Faculdade de Tecnologia de Americana "Ministro Ralph Biasi"

Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Willian dos Santos Almeida Alves

Utilização da Capacidade Computacional de uma Rede de Computadores para Aprendizado de Máquina utilizando BEAM VM e Elixir

Americana, SP

2025

Willian dos Santos Almeida Alves

Utilização da Capacidade Computacional de uma Rede de Computadores para Aprendizado de Máquina utilizando BEAM VM e Elixir

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas na área de concentração em algoritmos de aprendizado de máquina.

Orientador: Prof. Me. Rossano Pablo Pinto

Este trabalho corresponde à versão final do Trabalho de Conclusão de Curso apresentado por Willian dos Santos Almeida Alves e orientado pelo Prof. Me. Rossano Pablo Pinto

Americana, SP

2025

Willian dos Santos Almeida Alves

Utilização da Capacidade Computacional de uma Rede de Computadores para Aprendizado de Máquina utilizando BEAM VM e Elixir

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas pelo Centro Paula Souza – FATEC Faculdade de Tecnologia de Americana Ministro Ralph Biasi.
Área de concentração: Análise e Desenvolvimento de Sistemas.

Americana, 26 de junho de 2025.

Banca Examinadora:

Rossano Pablo Pinto
Mestre
Fatec Americana "Ministro Ralph Biasi"

Evandro Santaclara
Especialista
Fatec Americana "Ministro Ralph Biasi"

Odilon Delmont Filho
Doutor
Fatec Americana "Ministro Ralph Biasi"

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana Ministro Ralph Biasi- CEETEPS Dados Internacionais de Catalogação-na-fonte

ALVES, Willian dos Santos Almeida

Utilização da Capacidade Computacional de uma Rede de Computadores para Aprendizado de Máquina utilizando BEAM VM e Elixir . / Willian dos Santos Almeida ALVES – Americana, 2025.

52f.

Monografia (Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) - - Faculdade de Tecnologia de Americana Ministro Ralph Biasi – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Ms. Rossano Pablo PINTO

1. Inteligência artificial 2. Linguagem de programação 3. Sistemas distribuidos. I. ALVES, Willian dos Santos Almeida II. PINTO, Rossano Pablo III. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana Ministro Ralph Biasi

CDU: 007.52
681.3.061
681.3.047

Elaborada pelo autor por meio de sistema automático gerador de ficha catalográfica da Fatec de Americana Ministro Ralph Biasi.

DEDICATÓRIA

À Deus seja a primeira honra, que me permitiu alcançar tudo o que conquistei. Depois a minha família, que me fez sonhar com um amanhã melhor.

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Me. Rossano Pablo Pinto, que me guiou por essa jornada.

RESUMO

O presente texto conceitua elementos fundamentais no desenvolvimento de sistemas distribuídos utilizando a linguagem Elixir e seu ferramental herdado da linguagem Erlang. Foram utilizadas bibliotecas para os mais diversos fins, em especial para a conexão entre os nós do sistema (com a biblioteca Libcluster), e nas tarefas referentes a aprendizado de máquina (especialmente Axon e em segundo plano Nx). A justificativa da pesquisa é iniciada pelo aumento das tecnologias empregadas nas funções de algoritmos de aprendizado de máquinas e algumas preocupações nas limitações da utilização dessas mesmas tecnologias. Como metodologia proposta ao trabalho, é realizada uma pesquisa exploratória do ferramental da linguagem Elixir para as atividades de algoritmos de aprendizado de máquina que tanto cresce no presente momento. O resultado dessa pesquisa é uma aplicação, que não tem como objetivo determinante a sua performance e otimização, apenas o seu pleno funcionamento no ambiente proposto. A aplicação foi desenvolvida com uma arquitetura versátil em mente, especialmente para o caso da adição de mais modelos de aprendizado de máquinas à aplicação. Para tal foi escolhida uma arquitetura baseada em componentes, esta que tangencia com naturalidade as proposições fundamentais de módulos e funções que uma linguagem de paradigma funcional como Elixir prezam tanto. A conclusão foi de que durante o desenvolvimento algumas barreiras foram encontradas, como a dificuldade de otimização no uso de tensores via mensageria e a falta de ferramental no ecossistema de aprendizado de máquinas na plataforma Elixir que abstraiam algumas dificuldade primordiais no desenvolvimento de aplicações que se utilizam de algoritmos de aprendizado de máquinas.

Palavras Chave: Sistemas Distribuídos; Linguagem de Programação; Inteligência Artificial

ABSTRACT

This text conceptualizes fundamental elements in the development of distributed systems using the Elixir language and its tools inherited from Erlang language. Libraries were used for a variety of purposes, especially for the connection between system nodes (with the Libcluster library), and in tasks related to machine learning (especially Axon and in the background Nx). The justification for the research begins with the increase in technologies used in machine learning algorithm functions and some concerns about the limitations of using these same technologies. As a methodology proposed for the work, an exploratory research of the Elixir language tools for machine learning algorithm activities is carried out, which is growing so much at the present time. The result of this research is an application, which does not have as its determining objective its performance and optimization, only its full functioning in the proposed environment. The application was developed with a versatile architecture in mind, especially for the case of adding more machine learning models to the application. To achieve this, a component-based architecture was chosen, which naturally touches on the fundamental propositions of modules and functions that a functional paradigm language like Elixir values so much. The conclusion was that during development some barriers were encountered, such as the difficulty of optimizing the use of tensors via messaging and the lack of tools in the machine learning ecosystem on the Elixir platform that abstracted some fundamental difficulties in developing applications that use machine learning algorithms.

Keywords: *Distributed Systems; Programing Language; Artificial inteligence*

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Aplicações que utilizam aprendizado de máquina para os mais diversos fins e com as mais variadas técnicas estão se tornando cada vez mais comuns. Segundo Alexander Linden, vice-presidente de pesquisas na revista Gartner (Moore, 2016) “A 10 anos, tínhamos dificuldade de encontrar 10 aplicações comerciais baseadas em aprendizado de máquina. Agora temos dificuldade de encontrar 10 que não usam”. Isso demonstra o constante crescimento de algoritmos que utilizam alguma forma de aprendizado de máquina, e sua presença em ações simples do dia-a-dia, como pesquisas na web, carros cada vez mais inteligentes ou reconhecimento de voz (Moore, 2016).

De sistemas de reconhecimento facial à geração de texto, a inteligência artificial estende-se a setores inteiros da indústria, tornando processos mais eficientes e precisos. Como lista a revista Gartner (Moore 2016), alguns setores já possuem exemplos sólidos de organizações que se baseiam em aplicações com algoritmos de aprendizado de máquina. A mesma destaca algumas áreas, como por exemplo vendas e marketing, onde é muito utilizado na recomendação de novos produtos, utilizando os dados do próprio cliente e de toda a base para entregar valor na forma de possíveis vendas e fidelização dos clientes. Algo similar é utilizado em redes sociais ou plataformas de streaming de vídeo e áudio. Kaufman e Santaella (2020) destrincham como o algoritmo da rede social *Facebook* constrói o fluxo de histórias (*feed*) do usuário. Segundo as mesmas, utilizando como base a divulgação do próprio *Facebook*, o algoritmo de *deep learning* constrói uma classificação qualitativa de afinidade, utilizando para isso diversos parâmetros, alguns sendo privilegiados, como o tempo e as publicações mais “ativas” (que geram mais interações dos usuários).

Baji (2017) destaca a diferença essencial entre as GPU (*Graphical Process Unit*), que é o poder massivo de centenas ou milhares de núcleos da GPU, proporcionando desta forma a capacidade de paralelizar trabalho que não precisa ser sequencial, ao contrário das CPU (*Central Process Unit*), onde o foco está no trabalho sequencial (embora também existe paralelização em CPU, apenas não sendo massivo). Originalmente pensadas como processadores exclusivos para a renderização de gráficos 3D, em 2006, no desenvolvimento das GPU aconteceu um processo de generalização do seu uso, com a *NVIDIA* protagonizando este processo

com a criação do CUDA (*Compute Unified Device Architecture*), uma plataforma para desenvolvimento de software que opera diretamente na GPU, beneficiando-se desta forma de seu poder incomparável de paralelismo. Computadores capazes de acelerar os processos de aprendizado de máquina precisam ser equipados com placas de vídeo de alto desempenho, componentes especializados em realizar cálculos de matrizes — a base do aprendizado de máquina (Deisenroth, Faisal, Ong, 2020, p. 98).

Hoje, existe uma onipresença da computação em nuvem, tornando-a o caminho natural para desenvolvedores que necessitam de capacidade computacional que, muitas vezes, não possuem. Como demonstrado na revista Gartner (Stamford, 2023), haverá um crescimento de aproximadamente 67% no mercado global de *cloud*, considerando os anos de 2022 a 2024. Isto inclui diversos serviços, como os da categoria *Plataform as a Service*, que visa disponibilizar para os desenvolvedores ambientes para a criação de aplicações com facilidades no desenvolvimento e distribuição do software. Outras categorias muito conhecidas são *Infrastructure as a Service* e *Software as a service*. A primeira entrega ao cliente a infraestrutura necessária para a execução de sua aplicação, enquanto a segunda entrega um software propriamente dito. Essas três categorias somadas representaram aproximadamente 80% do faturamento das *cloud* em 2022, com 392 milhões de dólares. Além do faturamento expressivo, para o cliente final, especialmente para os desenvolvedores, elas representam a principal forma de desenvolver e executar aplicações que se utilizam de aprendizado de máquinas.

Plataformas como AWS, Google Cloud e Azure oferecem uma variedade de serviços que vão desde a infraestrutura básica para aplicações que utilizam aprendizado de máquina até serviços específicos, como inteligências artificiais generativas de texto, imagens e até vídeos. No entanto, o custo financeiro é um fator limitante, que pode dificultar a adesão dessas plataformas por parte de instituições de ensino ou o uso por desenvolvedores durante o aprendizado.

1.1 MOTIVAÇÃO

Em diversas instituições de ensino, sejam elas públicas, privadas de ensino superior ou de ensino fundamental/médio, existem laboratórios de informática e centros de tecnologia que constituem um parque computacional considerável, ao menos em termos de número de dispositivos. A capacidade computacional conjunta

dessas dezenas ou centenas de máquinas é significativa. Contudo, muitos desses recursos permanecem ociosos e subutilizados em casos simples que não envolvem uso intensivo de processamento; em particular, o poder de processamento das placas de vídeo raramente é aproveitado.

