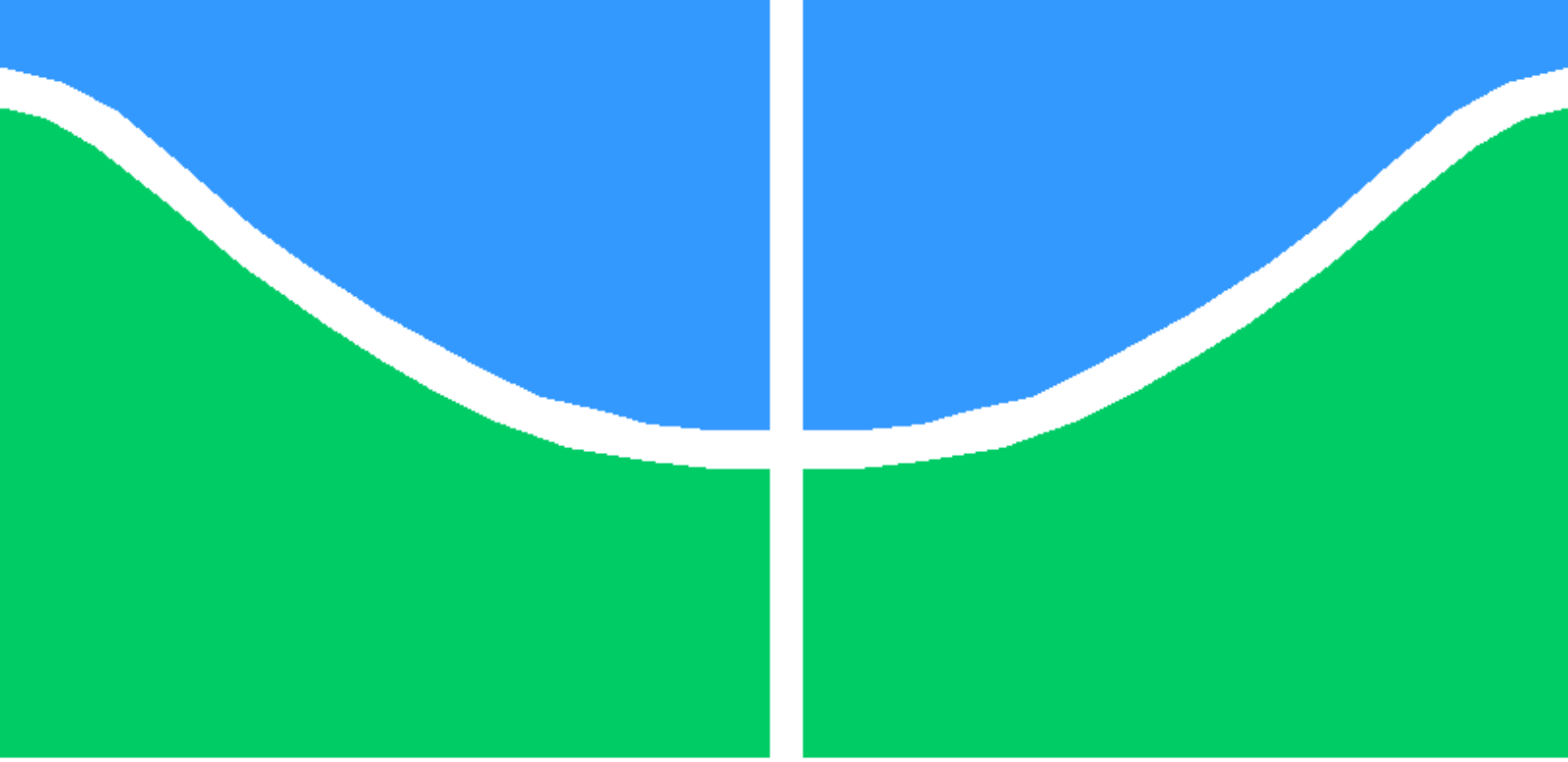

TRABALHO DE GRADUAÇÃO

Uma Arquitetura Descentralizada para Redes Neurais Artificiais sem Peso

Arthur Costa de Lisbôa Vaz

Rayssa Moreira Cardoso

Brasília, Maio de 2021

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

Uma Arquitetura Descentralizada para Redes Neurais Artificiais sem Peso

Arthur Costa de Lisbôa Vaz

Rayssa Moreira Cardoso

Relatório submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiro de Redes de Comunicação

Banca Examinadora

Prof. Alexandre Solon Nery, ENE/UnB _____
Orientador

Prof. Georges Amvame Nze., ENE/UnB _____
Examinador Interno

Prof. William Ferreira Giozza, ENE/UnB _____
Examinador Interno

Agradecimentos

Agradeço,
à Universidade de Brasília e ao seu corpo docente, em especial ao Prof. Alexandre Solon Nery, por me proporcionarem um ensino de excelência, gratuito e que contribui para a construção de uma sociedade melhor;
à minha dupla Rayssa Moreira que esteve ao meu lado ao longo de todo curso;
à minha família que sempre foi a minha base e fonte de motivação e incentivo;
e, acima de tudo, à Deus pelo dom da vida e por ter me dado saúde e força durante essa jornada.

Arthur Costa de Lisbôa Vaz

Agradeço a Deus pela fortaleza durante a jornada e à Universidade pública, gratuita e de qualidade.
Aos meus pais, Manoel e Aurea, que sempre me incentivaram e possibilitaram que eu pudesse cursar uma faculdade.
À minha irmã Larissa e ao meu cunhado Giovane que sempre me apoiaram.
Agradeço a todos os amigos, tanto os que fiz dentro da faculdade durante esses anos de graduação como os que me acompanharam fora da Universidade.
À minha dupla, Arthur Vaz, com quem compartilhei alegrias e tristezas e aprendi muito durante o convívio.
À UnB e a todos os professores, por me proporcionarem um ambiente de aprendizado e amadurecimento, viabilizando oportunidades únicas. Em especial ao meu orientador Prof. Alexandre Solon Nery por aceitar conduzir este trabalho de pesquisa.

Rayssa Moreira Cardoso

RESUMO

É cada vez mais frequente a necessidade de solucionar situações onde um modelo matemático simples é incapaz de modelar o problema. Os avanços tecnológicos na última década, em especial o aumento do poder computacional e volume de dados disponíveis, possibilitaram a popularização do uso de técnicas de aprendizado de máquina e inteligência artificial como alternativa para resolver esse desafio.

Entre as técnicas que ganharam destaque estão as Redes Neurais Artificiais, modelo computacional que se baseia vagamente no funcionamento do cérebro humano. Nesse tipo de modelo, o conhecimento é adquirido por meio do ajuste de pesos sinápticos entre os diversos neurônios da rede. Geralmente, esse processo utiliza uma grande quantidade de operações matemáticas como somas e multiplicações. Essa técnica é muito poderosa, mas traz algumas desvantagens importantes: a exigência de grande poder computacional e treinamento lento.

Em contraponto, o modelo abordado nesse trabalho, as Redes Neurais Artificiais sem Peso, não precisam ajustar os pesos sinápticos e todo conhecimento fica armazenado em memórias RAM. Dessa maneira, a aprendizagem do modelo consiste em realizar operações de leitura e escrita, tornando o processo menos custoso computacionalmente e mais rápido.

Além disso, esse trabalho propõe uma arquitetura descentralizada para implementação de uma Rede Neural Artificial sem Pesos. Na arquitetura apresentada, os componentes do modelo são distribuídos em uma rede de computadores e atuam como servidores que prestam serviços.

Assim, é possível expandir as funcionalidades da rede possibilitando a realização de aprendizado federado em paralelo, elevada robustez e paralelização do processamento. Com a arquitetura proposta, o tempo de treinamento caiu em até 27% e com a falha de alguns componentes a acurácia do modelo caiu apenas parcialmente.

Palavras-chaves: inteligência artificial. aprendizado de máquina. redes neurais artificiais sem peso. processamento descentralizado.

ABSTRACT

The need to solve situations where a simple mathematical model is unable to model the problem is becoming more and more frequent. Technological advances in the last decade, in particular the increase in computing power and the volume of data available, made it possible to use machine learning and artificial intelligence tehniques to solve this challenge.

Among the technologies that have gained prominence are Artificial Neural Networks, a computational model that is based loosely on the workings of the human brain. In this type of model, knowledge is acquired by adjusting synaptic weights between the different neurons in the network. This process generally uses a large number of mathematical operations such as sums and multiplications. This technique is very powerful, but it has some important drawbacks: the requirement of great computational power and slow training.

In contrast, the model addressed in this document, Weightless Artificial Neural Networks, does not need to adjust synaptic weights and all knowledge stored in RAM memories. Thus, the learning of the model consists of performing reading and writing operations, making the process less computationally expensive and faster.

In addition, this work proposes a decentralized architecture for implementing a Weightless Artificial Neural Networks. In the proposed architecture, the components of the model are distributed in a computer network and act as servers providing services. Thus, it is possible to expand the network's functionalities, enabling federated learning, high robustness and parallel processing.

keywords: artificial intelligence. machine learning. weightless neural networks. decentralized processing.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

Acrônimos

FL	*Fedareted Learning*
IA	Inteligência Artificial
IID	Independente e identicamente distribuído
LAN	*Local Area Network*
ML	*Machine Learning*
no-IID	Não independente e identicamente distribuído
POO	Programação orientada a objetos
Pyro	*Python Remote Objects*
RAMs	*Random Access Memories*
RMI	*Remote method invocation*
RNSP	Redes Neurais Sem Peso
WiSARD	*Wilkes, Stonham and Aleksander Recognition Device*
ZB	zettabyte

Capítulo 1

Introdução

A medida que o tempo passa os problemas relevantes para a sociedade e para a ciência vêm se tornando cada vez mais complexos. Para solucionar alguns desses desafios, as ferramentas computacionais disponíveis vem se tornando cada vez mais poderosas e eficientes. Nesse contexto, surgiram técnicas de computação paralela e sistemas distribuídos, que por sua vez impulsionaram o desenvolvimento de técnicas de inteligência artificial e aprendizado de máquina.

Na ciência da computação, inteligência artificial pode ser definida como o comportamento de computadores, robôs ou outras maquinas que apresentam inteligência semelhante à humana (EDUCATION, 2021). Já o aprendizado de máquina, geralmente se refere ao campo de estudos que utiliza algoritmos computacionais e modelos matemáticos projetados para gerar previsões, ou tomar decisões com base em dados amostrais (ZHANG, X.-D., 2020). Um grande número de pesquisadores acredita que o aprendizado de máquina é o melhor caminho para atingir inteligência artificial (NG, 2021).

Apesar dos termos aprendizado de máquina e inteligência artificial terem surgido pela primeira vez em 1952 e 1955 respectivamente (PRESS, 2021), foi na última década que essas técnicas ganharam enorme popularidade e permearam praticamente todos os aspectos da sociedade moderna. Esse avanço foi impulsionado pelo aumento da disponibilidade de dados digitais, poder computacional e algoritmos aplicados. De 2010 a 2020, o desempenho do maior supercomputador disponível cresceu 17 vezes (Figura 1.1) e o limiar de mais de um zettabyte (ZB) de informações transmitidas pela internet anualmente foi atingido. (JR., 2021).

Desta maneira, surgiram diversos avanços em IA. Em 2011, o sistema Watston da IBM venceu os campeões de longa data no popular jogo de perguntas e resposta americano Jeopardy! (IBM, 2021). Pouco tempo depois, em 2012, o algoritmo DeepFace do Facebook usado para detectar rostos e marcar pessoas em fotos atingiu 97.35% de acurácia (TAIGMAN et al., 2014). Em 2016, um trabalho publicado pela divisão de Inteligência Artificial e Pesquisa da Microsoft, mostrou um sistema de reconhecimento de fala com uma taxa de erro de apenas 5.1% aproximadamente igual ao resultado de uma pessoa média (XIONG et al., 2018). Alguns anos depois, a inteligência artificial já despontava como uma poderosa ferramenta na área de saúde. Em 2018, pesquisadores treinaram uma rede neural com mais de 100.000 imagens de melanomas malignos e manchas

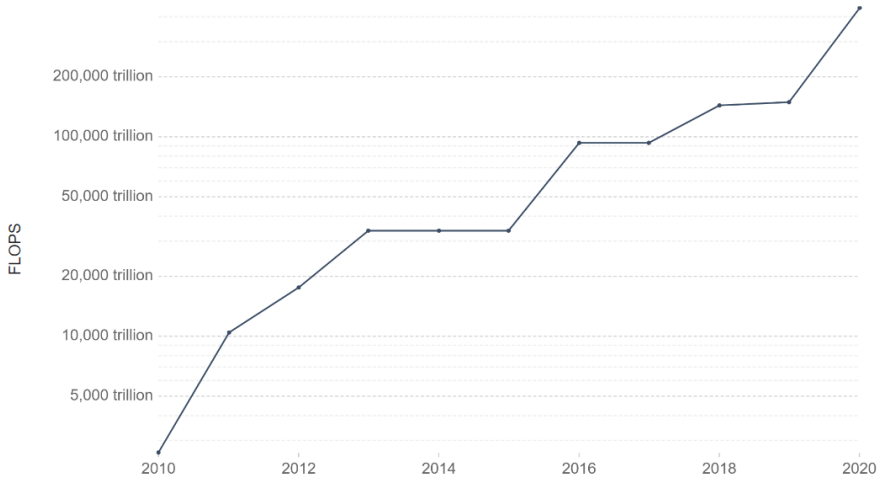

Figura 1.1: Poder computacional de supercomputadores em operações de ponto flutuante por segundo (FLOPS) (ROSER; RITCHIE, 2013)

benignas, para atingir 95% de sucesso na detecção de câncer de pele, melhor que a performance de 58 dermatologistas usados como referência (PITCHFORD, 2021).

Avançando para 2020, técnicas de IA e aprendizado de máquina estão na linha de frente do combate a pandemia de Covid-19. Pesquisadores estão utilizando essas ferramentas para prever a transmissão do vírus, realizar testes virtuais de medicamentos existentes e para desenvolver vacinas em potencial (TERRA, 2021).

O impacto econômico resultante da utilização crescente da IA será significante. Estima-se que tecnologias que melhorem a eficiência e otimizem processos aumentem a produtividade do trabalho em até 40% (SZCZEPANSKI, 2019). Também espera-se que a inteligência artificial possa ser usada para criar uma força de trabalho "virtual"capaz de solucionar problemas e aprender sozinha. A economia também poderá se beneficiar da difusão da inovação gerada por essas tecnologias, criando novas oportunidades de geração de receita em diversos setores. Um estudo da Pricewaterhouse-Coopers (GILLHAM et al., 2018) estima que o PIB global pode crescer até 14% até 2030 como resultado da adoção generalizada da IA.

Os avanços em IA foram tão rápidos que as normas da sociedade e legisladores de todo mundo não foram capazes de acompanhar. Algoritmos de aprendizado de máquina podem reproduzir comportamentos discriminatórios, e até ilegais, quando utilizados sem o devido cuidado. Um exemplo de mau uso dessa tecnologia é a utilização de algoritmos de IA para prever se um indivíduo será capaz de pagar um empréstimo ou desempenhar um trabalho com base em seu endereço, fenótipo ou idioma.

Outra preocupação de pesquisadores e especialistas é a crescente falta de confiança nas informações devido à utilização maliciosa da inteligência artificial. Informações adulteradas ou criadas por algoritmos podem ser utilizada para causar danos financeiros ou de reputação. Um exemplo que ganhou destaque recentemente foram os conteúdos de vídeo e áudio alterados, manipulados ou criados por técnicas de inteligência artificial que pareçam genuínos, chamados de *deepfakes*.

Essa tecnologia pode ser usada por criminosos e até governos autoritários para controlar a opinião pública, manipular o mercado financeiro, chantagear vítimas ou criar evidencias falsas na justiça.

Esses desafios não diminuíram o ritmo de evolução da IA. Na verdade, essa tecnologia vem ganhando cada vez mais importância, impulsionando o surgimento de novos avanços tecnológicos e campos de estudo. Um deles, é o estudo da ética aplicada a IA, campo que estuda o impacto na sociedade do uso malicioso de algoritmos de aprendizado de máquina.

Já os novos avanços tecnológicos vêm sido alcançados pela adoção de técnicas de outros campos do conhecimento da tecnologia da informação, em especial da computação e das redes de computadores. Por exemplo, utilizando uma arquitetura de computação distribuída, é possível desenvolver e implantar sistemas de IA mais robustos, escaláveis e confiáveis (CHAIB-DRAA, 1995).

A computação distribuída, como o nome já sugere, é uma técnica que utiliza várias máquinas, como múltiplos elementos de processamento, visando resolver problemas de forma que cada unidade do sistema execute uma pequena fatia do problema.

Além de simplesmente dividir o processamento, a computação distribuída apresenta uma boa tolerância a falhas, visto que a perda de uma máquina implica um comprometimento, tão somente, de uma porção do sistema. Essa técnica evoluiu principalmente depois de dois avanços tecnológicos: o desenvolvimento de microprocessadores de grande capacidade e as LANs (Local-are-networks), que permitiram a troca de informações entre diversos equipamentos conectados a uma mesma rede local. (TANENBAUM, 2007)

Existem várias aplicações interessantes de sistemas de IA distribuídos. Um exemplo notável é o sistema LODES (*Large-Internetwork Observation and Diagnostic Expert System*) (CHAIB-DRAA, 1995) utilizado na área de segurança de sistemas de telecomunicações. Esse sistema distribui cópias de si mesmo, cada uma atuando como um agente independente, para monitorar e gerenciar diferentes segmentos de uma rede local. O diferencial, é que cada agente é capaz de cooperar com outras cópias atualizando o sistema completo de possíveis ameaças e problemas de conexão (SUGAWARA, 1990).

Outro exemplo interessante é utilizar sistemas de IA distribuído para o controle de tráfego urbano, como apresentado em (BOMARIUS, 1992). Nesse cenário, todos os agentes que participam da atividade sejam motoristas, policiais, pedestres ou máquinas como veículos e semáforos, continuamente colaborariam para ajustar suas ações com o propósito de evitar conflitos. Com essa arquitetura é possível realizar a previsão do volume de congestionamento e controlar o fluxo de trânsito para otimizar a vazão e minimizar engarrafamentos. Por exemplo, esse tipo de rede pode ser capaz de acionar os semáforos para permitir a passagem mais rápida de uma ambulância por uma encruzilhada (Figura 1.2)

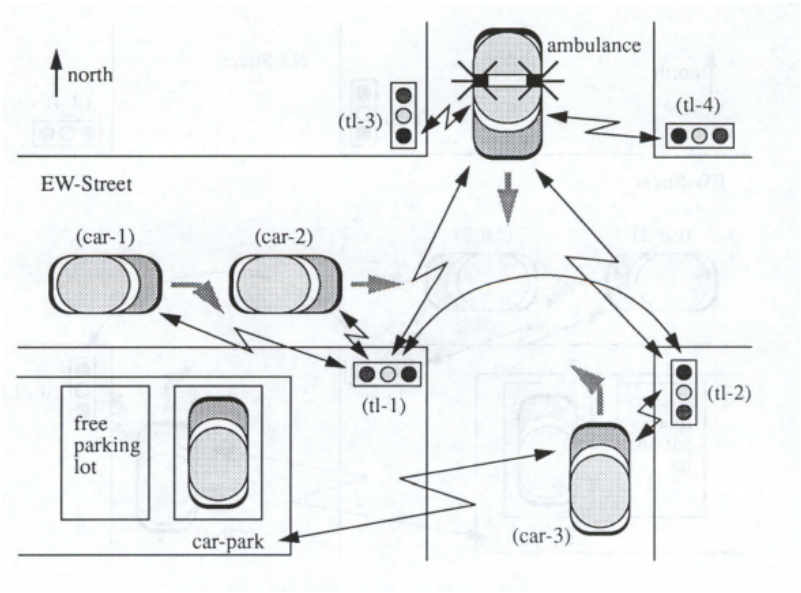

Figura 1.2: Sistema de controle de tráfego com IA distribuído - encruzilhada com ambulância (BOMARIUS, 1992)

O uso de sistemas de IA distribuídos também contribui para avanços na segurança da informação e a privacidade dos dados. Nesse contexto, técnicas de aprendizado federado permitem que diversos clientes utilizem suas próprias máquinas para fornecer dados e contribuir para o aprendizado e melhoria de um modelo de inteligência artificial.

