Virtualized Systems. In: . [S.l.]: IEEE Computer Society, 2018. (HPCS - International Conference on High Performance Computing and Simulation), p. 901–908. ISBN 978-1-5386-7879-4.

MATTEUSSI, K. J. et al. Um guia para a modelagem e desenvolvimento de aplicações big data sobre o modelo de arquitetura zeta. In: **Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul**. [S.l.: s.n.], 2017. ISBN 85-88442-74-4.

MENG, X. et al. Mllib: Machine learning in apache spark. **The Journal of Machine Learning Research**, JMLR. org, v. 17, n. 1, p. 1235–1241, 2016.

MOLLAHOSSEINI, A.; CHAN, D.; MAHOOR, M. H. Going deeper in facial expression recognition using deep neural networks. In: IEEE. **Applications of Computer Vision (WACV), 2016 IEEE Winter Conference on**. [S.l.], 2016. p. 1–10.

MORITZ, P. et al. Sparknet: Training deep networks in spark. **arXiv preprint arXiv:1511.06051**, 2015.

PANG, B.; LEE, L. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. **Proceedings of the 43rd annual meeting on association for computational linguistics**. [S.l.], 2005. p. 115–124.

PENNINGTON, J.; SOCHER, R.; MANNING, C. Glove: Global vectors for word representation. In: **Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)**. [S.l.: s.n.], 2014. p. 1532–1543.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological review**, American Psychological Association, v. 65, n. 6, p. 386, 1958.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **nature**, Nature Publishing Group, v. 323, n. 6088, p. 533, 1986.

RUSSAKOVSKY, O. et al. Imagenet large scale visual recognition challenge. **International Journal of Computer Vision**, Springer, v. 115, n. 3, p. 211–252, 2015.

SEIDE, F.; AGARWAL, A. Cntk: Microsoft's open-source deep-learning toolkit. In: ACM. **Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. [S.l.], 2016. p. 2135–2135.

SERMANET, P. et al. Pedestrian detection with unsupervised multi-stage feature learning. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2013. p. 3626–3633.

SILBERMAN, N. Tf-slim: A lightweight library for defining, training and evaluating complex models in tensorflow. Georgia Institute of Technology, 2017.

SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. **arXiv preprint arXiv:1409.1556**, 2014.

SINGH, D.; REDDY, C. K. A survey on platforms for big data analytics. **Journal of Big Data**, Springer, v. 2, n. 1, p. 8, 2015.

SOUSA, F. R.; MOREIRA, L. O.; MACHADO, J. C. Computação em Nuvem: Conceitos, Tecnologias, Aplicações e Desafios. **II Escola Regional de Computação Ceará, Maranhão e Piauí ERCEMAPI'09**, p. 150–175, 2009.

SOUZA, P. R. R. de et al. Aten: A dispatcher for big data applications in heterogeneous systems. In: **2018 International Conference on High Performance Computing Simulation (HPCS)**. [S.l.: s.n.], 2018. p. 585–592.

SUTSKEVER, I.; VINYALS, O.; LE, Q. V. Sequence to sequence learning with neural networks. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2014. p. 3104–3112.

SZEGEDY, C. et al. Going deeper with convolutions. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2015. p. 1–9.

SZEGEDY, C. et al. Rethinking the inception architecture for computer vision. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2016. p. 2818–2826.

TAIGMAN, Y. et al. Deepface: Closing the gap to human-level performance in face verification. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2014. p. 1701–1708.

TOKUI, S. et al. Chainer: a next-generation open source framework for deep learning. In: **Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)**. [S.l.: s.n.], 2015. v. 5, p. 1–6.

VENKATESAN, N. J.; NAM, C.; SHIN, D. R. Deep learning frameworks on apache spark: A review. **IETE Technical Review**, Taylor & Francis, p. 1–14, 2018.

VU, T. H.; WANG, J.-C. Acoustic scene and event recognition using recurrent neural networks. **Detection and Classification of Acoustic Scenes and Events**, v. 2016, 2016.

WANG, Y. et al. Bigdl: A distributed deep learning framework for big data. **arXiv preprint arXiv:1804.05839**, 2018.

WHITE, T. **Hadoop: The Definitive Guide**. [S.l.]: O'Reilly Media, Inc. 2009. 3rd. pp. 657, 2009. 657 p.

XAVIER, M. G. et al. Understanding performance interference in multi-tenant cloud databases and web applications. In: **2016 IEEE International Conference on Big Data (Big Data)**. [S.l.: s.n.], 2016. p. 2847–2852.

XAVIER, M. G. et al. A performance isolation analysis of disk-intensive workloads on container-based clouds. In: **2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. [S.l.: s.n.], 2015. p. 253–260. ISSN 1066-6192.

YANG, L. et al. **Open sourcing tensorflowonspark: Distributed deep learning on big-data clusters**. 2017.

ZHANG, H. et al. In-memory big data management and processing: A survey. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 27, n. 7, p. 1920–1948, 2015.