Embora esse poder computacional, quando comparado ao virtualmente infinito oferecido pelos grandes provedores de nuvem, possa parecer modesto, ele tem seu valor para fins educacionais. Ao permitir que alunos, professores ou pesquisadores utilizem os equipamentos locais, cria-se um ecossistema independente de grandes corporações, que podem não ter interesse nessas instituições. De fato, como destaca Bergmann (2024), é projetado um uso de modelos menores e possivelmente mais focados em situações mais específicas. Ainda em Bergmann (2024), é citado a cada vez mais baixa oferta de GPU potentes no mercado. O recente aumento da demanda faz com que encareça a oferta presente e mesmo as organizações dispostas a adquiri-las, é um estoque reduzido. Com tudo isso em mente, é um caminho natural para organizações como um todo procurar uma alternativa viável e eficiente.

Além destes pontos, a criação e utilização de aplicações de aprendizado de máquina localmente, aproveitando recursos pré-existentes, permite a otimização dos recursos da instituição e, claro, proporciona um aprendizado valioso ao estruturar aplicações em ambientes que não contam com os facilitadores oferecidos pelos provedores de nuvem. Adicionalmente, ao treinar modelos de IA localmente, a instituição pode garantir maior controle sobre o sigilo e a segurança dos dados. Isso é particularmente importante quando se lida com informações sensíveis ou reguladas, já que evita a necessidade de transferir dados para servidores externos, reduzindo o risco de exposição e garantindo conformidade com normas de privacidade e proteção de dados.

1.2 OBJETIVOS

Esta seção apresenta os objetivos do trabalho.

1.2.1 OBJETIVO GERAL

Desenvolver um sistema distribuído que utilize a capacidade computacional de uma rede para o aprendizado de máquinas.

1.2.2 OBJETIVOS ESPECÍFICOS

1. Estudar e aplicar conceitos de sistemas distribuídos e sua implementação na BEAM VM.
2. Investigar a aplicação de aprendizado de máquinas em sistemas distribuídos.
3. Examinar bibliotecas de aprendizado de máquinas na linguagem Elixir.
4. Distribuir processos de aprendizado de máquinas em uma rede computacional.

1.3 METODOLOGIA

Nesta seção será descrito a metodologia que será utilizada no trabalho.

1.3.1 JUSTIFICATIVA

Como abordado na seção 1, existe uma inclinação do desenvolvimento de aplicações que utilizam aprendizado de máquina para utilizarem uma infraestrutura local. Seja para o preparo dos alunos ou a utilização dos pesquisadores, este trabalho busca investigar um dos caminhos possíveis para o problema que pode se apresentar. Utilizar localmente modelos de inteligência artificial também é mais seguro, visto que necessitam de muitos dados, e desta forma os mesmo não serão transferidos para servidores externos. Os requisitos, portanto, são a utilização da infraestrutura vigente de forma eficiente.

1.3.2 METODOLOGIA DE PESQUISA

A presente pesquisa caracteriza-se em pesquisa aplicada, visto que tem como objetivo propor uma solução ao problema investigado. Tem como carater ser exploratória e de ordem quantitativa, ao observar se a solução proposta é viável ou não, por este mesmo motivo o método utilizado será hipotético-dedutivo.

Primeiramente será realizada uma exploração teórica do tema, investigando as tecnologias bases que o constituem, como o próprio conceito de computação distribuída e um sistema distribuído.

Após, um teste das ferramentas propostas deverá ser realizado (o uso da linguagem Elixir, a plataforma da BEAM VM, a distribuição através de containers, etc).

Com uma base sedimentada será desenvolvida uma prova de conceito do proposto. O desenvolvimento deste protótipo tem um valor crucial para a pesquisa, sendo este seu principal objetivo.

Por fim, será feito o levantamento dos resultados, para aferir se a solução proposta é viável e possíveis caminhos de um desenvolvimento futuro.

2 REFERENCIAL TEÓRICO

Este capítulo descreve alguns dos conceitos fundamentais que embasam o desenvolvimento de sistemas distribuídos para aprendizado de máquina, tendo como ponto focal a utilização da infraestrutura da BEAM VM, utilizando para operá-la a linguagem Elixir. Além disso, a construção de uma base teórica que permita compreender as nuances do problema apresentado e os motivos que tornam as possíveis soluções promissoras.

2.1 SISTEMAS DISTRIBUÍDOS

Steen & Tanenbaum (2016) assim definem sistemas distribuídos: “Um sistema distribuído é uma coleção de elementos computacionais autônomos que aparentam à seus usuários ser um único e coerente sistema”.

Sistemas distribuídos se tornaram comuns no mundo atual, e isso se dá início em meados de 1980, quando dois avanços tecnológicos impulsionam essa área da engenharia de *software*, estes sendo a miniaturização dos processadores e tecnologias de rede de alta velocidade (Steen & Tanenbaum, 2016).

Tanenbaum (2007), ao descrever os objetivos de um sistema distribuído deixa claro que este não é necessário para todos os casos. Por isso, se um sistema visa cumprir esses objetivos em algum grau, este é considerado um sistema distribuído. Os objetivos são: disponibilidade de recursos compartilhados, transparência, abertura e escalabilidade. Estes conceitos não são restritos a sistemas distribuídos, tão pouco se mostram de forma homogênea nos mesmos, porém estão presentes nesses.

Disponibilidade dos recursos compartilhados é o conceito primordial de um sistema distribuído. O recurso pode ser materializado em qualquer recurso computacional de processamento, armazenamento ou transmissão de dados. Uma impressora é um recurso, uma página web é um recurso, processamento de dados para treinamento de algoritmos de aprendizado de máquinas também é um recurso. Um sistema distribuído deve ser capaz de compartilhar recursos com os usuários, independente da forma que o faz.

A transparência é a capacidade do sistema de se ocultar do usuário. Existem diversas formas de transparência, além de diversos graus de transparência, e de

maneira geral, limitações de transparência. Os tipos podem ser classificados da seguinte maneira, como fez Tanenbaum (2007):

Tabela 1 - Tipos de transparência em um sistema distribuído

Tipo de transparência	**Descrição**
Acesso	Esconde diferenças na representação do dado e em seu acesso
Localização	Esconde aonde um recurso é acessado
Migração	Esconde que o recurso poder mudar de localização
Realocação	Esconde que o recurso pode se mudar em uso
Replicação	Esconde que o recurso é replicado
Concorrência	Esconde que um recurso pode ser compartilhado por vários usuários ao mesmo tempo
Falha	Esconde a falha e recuperação de um recurso

Fonte: ISO 1995

Além dos tipos de transparência, essa característica também possui graus. Um exemplo é a transparência a falhas. Caso uma requisição de recurso mal sucedida se repita indefinidamente antes de se mostrar ao usuário (ou seja, o usuário não fica ciente da falha), esta repetição da requisição pode acarretar em um completo congestionamento de todo o sistema, e ainda assim o objetivo principal (entregar o recurso) não será alcançado. Por isso, nem sempre a transparência será utilizada integralmente.

Além disso, a transparência também possui limitações, como por exemplo o limite teórico e/ou prático da velocidade de transferência entre longas distâncias.

Um sistema é aberto quando:

oferece interfaces bem definidas e pública essas interfaces para permitir a interoperabilidade e a portabilidade entre seus componentes (Tanenbaum, 2007).

Este objetivo possibilita o sistema ser estendido, acessível e integrável por outros sistemas, já que toda a comunicação entre este é feita através de uma mensageria

abstrata, cujo qual ambos os sistemas implementam. Um exemplo disso são protocolos de comunicação como o HTTP, utilizado pela web, ou o OTP utilizado na BEAM VM por Erlang/Elixir.

Um dos objetivos de arquitetura mais importante quando se constrói um sistema distribuído é a escalabilidade (Tanenbaum, 2007). A escalabilidade pode ser em tamanho (o comum em se pensar), geográfica ou administrativa. A escalabilidade de tamanho é a capacidade de um sistema de alocar novos recursos ao sistema sem a perda significativa de desempenho. A escalabilidade geográfica se define pela confiabilidade do sistema, mesmo com as suas partes distantes geograficamente. Por fim, a escalabilidade administrativa é a capacidade do sistema de ser administrado por múltiplas organizações.

Existem várias técnicas possíveis de escalonamento. De maneira geral, se resolve problemas de tamanho em um sistema adicionando mais poder computacional, o que gera poucas complicações mas nem sempre é viável. Sobre o escalonamento geográfico, técnicas de distribuição, replicação e *caching* entram em cena. Quanto ao escalonamento administrativo, normalmente as raízes são humanas. De um ponto de vista técnico, uma arquitetura P2P, pode ser uma opção.

Existem, porém, os problemas naturais aos sistemas distribuídos, como o diagnóstico de problemas (que podem estar em diferentes componentes do software ou em diferentes nós), e o fato de que recaem-se muito sobre a qualidade da rede. Outro ponto focal dos problemas que ocorrem em sistemas distribuídos é a natureza assíncrona dos mesmos (Thoke, 2016).

Por fim, diversos tipos de aplicações fazem bom uso desta arquitetura, como telecomunicações de forma geral, aplicações de tempo real, como sistemas industriais ou de aviação, aplicações que usam uma rede de computadores essencialmente fazem uso de sistemas distribuídos, como aplicação da World Wide Web, e é claro, computação distribuída.

Sistemas distribuídos podem ser divididos em duas grandes linhas de arquiteturas: Os sistemas centralizados e os descentralizados.

Um sistema distribuído centralizado pode ser resumido em um sistema dependente de um único componente, este que será encarregado de todas as tarefas críticas do sistema ou ao menos terá responsabilidade o suficiente para que o sistema não seja funcional sem ele. Este tipo de sistema normalmente classifica seus nós com dois papéis: clientes e servidores.

Estes papéis não são mutuamente exclusivos, pelo contrário, na maior parte das implementações da arquitetura cliente/servidor, o servidor também faz papel de cliente de outros servidores. De maneira geral, porém, o usuário sempre será determinado como o cliente. Eis a definição:

Cliente: Um cliente é um nó do sistema que faz uma requisição (essa de qualquer tipo de recurso) a um servidor.

Servidor: Um servidor é um nó do sistema que responde às requisições (essas de qualquer tipo de recurso), sendo que o cumprimento desta requisição pode levá-lo a se tornar cliente de um outro servidor.

Embora exista essa distinção entre nós clientes e nós servidores, quando se arquiteta um sistema ela deixa de ser suficientemente clara. Por isso, a arquitetura de sistemas traz em seu arcabouço uma outra classificação no que tange a sistemas distribuídos, essa representada em camadas:

1. Camada de interface de usuário
2. Camada de processamento
3. Camada de dados

Esta divisão de um sistema em 3 fatores não é incomum na arquitetura de sistemas, considerando uma margem de equivalência, está presente em padrões de arquitetura como o MVC (*Model, View, Control*), onde existe um modelo representativo de dados (*model*), uma interface para o usuário (*view*) e um controlador dos processos (*control*) (embora mesmo no padrão MVC componentes de domínio, como regras de negócio, possam estar distanciados dessa arquitetura).