1.1 Definição do Problema

Modelos de sistemas com alta disponibilidade e boa tolerância a falhas tem se estabelecido como o novo paradigma, visto que as organizações, sejam elas públicas ou privadas, necessitam de informações confiáveis e de fácil acesso, criando elevada dependência em relação aos ativos tecnológicos que sustentam a operação. Qualquer falha ou indisponibilidade nestes componentes podem gerar prejuízos incalculáveis para os negócios. Por esse motivo, é importante elaborar estratégias para minimizar o impacto causado por falhas tecnológicas, criando ambientes adequados para a realidade de cada aplicação, reduzindo ao máximo os impactos gerados pela indisponibilidade de serviços críticos para o negócio.

Entretanto, essa disponibilidade tem um custo, é necessário investir em infraestrutura física e lógica para que a redundância seja garantida de alguma forma. Esse investimento costuma ser compensado quando ocorre uma falha que comprometeria a operação e a tolerância a falhas permite que o sistema continue funcionando, mesmo que parcialmente, evitando o possível dano financeiro. Dependendo da aplicação, as falhas podem impactar não somente em questões econômicas, mas também em casos de vida ou morte, como sistemas hospitalares, por exemplo.

As redes neurais, amplamente utilizadas nos dias atuais, possuem finalidades variadas com aplicabilidade em diversas áreas de atuação, incluindo áreas sensíveis e críticas. Pensando na

disponibilidade dos dados e na necessidade dos clientes que utilizam esse tipo de modelo, o impacto causado por falhas inesperadas é uma preocupação real, portanto é importante que o sistema não seja acometido em sua totalidade.

1.2 Objetivos

Este projeto apresenta uma proposta de um sistema descentralizado para implementação de uma Rede Neural Artificial sem Peso (RNSP) de alta disponibilidade. O sistema utiliza a linguagem de programação *Python* para treinar um algoritmo de detecção de imagens organizado em múltiplas camadas lógicas e distribuído em uma rede de computadores. O objetivo deste projeto foi aumentar a robustez da rede neural, de forma que os processos de treinamento e classificação sejam executados em vários servidores independentes a fim de garantir que a perda de uma dessas máquinas comprometa a disponibilidade da rede de forma parcial e não em sua totalidade. Também faz parte do escopo desse projeto a realização de treinamento colaborativo da rede.

1.3 Justificativa

De acordo com (MICHAEL E. PORTER, 2019), os maiores desafios para iniciativas de implementação de sistemas de IA nas empresas são a dificuldade de integrar a tecnologia a processos já existentes e o custo do sistema ou da expertise necessária para utilização. O modelo proposto nesse trabalho, de implementação de uma rede distribuída e capaz de realizar treinamento colaborativo, pode ser uma proposta atraente para mitigar essas dificuldades.

Figura 1.3: Os desafios da inteligência artificial (MICHAEL E. PORTER, 2019)

A arquitetura distribuída implica que esse tipo de rede pode ser construída sobre um *cluster* de milhares de computadores sem que seja preciso implementar uma infraestrutura sob medida e custosa. Já o treinamento colaborativo, pode facilitar o treinamento e uso de redes neurais sem peso sem a necessidade da construção de um novo modelo específico para cada empresa.

Além disso, a relevância acadêmica do estudo de novas arquiteturas e soluções de IA fica evidente com o crescente número de publicações no setor. Na última década, o número de trabalhos publicados mais que dobrou (Figura 1.4).

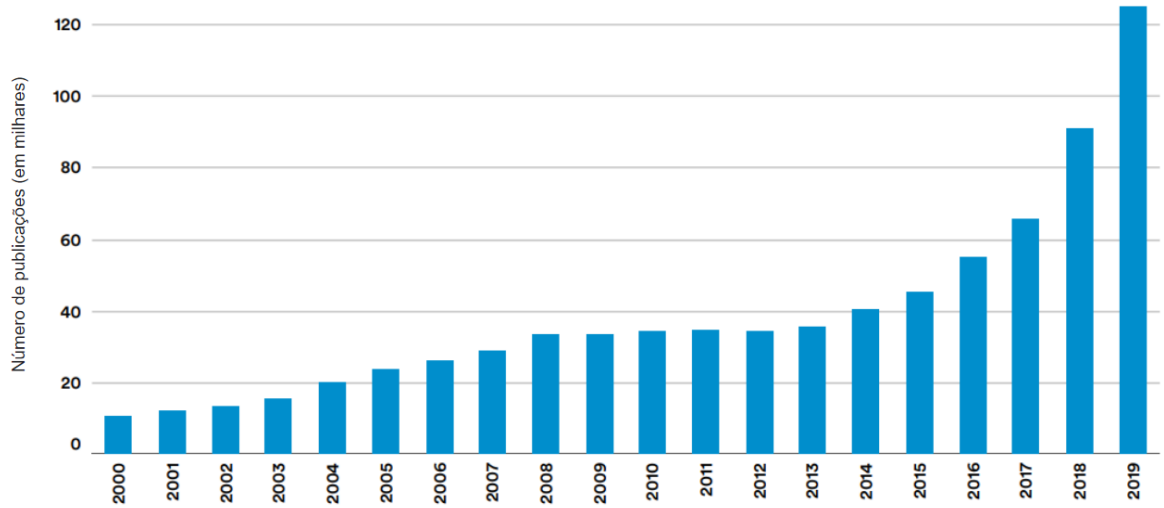

Figura 1.4: Número de publicações de IA revisadas por pares (ZHANG et al., 2021)

1.4 Trabalhos Relacionados

Esta Seção apresenta alguns trabalhos relacionados à metodologia do projeto, treinamento de RNSPs, e ao problema do reconhecimento de imagens em geral.

Sabe-se que uma aplicação clássica do modelo estudado é a classificação de imagens. Por esse motivo existem diversas abordagens a respeito das RNSPs quanto à suas características de aprendizado e capacidade de generalização. O presente trabalho faz uso de uma RNSPs baseadas no modelo WiSARD (Wilkes, Stonham and Aleksander Recognition Device). Esse modelo é de longe a arquitetura mais popular para RNSPs, de forma que, até hoje, este é o principal alvo de estudos onde novas melhorias e otimizações são desenvolvidas. Assim, foram utilizados como referência os seguintes trabalhos:

1. Projeto Dedicado de Redes Neurais Sem Peso Baseadas em Neurônios de Lógica Probabilística Multi-valorada (MACHADO et al., 2017)

2. NC-WISARD: Uma Interpretação Sem Pesos do Modelo Neural Neocognitron (BANDEIRA; FRANÇAA, 2010)

3. RWISARD: Um Modelo de Rede Neural Sem Peso para Reconhecimento e Classificação de

Imagens em Escala de Cinza (ARAÚJO, 2011)

4. Classificação de Emoções Faciais Utilizando a Rede Neural Sem Pesos WiSARD (LUSQUINO FILHO, 2018)

5. Análise de Desempenho da Rede Neural Artificial do tipo Multilayer Perceptron na era Multicore (SOUZA, 2012)

1.5 Estrutura do Trabalho

No Capítulo 2, Fundamentação Teórica, são abordados os principais fundamentos necessários para esse projeto, incluindo aqueles que dizem respeito ao funcionamento do Pyro.

No Capítulo 3, Metodologia, são descritos os métodos, arquitetura, ferramentas utilizadas e a descentralização proposta ao funcionamento da rede neural sem peso.

No Capítulo 4, Análise e Resultados, são mostrados os resultados e comparativos em diferentes perspectivas da rede, considerando diferentes *datasets*.

No Capítulo 5, Conclusão, são discutidos os argumentos finais dos autores e propostas de implementação do projeto em cenários mais complexos.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta os conceitos fundamentais utilizados para o desenvolvimento desse trabalho.

2.1 Introdução ao aprendizado de máquina

De maneira abrangente, o termo aprendizado de máquina (do inglês *machine learning* - ML) pode ser definido como um conjunto de métodos computacionais que utilizam informações anteriores para melhorar sua performance ou realizar previsões precisas (MOHRI; ROSTAMIZADEH; TALWALKAR, 2018). Tipicamente, essas informações anteriores são dados eletrônicos coletados e disponibilizados para análise. Esses dados podem ser conjuntos de treino rotulados por pessoas ou outros tipos de informação obtidas pela interação com ambiente. Técnicas de aprendizado de máquina podem ser divididas em três grupos principais:

- **Aprendizado não supervisionado:** Esse conjunto de técnicas busca descobrir estruturas ocultas em um conjunto de dados não rotulados. Se existe uma estrutura nos dados de entrada de forma que certos padrões são encontrados mais frequentemente que outros, deseja-se encontrar quais são esses padrões para determinar o comportamento esperado desse conjunto de dados (DELUA, 2021). As aplicações de algoritmos de aprendizado não supervisionado são divididas em três categorias:
  - **Clusterização:** Consiste em agrupar os dados baseado em suas similaridades ou diferenças.
  - **Associação:** Técnica que utiliza diferentes regras para encontrar relacionamentos entre as variáveis de um conjunto de dados.
  - **Redução de dimensionalidade:** Técnica de aprendizado que busca diminuir as características, ou dimensões de um conjunto de dados de forma a simplificar análises subsequentes.
- **Aprendizado supervisionado:** Ao contrário do aprendizado não supervisionado, esse tipo de técnica utiliza conjuntos de dados rotulados para treinar ou "supervisionar"algoritmos

de forma que seja possível classificar dados ou realizar predições. A ideia é aprender o mapeamento das entradas e saídas do conjunto de treino e utilizar esse mapeamento a fim de realizar previsões para dados de entrada não vistos anteriormente. Esse tipo de modelo deve ser capaz de mensurar a acurácia das previsões durante sua execução e utilizar esse resultado para aprender a cada iteração. Esse conjunto de técnicas é utilizado para resolver dois tipos de tarefa:

  - **Classificação:** Envolve mapear cada entrada a uma ou mais categorias distintas. Esse tipo de modelo tem como saída a previsão de valores discretos. O presente trabalho faz uso desta técnica para a classificação de imagens de acordo com suas especificidades.
  - **Regressão:** Envolve utilizar um algoritmo para determinar a relação entre as entradas e saídas de um conjunto de dados para realizar previsões. Esse tipo de modelo tem como saída valores contínuos.

- **Aprendizado por reforço:** Consiste no treino de modelos de aprendizado de máquina para tomar uma sequência de decisões ou ações. Durante o processo de treinamento, o modelo recebe punições ou recompensas dependendo do conjunto de ações tomadas. Dessa maneira o objetivo é que o programa aprenda a sequência de ações que maximiza as recompensas futuras (ou minimize as punições) (GHAHRAMANI, 2004).

2.2 Redes neurais artificiais

Redes neurais artificiais são um campo de aplicação de técnicas de aprendizado de máquina que se baseiam vagamente em uma representação do cérebro humano. Foram primeiro estudadas em 1940 por Warren McCulloch e o matemático Walter Pitts, que buscaram utilizar modelos matemáticos para descrever o comportamento de um neurônio (MCCULLOCH; PITTS, 1943).

As unidades elementares de redes neurais artificiais são os nós, ou neurônios artificiais, responsáveis pelo processamento das informações recebidas por seus terminais de entrada para determinar uma saída. A arquitetura de uma rede neural artificial define como seus diversos nós são posicionados em relação uns aos outros. Esses arranjos são estruturados essencialmente pelo direcionamento das conexões sinápticas dos neurônios em camadas.

2.2.1 Neurônio artificial

Os neurônios artificias são inspirados de maneira abrangente nos neurônios biológicos presentes no cérebro humano, como mostra a figura 2.1. De maneira simplificada, um neurônio biológico pode ser separado em três estruturas principais: os dendritos, o corpo celular ou soma e o axônio. Os dendritos de um neurônio são responsáveis por receber sinais de entrada provenientes de estímulos exteriores ou de outros neurônios acima. A soma é responsável por processar esses sinais de entrada para produzir um potencial de ativação que indica se o neurônio pode ou não emitir um impulso elétrico até o axônio. No axônio, esses sinais processados são transmitidos até o terminal de saída,

de onde podem ser enviados a outros neurônios abaixo na cadeia, ou até a órgãos como os músculos para que executem alguma ação. (SILVA et al., 2017)

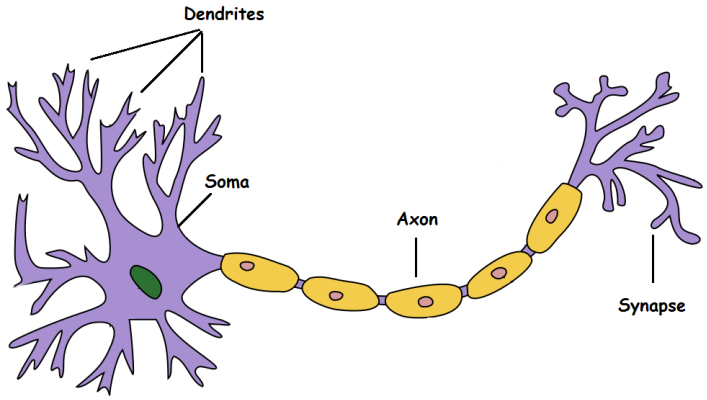

Figura 2.1: Esquema simplificado de um neurônio biológico (SEER, 2021)

Um neurônio artificial, ou nó de uma rede neural tradicional, se baseia nesse comportamento para seu funcionamento. Primeiramente o neurônio artificial coleta sinais de entrada, que proveem informação para o aprendizado e, a cada um desses sinais, atribui pesos que amplificam ou diminuem sua intensidade. Em um neurônio biológico são os dendritos que realizam essa função (ZHANG, Z., 2016).

Dessa maneira, cada neurônio artificial é capaz de atribuir significância às entradas em respeito a tarefa que o algoritmo busca aprender, por exemplo, determinar qual entrada é mais útil em classificar os dados sem erro.

Em seguida, os sinais ponderados são somados e processados por uma função de ativação de onde é possível extrair o sinal de saída, processo análogo ao que ocorre na soma de um neurônio biológico. A Figura 2.2 mostra os elementos de um neurônio artificial clássico.

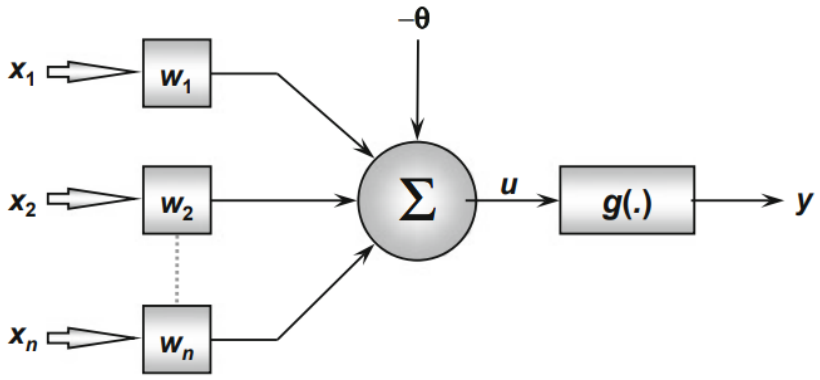

Figura 2.2: Esquema simplificado de um neurônio artificial (SILVA et al., 2017)

2.2.2 Arquitetura

De acordo com (SILVA et al., 2017), esses neurônios artificiais podem ser estruturados em camadas conforme a Figura 2.3.

- Camada de entrada: Responsável por receber as informações de entrada como sinais, atributos ou medidas de um ambiente externo. Essas entradas (amostras ou padrões) são frequentemente normalizadas dentro dos limites dos valores produzidos pelas funções de ativação;
- Camadas ocultas: Camada onde os neurônios estão localizados. São responsáveis por extrair padrões associados ao processo ou sistema analisado. Essas camadas executam a maior parte do processamento interno da rede;
- Camada de saída: Essa camada também é composta de neurônios e tem a função de produzir a saída final da rede resultante do processamento executado pelas camadas anteriores.

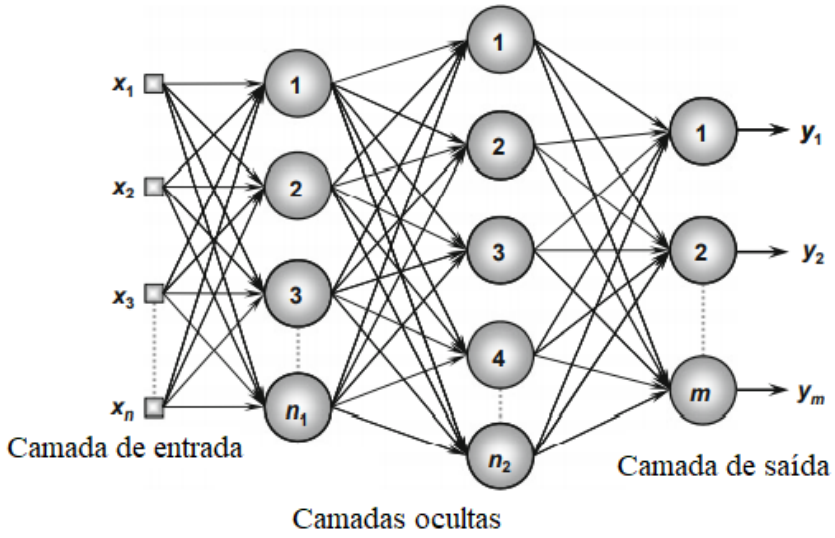

Figura 2.3: Organização de neurônios artificiais em múltiplas camadas (SILVA et al., 2017).

2.2.3 Treinamento

Redes neurais artificiais tem a capacidade de aprender o comportamento de um sistema quando exemplos suficientes são apresentados a elas. Assim, depois de compreender o relacionamento do conjunto de dados de treino, a rede se torna capaz de produzir uma saída que se aproxima da esperada dado qualquer valor de entrada.

O processo de treinamento de uma rede neural artificial envolve uma série de etapas com objetivo de calibrar o vetor de pesos e os limiares de ativação de cada neurônio. Durante esse processo de ajuste, a rede é capaz de extrair traços importantes do sistema que está sendo mapeado.

Redes neurais podem ser utilizadas nas três formas de aprendizado descritas na subseção 2.1. No caso do aprendizado não supervisionado, a rede deve se auto organizar para identificar subconjuntos

nos dados de entrada que apresentam padrões semelhantes. Assim o algoritmo de aprendizagem deve ajustar os pesos e limiares de ativação para refletir esses grupos na própria rede. O processo de aprendizagem supervisionado de redes neurais envolve o ajuste dos pesos e limiares por meio da comparação entre a saída do sistema e a saída esperada. Essa diferença é então utilizada no processo de ajuste. Quando essa discrepância estiver em um limite considerado aceitável, a rede pode ser considerada treinada. Já o treinamento de algoritmos de aprendizagem por reforço envolve um processo de tentativa e erro, uma vez que a única resposta a uma entrada é se ela foi adequada ou não. Caso a resposta tenha sido adequada, os pesos são gradualmente incrementados afim de incentivar esse comportamento.