A camada de interface de usuário (ou cliente), é a forma que o sistema receberá as requisições dos clientes e responderá de forma apresentável a eles, um sistema distribuído tem como objetivo ser aberto, e isso fica muito claro na camada de interface de usuário. Um exemplo são aplicações web, que normalmente utilizam protocolos de comunicação (como HTTP ou gRPC), além de protocolos de apresentação (como HTML ou JSON). Embora estes protocolos sejam abertos e amplamente implementados em ferramentas como buscadores web, nada impede a utilização de protocolos próprios e/ou fechados, apenas devem ser descritos e implementados por ambos os sistemas.

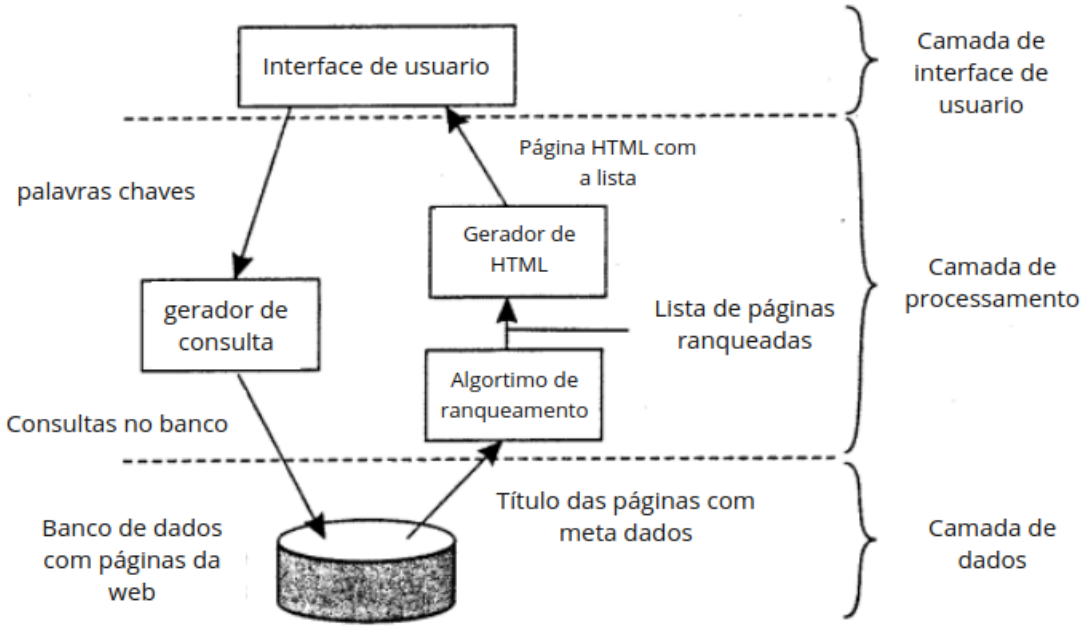
A camada de processamento é responsável pela orquestração dos recursos do sistema. Após a requisição feita na camada acima, ela passa a ordenar os processos necessários para cumpri-la.

Na camada de processamento é contido parte das regras de domínio do sistema, como a apresentação que será enviada à camada de interface de usuário e os dados necessários para tal.

Por fim, a camada de dados contém todos os dados da aplicação. Esses dados podem ser dados de usuários, ou simplesmente dados que o sistema utiliza no processamento de suas requisições. Por exemplo, uma aplicação web busca uma determinada página baseado em um parâmetro, como o nome da página.

A figura 1 exemplifica o fluxo de uma aplicação web, no caso um buscador de páginas web:

Figura 1 - Fluxo de um buscador de páginas web



Fonte: Distributed System: Principles and Paradigms (adaptado)

Observando essas camadas, as relações entre os nós do sistema ficam mais claras, pois seus papéis dentro do sistemas são bem definidos. De maneira geral, a centralização do sistema ocorre na camada de processamento, sendo esta a que controla todo o fluxo interno do sistema.

Embora usado como exemplos, muitas outras formas de arquitetura podem ser utilizadas para sistemas web. Em um sistema centralizado a maneira mais comum de distribuição é a vertical, exemplificada acima. Esta concentra partes lógicas do sistema em uma única respectiva máquina. Existe também a distribuição horizontal, que será tratada a seguir.

A distribuição horizontal se caracteriza pela uniformidade de papéis que os nós possuem dentro do sistema, sendo todos estes conectados de alguma forma. Por conta disso, nenhum nó é crítico e pontos únicos de falha deixam de existir.

Essa descentralização é alcançada pelo princípio de que todos os nós possuem ou:

- Todos os dados necessários para o processamento de uma requisição; ou
- Meios necessários de obter os dados necessários para o processamento de uma requisição.

Tendo todos nós um mesmo modelo de referência de dados, qualquer requisição pode ser atendida por qualquer nó ou por vários nós paralelamente, o que traz ao sistema mais capacidade computacional nesse tipo de tarefa.

Sistemas descentralizados trazem consigo complexidades adicionais. A primeira é a organização da rede que interliga os nós. Em uma rede estruturada, todos os nós e os recursos que detém (se estes estiverem particionados) são indexados em uma tabela distribuída.

Uma rede desestruturada não possui uma forma de indexação, por isso a requisição de algum recurso é propagada por toda a rede e respondida pelos nós aptos.

Essa generalização das arquiteturas de sistemas distribuídos é necessária devido a extensão do assunto.

2.2 APRENDIZADO DE MÁQUINAS E SISTEMAS DISTRIBUÍDOS

O aprendizado de máquina (Machine Learning - ML) é um campo da ciência da computação, sendo este um ramo da inteligência artificial. Na programação tradicional, os sistemas são desenvolvidos através de algoritmos que definem de forma explícita o que um sistema deve fazer, isto é: um sistema não poderá adequar-se a diferentes contextos, caso isso não tenha sido previsto durante a construção do mesmo.

Algoritmos de ML por sua vez, são desenvolvidos para aprender padrões diretamente a partir de dados (o contexto em que o sistema está inserido), e possuem a capacidade de se adequar a um novo conjunto de dados, reagindo aos padrões deste novo conjunto, utilizando o que “aprendeu” durante o contexto inicial. Essa classe de algoritmos cria modelos matemáticos e estatísticos internos, e desta

forma são capazes de realizar diferentes tipos de tarefas, como classificação, predição, agrupamentos, otimizações, entre outras.

Diversas técnicas são utilizadas na construção dos modelos de ML, como por exemplo o aprendizado por reforço, que faz o algoritmo interagir com o conjunto de dados e otimizando-o por meio de “recompensas” para determinar os padrões daquele contexto.

A melhoria contínua no *hardware* e na engenharia de *software* impulsionam cada vez mais sistemas que incorporam algoritmos de ML. Um produto notável deste cenário é o ChatGPT (Deisenroth et al 2020, p. 98). Este produto da OpenAi exemplifica o potencial de tecnologias de ML, sendo um propulsor do interesse do público neste campo.

O ML tornou-se integrante de inúmeras tecnologias, sendo a base de tantas destas, como reconhecimento de voz, visão computacional, sistemas de recomendação, processamento de linguagem natural, e muitas outras.

O processo de treinamento de um modelo de aprendizado de máquina é intensivo em recursos computacionais. A depender da complexidade do modelo e da quantidade de dados, o tempo necessário para a conclusão de um treinamento pode variar de minutos a semanas, mesmo em hardware especializado. Esse custo é justificado pelo volume de operações matemáticas, principalmente multiplicações de matrizes de alta dimensão, necessárias para ajustar os parâmetros internos do modelo. Em razão disso, a necessidade de paralelismo se tornou um pilar central no desenvolvimento de soluções em ML, buscando reduzir o tempo de treinamento e otimizar o uso dos recursos disponíveis.

As principais abordagens para paralelizar o processo de aprendizado nos algoritmos de ML são o paralelismo de modelo e o paralelismo de dados.

No paralelismo de modelo, partes diferentes do modelo são distribuídas entre múltiplos nós. Este processo exige a divisão do modelo, a alocação dos dispositivos processadores, o fluxo de dados que será recebido pelos nós e enviados por eles, o processamento paralelo do treinamento do modelo e por fim a agregação e sincronização dos pesos dados aos modelos, para que o treinamento prossiga de forma eficiente. Esta técnica é útil para modelos de grande escala, onde um modelo

inteiro ultrapasse a capacidade de um único nó. O maior limitador desta técnica é a latência envolvida no processo, além das dificuldades de implementação que rodeiam a comunicação, balanceamento de carga e particionamento do modelo.

No paralelismo de dados, o modelo completo será replicado entre todos os nós, limitando o treinamento à capacidade do nó mais fraco, já que todos estes devem ser capazes de alocar recursos para o treinamento. Nessa técnica, os dados são particionados entre nós, seguindo um fluxo de trabalho similar ao do paralelismo de modelo, incluindo a sincronização dos pesos em todos os nós. É mais simples de ser implementada, porém requer uma cadeia de elementos processadores condizente com as dimensões do modelo.

Os desafios da implementação de ML em ambientes distribuídos são muitos e significativos. Segundo Verbraeken et al (2020), a comunicação eficiente entre os nós da rede pode ser o mais fundamental desses, dados que um grande volume de dados serão transferidos entre os nós durante o treinamento, criando assim gargalos de comunicação entre partes que frequentemente precisam se sincronizar. Outro desafio a se destacar é a heterogeneidade da rede, que dispõe de recursos variados. Como mostrado no parágrafo anterior, em certas técnicas de paralelização do treinamento, o processo todo será pautado pelo nó com menos recursos. Se posto em comparação, os desafios da distribuição de ML e de outros domínios são similares.

Assim como compartilham desafios, o ML distribuido também compartilha as benesses dos sistemas distribuídos de maneira geral. A escalabilidade é muitas vezes um foco, pois permite que problemas de uma dimensão superior a infraestrutura disposta possam ser resolvidos a um custo aceitável. A redução do tempo pode ser um benefício, embora exija uma infraestrutura para tal, levando em conta as limitações dos nós e especialmente da rede. A tolerância à falha é um outro ponto importante, que está diretamente ligado à arquitetura do sistema, sendo esta uma consequência de uma escolha de design. Também relacionado a arquitetura do sistema, pode existir uma proximidade aos dados, como por exemplo em sistemas de ML que atuam em tempo real, muitas vezes ligados a dados de sensores embarcados ou similares.