2.3 Redes neurais artificiais sem peso

Nesse trabalho serão abordadas especificamente redes neurais sem peso, apresentadas pela primeira vez em 1984 e focadas originalmente no processamento de imagens.

Nesse tipo de modelo computacional os neurônios artificiais têm entradas e saídas binárias e nenhum peso entre os nós. As funções computadas de cada neurônio são armazenadas em Tabelas verdade que podem ser implementadas utilizando memórias de acesso aleatório (*Random Access Memories* - RAMs) (GREGORIO; GIORDANO, 2016). Em contraste às redes neurais tradicionais, em que o processo de aprendizado envolve descobrir o valor dos pesos de cada camada, o aprendizado de redes neurais sem peso consiste em atualizar o conteúdo das entradas da tabela verdade.

O treinamento de redes neurais artificias sem peso é considerado rápido devido a independência mútua entre os nós quando as entradas são alteradas. Em modelos com pesos, o treinamento é mais complexo, já que ajustar os pesos para treinar um novo padrão de entrada e saída também envolve a alteração do comportamento desse nó em relação a outros padrões aprendidos anteriormente (LUDERMIR et al., 1999).

Como cada neurônio sem peso (RAM) individual é altamente funcional, redes desse tipo são muito flexíveis. Portanto, as principais vantagens quando comparadas a redes tradicionais são a flexibilidade, alta velocidade de aprendizagem e a possibilidade de implementar redes reais utilizando RAMs disponíveis comercialmente.

Assim como para as redes neurais tradicionais, também é possível analisar redes neurais sem peso pela ótica biológica e traçar um paralelo entre a decodificação dos endereços de memorias das RAMS, e o comportamento de sinalização excitatório e inibitório executado pelos dendritos de um neurônio: quanto mais próximo da soma de um neurônio a entrada da sinapse termina, maior é a influência dessa entrada na determinação da saída no axônio. (ALEKSANDER et al., 2009) Ou seja, a influência de um sinal de entrada depende da posição em que a conexão sináptica está localizada no dentrito, similar a maneira em que a decodificação do endereço da RAM é executada.

2.3.1 RAM

A unidade mais básica desse tipo de rede são as RAMs, memórias usadas para armazenar informações obtidas por meio do processo de treino. Essas unidades foram introduzidas para resolver problemas de reconhecimento de padrões há mais de 50 anos por Bledsoe and Browning (BLEDSOE; BROWNING, 1959).

Um nó RAM de N entradas deve possuir 2^N espaços de memória endereçados por um vetor $a = \{a_1, a_2, ..., a_N\}$ de N bits. Um sinal de entrada $I = \{I_1, I_2, ..., I_N\}$ só vai acessar uma dessas posições de memória, quando a=I. Portanto o bit $C[I]$ armazenado na posição de memória acessada I representa a saída r desse nó, ou seja $r = C[I]$. Dessa maneira, a função *Booleana* executada por cada neurônio é determinada pelo conteúdo da RAM.

O processo de aprendizagem de uma RAM consiste em escrever nas entradas da tabela verdade correspondente. Por isso, esse tipo de neurônio artificial é capaz de computar qualquer função *Booleana* de suas entradas enquanto neurônios artificiais de redes com peso só podem computar funções linearmente independentes. Uma RAM isoladamente não apresenta capacidade de generalização, porém, uma rede combinando vários nós apresenta essa capacidade.

2.3.2 Discriminador

Aleksander descreveu o conceito de Discriminador (ALEKSANDER et al., 2009) através da combinação de várias RAMs. Um Discriminador consiste em um conjunto de X RAMs de palavras de 1-bit com n entradas e um dispositivo de soma. Cada Discriminador pode receber um padrão binário de Xn bits como estímulo. As entradas da RAM são conectadas ao padrão de entrada do sistema por um mapeamento pseudoaleatório. É importante notar que para que haja consistência na informação os bits de endereçamento da rede devem sempre se referir à mesma informação. Por exemplo, no processamento de imagens cada RAM deve endereçar cada pixel na mesma posição de memória.

O dispositivo de soma é o componente que permite que a rede apresente generalização e tolerância ao ruído, da mesma maneira que as redes neurais com peso convencionais. Um esquemático de um Discriminador é mostrado na Figura 2.4.

O processo de treinamento de um Discriminador consiste em definir todas as posições de memória RAM em 0 e determinar um conjunto de treino formado por padrões binários de Xn bits. Para cada estímulo de entrada, é atribuído o valor 1 à posição de memória de cada RAM endereçada por esse padrão de entrada. Uma vez que o treino de padrões é encerrado, o conteúdo das memorias RAM conterão um determinado número de zeros e uns.

O processo de classificação envolve estímulos de entrada que não foram utilizados no processo de treino e, portanto, não são conhecidos pela rede. Quando um padrão desconhecido $\{e\}$ é utilizado como entrada, os conteúdos da memória RAM endereçados pelo estímulo são lidos e somados pelo dispositivo de soma. O resultado dessa soma r é chamado de resposta do Discriminador e é igual ao número de endereços que armazenam 1. Caso $\{e\}$ pertença ao conjunto de treino, r atinge seu

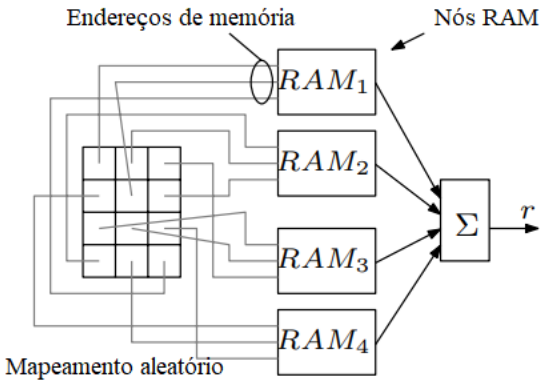

Figura 2.4: Esquemático de um Discriminador (ALEKSANDER et al., 2009)

valor máximo, caso nenhum componente de n-bit de $\{e\}$ apareça no conjunto de treino (nenhum endereço de memória armazena 1), r vale 0. Valores intermediários de r apresentam uma medida de similaridade de $\{e\}$ em relação ao conjunto de treino.

2.3.3 WiSARD

A WiSARD (Wilkes, Stonham and Aleksander Recognition Device) foi a primeira rede artificial sem pesos patenteada e utilizada comercialmente. Esse tipo de rede é composta por um conjunto de Discriminadores, conforme descritos na Seção anterior. A cada um deles, na etapa de treino, é introduzido um conjunto de padrões binários pertencentes à alguma classe específica como entrada. Uma representação de um nó WiSARD de n Discriminadores é mostrada na Figura 2.5.

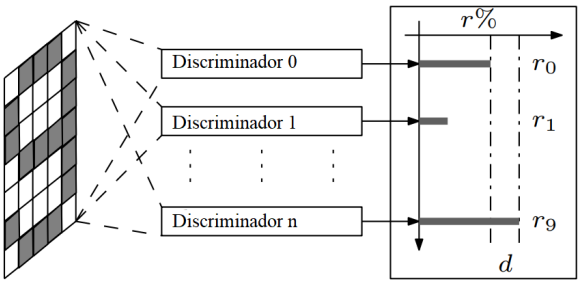

Figura 2.5: Esquemático de uma WiSARD (ALEKSANDER et al., 2009)

Portanto, em um sistema WiSARD, cada Discriminador é treinado para reconhecer uma classe diferente de padrões. Dessa maneira, durante o treinamento de um padrão pertencente a uma classe v, o processo de escrita ocorre apenas nas RAMs do Discriminador correspondente a essa classe.

Já o processo de inferência consiste em receber a resposta de cada Discriminador à um estimulo desconhecido de teste. À esse estímulo é atribuída a classe correspondente ao Discriminador que retornou a maior resposta r. A confiança relativa do resultado pode ser determinada por outro algoritmo, por exemplo, a diferença entre a quantidade de votos da maior resposta e a quantidade

de votos da segunda maior resposta segunda maior resposta divida pelo primeiro valor.

2.4 Computação distribuída

A computação distribuída consiste no compartilhamento de recursos computacionais interligados por uma rede. Mais do que a simples subdivisão de tarefas, este paradigma permite a repartição e a especialização das tarefas computacionais conforme a natureza da função de cada computador. Um exemplo típico é a chamada arquitetura cliente/servidor, onde muitos computadores "clientes" se comunicam com computadores "servidores" que nada mais são do que processos especializados na execução de certas tarefas, como cuidar de arquivos ou administrar bancos de dados.

A interação entre cada componente dessa rede é de suma importância, pois suas ações são coordenadas apenas por meio de troca de mensagens. Logo, as mensagens possivelmente não serão compreendidas caso o protocolo ou o canal de comunicação não esteja acessível a um ou mais elementos do sistema distribuído. Devem ser levados em consideração ainda cuidados especiais para que as mensagens inválidas sejam rejeitadas; caso contrário, o sistema como um todo pode ser comprometido.

2.4.1 Programação orientada a objetos

Programação Orientada a Objetos, comumente chamada de POO é uma abordagem para concepção de sistemas baseado na composição e interação entre diversas unidades chamadas de objetos. A POO surgiu com o objetivo de atender as necessidades dos programadores, aproximando o código dos problemas reais.

Uma vez que o domínio do problema do mundo real é caracterizado por objetos e suas interações, um aplicativo de software desenvolvido usando a abordagem de programação orientada a objetos resultará na produção de um sistema de computador que tem uma representação mais próxima do domínio do problema do mundo real(DANNY POO DEREK KIONG, 2007). Para entender esse paradigma da programação é necessário conceituar dois componentes importantes: os objetos e as classes.

Um objeto consiste em um elemento que possui atributos e métodos, já as classes são representações abstratas, responsáveis por instanciar os objetos à ela pertencentes. Para ilustrar a situação pode-se pensar em um carro, que é vermelho, tem quatro portas, quatro janelas, um motor 2.0 e várias outras características. Este carro possui comportamentos como aceleração, acender ou apagar os faróis, possibilidade de utilizar ou não a buzina, dentre outros. Sabe-se que é possível encontrar no mercado diversos tipos de carro, com diferentes cores, motores ou quantidade de portas. Dessa forma, para esse exemplo o carro vermelho citado no início é um objeto, da classe carro, suas características são os atributos e os comportamentos são os métodos, conforme a figura 2.6.

A partir das definições de classe e objeto, derivam mais três conceitos significantes, o encapsulamento, a herança e o polimorfismo.

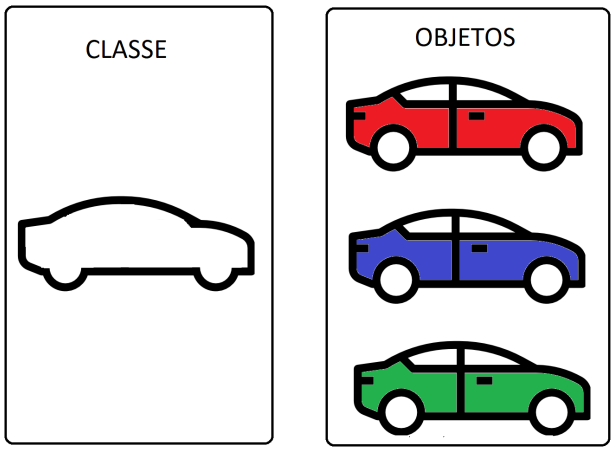

Figura 2.6: Esquema simplificado de uma classe com objetos

De maneira simplificada, encapsulamento significa reunir um conjunto de atributos e métodos de um objeto e ocultar sua estrutura de implementação dos usuários (DANNY POO DEREK KIONG, 2007). O encapsulamento não é obrigatório, mas é uma boa prática de programação, pois produz classes mais eficientes. Há três tipos de encapsulamento:

- Público: O atributo, o método ou a classe podem ser acessados livremente;
- Privado: O atributo, o método ou a subclasse somente podem ser acessados dentro da classe em que foram declarados;
- Protegido: O atributo, o método ou a subclasse podem ser acessados dentro da classe em que foram declarados e a partir de classes descendentes.

O conceito de herança permite definir uma nova classe, com base em outra classe. A subclasse criada automaticamente herda todas os atributos e métodos da classe já existente. O mecanismo de herança permite que a subclasse inclua ou sobreponha novas variáveis e métodos. O mecanismo de herança é recursivo, permitindo criar-se uma hierarquia de classes, do nível mais alto, onde as características herdadas são comuns a todos os objetos desta classe até os níveis inferiores, onde estão as especializações das classes superiores.

Os objetos respondem às mensagens que eles recebem através dos métodos. A mesma mensagem pode resultar em diferentes resultados. Esta propriedade é chamada de Polimorfismo(DANNY POO DEREK KIONG, 2007). Na prática, o polimorfismo permite que classes derivadas tenham métodos iguais, mas com resultados redefinidos em cada uma das classes filhas.

2.4.2 Programação distribuída

Programação distribuída é a atividade de criação de algoritmos e programas para serem executados concorrentemente através da troca de mensagens entre os componentes distribuídos. Tais

componentes são interligados por uma rede de internet ou intranet. De acordo com Andrew Tanenbaum, um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente. (TANENBAUM, 2007)

Na prática, a programação distribuída traz três benefícios principais (PALACH, 2014):

- **Tolerância a falhas**: Como o sistema é descentralizado, é possível distribuir o processamento para diferentes máquinas em uma rede e, assim, executar individualmente manutenção de máquinas específicas sem afetar o funcionamento do sistema como um todo.
- **Escalabilidade horizontal**: É possível aumentar a capacidade de processamento em sistemas distribuídos em geral. Visto que para conectar novos equipamentos não é necessário abortar os aplicativos em execução.
- **Computação em nuvem**: Para a redução dos custos de hardware, é possível utilizar esse tipo de arquitetura com máquinas atuando de forma cooperativa e executando programas de forma transparente para seus usuários.

A Figura 2.7 mostra um esquema de sistema distribuído, onde é possível perceber a possibilidade de compartilhar o processamento por troca de dados por meio de mensagens entre máquinas (nós) de computação, que estão fisicamente separados.

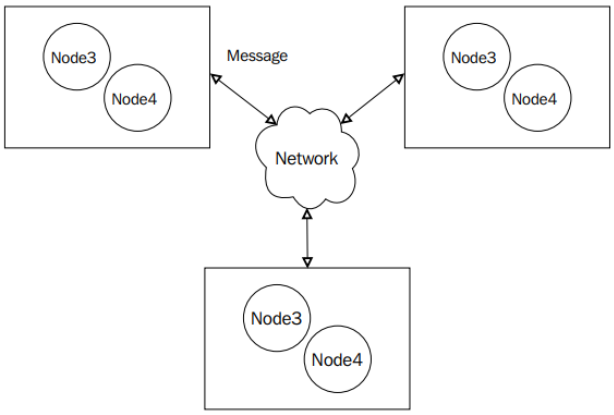

Figura 2.7: Esquema de programação distribuída (PALACH, 2014)

Mesmo com tantos benefícios, se faz necessário dar a devida atenção a algumas desvantagens desse paradigma da programação. Por exemplo, podem ocorrer falhas de rede, como a presença de enlaces de má qualidade, que resultem em comprometimentos ao sistema.

2.4.3 Objetos Distribuídos

O conceito de objetos distribuídos envolve a utilização da Programação Orientada a Objetos somada a Programação Distribuída, conforme descrito nas Seções 2.4.1 e 2.4.2, respectivamente. O paradigma da orientação a objetos é bastante apropriado para o desenvolvimento de sistemas

distribuídos. Ele provê um suporte natural para as características necessárias para a construção de sistemas distribuídos bem estruturados.

Já é conhecida a capacidade de encapsular atributos e métodos de um objeto, de modo efetivo, isso significa que não há nenhuma forma de manipular um objeto a não ser pela a invocação dos métodos disponibilizados por uma interface. Essa característica é o que permite que um cliente faça requisições, invocando métodos em uma máquina, enquanto o objeto, em si, reside em outra máquina, conforme ilustrado na Figura 2.8.

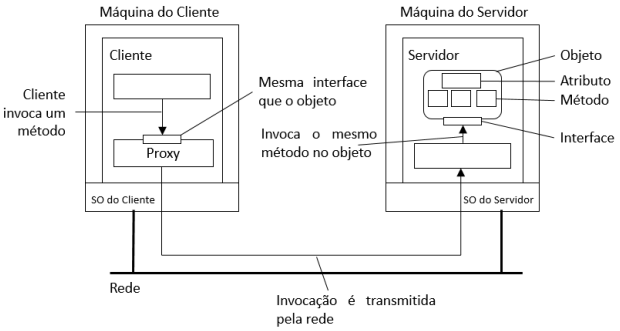

Figura 2.8: Esquema de um objeto distribuído (TANENBAUM, 2007)

Nesse contexto, tudo é tratado como objeto, e serviços e recursos são oferecidos a clientes na forma de objetos que eles possam invocar (TANENBAUM, 2007). Como ilustrado na Figura 2.8, os objetos ficam em máquinas distintas, que são chamadas de servidores de objeto. Esses servidores são configurados para suportar objetos distribuídos. A diferença básica entre este e um servidor tradicional é que um servidor de objetos, por si só, não fornece um serviço específico, visto que, os serviços específicos são implementados pelos objetos que residem no servidor. Em suma, o servidor é a casa dos objetos, fornecendo somente os meios para invocação dos métodos, com base em requisições de clientes remotos.

2.4.4 Troca de mensagens baseada em objetos

A comunicação é um ponto de atenção quando se trata de um sistema distribuído. De um modo geral, os meios para um cliente remoto invocar um objeto são baseados em chamadas de procedimento remoto (RMI - *remote method invocation*).

Uma RMI é capaz de transferir parâmetros do método e referências do objeto no âmbito do sistema (TANENBAUM, 2007). Essas invocações remotas dependem da linguagem de programação utilizada, sendo o Java RMI uma das mais conhecidas. Com o crescimento da popularidade da linguagem de programação *Python*, surgiu a necessidade de um RMI próprio. Em 1998 a primeira versão do *middleware* Pyro (*Python Remote Objects*) começou a ser desenvolvida, possibilitando que os programadores dessa linguagem construam sistemas distribuídos baseados em Objetos Distribuídos.

Essencialmente, o Pyro pode ser usado para distribuir e integrar vários tipos de recursos ou responsabilidades: recursos computacionais (cpu, armazenamento, impressoras), recursos informa-

cionais (dados, informações privilegiadas) e lógica de negócios (departamentos, domínios) (JONG, 2021). O Pyro é, em seu fundamento, uma biblioteca (*middleware*) escrita totalmente em *Python* que permite a comunicação de objetos pela rede. De maneira que as chamadas de um método *Python* são executadas normalmente, com quase todos os parâmetros e tipos de valor de retorno possíveis, ficando a cargo do Pyro localizar corretamente o objeto e o computador para execução do método.

Essa localização é feita por um servidor de nomes que armazena o caminho e os metadados dos objetos para que seja possível encontrá-los de maneira transparente. Esses metadados são etiquetas que o programador pode atribuir a cada objeto para facilitar o processo de busca posterior. Essa biblioteca oferece ainda flexibilidade, podendo ser utilizada entre diferentes arquiteturas de sistema e sistemas operacionais, com capacidade de se comunicar entre diferentes versões do Python de forma transparente. É possível instanciar os objetos remotos de três formas diferentes (JONG, 2021):

- *single*: (configuração padrão) Um novo objeto é criado para cada nova conexão de proxy e é reutilizado para todas as chamadas durante aquela sessão específica. Outras sessões lidarão com um objeto diferente.
- *session*: Um único objeto será criado e usado para todas as chamadas de método, independentemente da conexão proxy utilizada.
- *percall*: Um novo objeto é criado para cada chamada de método e descartado posteriormente.

2.5 Aprendizado Federado

O aprendizado federado (*Federated Learning - FL*) é proposto como uma solução de aprendizado colaborativo com foco na preservação da privacidade e eficiência de comunicação (MCMAHAN et al., 2017). Ele permite o treinamento de modelos em um grande corpo de dados descentralizados.

Essa preservação da privacidade é intrínseca a arquitetura do FL, visto que os dados do usuário não são compartilhados pela rede e apenas atualizações/parâmetros de modelo são transmitidas. Portanto, esse tipo de aprendizagem distribuída tem sido usada em aplicações do mundo real onde a privacidade do usuário é crucial, por exemplo, dados hospitalares e previsões de texto em dispositivos móveis. Idealmente, qualquer abordagem desse tipo é considerada segura, pois os metadados são considerados menos informativos do que os dados originais (QIANG YANG LIXIN FAN, 2020). É possível observar o comparativo entre os dados originais e os dados transmitidos pela rede na Figura 2.9, pode-se perceber que é difícil inferir de uma lista de vetores numéricos de que a imagem original é um gato.

O aprendizado federado é o treinamento de um modelo de aprendizado de máquina a partir de dados armazenados em dispositivos remotos. Assim é possível entender o paradigma como um conjunto de tarefas realizado por clientes e servidores.

[1, 0, 0 … 0]
cat

```
[[ 0.75,  1.26,  0.56,  ..., -0.19],
 [-0.99, -0.37, -0.93,  ...,  2.54],
 [ 0.06, -0.96,  0.78,  ..., -0.85],
 ...,
 [-0.55, -0.55, -1.31,  ...,  0.32]]
```

(a) Dado de treinamento.

(b) Metadados correspondentes.

Figura 2.9: Dados treinados vs. metadados.

Para treinar um modelo que permite aprendizado federado os clientes devem executar a recuperação dos parâmetros de treino no servidor. Em seguida, devem atualizar o modelo com seus próprios dados armazenados localmente, para, por fim, transferir para o servidor os novos parâmetros.

Já os servidores ficam responsáveis por agregar as atualizações advindas de diversos clientes, visando melhorar o modelo como um todo. Após essa agregação, um modelo mais atualizado é repassado a todos os clientes, para que estes sejam capazes de realizar testes ou aplicações. A Figura 2.10 ilustra esse processo.

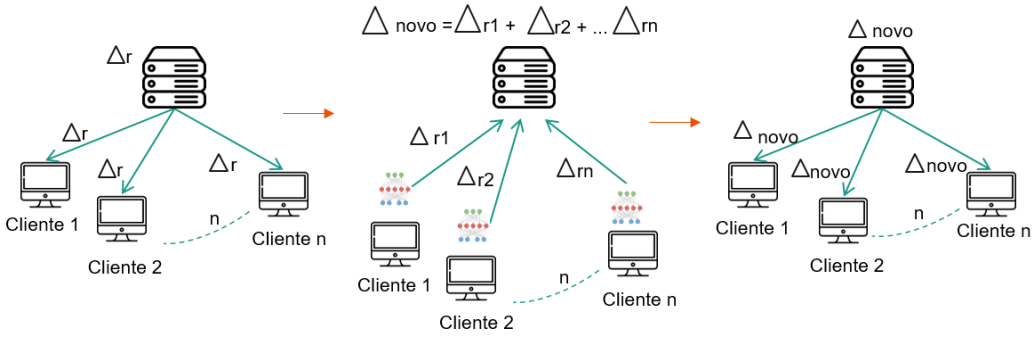

Figura 2.10: Esquema de aprendizado federado.

O FL pode utilizar dois tipos de abordagem, na primeira é considerado um cenário ideal, onde os dados locais dos participantes são independentes e identicamente distribuídos (IID). Por outro lado, a segunda abordagem considera um cenário mais próximo do real, onde tantos os dispositivos quanto a estatística dos dados são heterogêneos e por isso foram um conjunto de dados que não são independentes e podem ter distribuições variadas (No-IID).

Embora esse paradigma seja proposto como uma solução de privacidade para a viabilização de modelos de aprendizado de máquina que utilizam dados sensíveis, diversos ataques focam no próprio funcionamento do aprendizado para extrair informações dos clientes ou para perverterem o modelo gerado.

Então, o FL nem sempre fornece garantias de privacidade suficientes, uma vez que a comunicação de atualizações do modelo durante o processo de treinamento pode revelar informações confidenciais, mesmo incorrendo em vazamentos profundos, seja para terceiros ou para o servidor central. Mesmo uma pequena porção de metadados pode revelar informações sobre dados locais.

Um trabalho mais recente mostrou que o invasor mal-intencionado pode roubar completamente os dados de treinamento a partir dos metadados em algumas iterações (QIANG YANG LIXIN FAN, 2020).

Capítulo 3

Metodologia

3.1 Arquitetura

A estrutura desse trabalho utilizou máquinas virtuais para simular uma rede, na qual objetos Python remotos podem se comunicar entre si por meio da biblioteca Pyro 4.80.

Na arquitetura proposta, cada elemento constituinte de uma rede neural artificial sem pesos descritos na Seção 2.3 foi modelado como uma classe Python. Depois, com o auxílio da biblioteca Pyro4 essas classes foram espalhadas em uma rede de computadores de forma que cada etapa do processo reside em um objeto distribuído específico. Esses objetos podem ser compreendidos como um servidor que implementa as funcionalidades de cada uma das classes do modelo.

Dessa forma, para implementar um classificador de imagens, é preciso instanciar um servidor WiSARD para cada *dataset*. Aqui, entende-se um *dataset* como um conjunto de imagens de diferentes categorias etiquetadas, como ilustrado na Figura 3.1.

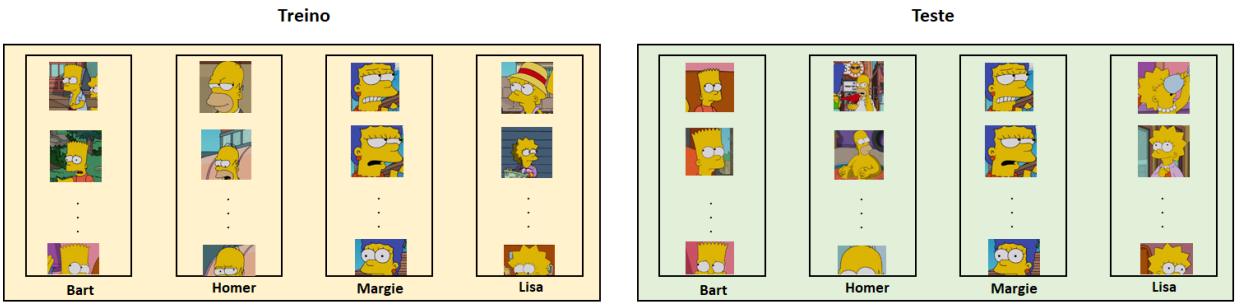

Figura 3.1: Exemplo de *dataset* Simpsons

Ou seja, por exemplo, para criar um detector de máscaras faciais primeiro é necessário instanciar um servidor WiSARD e publicá-lo no *nameserver* (seção 2.4.4) presente na rede local.

Em seguida, são identificadas as categorias do *dataset* do servidor WiSARD em questão e, para cada uma delas um servidor Discriminador é instanciado. A Figura 3.2 mostra um exemplo onde o *dataset* possui quatro categorias distintas. Por isso, foram instanciados quadro servidores Discriminador.

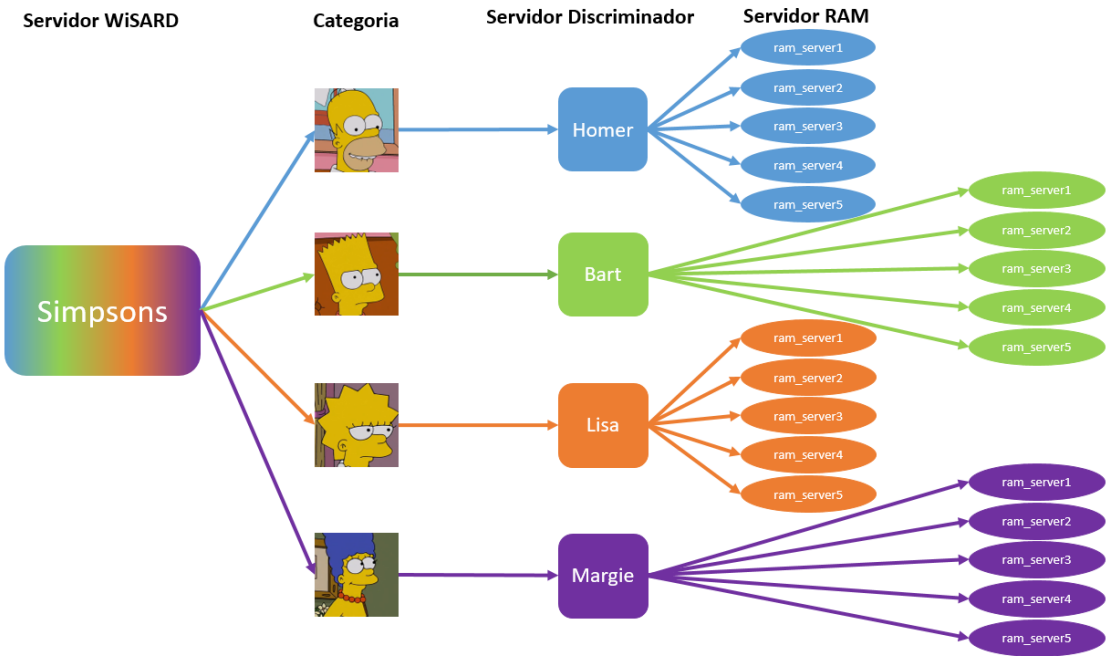

Figura 3.2: Arquitetura dos servidores: Simpsons

Já a Figura 3.3 apresenta apenas duas categorias: imagens de rostos com máscara e imagens de rostos sem máscara, portanto, para este exemplo é necessário instanciar apenas dois servidores de Discriminador. Os servidores Discriminador implementam as funcionalidades descritas na Seção 2.3.2.

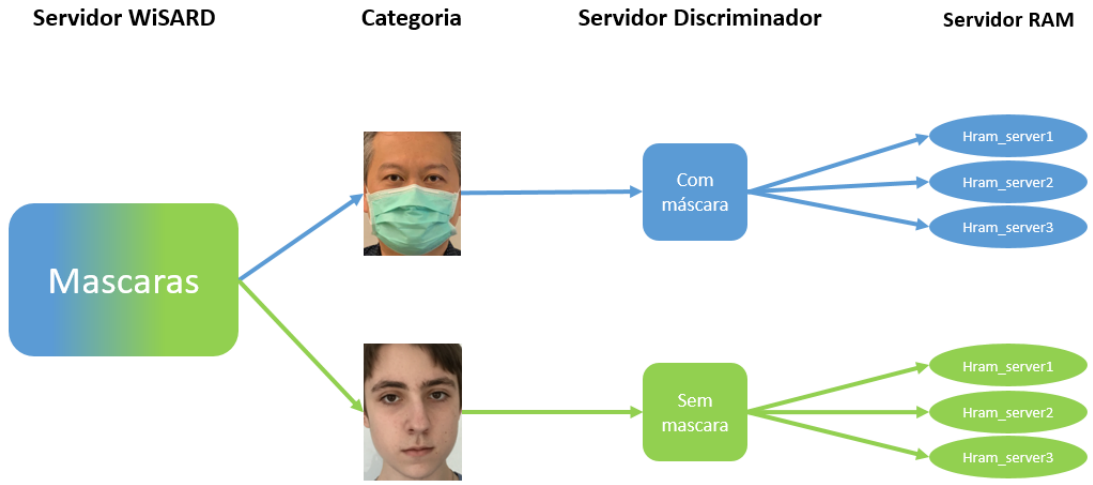

Figura 3.3: Arquitetura dos servidores: máscaras

Por fim, cada Discriminador instancia seus próprios servidores RAM. A quantidade de servidores é variável e pode ser definida de acordo com as necessidades do cliente como mostram as Figuras 3.2 e 3.3, na qual a primeira tem cinco servidores RAM para cada Discriminador e a segunda tem apenas três. Cada um desses servidores implementa as funcionalidades descritas na Seção 2.3.1.

Na arquitetura proposta, o papel do *nameserver* é manter o controle e divulgar os objetos Pyro

disponíveis na rede. O Snippet 3.1 ilustra o processo de registro de um dos servidores descritos anteriormente, no *nameserver* (linha 12). Os valores inseridos no campo *metadata* podem ser utilizados para buscar esse objeto na rede.

Cada objeto acessa as informações armazenadas no *nameserver* buscando essa entidade por meio do endereço IP desse host.

O processo de inicialização do *nameserver*, consiste em executar no terminal de uma máquina conectada a rede, o comando **$ pyro4-ns -n {IP_DA_MAQUINA}**, dessa maneira, todas as máquinas dessa rede poderão acessar os objetos registrados.

O *nameserver*, em si, é um objeto Pyro. Por isso, para acessá-lo utiliza-se um mecanismo de pesquisa de transmissão. Esse mecanismo permite a descoberta de forma automática de *nameservers* disponíveis em uma sub-rede. A linha 9 do Snippet 3.1 procura esses servidores e retorna o primeiro resultado.

Para publicar uma classe Python normal e transformá-la em um objeto Pyro, basta informar ao Pyro, por meio de um *Daemon*, que ficará responsável por escutar as solicitações recebidas e processá-las.

Para criar um *Daemon* basta executar o método Pyro4.Daemon() e registrar o objeto resultante (linha 11). Em seguida para que todos os outros objetos da rede sejam capazes de encontrar a classe publicada é preciso incluir o *decorator* **@Pyro4.expose** (linha 3)

Os valores inseridos no campo *metadata* na linha 12 podem ser utilizados para buscar um objeto na rede. Esses valores são marcas, ou *tags*, que possibilitam solicitar ao *nameserver* quaisquer objetos registrados que possuam esses metadados.

Por fim, é executado o laço de repetição do *Daemon* como mostrado na linha 13 do Snippet 3.1, para fazer com que o Pyro fique a espera de solicitações.

Snippet 3.1: Instanciar um servidor

```
1  import Pyro4
2
3  @Pyro4.expose
4  class Wisard:
5      # ...métodos que poderão ser chamados...
6
7      daemon = Pyro4.Daemon(host="192.168.0.116")
8      ns = Pyro4.locateNS()
9      print(ns)
10     uri = daemon.register(Wisard)
11     ns.register("wisard_server1", uri, metadata={"wisard","simpsons"})
12     daemon.requestLoop()
```

3.2 Treinamento

O treino da rede é executado da mesma maneira descrita na Seção 2.3 mas executado de maneira descentralizada. Isso quer dizer que, de fato, o treino ocorre nos servidores RAM. As funções de treino dos servidores WiSARD e Discriminador são usadas para controle e distribuição dos exemplares de treino.

Utilizando uma abordagem *top-down*, o processo de treino se inicia pela aplicação cliente onde as imagens utilizadas para treino e classificação são carregadas e redimensionadas por um processo de interpolação da biblioteca cv conforme o Snippet 3.2 .

Snippet 3.2: Redimensionamento de imagens

```
def image_resize(img, size, interpolation = cv.INTER_AREA):
    h, w = img.shape[:2]
    c = None if len(img.shape) < 3 else img.shape[2]
    if h == w:
        return cv.resize(img, (size, size), interpolation)
    if h > w:
        dif = h
    else:
        dif = w
    x_pos = int((dif - w)/2.)
    y_pos = int((dif - h)/2.)
    if c is None:
        mask = np.zeros((dif, dif), dtype=img.dtype)
        mask[y_pos:y_pos+h, x_pos:x_pos+w] = img[:h, :w]
    else:
        mask = np.zeros((dif, dif, c), dtype=img.dtype)
        mask[y_pos:y_pos+h, x_pos:x_pos+w, :] = img[:h, :w, :]
    return cv.resize(mask, (size, size), interpolation)
```

A aplicação cliente também é responsável por se conectar com um dos os servidores WiSARD disponíveis em sua rede. Para tal apenas é necessário que o usuário insira o nome do servidor desejado na flag –wisard quando for executar o programa. Essa entrada é armazenada em uma variável e é utilizada para procurar um servidor que possua essa string como metadado. O Snippet 3.3 mostra o como essa busca é realizada:

Snippet 3.3: Busca por servidores WiSARD

```
wsd = Pyro4.Proxy("PYROMETA:"+'wisard'+','+ wisard)
```

Caso seja a primeira vez que esse servidor WiSARD é utilizado, é preciso instanciá-lo conforme o Snippet 3.4. A Tabela 3.1 mostra o significado de cada um dos parâmetros:

Snippet 3.4: Instanciamento de um servidor WiSARD

```
wsd.instantiate(num_of_htables=entrySize//addressSize,
                input_addr_length=addressSize,
                wisard = wisard,
                qtd_hram_server = 5)
```

Tabela 3.1: Parâmetros do servidor WiSARD

Parâmetros	**Significado**
num_of_htables	Tamanho de cada unidade RAM
input_ addr_length	Tamanho do endereço de memória de cada entrada
wisard	Nome do servidor
qtd_hram_server	Quantidade de RAMS usadas na classificação de cada categoria

Em seguida, utilizando a biblioteca Pyro4 os dados de treino são enviados para o servidor WiSARD a ser treinado. Nesse servidor, dentro da função de treino, um servidor de Discriminador para cada categoria é instanciado, caso ainda não tenha sido, conforme mostrado nas linhas 2-7 do Snippet 3.5.

Snippet 3.5: Função de treino em um servidor WiSARD

```
1     def train(self, X, label):
2         if label not in self.discs:
3             self.discs[label] =  Pyro4.Proxy("PYROMETA:"+'wisard'+','+ self.wisard+','+label)
4             self.discs[label].instantiate(num_of_htables = self.num_of_htables,
5                                           qtd_hram_server = self.qtd_hram_server,label=label)
6             self.d_times_trained[label] = 0
7             self.d_relevance[label] = 0
8         tmp = X[self.mapping]
9         addresses = self.ranks(tmp)
10        self.discs[label].train(addresses)
```

Depois, ainda na função de treino, as imagens passam pelo processo de mapeamento aleatório descrito na Seção 2.3. Os endereços mapeados para cada imagem seguem para os servidores Discriminador. Nessa etapa, os endereços são encaminhados para o servidor específico de cada categoria, como mostra a linha 9. Por exemplo, endereços com a *label* Homer só serão enviados ao Discriminador responsável por treinar essa categoria.

A etapa de treino nos Discriminadores é simples: as imagens são divididas pelo número de servidores RAM disponíveis para cada categoria, e, cada fragmento é enviado para um servidor RAM responsável por realizar o treinamento. A Figura 3.4 ilustra esse processo no caso de existirem apenas 2 servidores RAM para cada categoria, e o Snippet 3.6 mostra como essa divisão acontece no código. Aonde **qtd_hram_server** é o número de servidores RAMs, esse valor é recebido das aplicações cliente (Tabela 3.1).