Existem várias ferramentas para a construção direta de sistemas de ML distribuído, como por exemplo o *Tensor Flow Distributed*, que é uma solução nativa de uma das mais importantes bibliotecas no ecossistema de ML. É uma solução

completa que abstrai do desenvolvedor a maior parte dos problemas com relação à rede, conexão entre nós e afins. Na parte do design, temos as duas grandes linhas do design distribuído, os sistemas centralizados e descentralizados.

Nos sistemas centralizados, um nó será definido como o responsável pela partição do modelo ou dados e pela eventual agregação do mesmo, seja na distribuição por dados ou modelo. Em um sistema descentralizado, todos os nós possuem o potencial para realizar todas as funções do sistema. A definição das funções é uma escolha de design do sistema e pode ser realizada de diversas formas.

Além do treinamento, o estilo de distribuição também afeta a forma de armazenar e utilizar o modelo, podendo ser armazenado de forma distribuída (análogo a uma rede bittorrent por exemplo) ou centralizada, como em um servidor de arquivos. A utilização do modelo normalmente é orquestrada pelas mesmas bibliotecas em que foram desenvolvidos, ou compatíveis. A forma da utilização distribuída também abrange diversas escolhas de design, desde do aprendizado federado em que vários modelos especialistas contribuem para o processamento da saída final, até a simples distribuição das tarefas de cálculo do modelo em diversos nós. Dito isso, normalmente a utilização do modelo fica restrita a um único nó, sendo sua distribuição motivada pelo balanceamento da infraestrutura e não por restrições da mesma.

Como sempre, a definição de que será utilizado a ML distribuída é contextual, e as formas de se fazer isso também.

2.3 INFRAESTRUTURA EM NUVEM

Como Bhardwaj et al (2010) definem “Computação em nuvem é uma forma de se referir ao uso de recursos computacionais compartilhados”. Ainda em Bhardwaj et al (2010) “Computação em nuvem reúne um grande número de computadores, servidores e outros recursos”. Como fica claro, uma nuvem pode ser definida em seu cerne como uma infraestrutura conjunta de nós computacionais, além de outros recursos necessários para a nuvem, alguns sendo essenciais como a rede que a conecta, outros opcionais, como periféricos.

A “nuvem” se tornou comum na última década. Este conceito, embora não esteja preso a um modelo comercial em específico, foi popularizado desta maneira. Como cita Mirashe & Kalyankar (2010), a forma mais comum de nuvem no âmbito

comercial são as nuvens privadas com algum tipo de acesso a internet. A nuvem da Google, por exemplo, permite a usuários de todo o mundo compartilhar arquivos, trabalhar em conjunto em documentos, realizar chamadas de vídeo, etc.

Os modelos de negócio mais comuns nestas nuvens são a venda de algum serviço, seja ele software (normalmente para o usuário final), plataforma (para a distribuição de software) e infraestrutura (qualquer tipo de computação em geral). No geral, essas nuvens seguem os mesmos princípios de sistemas distribuídos, buscando e proporcionando transparência, abertura e escalabilidade.

Bhardwaj et al (2010) descreve um fluxo viável e comum para a nuvem. Provedores de nuvem normalmente disponibilizam ambientes virtualizados, para melhor utilização dos servidores e mais liberdade ao consumidor. Junto de possivelmente outros serviços, como banco de dados, *firewalls* ou balanceadores de carga, uma aplicação distribuída desta forma poderia estar acessível ao usuário final.

2.4 HARDWARE DEDICADO E VIRTUALIZAÇÃO

Hardware dedicado pode ser entendido como um elemento computacional dedicado a uma única tarefa ou organização. Está no espectro contrário da nuvem, embora nuvens possam disponibilizar também um certo nível de hardware dedicado. Desde de meados dos anos 1960, existem ideias de virtualização, sendo a mais notável da IBM, no conhecido sistema operacional CP/CMS, onde uma grande máquina cuidava de múltiplas cópias de um mesmo sistema operacional, proporcionando assim o compartilhamento de recursos, algo notável em uma época onde computadores eram extremamente caros e precisavam ser usados de formas mais eficientes (Rodrizes-Haroa, 2012).

Práticas de virtualização evoluíram e se tornaram comuns, possuindo implementações diferentes, cada uma com suas vantagens e uso. Uma primeira distinção clara é a execução de um sistema operacional modificado ou não. A virtualização em nível de sistema é uma das técnicas que modifica o kernel para que o mesmo execute múltiplos sistemas operacionais utilizando o mesmo kernel (Rodrizes-Haroa, 2012). Outra é a para-virtualização, que modifica algumas instruções da máquina real, permitindo que múltiplos kernels diferentes sejam executados, embora os mesmos precisem ser modificados. No campo das técnicas que não exigem a modificação do sistema operacional, temos a tradução dos

binários do sistema hóspede para as instruções do sistema hospedeiro. A partir de 2006 as maiores fabricantes de CPU começaram a construir processadores com instruções específicas para facilitar a virtualização, assistindo dessa forma a mesma (Rodrizes-Haroa, 2012).

No geral, processadores possuem níveis de permissão de execução, conhecidos como *rings*, indo do 0 ao 3 normalmente. Nas técnicas de virtualização os sistemas hóspedes eram executados fora do nível 0, o nível tradicional do kernel. Porém com a assistência do hardware, um “nível -1” é criado para o gerenciador das máquinas virtuais, possibilitando sistemas não modificados de serem executados em nível 0.

Uma última forma de “virtualização”, é a virtualização em um processo. O sistema hóspede é executado apenas como um único processo do sistema hospedeiro e requisita permissão sempre que precisa executar operações em nível de kernel. A diferença para um processo comum se dá, de forma geral, que estes processo utilizam diversas threads, normalmente possuindo um agendador de threads próprio, assim como o gerenciamento de memória.

2.5 BEAM VM

A BEAM é a máquina virtual responsável pela execução de aplicações escritas em Erlang, Elixir e outras linguagens compatíveis. Apesar do nome "Bogdan's Erlang Abstract Machine" (posteriormente reinterpretado como "Björn's Erlang Abstract Machine"), a BEAM não é apenas uma definição abstrata: ela é a implementação concreta e única de uma Máquina Virtual Erlang (Stenman, 2025). Esta seção irá introduzir o ERTS (essencial para o entendimento da BEAM) e a própria BEAM.

A BEAM é constituída sobre um ERTS (*Erlang RunTime System*) e oferece o suporte para OTP.

De forma breve, o ERTS é o processo no sistema operacional, é a interface da BEAM com o sistema operacional e vice-versa. BEAM é a máquina virtual que executa códigos de seu *bytecode* (similar a *Java Virtual Machine*), códigos esses compilados de linguagens como Erlang ou Elixirn (Stenman, 2025). OTP são as bibliotecas base do Erlang e Elixir e definem padrões de comunicação e da plataforma de desenvolvimento (Stenman, 2025).

Como citado, o ERTS irá executar uma instância da BEAM, um coletor de lixo e por padrão um gerenciador de processos por núcleo do processador (Stenman, 2025).

O gerenciador de processos do ERTS é similar a um gerenciador de processos de um sistema operacional. Ele possui duas filas principais: processos prontos e processos que estão esperando alguma mensagem. O tempo de execução dos processos é compartilhado de forma concorrida (a preempção de processos é gerenciada internamente, independentemente de o sistema operacional hospedeiro oferecer suporte nativo a processos preemptivos.) (Stenman, 2025). Processos em Erlang são “processos baratos” e não são relacionados aos processos do sistema operacional. Processos erlang não são threads ou forks do sistema operacional, e todos os seus metadados e memória são privados, a comunicação com outros processos é feita por via de mensagens (Stenman, 2025).

O subsistema de memória do ERTS é responsável pelas alocações e realocações de memória. O coletor de lixo é responsável por liberar memória associada a processos finalizados ou de realocações que diminuíram a memória de processos ativos (Stenman, 2025). Tudo isso é feito de forma automática, sem necessidade de intervenção manual pelo desenvolvedor.

Por fim, cada instância da BEAM é chamada de nó. Cada nó possui sua própria máquina virtual, ambiente de execução e nome identificador, permitindo comunicação e distribuição de tarefas (Stenman, 2025).

2.6 ELIXIR

Elixir é uma linguagem de programação funcional e dinâmica projetada para o desenvolvimento de aplicações escaláveis e de fácil manutenção. A linguagem é executada sobre a máquina virtual Erlang (BEAM), que é reconhecida por sua capacidade de criar sistemas distribuídos, tolerantes a falhas e de baixa latência (Elixir, 2025). Esta base técnica permite que aplicações desenvolvidas em Elixir sejam altamente resilientes, além de aproveitarem ao máximo os recursos de hardware disponíveis.

Assim como no Erlang, a arquitetura do Elixir é fundamentada na utilização de processos leves e isolados. Esses processos comunicam-se exclusivamente por meio do envio de mensagens, não compartilhando a memória entre si. É importante lembrar que um processo Elixir não é como um processo do sistema operacional,

pois a BEAM VM possui um agendador próprio (Juric, 2016). Dessa forma, é possível executar centenas de milhares de processos simultaneamente em uma única máquina, de maneira eficiente. A comunicação entre processos não se limita a um único hospedeiro, sendo possível a integração de múltiplos nós, o que viabiliza tanto a escalabilidade vertical quanto horizontal (Juric, 2016).

A tolerância a falhas é outro aspecto central na concepção da linguagem. Partes críticas do sistema são monitoradas e em caso de falha, reiniciadas de forma controlada por supervisores. Essa abordagem garante que falhas localizadas não comprometam o funcionamento de todo o sistema, contribuindo para a construção de arquiteturas robustas e altamente disponíveis (Elixir, 2025).

A linguagem adota o paradigma funcional como pilar principal de desenvolvimento. Tal paradigma enfatiza a imutabilidade dos dados, a composição de funções puras e o uso de pattern matching para o controle de fluxo, permitindo a criação de aplicações concisas, previsíveis e de fácil manutenção. A programação funcional no Elixir facilita o desenvolvimento de sistemas que precisam operar sob restrições específicas e previsibilidade de execução (Elixir, 2025).

Elixir também foi concebido para ser extensível. A linguagem permite que novos comportamentos e funcionalidades sejam adicionados naturalmente através da construção de DSLs (Domain-Specific Languages) e da sobrecarga controlada de estruturas nativas, aumentando a produtividade dos desenvolvedores em domínios específicos de aplicação (Elixir, 2025).