Snippet 3.6: Função de treino em um servidor Discriminador

```
    def train(self,addresses):
        addresses_split = np.array_split(addresses, self.qtd_hram_server)
        for x in range(self.qtd_hram_server):
            self.dict_hrams_servers[x].train(addresses_split[x])
```

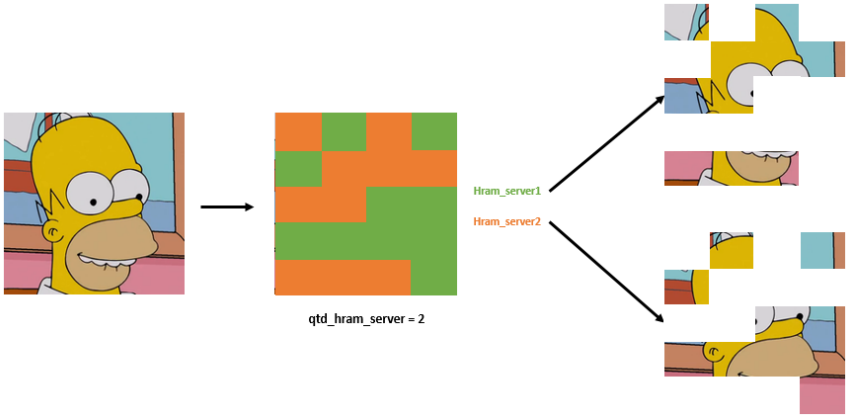

Figura 3.4: Ilustração do processo de treino

É nos servidores RAM que o processo de treino é, de fato, executado (seção 2.3), ou seja, são neles que as Tabelas verdade características de redes neurais sem peso são implementadas. O Snippet 3.7 mostra como cada servidor é instanciado.

Snippet 3.7: Instanciamento de um servidor RAM

```
    def instantiate(self, num_of_htables):
        self.num_of_htables = num_of_htables
        self.h_rams = [dict() for x in range(num_of_htables)]
        self.times_trained = 0
```

Nessa representação ,as RAMs são compostas por uma lista de dicionários. As chaves desses dicionários são os endereços de memória recebidos pelo servidor.

A função *train* recebe um fragmento de imagem, itera por cada um dos dicionários da lista RAMs e checa se um endereço de memória está entre suas chaves, caso positivo ele acrescenta 1 nos valores, caso contrário uma nova chave é inserida no dicionário já a iniciando com valor 1, conforme mostra o Snippet 3.8

Snippet 3.8: Função de treino de um servidor RAM

```
    def train(self, X):
        for i in range(0, self.num_of_htables):
            key = X[i]
```

```
4               if key in self.h_rams[i].keys():
5                   self.h_rams[i][key] += 1
6               else:
7                   self.h_rams[i][key] = 1
8           self.times_trained += 1
```

3.2.1 Treinamento Federado

Uma funcionalidade da rede proposta nesse trabalho é a possibilidade de realizar treinamento federado (seção 3.2.1). Para tal, cada aplicação cliente precisa simplesmente se conectar a um servidor WiSARD disponível. Esse servidor armazena o mapeamento aleatório das imagens para os endereços de memória de forma que, haja consistência entre os treinos e subsequentes classificações. A conexão com um servidor WiSARD pode ser vista no Snippet 3.3, a variável **wisard** é enviada pelo cliente, e determina qual tipo de WiSARD o *nameserver* deve pesquisar.

Assim, cada cliente, em separado, pode subir diversas imagens no intuito de treinar a rede com dados de maneira mais rápida e com informações mais diversificadas contribuindo para a regularização da rede.

3.3 Classificação

Após o treinamento da rede neural sem pesos é possível realizar a classificação das imagens nas categorias treinadas. Novamente, o processo de classificação, de fato, ocorre nos servidores RAM nas extremidades da arquitetura. Portanto, também é possível descrever esse processo utilizando uma abordagem *top-down*. Primeiro a imagem, ou conjunto de imagens a serem classificadas são carregadas e passam por um processo de redimensionamento idêntico ao realizado na etapa de treino na aplicação cliente.

Em seguida, utilizando o protocolo Pyro4, as imagens são enviadas para o servidor WiSARD desejado onde elas passam exatamente pelo mesmo processo de mapeamento das imagens de treino. O servidor WiSARD é responsável por armazenar esse mapeamento no atributo **selfmapping** (Snippet 3.9). Os endereços de memória resultantes são transmitidos para os servidores de Discriminador como mostra a linha 11 do Snippet 3.10.

Snippet 3.9: Mapeamento aleatório

```
1           self.mapping = list(range(0, self.num_of_htables*self.input_addr_length))
2           random.shuffle(self.mapping)
```

Snippet 3.10: Função de classificação em um servidor WiSARD

```
1       def classify(self, X, total_trained=0):
2           tmp = X[self.mapping]
```

```
        biggest = -1
        secon_biggest = -1
        label = -1
        d_label = None
        addresses = self.ranks(tmp)
        b = 0
        for label in self.discs.keys():
            try:
                self.d_votes[label] = self.discs[label].classify(addresses, b)
            except Exception as e:
                print(e)
                self.d_votes[label]=0
        votes = list(sorted(self.d_votes.values()))
        label = max(self.d_votes, key=self.d_votes.get)
        biggest = votes[-1]
        secon_biggest = votes[-2]
        conf = 0
        if biggest > 0:
            conf = (biggest - secon_biggest)/biggest
        return (label, biggest/self.num_of_htables, conf)
```

Nos servidores Discriminador as imagens são novamente dividas pela quantidade de servidores RAM disponíveis para essa categoria (Snippet 3.11). Cada servidor RAM contabiliza quantas vezes os endereços de memória recebidos foram acessados na etapa de treino e retornam esse valor, aqui chamados de votos, para os Discriminadores (Snippet 3.12).

Snippet 3.11: Função de classificação em um servidor Discriminador

```
    def classify(self,X,bleach=0): #no bleaching
        self.votos = 0
        addresses_split = np.array_split(X, self.qtd_hram_server)
        for x in range(self.qtd_hram_server):
            try:
                self.votos += self.dict_hrams_servers[x].classify(addresses_split[x])
            except:
                pass
        return self.votos
```

Snippet 3.12: Função de classificação em um servidor RAM

```
    def classify(self, X, bleach =0):
        votes = 0
        for i in range(0, self.num_of_htables):
            key = X[i]
            if key in self.h_rams[i].keys():
                if self.h_rams[i][key] > bleach:
                    votes += 1
        return (votes)
```

Em seguida, cada Discriminador soma todos os que votos recebidos e envia esse resultado para o servidor WiSARD. Na WiSARD, a categoria correspondente ao Discriminador que retornou o maior número votos é atribuída a imagem (Snippet 3.10, linha 16). Por fim, essa categoria é enviada para aplicação cliente que a exibe ao usuário juntamente com a confiança na predição.

A classificação segue conforme o mapeamento dos endereços, de forma que cada parte da imagem busca as informações armazenadas nos servidores RAM equivalentes. Por exemplo, a primeira porção de uma imagem do Homer, será comparada com as informações do treino do servidor de *RAM server 1* do Homer, do Bart, da Margie e da Lisa. Da mesma forma que a segunda porção da imagem será comparada com os servidores de *RAM server 2* e assim sucessivamente. Por esse motivo, é tão importante que o mapeamento realizado no treino e na classificação sejam iguais.

3.4 Métricas avaliadas

Para cada *dataset* foram realizadas simulações onde alguns servidores ficaram indisponíveis, essa quantidade de servidores indisponíveis implica em uma taxa de falhas. Para os cenários de teste propostos foram avaliadas métricas de:

- Acurácia, observando desde a taxa de acertos da rede no geral até a taxa individual de cada categoria com o auxílio da matriz de confusão;
- Atraso de processamento, observando o tempo total de classificação.

3.5 Cenários de teste

Como descrito na Seção 2.4.3, na arquitetura de testes proposta, cada elemento do modelo é um servidor Pyro distribuído em uma rede de computadores. Cada um desses servidores armazena um objeto Python que fornece meios invocação dos métodos, com base em requisições dos outros servidores Pyro. Dessa maneira, para cada *dataset*, um servidor WiSARD foi instanciado, e, para cada categoria que a WiSARD é capaz de classificar, um Discriminador foi instanciado. Por fim, para cada Discriminador, cinco servidores RAM foram instanciados.

A arquitetura final da rede, usada como base para execução dos testes pode ser conferida na Figura 3.5

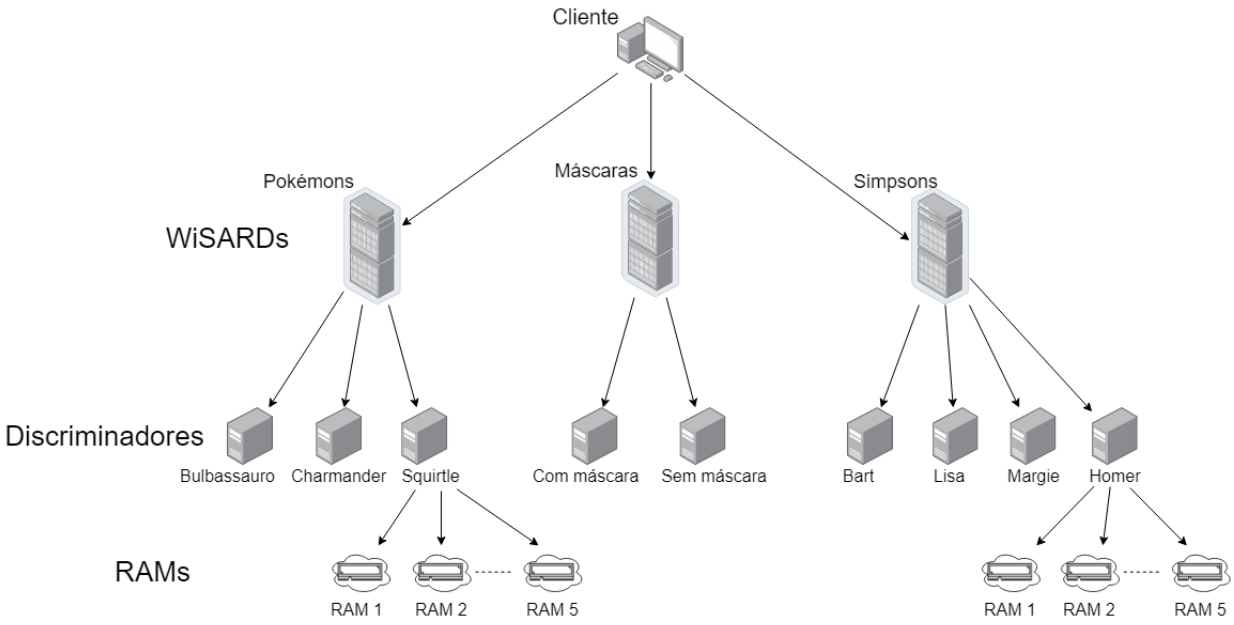

Figura 3.5: Arquitetura de testes do modelo

Para treinar cada uma das WiSARDs , foram utilizados três *datasets* balanceados, um para classificação de personagens dos Simpsons, outro para detecção da utilização de máscaras e outro para classificação de diferentes Pokémons.

O *dataset* dos Simpsons (ATTIA, 2018) possui quatro categorias, sendo elas, Homer, Bart, Margie e Lisa. O *dataset* das máscaras (GURAV, 2020) consiste em apenas duas: com máscaras ou sem máscaras. Por fim, o *dataset* de Pokémons (DWIVEDI, 2018) possui três categorias disponíveis: Charmander, Bulbassauro e Squirtle. A Tabela 3.2 mostra a quantidade de imagens no conjunto de treino e teste para cada *dataset*

Tabela 3.2: Tamanho dos *datasets*

	Tamanho *dataset* de treino	**Tamanho *dataset* de teste**
Simpsons	168 imagens	100 imagens
Pokémons	300 imagens	180 imagens
Máscaras	332 imagens	130 imagens

Quatro cenários de teste foram propostos para avaliar o desempenho da arquitetura apresentada. Primeiro, no cenário de testes I, a rede foi avaliada quanto a sua capacidade de executar treinamento colaborativo. Nos cenários de teste II a IV, a rede foi avaliada quanto a sua robustez e tolerância a falhas. Para isso, esses cenários simulam falhas em dois elementos da rede: servidores Discriminador e servidores RAM.

Entende-se falha como uma indisponibilidade completa do componente em questão da arquitetura proposta. A Figura 3.6 mostra um esquemático do processo de classificação com falhas. O "X" de cor vermelha representa uma falha em um servidor Discriminador e o "X" roxo representa falhas em servidores RAM.

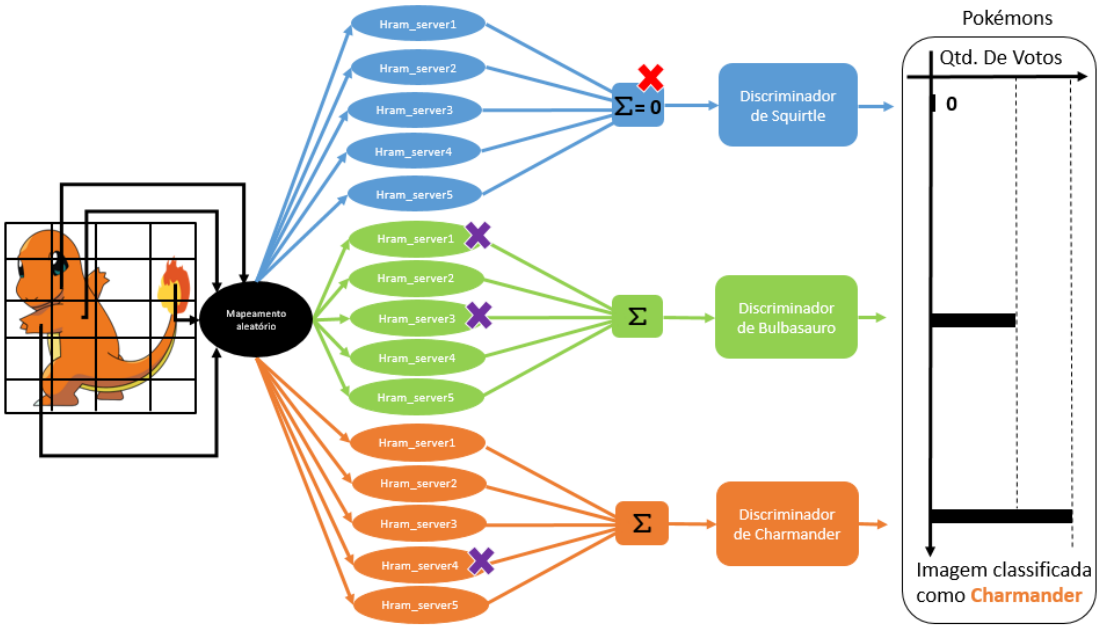

Figura 3.6: Fluxo de uma classificação com falhas

3.5.1 Cenário de testes I: Treinamento Federado

A arquitetura proposta é capaz de utilizar o treinamento colaborativo apresentado na Seção 3.2.1. Para comprovar a eficácia desse paradigma foram considerados três clientes e dados independentes e identicamente distribuídos (IID), ou seja, cada *dataset* utilizado para o treinamento foi dividido em três conjuntos disjuntos. A Figura 3.7 mostra a arquitetura proposta para o de treinamento federado de Pokémons.

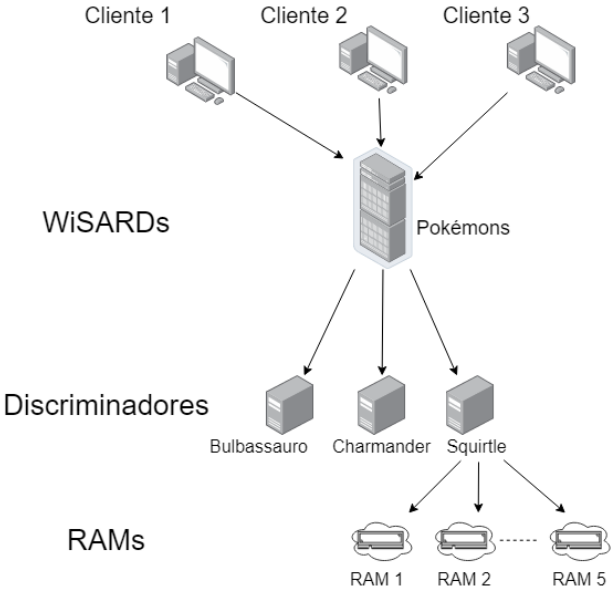

Figura 3.7: Arquitetura de testes para treinamento fedarado - Pokémons

Primeiro, cada conjunto foi treinado sequencialmente por um cliente diferente e, ao final de

cada treino, um processo de classificação foi executado para avaliar os impactos da colaboração e validar se depois do terceiro treino os resultados são semelhantes à um treino único com o *dataset* completo.

A fim de validar que o modelo não necessariamente irá melhorar, caso os dados não sejam independentes e identicamente distribuídos (no-IID), foi realizado um quarto processo de treino com imagens que ainda não tinham sido utilizadas. Nesse processo, uma categoria de cada *dataset* foi priorizada em detrimento das outras, ou seja, a quantidade de imagens desta categoria era muito maior, conforme a distribuição apresentada na Tabela 3.3

Tabela 3.3: Quantidade de imagens para cada categoria do *dataset*

	Categoria 1	**Categoria 2**	**Categoria 3**	**Categoria 4**
Simpsons	50	2	2	1
Pokémons	90	2	3	-
Máscaras	95	5	-	-

Depois, para avaliar o desempenho de um treinamento em paralelo, o *dataset* foi novamente dividido em três partes disjuntas e o treino foi executado por três clientes simultaneamente. A métrica avaliada aqui foi o tempo total de treino do modelo.

3.5.2 Cenário de testes II: Indisponibilidade de servidores Discriminador

Conforme descrito na sessão 3.1, um servidor de Discriminador é instanciado para cada categoria que uma WiSARD é capaz de identificar. Portanto, espera-se que mesmo que um ou mais servidores Discriminador fiquem indisponíveis, a rede ainda consiga classificar as demais categorias.

A Tabela 3.4 mostra os testes executados para validar essa proposição:

Tabela 3.4: Simulações de falha para servidores de Discriminador

	Simpsons	**Pokémons**	**Máscaras**
Servidores disponíveis (Início)	4 servidores	3 servidores	2 servidores
Servidores disponíveis (1ª execução)	3 servidores	2 servidores	1 servidor(máscaras)
Servidores disponíveis (2ª execução)	2 servidores	1 servidor	1 servidor(sem máscaras)
Servidores disponíveis (3ª execução)	1 servidor	-	-

Não foram realizados testes com todos os servidores indisponíveis, pois esse cenário consiste em uma falha de 100% de acesso a rede, caracterizando uma perda total e não parcial.

3.5.3 Cenário de testes III e IV: Indisponibilidade de servidores RAM

Sabe-se que cada Discriminador é conectado a um numero arbitrário de servidores de RAM, onde cada um desses servidores executa uma parte do treino.

Foram considerados 5 servidores RAM para cada Discriminador, de forma que a rede suporte uma quantidade razoável de falhas. Para determinar o quanto a indisponibilidade de cada RAM impacta na acurácia e também no tempo de execução, foram realizados testes variando as taxas de falha entre 20% e 80%, conforme mostra a Tabela 3.5.

É importante ressaltar que um *dataset* com 4 categorias, teria 4 servidores Discriminador e consequentemente 20 servidores de RAM no total. Já um *dataset* com 2 categorias, teria um total de 10 servidores de RAM.

Tabela 3.5: Simulações de falha para servidores RAM

Taxa de falha	**Simpsons**	**Pokémons**	**Máscaras**
0%	20 servidores RAM	15 servidores RAM	10 servidores RAM
20%	16 servidores RAM	12 servidores RAM	8 servidores RAM
40%	12 servidores RAM	9 servidores RAM	6 servidores RAM
60%	8 servidores RAM	6 servidores RAM	4 servidores RAM
80%	4 servidores RAM	3 servidores RAM	2 servidores RAM

Os testes da Tabela 3.5 foram realizados de duas formas, primeiro considerando um balanceamento na queda, ou seja, para o *Dataset* dos Simpsons uma taxa de falha = 20% significaria a queda de um servidor de RAM de cada Discriminador.

Como esse cenário está distante da realidade, foi realizado uma segunda bateria de testes considerando quedas aleatórias e não necessariamente balanceadas, respeitando a quantidade total de servidores RAM.

Capítulo 4

Análise de Resultados

O projeto analisa a execução de uma rede neural sem pesos descentralizada em diversos cenários de falha, considerando os efeitos que cada uma delas impõe na acurácia do modelo. A Seção 4.2 compara os comportamentos obtidos com o treinamento padrão e o treinamento federado. As seções 4.3 à 4.5 observam os resultados gerados em cada uma das execuções com os diferentes parâmetros utilizados. Finalmente, a Seção 4.6 compara os resultados observados entre as execuções.

4.1 Treinamento e classificação na arquitetura distribuída

Para estabelecer um *benchmark* do modelo proposto, uma bateria de dez treinos e dez classificações para cada WiSARD mostrada na Figura 3.5, foi executada. A Tabela 4.1 mostra a acurácia média dos testes. Os resultados apresentados serão usados para avaliar o impacto dos cenários de teste apresentados.

Tabela 4.1: Acurácia com Treinamento padrão

	Máscaras	**Simpsons**	**Pokémons**
Acurácia	78%	74%	68%

4.2 Cenário de testes I: Treinamento Federado

A fim de avaliar o comportamento do treinamento federado, para cada WiSARD, foram realizados três treinamentos separados. Em cada uma dessas execuções, o modelo foi treinado por um cliente diferente (Figura 3.7).

Cada um desses clientes treinou o modelo com uma parte diferente do conjunto de dados. Para tal, cada *dataset* foi dividido em três conjuntos, de mesmo tamanho e distintos. Por exemplo, no caso dos Simpsons, cada conjunto continha 56 imagens, uma vez que o *dataset* completo possuía 168 imagens. Assim, após cada cliente treinar o modelo, uma inferência com todo conjunto de dados de teste foi executada. O resultado pode ser conferido na Tabela 4.2

Tabela 4.2: Acurácia com Treinamento Federado

	Máscaras	**Simpsons**	**Pokémons**
Acurácia - 1° Treino	51%	48%	45%
Acurácia - 2° Treino	59%	56%	55%
Acurácia - 3° Treino	78%	75%	68%

Observando os resultados, pode-se perceber que o treinamento federado está em pleno funcionamento, visto que a acurácia da rede melhorou na medida em que os treinos foram executados, pois mais clientes forneceram dados para o modelo. No último treino, com três clientes fornecendo o mesmo volume de dados do treinamento de *benchmark*, a acurácia geral da rede foi semelhante.

Este resultado mostra o poder que o treinamento colaborativo possui de melhorar o modelo. No entanto, este cenário de conjuntos de dados disjuntos não representa, de fato, um treinamento no mundo real. Por esse motivo, para cada *dataset*, mais um treino foi executado onde um 4º cliente hipotético treinou o modelo usando uma quantidade de imagens extremamente desbalanceada. No caso do *dataset* Simpsons a rede foi treinada novamente com 50 imagens do personagem Homer; Do *dataset* Pokémons com 90 do personagem Charmander, e no caso do *dataset* mascaras com 95 imagens de pessoas usando máscaras. Para as demais categorias foram utilizadas no máximo 5 imagens. A Figura 4.1 ilustra o resultado desse cenário para a WiSARD dos Simpsons.

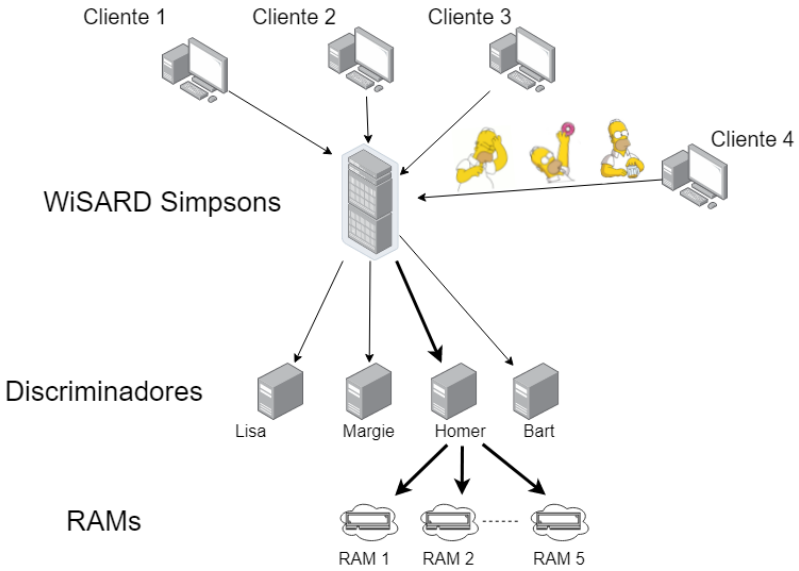

Figura 4.1: Exemplo de 4º Treino federado - Treinamento exclusivamente da categoria Homer

Esse novo treino esclareceu uma das desvantagens do treinamento federado. Como um dos Discriminadores foi treinando com muito mais imagens que os outros, o modelo ficou enviesado e apresentou uma piora de acurácia geral. Isso ocorreu pois, o Discriminador que recebeu mais dados influenciou na classificação de todas as outras classes, já que, durante o processo de treinamento, as memórias RAM desse Discriminador foram acessadas mais vezes. Em todos os casos a acurácia geral da rede piorou conforme mostra a Tabela 4.3

Tabela 4.3: Acurácia com o 4º treino colaborativo

	Máscaras	**Simpsons**	**Pokémons**
Acurácia - 4º treino	56%	38%	40%

A matriz de confusão referente a classificação do *dataset* dos Simpsons após o 4º treino, apresentada na imagem 4.2, ilustra claramente essa interferência.

	Bart	Homer	Lisa	Margie
Bart	14,3%	85,7%	0,0%	0,0%
Homer	0,0%	100,0%	0,0%	0,0%
Lisa	3,1%	81,3%	15,6%	0,0%
Margie	0,0%	71,8%	0,0%	28,2%

Figura 4.2: Matriz de confusão dos Simpsons - 4º treino

Constata-se que a acurácia do Discriminador Homer alcançou 100% de acerto, isso acontece porque esse Discriminador foi treinado com muito mais imagens que seus concorrentes. Dessa forma, a influência desse Discriminador começa a afetar negativamente os demais. É possível notar que 87,5% das imagens do Bart foram classificadas como Homer e que essa interferência exagerada também ocorreu nos Discriminadores da Lisa e da Margie. Assim, apesar da acurácia do Discriminador do Homer ter aumentado, o desempenho geral da rede diminuiu.

Esse resultado mostra que nem sempre o modelo será melhorado com mais informações de entrada, pode haver uma piora significativa de acurácia se os dados forem extremamente desbalanceados.

Por fim, o desempenho da rede foi avaliado quanto ao ganho de performance resultante da execução de treinamento colaborativo em paralelo. Novamente o *dataset* dos simpsons foi dividido em três conjuntos disjuntos. Depois, esses conjuntos foram usados por três clientes para treinar a rede simultaneamente. A Figura 4.3 mostra uma comparação entre o tempo total de treino do treinamento colaborativo em paralelo e o tempo total de treino do treinamento comum onde apenas um cliente treina a rede com o *dataset* completo.

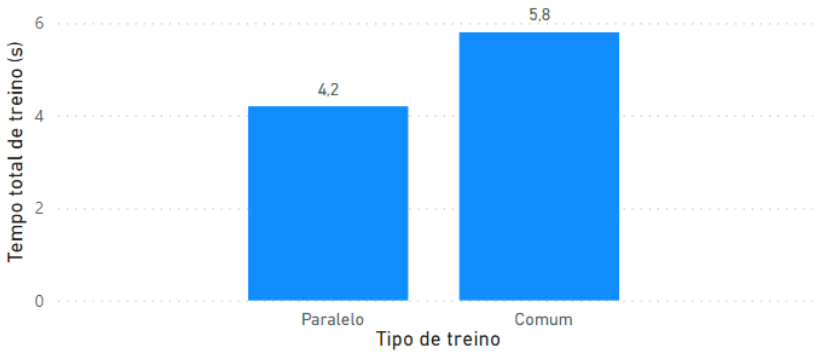

Figura 4.3: Tempo total de treino - cenário colaborativo em paralelo e único cliente

Fica evidente que o treino colaborativo traz ganhos de performance ao modelo. A acurácia continuou a mesma, e o tempo total de treino foi aproximadamente 27% menor.

4.3 Cenário de testes II: Indisponibilidade de servidores Discriminador

Neste cenário, foram analisados os impactos da perda de um servidor Discriminador. Conforme ilustrado na Figura 3.6, a indisponibilidade do servidor Discriminador da categoria Squirtle, faz com que ele não contribua na votação, pois a quantidade de votos enviada ao servidor WiSARD é igual a zero, portanto a rede não classificará nenhuma imagem como o sendo pertencente à essa categoria.

Para o *dataset* dos Simpsons, que possui quatro Discriminadores, foram desligados sequencialmente os servidores da Margie, Homer e Lisa e a acurácia da rede variou conforme o gráfico apresentado na Figura 4.4

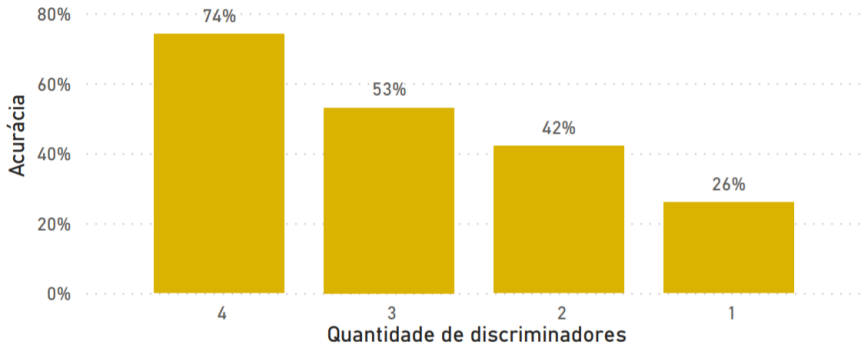

Figura 4.4: Variação da acurácia de acordo com a quantidade de Discriminadores para o *dataset* dos Simpsons

Para o *dataset* dos Pokémons, que possui três Discriminadores, foram desligados sequencialmente os servidores do Squirtle e do Bulbasauro. A acurácia da rede variou conforme o gráfico apresentado na Figura 4.5

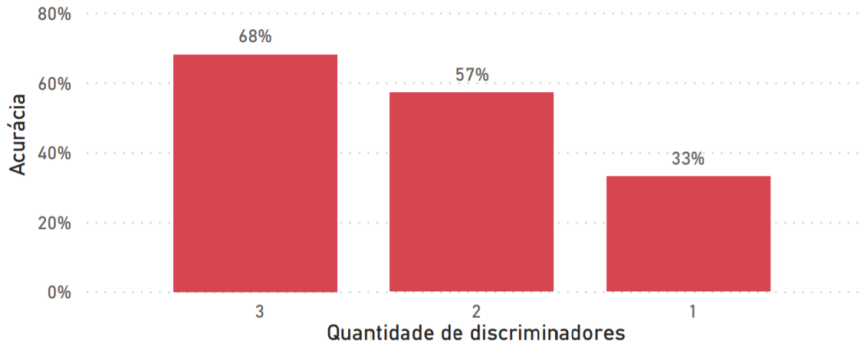

Figura 4.5: Variação da acurácia de acordo com a quantidade de Discriminadores para o *dataset* de Pokémons

Para o *dataset* de Máscaras, que possui dois Discriminadores, foi desligado o servidor de rostos com máscara e, em um segundo teste foi feita a configuração inversa, deixando o servidor de rostos com máscara ligado e o de rostos sem máscara desligado. O resultado obtido foi o mesmo para ambos os casos e a variação de acurácia está apresentada naFigura 4.6

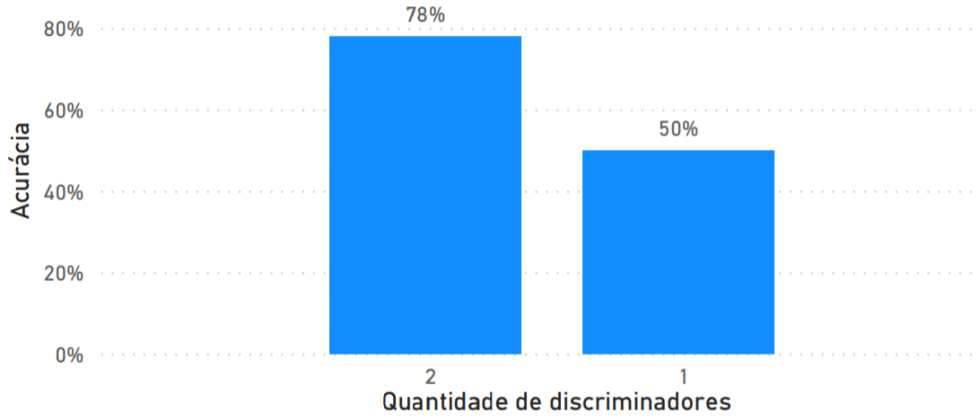

Figura 4.6: Variação da acurácia de acordo com a quantidade de Discriminadores para o *dataset* de máscaras

4.4 Cenário de testes III: Indisponibilidade balanceada de servidores RAM

Agora, foram analisados os impactos da perda de servidores RAM, em um cenário ideal com perdas igualmente distribuídas entre as categorias. Observando a Figura 3.6 é possível notar que a indisponibilidade de servidores RAM afeta a rede em menor proporção do que um cenário de falha de um Discriminador inteiro, uma vez que, todos os Discriminadores ainda são capazes de contribuir na votação, mesmo que alguns tenham mais influência que outros.

Os resultados foram coletados com base em uma queda progressiva no número de servidores disponíveis, como descrito na Seção 3.5.3. Essa indisponibilidade foi testada até o limite para que a rede continuasse funcionando minimamente.

Na Figura 4.7 pode-se perceber que mesmo com uma indisponibilidade de 80% dos servidores RAM, a queda na acurácia é baixa. Esse resultado se deve a perda de informações de forma identicamente distribuída entre os Discriminadores.

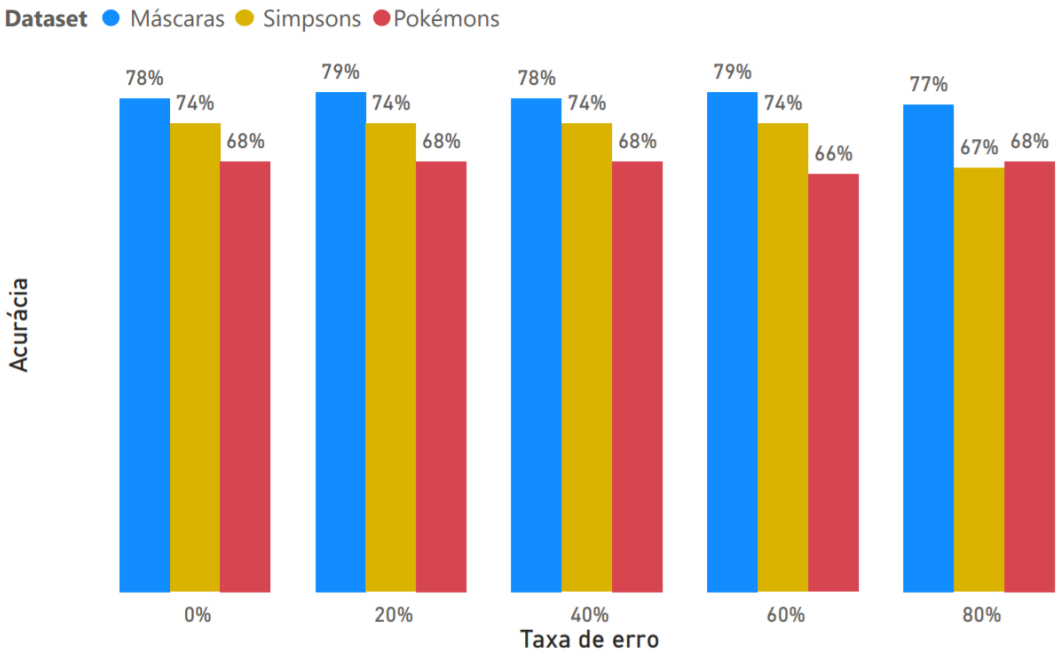

Figura 4.7: Acurácia vs Taxa de Erro para falhas balanceadas

Utilizando a imagem 4.8 como referência, uma taxa de erro de 40% significar eliminar dois servidores RAM de cada Discriminador. A partir dessa visualização fica claro que ao final da votação não há nenhum Discriminador com mais influência que o outro, em virtude das falhas, portando o comportamento da rede é mantido.

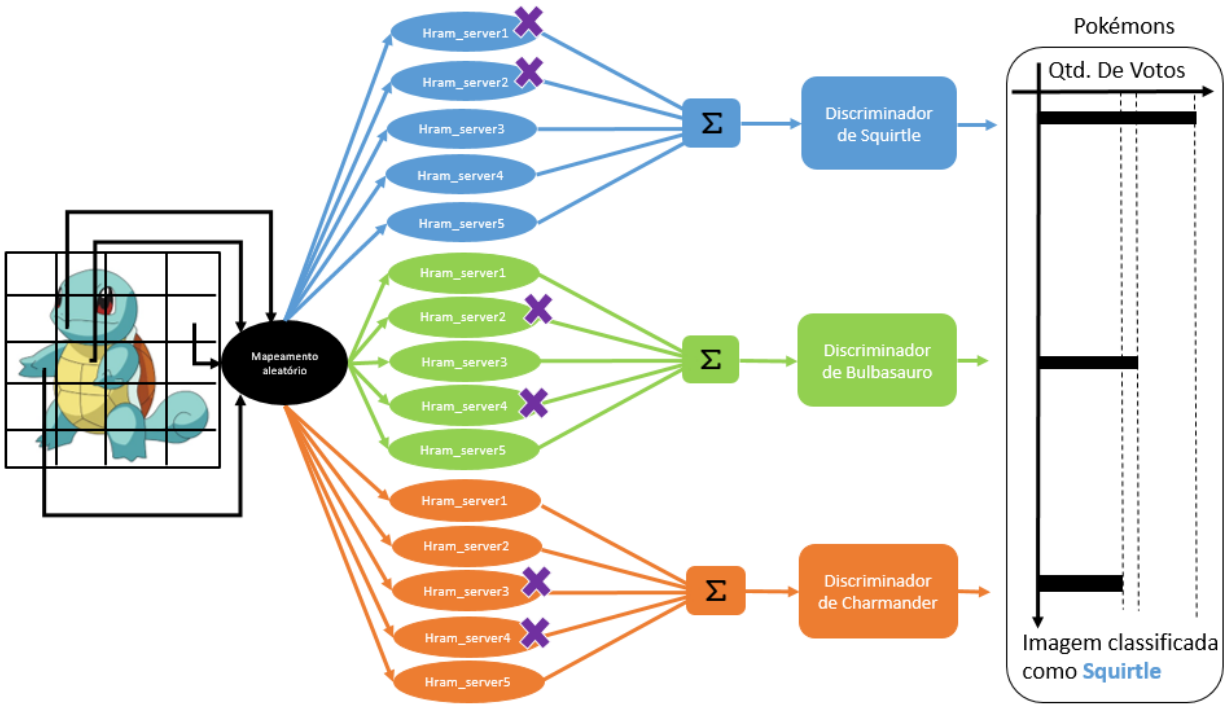

Figura 4.8: Fluxo de uma classificação com falhas de servidores RAM

4.5 Cenário de testes IV: Indisponibilidade aleatória de servidores RAM

Neste caso, foram analisados os impactos da falha de servidores RAM, em um cenário mais próximo da realidade com indisponibilidades aleatórias, ou seja, não necessariamente balanceadas, como ilustrado pelos "X"roxos da imagem 3.6.