Finalmente, por ser executado sobre a BEAM, Elixir mantém compatibilidade total com a vasta gama de bibliotecas e ferramentas construídas para a plataforma Erlang. Essa compatibilidade permite que aplicações escritas em Elixir integrem-se diretamente ao ecossistema Erlang, aproveitando tecnologias maduras usadas em sistemas de alta demanda por empresas de grande porte (Juric, 2016).

2.7 ARQUITETURA BASEADA EM COMPONENTES

A arquitetura baseada em componentes tem como princípio a separação de responsabilidades em módulos e a programação orientada a interfaces. É um paradigma que tem como objetivo acelerar o desenvolvimento e promover

reusabilidade e manutenção no *software* pela composição de componentes como blocos para construção do sistema.

Um componente pode ser definido como uma unidade com uma única responsabilidade dentro do sistema. Ele possui interfaces que definem seu comportamento e contratos para interagir com componentes externos. Embora com uma única responsabilidade, pode englobar um escopo maior ou menor, a depender das necessidades do sistema.

Uma característica fundamental dessa arquitetura é a especificação dos componentes, sendo a interface de uso para componentes clientes e a definição abstrata para provedores. A composição dos componentes é outra característica essencial, sendo esta feita através da conexão de diferentes componentes via suas interfaces.

Sendo uma arquitetura simples e fundamentada em conceitos primordiais das linguagens que a implementa, é extremamente flexível, podendo oferecer alta, baixa ou nenhuma forma de encapsulamento (embora não seja recomendado expor a implementação, nem todos os sistemas possuem essa necessidade). Também permite acoplamento fraco ou forte entre os módulos, componentes possuírem estado ou não e finalmente, a substituição de módulos é possível, embora não facilitada por esta arquitetura.

O desenvolvimento envolve as etapas comuns do desenvolvimento de sistemas, como a elicitação de requisitos, design do sistema, desenvolvimento, teste e implantação, porém adiciona a possibilidade do desenvolvimento em total paralelismo, visto que idealmente um componente não depende dos detalhes de implementação de outro.

Ainda que Elixir não ofereça um suporte explícito/dedicado a este tipo de arquitetura, componentes são análogos a módulos Elixir que utilizam comportamentos previamente definidos com diretrizes como @callback. Essas mesmas diretrizes oferecem a infraestrutura de design e tempo de execução para os componentes. Embora simples, essa arquitetura permite um padrão de desenvolvimento eficaz, mas não resolve alguns dos problemas encontrados no desenvolvimento de software moderno.

3 ARQUITETURA

Este capítulo apresenta os principais aspectos estruturais da aplicação. Inicialmente, é descrito o domínio geral do sistema, seguido pelas tecnologias empregadas em sua implementação, pela arquitetura adotada e, por fim, uma justificativa para a escolha dessa arquitetura.

3.1 DOMÍNIO

Seguindo a proposta, a aplicação busca explorar uma possível solução para o atual cenário dos parques computacionais em escolas para o uso contínuo e local de algoritmos de aprendizado de máquina (estas públicas ou privadas, mas imagina-se soluções mais imediatas em escolas privadas). Tendo este problema em mente, podemos analisar melhor o domínio da aplicação, isto é, as possíveis causas do problema motivador, seus desdobramentos, a solução proposta e os motivos para tal.

Mesmo tendo produtos diversificados de aprendizado de máquina, existem motivos para a utilização local desta tecnologia, em escolas por exemplo. Desde a parte administrativa, criando modelos especializados para resolução de problemas ou previsão de outros mais, até na parte pedagógica, auxiliando professores no ensino de suas matérias específicas, ou no ensino de inteligência artificial como um todo.

A aplicação proposta é construída utilizando a linguagem Elixir. O escopo da aplicação se resume em: uma plataforma que possibilite o treinamento e utilização de modelos de aprendizado de máquinas utilizando uma rede de computadores para o processamento requerido em tais operações[1].

3.2 TECNOLOGIAS DA APLICAÇÃO

Posto o caráter exploratório da aplicação, contenta-se com a sua limitação, não tendo todas as funções imagináveis para uma aplicação do gênero.

Bibliotecas de funções são um modelo de compartilhamento de funções, procedimentos e estruturas de dados presentes em quase todas as linguagens de programação modernas. O compartilhamento dessas bibliotecas que as tornam dependências dentro do projeto acontecem da mais variada forma, mas a utilização dessas bibliotecas cessam aos desenvolvedores a tarefa de implementar a solução

[1] Código disponivel em https://github.com/w1lli4n/hive.git

de um mesmo problema mais de uma vez. Esta seção irá destrinchar as principais bibliotecas utilizadas na aplicação.

3.2.3 NX

Nx é uma biblioteca de tensores multidimensionais para Elixir, focada em compilações multi-estágio para CPU e GPU. Sua proposta é oferecer uma base robusta para computação numérica de alta performance no ecossistema Elixir, contemplando diferentes aspectos da construção de aplicações que demandam processamento intensivo.

A biblioteca introduz o conceito de tensores tipados e multidimensionais, permitindo que dados sejam representados em formatos como inteiros sem sinal (u2, u4, u8, u16, u32, u64), inteiros com sinal (s2, s4, s8, s16, s32, s64), números de ponto flutuante (f8, f16, f32, f64), floats de precisão reduzida para aplicações de aprendizado de máquina (bf16) e números complexos (c64, c128).

Outro destaque é o suporte a tensores nomeados, onde cada dimensão do tensor pode ser identificada por nomes simbólicos, favorecendo a clareza e a legibilidade dos códigos, além de reduzir a possibilidade de erros em manipulações de alta dimensão. Nx também disponibiliza diferenciação automática, ou autograd, que é a capacidade de computar gradientes automaticamente a partir de funções definidas sobre tensores. Essa funcionalidade é especialmente útil em simulações, otimizações e treinamento de modelos probabilísticos.

A biblioteca implementa ainda mecanismos de auto-vetorização e auto-batching, permitindo a transformação de códigos desenvolvidos para dimensões específicas em implementações eficientes capazes de operar sobre dimensões superiores de maneira paralela.

Outro conceito importante dentro da arquitetura do Nx é o de definições numéricas, conhecidas como defn. Este é um subconjunto da linguagem Elixir desenhado para ser compilado em múltiplos alvos, incluindo suporte para CPUs, GPUs e TPUs, o que é viabilizado por backends como EXLA, Torchx e Candlex.

Além disso, Nx possibilita a construção de pipelines numéricos encapsulados em Nx.Serving, que oferece funcionalidades como agrupamento, transmissão contínua de dados e particionamento automático. Estes servidores podem ser distribuídos entre múltiplos núcleos de CPU, dispositivos de GPU e até mesmo em clusters de máquinas.

O suporte a hooks amplia a capacidade de interação dos desenvolvedores com o ambiente de execução, permitindo o envio e o recebimento de dados em tempo de execução entre CPUs, GPUs e TPUs.

Complementariamente, a biblioteca fornece primitivas para operações de álgebra linear através do módulo Nx.LinAlg, abrangendo operações essenciais para aplicações científicas e de aprendizado de máquina.

Nx reúne, em um único ambiente e proposta de desenvolvimento, funcionalidades que são tradicionalmente fragmentadas em diversas bibliotecas do ecossistema Python, como Numpy, JAX, HuggingFace Pipelines e frameworks serving como TorchServing e TensorServing, proporcionando uma experiência unificada para o desenvolvedor Elixir.

3.2.4 AXON

O Axon foi projetado para oferecer abstrações que possibilitam uma integração fácil, mantendo a separação entre seus componentes. É possível utilizar qualquer uma das APIs de maneira independente, garantindo controle total sobre a criação e o treinamento de redes neurais. Axon utiliza Polaris como sua API de otimização.

Axon se define pela sua API funcional de baixo nível, composta por definições numéricas (defn), sobre a qual todas as outras APIs são construídas. Mais próximo do desenvolvimento existem a API de Criação de Modelos, essa sendo de alto nível que gerencia a inicialização e aplicação de modelos e a API para treinamento rápido de modelos, inspirada no PyTorch Ignite.

Os principais módulos são Axon.Activations, que contém funções de ativação aplicadas elemento a elemento; Axon.Initializers para funções para inicialização de parâmetros de modelos; Axon.Layers define as implementações de camadas comuns em redes neurais; Axon.Losses são funções de perda padrão utilizadas em treinamentos e Axon.Metrics mede acurácia, erro absoluto, precisão, entre outras.

Todos os métodos da API funcional são implementados utilizando definições numéricas (defn), o que permite que Axon seja acelerado por qualquer compilador ou backend do Nx. Além disso, os métodos podem ser compostos livremente com definições numéricas próprias, proporcionando grande flexibilidade.

Como as APIs de alto nível do Axon são construídas sobre a API funcional, todos os benefícios são herdados. Qualquer rede neural criada com Axon pode ser compilada just-in-time (JIT) ou ahead-of-time (AOT) usando qualquer backend do

Nx, ou ainda convertida para formatos de redes neurais de alto nível, como TensorFlow Lite e ONNX.

3.2.5 LIBCLUSTER

Libcluster é uma biblioteca que fornece mecanismos para a formação automática de *clusters* de nós Erlang, com suporte tanto a membros estáticos quanto dinâmicos.

Inicialmente fazia parte de uma projeto maior, chamado Rancher, mas foi desacoplado por razões de arquitetura e para permitir que as funcionalidades gerais da nova biblioteca, Libcluster, fossem utilizadas em outros contextos.

Dentre as principais funcionalidades, duas se destacam: formação automática de *clusters* (configurável para utilizar diferentes estratégias) e um conjunto próprio de funções para gerenciar a distribuição de trabalho em uma rede Erlang.

Embora existam diversas opções para configurar a formação de *clusters*, a utilizada neste projeto é a estratégia de *Gossip*, via multicast UDP. Essa estratégia se baseia na propagação de pacotes através da rede, utilizando o protocolo UDP (que tende a ser mais rápido que o TCP e é utilizado em situações semelhantes, como no DNS). Esses pacotes utilizam o multicast (de um nó para muitos) ou broadcast (de um nó para todos), sendo a escolha sujeita às liberdades de trânsito de pacotes pela rede. Cada nó envia um pacote e é escutado por outros nós (sendo estes muitos ou todos), caso a palavra secreta (*cookie* do Erlang) seja compartilhada pelo transmissor e receptor, é feita a conexão.

Desta forma, toda a complexidade na formação de uma rede é abstraída dentro desta biblioteca.

3.3 ARQUITETURA DA APLICAÇÃO

Garlan (2008) evidência a necessidade da arquitetura de software, está sendo proporcional ao tamanho e complexidade do software. A arquitetura de software está ligada a questões estruturais, desde protocolos de comunicação até a composição dos elementos de design.