Os resultados, assim como no Cenário de testes II, foram coletados com base em uma queda progressiva no número de servidores disponíveis. No entanto, diferente do comportamento obtido na execução anterior, agora a rede sofreu uma redução quase linear na acurácia a medida que a taxa de erro aumentou. A Figura 4.9 mostra esse resultado.

A probabilidade dos servidores indisponíveis de forma aleatória ser identicamente distribuída é muito baixa, para os *datasets* dos Simpsons e dos Pokémons está próxima de 0,3% e piora a medida que o número de classes aumenta. Essa probabilidade justifica o comportamento da rede, dado que, ao final da votação, alguns Discriminadores terão mais servidores RAM ativos e por consequência terão uma contribuição maior de votos.

A distribuição aleatória da indisponibilidade de servidores RAM leva a rede a encontrar uma quantidade de falsos positivos maior, pois a classificação irá preterir algumas categorias. Esse comportamento é semelhante ao descrito na Seção 4.2, onde um Discriminador é mais treinado que o outro.

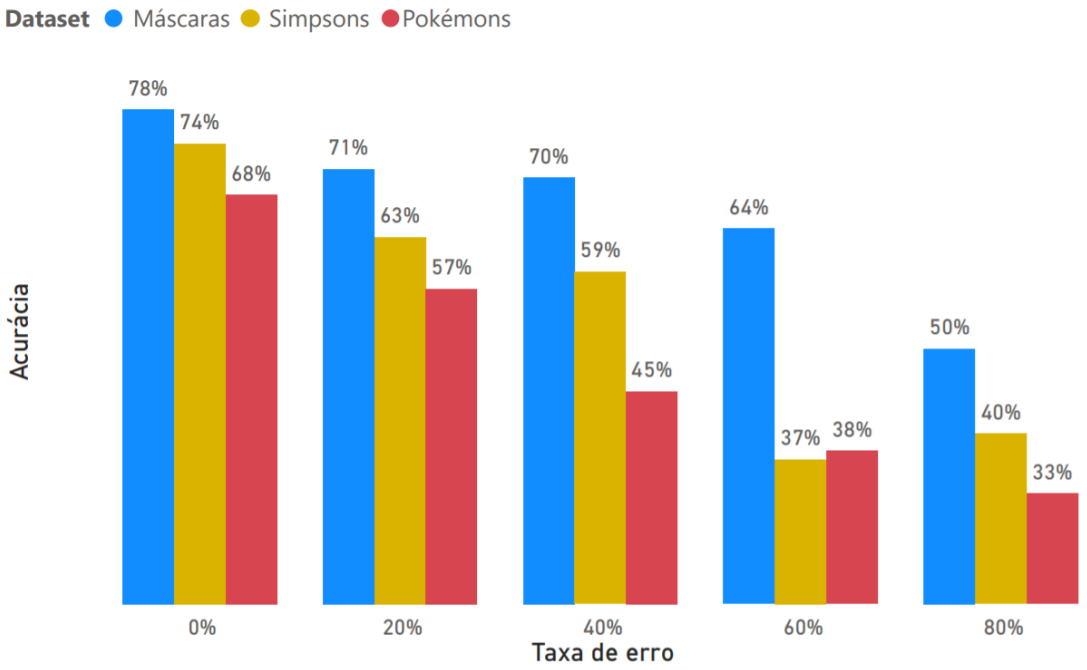

Figura 4.9: Taxa de Erro vs. Acurácia para quedas aleatórias

4.6 Comparação de resultados

Para comparar os resultados obtidos em cada uma das execuções, foi medida não só acurácia total, mas também a acurácia individual de cada categoria por meio da matriz de confusão.

Como referência para a comparação será utilizada a matriz de confusão do *dataset* dos Simpsons, a Figura 4.10 mostra os resultados para a rede com 0% de indisponibilidade.

É possível observar nessa matriz que a classe com menos acurácia é a da personagem Lisa, que teve 15,6% das suas imagens classificadas como Bart; 15,6%, como Homer; 9,4%, como Margie e; somente 59,4% classificadas corretamente como Lisa.

	Bart	Homer	Lisa	Margie
Bart	85,7%	14,3%	0,0%	0,0%
Homer	12,9%	67,7%	9,7%	9,7%
Lisa	15,6%	15,6%	59,4%	9,4%
Margie	5,1%	7,7%	7,7%	79,5%

Figura 4.10: Matriz de confusão do *dataset* Simpsons para uma taxa de erro= 0%

Para ilustrar a matriz de confusão do Cenário de testes I, a Figura 4.11 traz os resultados referentes a três servidores de Discriminador disponíveis e um servidor indisponível, sendo esse o da Margie.

A Figura 4.11, mostra claramente o efeito da perda de um Discriminador, é possível observar que a coluna da matriz referente a essa categoria está zerada, ou seja, nenhuma imagem de todo o *dataset* foi classificada como Margie. Esse comportamento não afeta apenas a acurácia dessa categoria, visto que as imagens referentes a esse personagem serão agora classificadas como outros personagens de acordo com a votação dos Discriminadores disponíveis. É possível observar essa interferência principalmente nas colunas do Homer e da Lisa.

	Bart	Homer	Lisa	Margie
Bart	85,7%	11,4%	2,9%	0%
Homer	16,1%	80,7%	3,2%	0%
Lisa	21,9%	25,0%	53,1%	0%
Margie	10,3%	53,9%	35,9%	0%

Figura 4.11: Matriz de confusão do *dataset* Simpsons para um Discriminador indisponível

Então, no pior caso ainda funcional, quando existir apenas um servidor de Discriminador, todas as imagens serão classificadas como personagem restante. No caso do Simpsons, essa acurácia equivale a: $\frac{1}{4} = 25\%$, já para os Pokémons, no pior caso, a acurácia vale 33%. Como no *dataset* de máscaras há apenas duas possibilidades, o pior caso é 50%, independente de qual Discriminador está disponível.

A Figura 4.12 mostra a matriz de confusão referente ao Cenário de testes III, com uma taxa de falhas de 20%. Para obter essa taxa foram desligados dois servidores RAM pertencentes ao Discriminador de Homer e mais dois pertencentes a Margie.

	Bart	Homer	Lisa	Margie
Bart	94,3%	2,9%	2,9%	0,0%
Homer	51,6%	29,0%	16,1%	3,2%
Lisa	31,3%	0,0%	65,6%	3,1%
Margie	15,4%	2,6%	23,1%	59,0%

Figura 4.12: Matriz de confusão do *dataset* Simpsons para uma taxa de erro= 20%

Com essa Figura é possível perceber como a acurácia do Homer diminuiu ao perder dois de seus servidores RAM, e como essa perda influenciou no aumento da acurácia de outros Discriminadores. Por exemplo, a categoria Bart que tinha 14,3% das suas imagens classificadas erroneamente como Homer no cenário sem falhas, passou a ter apenas 2,9% após a indisponibilidade dos servidores RAM.

O mesmo pode ser observado para a Margie que teve sua acurácia reduzida de 79,5% para 59,0%. Sobre a sua influência pode-se notar que inicialmente 9,4% das imagens da Lisa foram classificadas como Margie, essa porcentagem caiu para 3,1% quando os dois servidores RAM da Margie foram desligados.

Portanto, observando as três execuções, constata-se pelo Cenário de testes I que a perda de um servidor Discriminador é muito impactante na acurácia geral do modelo, visto que um Discriminador indisponível não contribuirá na votação e nenhuma imagem pertencente a esta categoria será classificada corretamente. Já a perda de servidores RAM, dependendo da quantidade, permite que a rede mantenha uma acurácia razoável. A Cenário de testes II é a que sustenta a maior tolerância a falhas sem comprometer a acurácia, porém é um cenário ideal, muito difícil de ser alcançado na realidade.

Capítulo 5

Conclusão

Nesse trabalho propôs-se uma arquitetura descentralizada de Redes Neurais Artificiais sem Peso, visando o aumento da eficiência em ambientes de treinamento federado e a tolerância a falhas em casos de indisponibilidade dos servidores.

Apesar de ser possível observar a existência da correlação entre o treinamento federado e a melhora geral do modelo da rede, os resultados observados no capítulo 4 indicam algumas ressalvas. Caso os clientes treinem a rede utilizando dados no-IDD, a acurácia da rede pode sofrer um impacto negativo. Porém, o presente trabalho mostra que para uma piora significativa do modelo os dados fornecidos pelos clientes devem ser extremamente desbalanceados.

O capítulo 4 também mostra as vantagens da arquitetura proposta. Com a execução de um treinamento colaborativo em paralelo, o tempo total de execução foi até 27% menor que no caso da execução de um único treino.

Também possível observar que a tolerância às falhas promovida pela implementação de um modelo descentralizado resultou em uma maior robustez da rede. Esse ganho em robustez permitiu que o sistema continuasse funcionando, mesmo com a indisponibilidade de grande parte dos servidores do sistema. As falhas impactaram diretamente na acurácia, porém o modelo foi capaz de continuar a classificação de imagens com a rede funcionando parcialmente.

5.1 Trabalhos futuros

O projeto proposto possui limitações que podem ser exploradas em trabalhos futuros, como:

1. Ampliação do escopo de testes: expandir a quantidade de servidores RAM para ambientes em larga escala, com o intuito de observar se os ganhos mostrados no presente trabalho acompanham o crescimento do tamanho do ambiente;
2. Ampliação do escopo de clientes: expandir a quantidade de clientes no treinamento federado para acompanhar o processo de evolução da rede em um ambiente mais diversificado;

3. Segurança no treinamento federado: implementar camadas de segurança para proteger dados sensíveis e impedir colaboradores maliciosos, com criptografia de dados, por exemplo.

Referências

ALEKSANDER, Igor et al. A brief introduction to weightless neural systems. In:

ARAÚJO, Leandro Almeida de. RWISARD: UM MODELO DE REDE NEURAL SEM PESO PARA RECONHECIMENTO E CLASSIFICAÇÃO DE IMAGENS EM ESCALA DE CINZA, 2011.

ATTIA, alex. **The Simpsons Characters Data**. 2018. Disponível em: <https://www.kaggle.com/alexattia/the-simpsons-characters-dataset>. (acessado em: 17.04.2021).

BANDEIRA, Lawrence Cruvinel; FRANÇAA, FMG. **NC-WISARD: Uma interpretação sem pesos do modelo neural neocognitron**. 2010. Tese (Doutorado) – M. Sc. thesis, Rio de Janeiro, RJ, Brasil.

BLEDSOE, Woodrow Wilson; BROWNING, Iben. Pattern recognition and reading by machine. In: PAPERS presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference. [S.l.: s.n.], 1959. p. 225–232.

BOMARIUS, Frank. A multi-agent approach towards modeling urban traffic scenarios, 1992.

CHAIB-DRAA, Brahim. Industrial applications of distributed AI. **Communications of the ACM**, ACM New York, NY, USA, v. 38, n. 11, p. 49–53, 1995.

DANNY POO DEREK KIONG, Swarnalatha Ashok. **Object-Oriented Programming and Java**. 2nd. [S.l.]: Springer, 2007.

DELUA, Julianna. **Supervised vs. Unsupervised Learning: What's the Difference?** 2021. Disponível em: <https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning>. (acessado em: 04.04.2021).

DWIVEDI, Harshit. **Pokemon Generation One**. 2018. Disponível em: <https://www.kaggle.com/thedagger/pokemon-generation-one>. (acessado em: 04.04.2021).

EDUCATION, IBM Cloud. **Artificial Intelligence (AI)**. 2021. Disponível em: <https://www.ibm.com/cloud/learn/what-is-artificial-intelligence>. (acessado em: 04.04.2021).

GHAHRAMANI, Zoubin. **Unsupervised Learning**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 72–112.

GILLHAM, Jonathan et al. The macroeconomic impact of artificial intelligence. **PwC Report.–PricewaterhouseCoopers.–2018**, 2018.

GREGORIO, Massimo De; GIORDANO, Maurizio. The WiSARD Classifier, 2016.

GURAV, OMKAR. **Face Mask Detection Dataset**. 2020. Disponível em: <https://www.kaggle.com/omkargurav/face-mask-dataset/metadata>. (acessado em: 04.04.2021).

IBM. **A Computer Called Watson**. 2021. Disponível em: <https://www.ibm.com/ibm/history/ibm100/us/en/icons/watson/>. (acessado em: 25.04.2021).

JONG, Irme. **Servers: hosting Pyro objects**. 2021. Disponível em: <https://pyro4.readthedocs.io/en/stable/servercode.html?highlight=instance_mode#controlling-instance-modes-and-instance-creation>. (acessado em: 26.04.2021).

JONG, Irme de. **Pyro: Intro and Example**. 2021. Disponível em: <https://pyro4.readthedocs.io/en/stable/intro.html#>. (acessado em: 25.04.2021).

JR., Thomas Barnett. **The Zettabyte Era Officially Begins**. 2021. Disponível em: <https://blogs.cisco.com/sp/the-zettabyte-era-officially-begins-how-much-is-that>. (acessado em: 04.04.2021).

LUDERMIR, Teresa et al. Weightless neural models: A review of current and past works. **Neural Computing Surveys**, v. 2, 1999.

LUSQUINO FILHO, Leopoldo André Dutra. Classificação de emoções faciais utilizando a rede neural sem pesos WiSARD. Universidade Federal do Rio de Janeiro, 2018.

MACHADO, Tarso Mesquita et al. Projeto dedicado de redes neurais sem peso baseadas em neurônios de lógica probabilística multi-valorada. Universidade do Estado do Rio de Janeiro, 2017.

MCCULLOCH, Warren S; PITTS, Walter. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, Springer, v. 5, n. 4, p. 115–133, 1943.

MCMAHAN, Brendan et al. Communication-efficient learning of deep networks from decentralized data. In: PMLR. ARTIFICIAL Intelligence and Statistics. [S.l.: s.n.], 2017. p. 1273–1282.

MICHAEL E. PORTER, James E. Heppelmann. **On AI, Analytics, and the New Machine Age**. Paperback. [S.l.]: Harvard Business Review Press, 2019. (HBR's 10 Must Reads). ISBN 1633696847,9781633696846. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=412dd921d509dd99a6bb58707976dbc7>.

MOHRI, Mehryar; ROSTAMIZADEH, Afshin; TALWALKAR, Ameet. **Foundations of machine learning**. [S.l.]: MIT press, 2018.

NG, Andrew. **Machine Learning**. 2021. Disponível em: <http://mlclass.stanford.edu/>. (acessado em: 16.04.2021).

PALACH, Jan. **Parallel Programming with Python**. First. [S.l.]: Packt Publishing, 2014.

PITCHFORD, Daniel. **A Decade Of Advancements As We Enter A New Age Of AI**. 2021. Disponível em: <https://www.forbes.com/sites/danielpitchford/2020/12/31/a-decade-of-advancements-as-we-enter-a-new-age-of-ai/?sh=4ad97c1c4055>. (acessado em: 04.04.2021).

PRESS, Gil. **A Very Short History Of Artificial Intelligence (AI)**. 2021. Disponível em: <https://www.forbes.com/sites/gilpress/2016/12/30/a-very-short-history-of-artificial-intelligence-ai/?sh=3bff6c876fba>. (acessado em: 12.04.2021).

QIANG YANG LIXIN FAN, Han Yu. **Federated Learning: Privacy and Incentive**. 1st ed. [S.l.]: Springer International Publishing;Springer, 2020. (Lecture Notes in Computer Science 12500).

ROSER, Max; RITCHIE, Hannah. Technological Progress. **Our World in Data**, 2013. https://ourworldindata.org/technological-progress.

SEER. **Anatomy Physiology**. 2021. Disponível em: <https://training.seer.cancer.gov/anatomy/>. (acessado em: 15.04.2021).

SILVA, I.N. da et al. **Artificial Neural Networks: A Practical Course**. 1. ed. [S.l.]: Springer International Publishing, 2017.

SOUZA, Francisco Ary Alves de. **Análise de desempenho da rede neural artificial do tipo multilayer perceptron na era multicore**. 2012. Diss. (Mestrado) – Universidade Federal do Rio Grande do Norte.

SUGAWARA, T. A cooperative LAN diagnostic and observation expert system. IEEE Computer Society, Los Alamitos, CA, USA, 1990. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/PCCC.1990.101684>.

SZCZEPANSKI, M. Economic impacts of artificial intelligence (AI). **European Parliamentary Research Service (PE 637.967)**, 2019.

TAIGMAN, Yaniv 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.

TANENBAUM, Andrew S. **Sistemas Distribuídos - Princípios e Paradigmas**. 2. ed. [S.l.]: Pearson, 2007.

TERRA, John. **10 Years of Artificial Intelligence and Machine Learning**. 2021. Disponível em: <https://www.simplilearn.com/ten-years-of-artificial-intelligence-and-machine-learning-article>. (acessado em: 13.04.2021).

XIONG, Wayne et al. The Microsoft 2017 conversational speech recognition system. In: IEEE. 2018 IEEE international conference on acoustics, speech and signal processing (ICASSP). [S.l.: s.n.], 2018. p. 5934–5938.

ZHANG, Daniel et al. The AI Index 2021 Annual Report. **arXiv preprint arXiv:2103.06312**, 2021.

ZHANG, Xian-Da. **A Matrix Algebra Approach to Artificial Intelligence**. 1. ed. [S.l.]: Springer, 2020.

ZHANG, Zhongheng. A gentle introduction to artificial neural networks. **Annals of Translational Medicine**, v. 4, p. 370–370, out. 2016.

ANEXOS

ANEXO I

Código da Rede Neural

Esse anexo apresenta os códigos utilizados para configuração, treino e classificação da rede neural.