Arquitetura de software é algo imprescindível a qualquer projeto de software moderno. Esta seção busca detalhar a arquitetura e design de uma aplicação que utiliza modelos de redes neurais Axon, seus componentes, interações e os princípios no seu desenvolvimento. Desta forma surge um entendimento em como a aplicação

está estrutura e em como essa estrutura busca sustentar os requisitos funcionais e não funcionais da mesma.

3.3.1 REQUISITO FUNCIONAIS E NÃO FUNCIONAIS

Os objetivos (ou requisitos) funcionais da aplicação, aqueles que ditam o comportamento da aplicação, são:

1. O sistema deve permitir ao usuário selecionar um modelo disponível.
2. O sistema deve receber uma requisição de previsão contendo os dados de entrada fornecidos pelo usuário.
3. O sistema deve distribuir a requisição para uma máquina disponível que contenha o modelo solicitado.
4. O sistema deve carregar o modelo selecionado, caso ainda não esteja carregado.
5. O sistema deve executar a previsão com base nos dados recebidos.
6. O sistema deve retornar o resultado da previsão ao usuário.
7. O sistema deve registrar logs das requisições realizadas (para fins de auditoria e análise).

Os objetivos (ou requisitos) não funcionais, aqueles que determinam os objetivos da aplicação para além do seu objetivo prático, são:

1. O sistema deve ser escalável horizontalmente, podendo adicionar ou remover máquinas sem interrupção.
2. O sistema deve garantir a integridade e a consistência das respostas de previsão.
3. O sistema deve ser tolerante a falhas, redirecionando requisições automaticamente caso uma máquina fique indisponível.

Além dos objetivos funcionais e não funcionais, os princípios de construção da aplicação são muito importantes, pois determinam o contínuo desenvolvimento da mesma. Modelos de aprendizado de máquina são artefatos que possuem dois estados: um modelo pode estar treinado ou não. O processo de treinamento e entrega do modelo aos usuários exige um fluxo bem determinado para um desenvolvimento mais seguro, este sendo determinado pelos princípios que serão seguidos e suportados pelo fluxo de desenvolvimento e entrega da aplicação.

Dentre este e outros motivos foram escolhidos como focos na construção da aplicação, os seguintes princípios:

1. Separação de responsabilidades. Cada módulo da aplicação possui uma única responsabilidade clara, dessa forma a manutenção, teste e evolução de cada parte do sistema é isolada.
2. Programação orientada a interfaces. Cada modelo implementa interfaces de comportamento, promovendo flexibilidade e intercambialidade, facilitando a adição de novos tipos de modelos ou componentes (este princípio é ainda mais relevante pela capacidade do Elixir de substituição de módulos sem desligar a aplicação).
3. Tolerância a falhas e resiliência. A arquitetura ponto-a-ponto naturalmente permite redistribuição das tarefas, promovendo escalabilidade e modularidade.
4. Design para escalabilidade horizontal. Cada nó pode executar uma tarefa diferente, permitindo o sistema crescer adicionando mais nós.

3.3.2 ESTILO DA ARQUITETURA

A arquitetura baseada em componentes foi escolhida devido a sua simplicidade e implementação quase direta na linguagem Elixir, em detrimento a outros padrões mais acessíveis a linguagens que possuem um paradigma de orientação a objetos, por exemplo.

3.3.3 COMPONENTES DA APLICAÇÃO

Esta seção descreve os componentes do sistema proposto.

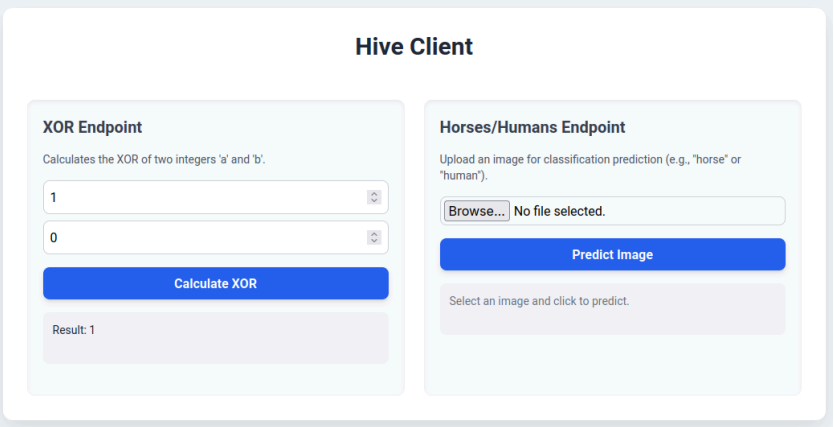
CLIENT

O componente Client é a forma que o usuário interage com a aplicação. Embora não precise existir dentro da aplicação, visto que a mesma oferece uma comunicação padronizada com sistemas externos, ela é proposta como uma forma padrão de interação.

Um cliente interno pode existir em variadas formas, para essa implementação, será uma interface de linha de comando.

Como proposta, utilizando a API HTTP disponibilizada pelo componente Router, eis um cliente HTML:

Figura 2: cliente HTML



Fonte: Autor

É possível também a simples leitura do JSON enviado pelo componente Router, como demonstra a requisição feita via o programa Curlie:

Figura 3: Cliente JSON

```
willian@opensuse:~> curlie -s "http://localhost:3177/xor?a=0&b=0"
{
    "result": 0
}
willian@opensuse:~> curlie -s "http://localhost:3177/xor?a=0&b=1"
{
    "result": 1
}
willian@opensuse:~> curlie -s "http://localhost:3177/xor?a=1&b=0"
{
    "result": 1
}
willian@opensuse:~> curlie -s "http://localhost:3177/xor?a=1&b=1"
{
    "result": 0
}
```

Fonte: Autor

SUPERVISOR

Supervisores são componentes Elixir ligados a OTP e exercem um papel fundamental em qualquer aplicação Elixir como parte da estratégia de tolerância a falhas, resiliência e robustez do sistema.

Em princípio, um Supervisor é um processo que monitora e gerencia outros processos de longa duração, controlando todo o seu ciclo de vida: a criação, monitoramento, interrupção (se necessário) e reiniciar (se necessário). É uma das bases da filosofia do Elixir de *“let it crash”*, em outras palavras, processos que apresentem problemas devem morrer para que um novo processo tome seu lugar,

poupando assim recursos computacionais e de desenvolvimento na tentativa de correção dos erros.

Dentro da aplicação proposta existem alguns supervisores:

- Supervisor da biblioteca Libcluster, para gerenciar os processos relacionados ao aparelhamento na rede.
- Supervisor da biblioteca Bandit (que atua através da biblioteca Plug) que controla o servidor web da aplicação.
- Supervisor do componente Controller, um processo que deve estar apto a receber as requisições provindas do Router.
- Supervisor do componente Model Trainer que atua como um processo estilo servidor dentro da aplicação.
- Supervisor de cargas de trabalho externas, Task.Supervisor, que gerencia as chamadas a funções do componente Fragment Trainer.

É importante ressaltar que supervisores são necessários apenas para processos com um ciclo de vida longo, em que a gestão deles é relevante para o funcionamento da aplicação.

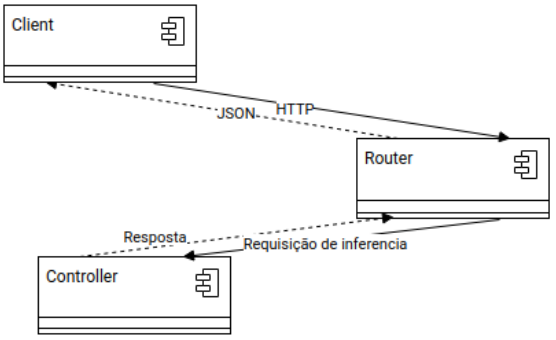
ROUTER

O componente Router funciona como a porta de entrada da aplicação. Ele recebe requisições HTTP e responde no formato JSON. Sua construção é bem simples, utilizando a biblioteca Plug e Cowboy, um servidor HTTP é iniciado no componente do Supervisor, e nele é configurado o uso do componente Router para definição das rotas HTTP do servidor.

Para cada modelo no sistema existe uma rota, em uma relação 1:1. A função do componente Router é receber a requisição e acionar o componente Controller correspondente. Utilizando a resposta provinda do Controller, o componente Router responde a requisição no formato JSON.

O componente Router é a única entrada para sistemas externos.

Figura 4: Fluxo do componente Router



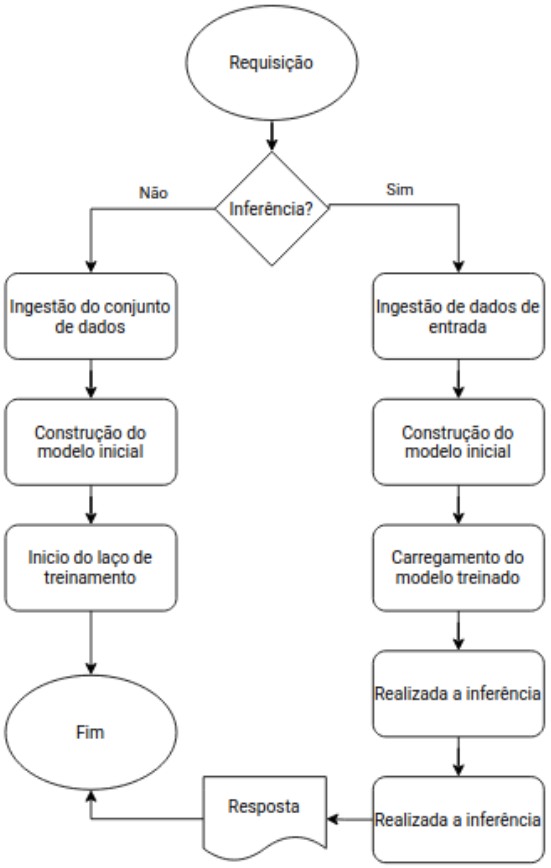
Fonte: Autor

CONTROLLER

O componente Controller é responsável pela orquestração da utilização dos modelos da aplicação. Primeiro é feito o carregamento do modelo em paralelo com a ingestão de dados que será realizada a inferência. Após isso, é realizada a inferência e através do componente de egestão de dados, o resultado será formatado no padrão esperado e retornado ao componente Router, onde será entregue ao cliente.

No jargão da arquitetura baseada em componentes, o Controller seria um componente de cola para o restante do sistema, tendo como única responsabilidade a orquestração do mesmo.

Figura 5: Fluxograma do componente Controller



Fonte: Autor

DATA INGESTION

O componente Data Ingestion tem como finalidade trazer dados externos para dentro do domínio, sendo crucial para a tradução de dados em formatos variados para um tensor Nx.