I.1 Código do servidor WiSARD

```
import itertools as ito
from operator import itemgetter
import numpy as np
import random
import pickle
import shelve
import math
import Pyro4
Pyro4.config.SERIALIZERS_ACCEPTED = set(['pickle'])
Pyro4.config.SERIALIZER = 'pickle'

@Pyro4.expose
@Pyro4.behavior(instance_mode="single")
class Wisard:
    def __init__(self):
        return

    def instantiate(self, num_of_htables, input_addr_length, discriminador,
    qtd_hram_server):
        self.qtd_hram_server = qtd_hram_server
        self.discs = {}
        self.discriminador = discriminador
        if input_addr_length < 11:
            self.input_addr_length = input_addr_length
        else:
            self.input_addr_length = 11

        self.num_of_htables = num_of_htables
        self.d_votes = {}
```

```
29          self.d_times_trained = {}
30          self.d_relevance = {}
31          self.mapping = list(range(0, self.num_of_htables*self.input_addr_length))
32          random.shuffle(self.mapping)
33          self.last_rank = 0
34          self.rank_table = {}
35          self.down_discs = []
36
37      def ranks(self,tmp):
38          addresses = []
39          for i in range(0, len(tmp), self.input_addr_length):
40              tuples = sorted(list(zip(tmp[i:i+self.input_addr_length],list(range(0,
    self.input_addr_length)))))
41              _,t = zip(*tuples)
42              if str(t) not in self.rank_table.keys():
43                  self.rank_table[str(t)] = self.last_rank
44                  self.last_rank += 1
45              addresses.append(self.rank_table[str(t)])
46          return addresses
47
48      def train(self, X, label):
49          if label not in self.discs:
50              self.discs[label] =  Pyro4.Proxy("PYROMETA:"+'discriminator'+','+ self.
    discriminador+','+label)
51              self.discs[label].instantiate(num_of_htables = self.num_of_htables,
    qtd_hram_server = self.qtd_hram_server,label=label)
52            #self.d_votes[label] = 0
53              self.d_times_trained[label] = 0
54              self.d_relevance[label] = 0
55
56          tmp = X[self.mapping]
57          addresses = self.ranks(tmp)
58          self.discs[label].train(addresses)
59
60      def classify(self, X, total_trained=0):
61          tmp = X[self.mapping]
62          biggest = -1
63          secon_biggest = -1
64          label = -1
65          d_label = None
66          addresses = self.ranks(tmp)
67          b = 0
68          for label in self.discs.keys():
69              if label in self.down_discs:
70                  self.d_votes[label]=0
71              else:
72                  try:
73                      self.d_votes[label] = self.discs[label].classify(addresses, b)
74                  except Exception as e:
75                      self.down_discs.append(label)
76                      print(e)
77                      self.d_votes[label]=0
```

```
78
79          votes = list(sorted(self.d_votes.values()))
80          label = max(self.d_votes, key=self.d_votes.get)
81
82          biggest = votes[-1]
83          secon_biggest = votes[-2]
84
85          conf = 0
86          if biggest > 0:
87              conf = (biggest - secon_biggest)/biggest
88
89          return (label, biggest/self.num_of_htables, conf)
90
91  def main():
92      daemon = Pyro4.Daemon(host="192.168.56.1")
93      ns = Pyro4.locateNS()
94      print(ns)
95      uri = daemon.register(Wisard)
96      ns.register("wisard_server1", uri, metadata={"wisard","simpsons"})
97      daemon.requestLoop()
98
99  if __name__=="__main__":
100     main()
```

I.2 Código do servidor de Discriminador

```
1  import itertools as ito
2  from operator import itemgetter
3  import numpy as np
4  import random
5  import pickle
6  import shelve
7  import math
8  import Pyro4
9  Pyro4.config.SERIALIZERS_ACCEPTED = set(['pickle'])
10 Pyro4.config.SERIALIZER = 'pickle'
11
12 @Pyro4.expose
13 @Pyro4.behavior(instance_mode="single")
14 class Discriminator:
15     def __init__(self, num_of_htables=28):
16         return
17
18     def instantiate(self, num_of_htables, qtd_hram_server,label):
19         self.times_trained = 0
20         self.num_of_htables = num_of_htables
21         self.qtd_hram_server = qtd_hram_server #quantidade de servidores de h_ram
       POR CLASSE
22         self.dict_hrams_servers = {} #armazena os servidores de h_ram
23         self.votes =0
```

```
24          self.down_h_rams = []
25          for x in range(self.qtd_hram_server):
26              self.dict_hrams_servers[x] = Pyro4.Proxy("PYROMETA:"+"hram"+","+str(x)+
    ","+label) #busca os servidores de h_ram de cada classe
27              self.dict_hrams_servers[x].instantiate(int((self.num_of_htables/self.
    qtd_hram_server)))
28
29      def train(self,addresses):
30          addresses_split = np.array_split(addresses, self.qtd_hram_server) #divide
    os endere os pela quantidade de servidores de h_ram de cada classe
31          for x in range(self.qtd_hram_server):
32              self.dict_hrams_servers[x].train(addresses_split[x]) #treina em cada
    pedaco
33
34      def classify(self,X,bleach=0): #no bleaching
35          self.votos = 0
36          addresses_split = np.array_split(X, self.qtd_hram_server)
37          for x in range(self.qtd_hram_server):
38              if x in self.down_h_rams:
39                  pass
40              else:
41                  try:
42                      self.votos += self.dict_hrams_servers[x].classify(
    addresses_split[x]) #executa a classificacao de cada metade dos endere os na
    hram correta
43                  except:
44                      self.down_h_rams.append(x)
45                      pass
46          return self.votos
47
48      def get_num_trainings(self):
49          return self.times_trained
50
51  def main():
52      daemon = Pyro4.Daemon(host="192.168.56.1")
53      ns = Pyro4.locateNS()
54      print(ns)
55      uri = daemon.register(Discriminator)
56      ns.register("discriminator_server1", uri, metadata={"discriminator",'simpsons',
    'Margie'})
57      daemon.requestLoop()
58
59  if __name__=="__main__":
60      main()
```

I.3 Código do servidor de RAM

```
import Pyro4
Pyro4.config.SERIALIZERS_ACCEPTED = set(['pickle'])
Pyro4.config.SERIALIZER = 'pickle'

@Pyro4.expose
@Pyro4.behavior(instance_mode="single")

class Hram:
    def __init__(self):
        return

    def instantiate(self, num_of_htables):
        self.num_of_htables = num_of_htables
        self.h_rams = [dict() for x in range(num_of_htables)]
        self.times_trained = 0

    def train(self, X):
        for i in range(0, self.num_of_htables):
            key = X[i]
            if key in self.h_rams[i].keys():
                self.h_rams[i][key] += 1
            else:
                self.h_rams[i][key] = 1
        self.times_trained += 1

    def get_times_trained(self):
        return self.times_trained

    def get_h_rams(self):
        return self.h_rams

    def classify(self, X, bleach =0):
        votes = 0
        for i in range(0, self.num_of_htables):
            key = X[i]
            if key in self.h_rams[i].keys():
                if self.h_rams[i][key] > bleach:
                    votes += 1
        return (votes)

def main():
    daemon = Pyro4.Daemon(host="192.168.56.1")
    ns = Pyro4.locateNS()
    print(ns)
    uri = daemon.register(Hram)
    ns.register("hram_server1", uri, metadata={"hram","0","Margie"})
    daemon.requestLoop()

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

I.4 Código para execução do Treino

```
import numpy as np
import random
import sys
import matplotlib.pyplot as plt
import os
import cv2 as cv
from math import exp
from sklearn.metrics import confusion_matrix
import argparse
import time
from colorama import init, deinit, Fore, Style
import Pyro4
Pyro4.config.SERIALIZERS_ACCEPTED = set(['pickle'])
Pyro4.config.SERIALIZER = 'pickle'

init()

def image_resize(img, size, interpolation = cv.INTER_AREA):
    h, w = img.shape[:2]
    c = None if len(img.shape) < 3 else img.shape[2]

    if h == w:
        return cv.resize(img, (size, size), interpolation)
    if h > w:
        dif = h
    else:
        dif = w

    x_pos = int((dif - w)/2.)
    y_pos = int((dif - h)/2.)

    if c is None:
        mask = np.zeros((dif, dif), dtype=img.dtype)
        mask[y_pos:y_pos+h, x_pos:x_pos+w] = img[:h, :w]
    else:
        mask = np.zeros((dif, dif, c), dtype=img.dtype)
        mask[y_pos:y_pos+h, x_pos:x_pos+w, :] = img[:h, :w, :]

    return cv.resize(mask, (size, size), interpolation)

def load_dataset_split(path, limit=sys.maxsize):
    if path[-1] != '/':
        path += '/'

    X_train = []
    y_train = []

    X_test = []
    y_test = []

```

```
51      classes = []
52
53      dirs = os.listdir(path)
54      for t in sorted(dirs):
55          if t == 'train':
56              dirs2 = os.listdir(path + t + '/')
57              for d_id, d in enumerate(sorted(dirs2)):
58                  classes.append(d)
59                  files = os.listdir(path + t + '/' + d  + '/')
60                  for i,f in enumerate(files):
61                      if i == limit:
62                          break
63
64                      X_train.append(path + t + '/' + d + '/' + f)
65                      y_train.append(d_id)
66
67          if t == 'test':
68              dirs2 = os.listdir(path + t + '/')
69              for d_id, d in enumerate(sorted(dirs2)):
70                  files = os.listdir(path + t + '/' + d  + '/')
71                  for i,f in enumerate(files):
72                      if i == limit:
73                          break
74
75                      X_test.append(path + t + '/' + d + '/' + f)
76                      y_test.append(d_id)
77
78      c = list(zip(X_train,y_train))
79      random.shuffle(c)
80      X_train,y_train = zip(*c)
81
82      c = list(zip(X_test,y_test))
83      random.shuffle(c)
84      X_test,y_test = zip(*c)
85
86      return np.asarray(X_train), np.asarray(y_train), np.asarray(X_test), np.asarray
        (y_test), classes
87
88  def load_dataset(path,p=0.25,limit=sys.maxsize,shuffle=True):
89
90      if path[-1] != '/':
91          path += '/'
92
93      X_train = []
94      y_train = []
95
96      X_val = []
97      y_val = []
98
99      classes = []
100
101     dirs = os.listdir(path)
```

```
102     for d_id,d in enumerate(sorted(dirs)):
103         classes.append(d)
104         files = os.listdir(path + d + '/')
105         for i,f in enumerate(files):
106             if i == limit:
107                 break
108 
109             if random.random() > p:
110                 X_train.append(path + d + '/' + f)
111                 y_train.append(d_id)
112             else:
113                 X_val.append(path + d + '/' + f)
114                 y_val.append(d_id)
115 
116     if shuffle:
117         c = list(zip(X_train,y_train))
118         random.shuffle(c)
119         X_train,y_train = zip(*c)
120 
121         c = list(zip(X_val,y_val))
122         random.shuffle(c)
123         X_val,y_val = zip(*c)
124 
125     return np.asarray(X_train), np.asarray(y_train), np.asarray(X_val), np.asarray(
    y_val), classes
126 
127 def run_train(wisard_obj, X_train, y_train,im_width, im_height, classes):
128     avg = 0
129     t1 = time.time()
130     for i, x in enumerate(X_train):
131         print('%4d/%4d >> Training class "%s" image "%s"' % (i, len(X_train),
    classes[y_train[i]], x ))
132         img = cv.imread(x,0) #0 => grayscale
133         img = image_resize(img, im_width)
134         img = img.reshape(im_width*im_height)
135 
136         ts = time.time()
137         wisard_obj.train(img,classes[y_train[i]])
138         tf = time.time()
139         avg += (tf - ts)
140 
141     print('training time:',(time.time()-t1), ' sec')
142     print('average training time: %6.2f sec' % (avg/len(X_train)))
143 
144     t1 = time.time()
145 
146 def main():
147 
148     parser = argparse.ArgumentParser(description='train and classify a dataset
    using Wisard Grayscale WNN')
149     parser.add_argument('--n', required=False, type=int, default=8, help='address
    size, default is 8')
```

```
150     parser.add_argument('--dataset_path', required=True, type=str, help='dataset to
        train/test')
151     parser.add_argument('--discriminator', required=True, type=str, help='
        discriminator to train/test')
152     args = parser.parse_args()
153     addressSize = args.n
154     im_width = 200
155     im_height = 200
156 
157     print('loading dataset...')
158     X_train,y_train,X_val,y_val,classes = load_dataset_split(args.dataset_path)
159     im_width = 200
160     im_height = 200
161     y_val_bin = np.bincount(y_val)
162     verbose = False
163     ignoreZero = False
164     entrySize = im_width*im_height
165     total_trained = np.sum(np.bincount(y_train))
166 
167     print('Train Dimensions: %s ' % len(X_train))
168     print('Test Dimensions: %s ' % len(X_val))
169     print('Classes:', classes)
170     print('labels: %s' % np.unique(y_train))
171     print('Train Class distribution: %s' % np.bincount(y_train))
172     print('Test Class distribution: %s' % y_val_bin)
173     print('Total training data %d' % (total_trained))
174     print('addressSize=',addressSize)
175     print('entrySize=',entrySize)
176 
177     discriminador = args.discriminator
178     wsd = Pyro4.Proxy("PYROMETA:"+'wisard'+','+ discriminador)
179     wsd.instantiate(num_of_htables=entrySize//addressSize, input_addr_length=
        addressSize,discriminador = discriminador, qtd_hram_server = 5)
180     avg = 0
181     y_pred = np.zeros(len(y_val))
182     acertos = 0
183 
184     run_train(wsd, X_train, y_train,im_width, im_height,classes)
185 
186     print('done')
187     deinit()
188 
189 if __name__ == "__main__":
190     main()
```

I.5 Código para execução da Classificação

```
import numpy as np
import random
import sys
import matplotlib.pyplot as plt
import os
import cv2 as cv
from math import exp
from sklearn.metrics import confusion_matrix
import argparse
import time
from colorama import init, deinit, Fore, Style
import Pyro4

Pyro4.config.SERIALIZERS_ACCEPTED = set(['pickle'])
Pyro4.config.SERIALIZER = 'pickle'
init()

def image_resize(img, size, interpolation = cv.INTER_AREA):
    h, w = img.shape[:2]
    c = None if len(img.shape) < 3 else img.shape[2]

    if h == w:
        return cv.resize(img, (size, size), interpolation)
    if h > w:
        dif = h
    else:
        dif = w

    x_pos = int((dif - w)/2.)
    y_pos = int((dif - h)/2.)

    if c is None:
        mask = np.zeros((dif, dif), dtype=img.dtype)
        mask[y_pos:y_pos+h, x_pos:x_pos+w] = img[:h, :w]
    else:
        mask = np.zeros((dif, dif, c), dtype=img.dtype)
        mask[y_pos:y_pos+h, x_pos:x_pos+w, :] = img[:h, :w, :]

    return cv.resize(mask, (size, size), interpolation)

def load_dataset_split(path, limit=sys.maxsize):
    if path[-1] != '/':
        path += '/'

    X_train = []
    y_train = []

    X_test = []
    y_test = []

```

```
51      classes = []
52
53      dirs = os.listdir(path)
54      for t in sorted(dirs):
55          if t == 'train':
56              dirs2 = os.listdir(path + t + '/')
57              for d_id, d in enumerate(sorted(dirs2)):
58                  classes.append(d)
59                  files = os.listdir(path + t + '/' + d  + '/')
60                  for i,f in enumerate(files):
61                      if i == limit:
62                          break
63
64                      X_train.append(path + t + '/' + d + '/' + f)
65                      y_train.append(d_id)
66
67          if t == 'test':
68              dirs2 = os.listdir(path + t + '/')
69              for d_id, d in enumerate(sorted(dirs2)):
70                  files = os.listdir(path + t + '/' + d  + '/')
71                  for i,f in enumerate(files):
72                      if i == limit:
73                          break
74
75                      X_test.append(path + t + '/' + d + '/' + f)
76                      y_test.append(d_id)
77
78      c = list(zip(X_train,y_train))
79      random.shuffle(c)
80      X_train,y_train = zip(*c)
81
82      c = list(zip(X_test,y_test))
83      random.shuffle(c)
84      X_test,y_test = zip(*c)
85
86      return np.asarray(X_train), np.asarray(y_train), np.asarray(X_test), np.asarray
        (y_test), classes
87
88  def load_dataset(path,p=0.25,limit=sys.maxsize,shuffle=True):
89      if path[-1] != '/':
90          path += '/'
91
92      X_train = []
93      y_train = []
94
95      X_val = []
96      y_val = []
97
98      classes = []
99
100     dirs = os.listdir(path)
101     for d_id,d in enumerate(sorted(dirs)):
```

```
102         classes.append(d)
103         files = os.listdir(path + d + '/')
104         for i,f in enumerate(files):
105             if i == limit:
106                 break
107 
108             if random.random() > p:
109                 X_train.append(path + d + '/' + f)
110                 y_train.append(d_id)
111             else:
112                 X_val.append(path + d + '/' + f)
113                 y_val.append(d_id)
114 
115     if shuffle:
116         c = list(zip(X_train,y_train))
117         random.shuffle(c)
118         X_train,y_train = zip(*c)
119 
120         c = list(zip(X_val,y_val))
121         random.shuffle(c)
122         X_val,y_val = zip(*c)
123 
124     return np.asarray(X_train), np.asarray(y_train), np.asarray(X_val), np.asarray(
    y_val), classes
125 
126 def run_classify(wisard_obj,X_val,y_val,y_pred,im_width,im_height,classes,
    total_trained,acertos,y_val_bin):
127     avg=0
128     dict_acertos = {}
129     dict_erros = {}
130     for classe in classes:
131         dict_erros[str(classe)] =0
132         dict_acertos[str(classe)] =0
133     for i, x in enumerate(X_val):
134         print('%4d/%4d >> Testing class "%s" image "%s" ' % (i, len(X_val), classes
    [y_val[i]], x, ))
135         img = cv.imread(x, 0) #0 => grayscale
136         img = image_resize(img, im_width)
137         img = img.reshape(im_width*im_height)
138         ts = time.time()
139         (disc, acc, conf) = wisard_obj.classify(X=img, total_trained=total_trained)
140         tf = time.time()
141         avg += (tf - ts)
142         y_pred[i] = classes.index(disc)
143 
144         if y_pred[i] == y_val[i]:
145             print(Fore.GREEN + ' %2.2f => class %s predicted as %s (a=%2.2f, conf
    =%2.2f)' % (acertos/len(X_val),classes[y_val[i]], disc,acc,conf))
146             acertos += 1
147             for classe in classes:
148                 if classes[y_val[i]] == classe:
149                     dict_acertos[str(classe)] +=1
```

```
150
151         else:
152             print(Fore.RED + ' %2.2f => class %s mistaken as %s (a=%2.2f, conf=%2.2
    f)' % (acertos/len(X_val),classes[y_val[i]], disc,acc,conf))
153             for classe in classes:
154                 if classes[y_val[i]] == classe:
155                     dict_erros[str(classe)] +=1
156
157         print(Style.RESET_ALL)
158
159     print('Acuracia: %6.2f' % (acertos/len(X_val)))
160     print('Tempo m dio de classificacao: %6.2f sec' % (avg/len(X_val)))
161
162     for classe in classes:
163         print('Porcentagem de acertos para %s: %6.2f' % (classe, dict_acertos[
    classe]/(dict_acertos[classe]+ dict_erros[classe])))
164
165     for i in range(0,len(classes)):
166         print(' %s ' % (classes[i]),end='')
167
168     print()
169     cfm = confusion_matrix(y_val, y_pred)
170     for i in range(0,cfm.shape[0]):
171         for j in range(0,cfm.shape[1]):
172             #print(' {0:2.2f >3} '.format((cfm[i,j]/y_val_bin[i])*100),end='')
173             print(' {:06.2f} '.format((cfm[i,j]/y_val_bin[i])*100),end='')
174         print()
175
176 def main():
177
178     parser = argparse.ArgumentParser(description='train and classify a dataset
    using Wisard Grayscale WNN')
179     parser.add_argument('--n', required=False, type=int, default=8, help='address
    size, default is 8')
180     parser.add_argument('--dataset_path', required=True, type=str, help='dataset to
     train/test')
181     parser.add_argument('--discriminator', required=True, type=str, help='
    discriminator to train/test')
182     args = parser.parse_args()
183     addressSize = args.n    # number of addressing bits in the ram.
184
185     print('loading dataset...')
186     X_train,y_train,X_val,y_val,classes = load_dataset_split(args.dataset_path)
187     im_width = 200
188     im_height = 200
189     y_val_bin = np.bincount(y_val)
190     verbose = False
191     ignoreZero = False
192     entrySize = im_width*im_height
193     total_trained = np.sum(np.bincount(y_train))
194
195     print('Train Dimensions: \%s ' \% len(X_train))
```

```
196     print('Test Dimensions: \%s ' \% len(X_val))
197     print('Classes:', classes)
198     print('labels: \%s' \% np.unique(y_train))
199     print('Train Class distribution: \%s' \% np.bincount(y_train))
200     print('Test Class distribution: \%s' \% y_val_bin)
201     print('Total training data \%d' \% (total_trained))
202     print('addressSize=',addressSize)
203     print('entrySize=',entrySize)
204 
205     discriminador = args.discriminator
206     wsd = Pyro4.Proxy("PYROMETA:"+'wisard'+','+ discriminador)
207     avg = 0
208     y_pred = np.zeros(len(y_val))
209     acertos = 0
210     run_classify(wsd,X_val,y_val,y_pred,im_width,im_height,classes,total_trained,
        acertos,y_val_bin)
211     print('done')
212     deinit()
213 
214 if __name__ == "__main__":
215     main()
```