Seu funcionamento se baseia na seguinte interface:

- load_data/1: recebe uma fonte de dados e insere no sistema;
- process_data/2: processa os dados, normalmente aplicando transformações a ele. Esta função normalmente será repetida em diferentes padrões de argumento para diferentes transformações (uma característica do polimorfismo em Elixir);
- ingest_data/2: seguindo a mesma característica polimórfica de process_data/2, funciona como uma definição de um fluxo de processamento dos dados, podendo ter vários destes fluxos.

Este componente é utilizado em dois momentos na aplicação: durante o treinamento para processar os dados nos quais os modelos serão treinados, e durante a inferência para processar os dados necessários de entrada. É importante mencionar que lidar com diferentes tipos de dados não é trivial e bibliotecas externas normalmente são utilizadas para tal, como uma biblioteca de processamento de imagens, por exemplo.

MODEL

Neste componente é feito a definição do modelo Axon que será utilizado pela aplicação. Um módulo muito simples, com apenas duas funções: build_model/0, que retorna o modelo e run_inference/3, que executa uma inferência no modelo (este passado por parâmetro).

Vale ressaltar que em uma linguagem orientada a objetos, Model provavelmente seria uma classe e componentes como Model Trainer e Model Loader seriam apenas métodos dessa classe.

DATA EGESTION

Este componente é muito similar em funcionamento ao componente Data Ingestion, mas é diametralmente oposto. Dado um conjunto de tensores Nx, o componente irá processá-los para que tenha o formato desejado.

MODEL TRAINER

O componente Model Trainer é um tipo de componente identificado como “cola” na arquitetura baseada em componentes. Embora possa se dizer o mesmo do Controller, Model Trainer faz algo muito específico na aplicação, isto sendo o treinamento do modelo.

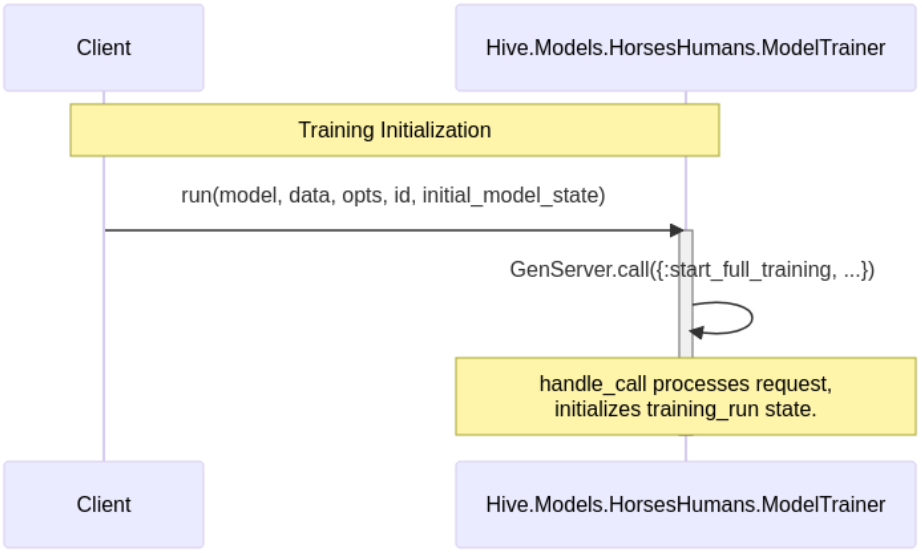
Baseado em um comportamento de servidor (definido no Elixir com a interface GenServer), este componente é de uso interno, criando/modificando um modelo de aprendizado de máquina. Sua utilização é imperativa ao desenvolvedor, que após implementar todos os componentes necessários para a criação de um novo modelo, ele utiliza as funções de Model Trainer para efetivamente treinar o modelo de forma distribuída na rede.

Por motivos de segurança este módulo não é exposto publicamente, sendo restrito ao uso da rede local de nós Erlang.

Seu funcionamento é o mais complexo dentre todos os componentes da aplicação, e tem o seguinte fluxo:

Primeiramente temos a inicialização:

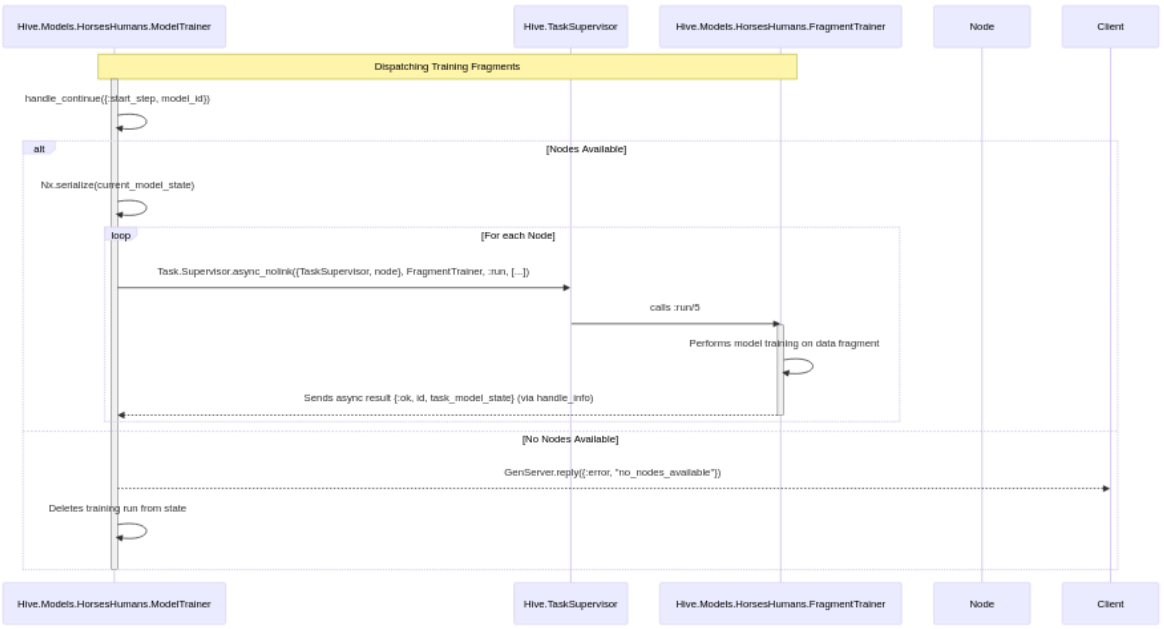
Figura 6 - Inicialização do laço de treinamento



Fonte: Autor

Iniciado o laço, temos a distribuição das tarefas pelos nós da rede:

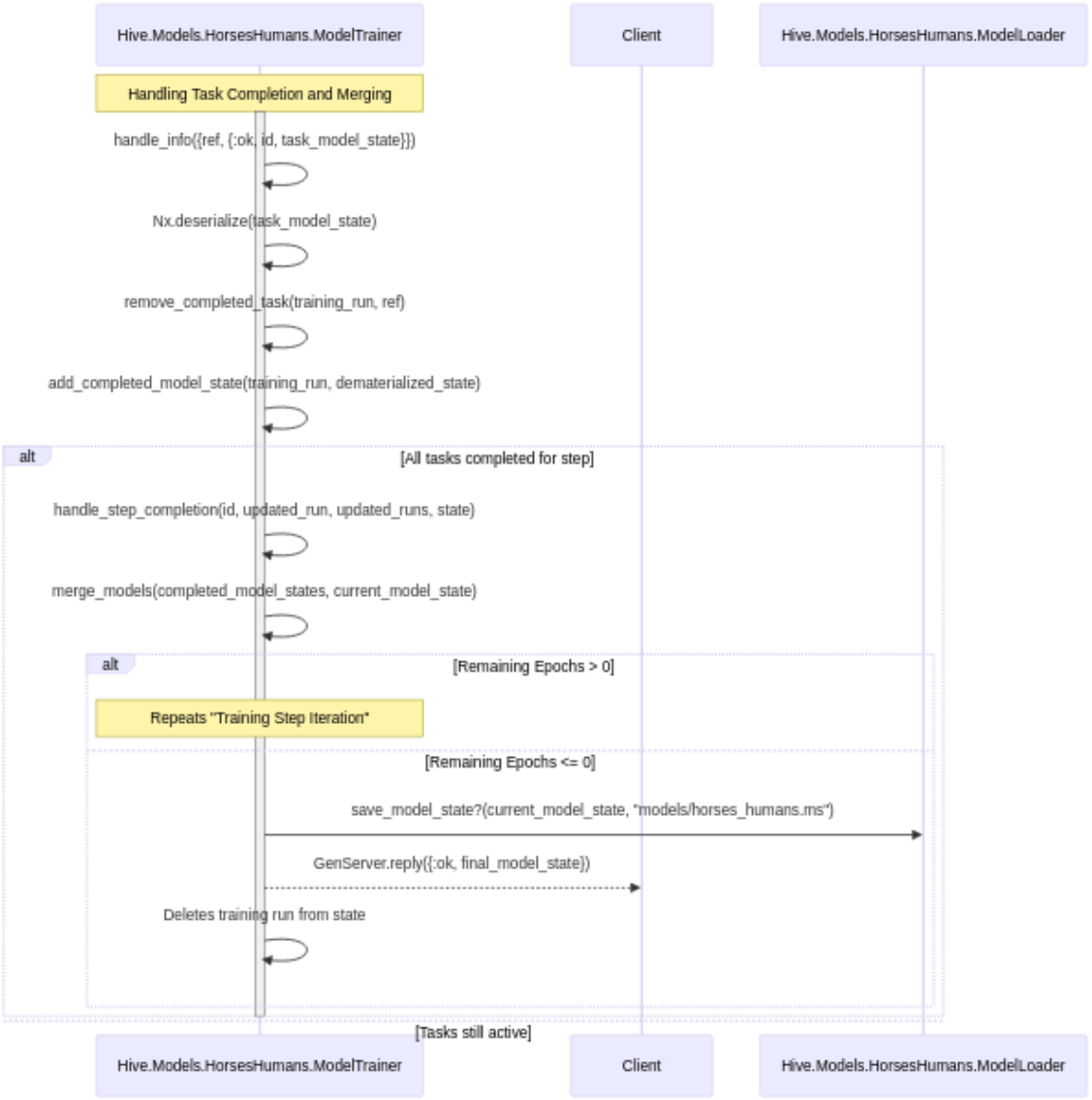
Figura 7 - Distribuição de tarefas



Fonte: Autor

Então é feito a junção dos modelos e o laço se repete até sua conclusão:

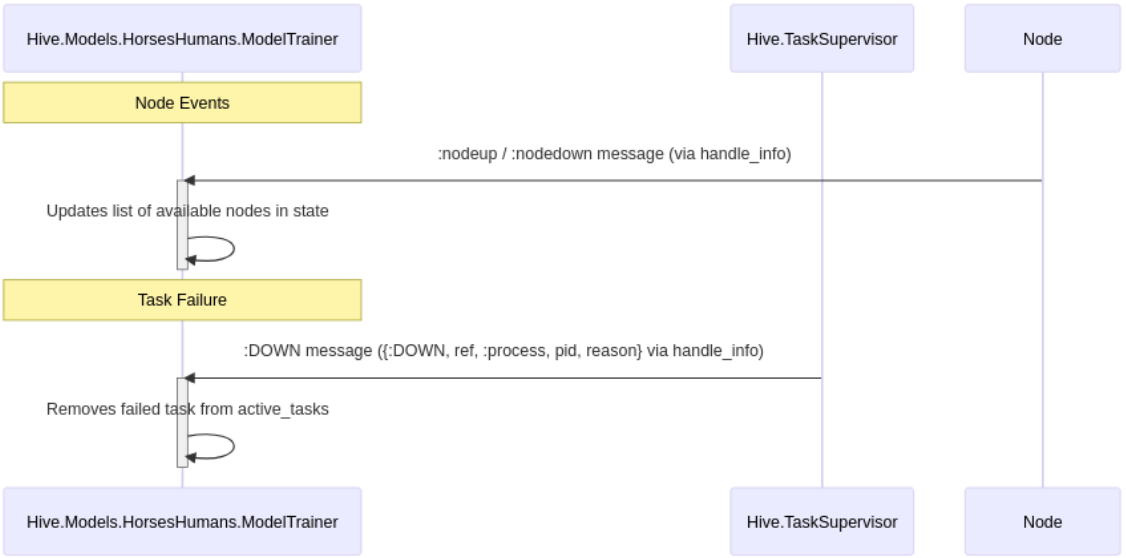
Figura 8 - Junção de modelos e repetição do laço de treinamento



Fonte: Autor

Por fim, o tratamento de erros que gera a tolerância a falhas nesse componente da aplicação:

Figura 9 - Tratamento de erros



Fonte: Autor

Quando chamado, todos os dados necessários são passados por meio de parâmetros.

O estado deste componente consiste em uma lista de nós disponíveis e os modelos que estão sendo treinados, além de contadores globais para o controle desse estado.

O fluxo de treinamento se inicia averiguando se o estado é coerente com a utilização. Primeira é feita a averiguação dos nós disponíveis, e depois se existem cargas de trabalho ativas, caso existam, a função retorna imediatamente.

Após isso é feita a junção dos estados do modelo, provindo das cargas de trabalho já realizadas e do argumento de estado inicial da função.

Com a reinicialização do estado uma nova série de cargas de trabalho se inicia e o estado é atualizado coerentemente.

Este fluxo acontece repetidamente até que o modelo esteja devidamente treinado e testado e o laço de repetição é controlado por uma função externa que se repete recursivamente (a função chamada pelo desenvolvedor).

Um exemplo do fluxo de chamada do treinamento seria:

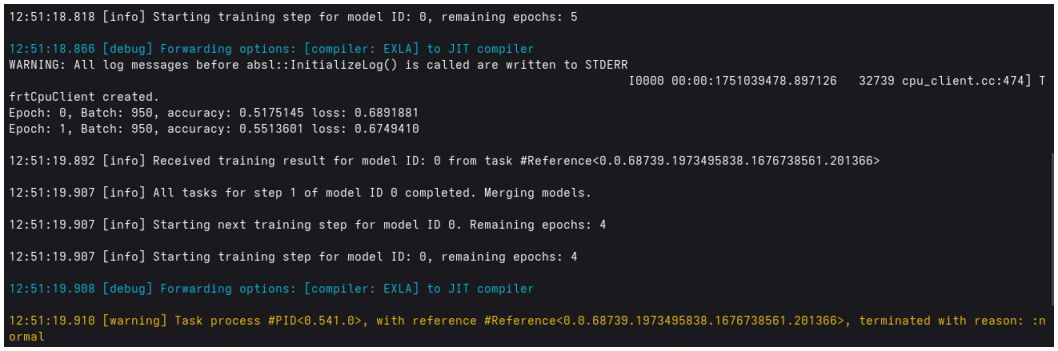
Figura 10: chamada do treinamento

```
iex(2)> Hive.Models.Xor.Controller.training_pipeline nil
```

Fonte: Autor

Após a chamada, mensagens de log são exibidas:

Figura 11: Logs do treinamento



Fonte: Autor

Finalmente, a mensagem de treinamento concluída é exibida, após o salvamento do modelo. Note que devido a natureza assíncrona da operação, a mensagem de recebimento da conclusão da tarefa ocorre após a mensagem de conclusão do treinamento:

Figura 12: Encerra-se o treinamento

Fonte: Autor

MODEL LOADER

Model Loader é responsável por carregar um modelo previamente treinado, e guardá-lo utilizando o mesmo formato. A princípio, não é definido uma forma para armazenar os modelos e nem para carregá-los para dentro do sistema posteriormente, sendo que os métodos de implementação podem variar.

Um exemplo de armazenamento totalmente distribuído seria uma tabela de hash distribuída, similar ao modelo que o protocolo bittorrent implementa (BITTORRENT, 2025). Outra forma seria um servidor centralizado e externo de modelos (seria necessário ser externo pois não é possível determinar se um nó participante da rede continuará sendo utilizado), ou mesmo todos os modelos estarem armazenados em todos os nós.

Para esta simples implementação do sistema, todos os nós terão uma cópia do modelo.

FRAGMENT TRAINER

O responsável pela carga de trabalho do treinamento é o componente Fragment Trainer. Este componente é utilizado como um *worker* assíncrono para as tarefas de treinamento e está no coração da distribuição desta tarefa.

Ao contrário do componente Model Trainer, Fragment Trainer simplesmente executa a função do loop de treinamento para o fragmento do conjunto de dados ou modelo designado a ele através dos parâmetros da função.

Este componente é supervisionado por um módulo do Elixir, Task.Supervisor, que gerencia trabalhos assíncronos, provendo uma API para este paradigma.

3.3.4 CONSIDERAÇÕES E DECISÕES DA ARQUITETURA

A escolha de uma arquitetura baseada em componentes é bastante favorecida pela natureza do Elixir e simplicidade da aplicação. Embora padrões de arquitetura como MVC ou Hexagonal sejam possíveis e frameworks e outros sistemas as utilizem, a implementação pode parecer antinatural, devido ao fato de Elixir ser uma linguagem funcional e a ampla documentação destes padrões se basearem em linguagem orientadas a objetos. Outro ponto é a forma que Elixir faz conexões entre os módulos e processos, trazendo de forma natural a utilização de interfaces e comunicação por mensageria.

Os princípios de separação de responsabilidades, tolerância a falhas e escalabilidade horizontal foram adotados seguindo os requisitos não funcionais. O uso de supervisores nos componentes críticos da aplicação garantem a robustez do sistema.

Parte da lógica foi abstraída através de bibliotecas fora do domínio da aplicação, como o servidor HTTP através das bibliotecas bandit e plug, e a conexão entre os nós através da biblioteca Libcluster.

De maneira geral, as decisões foram tomadas baseadas no escopo definido, este não muito grande, permitindo portanto que conceitos simples fossem adotados em detrimento a arquiteturas complexas e robustas, que não trariam muitas mais vantagens à aplicação.

4 RESULTADOS

Este capítulo apresenta os resultados da pesquisa.

4.1 TESTES

Os testes realizados foram utilizando dois modelos simples de redes neurais: um modelo emulando o portão lógico XOR e outro modelo de identificação de cavalos e humanos. Ambos os modelos foram bem sucedidos em suas previsões.

As configurações dos testes foram distribuindo o treinamento em 3 máquinas virtuais diferentes, cada uma utilizando o sistema operacional Ubuntu 24.04, com 1vCPU disponível e 4 GiB de memória RAM. As 3 máquinas estavam dispostas em uma mesma rede virtual e utilizaram o protocolo de multicast UDP para a comunicação através da biblioteca Libcluster e sua opção de Gossip.

Vale ressaltar que os objetivos da pesquisa não eram o de encontrar uma melhor performance no uso de algoritmos de aprendizado de máquina, como será retomado na seção 4.2 VALIDAÇÃO DA VIABILIDADE DA SOLUÇÃO, por isso a minimização desta linha de análise.

RESULTADO DAS FUNÇÕES DE FUSÃO DE MODELO

As Tabelas 2 e 3 apresentam as médias de três testes das métricas comparativas obtidas após a fusão dos modelos treinados, utilizando as funções de Média e SLERP.

Tabela 2: Métricas comparativas do modelo XOR

Métrica	**Treinamento local**	**Função: Média (2 nós)**	**Função: SLERP (2 nós)**
Acurácia	100%	91%	75%
Precisão	100%	85,7%	66,7%
Tempo	4s	6s	5s

Fonte: Autor

Tabela 3: Métricas comparativas do modelo Cavalos e Humanos

Métrica	**Treinamento local**	**Função: Média (2 nós)**	**Função: SLERP (2 nós)**
Acurácia	100%	83%	56%
Precisão	100%	78%	56%
Tempo	40min	23min	22min

Fonte: Autor

As Tabelas 2 e 3 revelam que a fusão dos modelos impactou negativamente as métricas de desempenho. A função de média resultou em uma redução significativa na acurácia e precisão para ambos os modelos. Para o modelo XOR, a acurácia caiu de 100% para 91% (média) e 75% (SLERP), e para o modelo Cavalos e Humanos, de 100% para 83% (média) e 56% (SLERP).

A função SLERP demonstrou um desempenho inferior à média. No modelo XOR, a degradação foi mais acentuada, enquanto no modelo Cavalos e Humanos, a SLERP se mostrou ineficaz, impossibilitando o aprendizado.

Embora a distribuição do treinamento tenha sido tecnicamente viável, os resultados indicam uma tendência de diminuição do tempo de treinamento, mas acompanhada de uma piora nas métricas de desempenho, como acurácia e precisão. Acredita-se que essa degradação seja consequência da distribuição em um nível de abstração elevado, diferente de plataformas existentes que operam em níveis mais baixos.

A principal conclusão é que a abordagem proposta, ao fundir os modelos, resulta em uma perda substancial de desempenho.

4.2 VALIDAÇÃO DA VIABILIDADE DA SOLUÇÃO

A seguir uma tabela da relação dos objetivos específicos e se os mesmos foram alcançados com a pesquisa:

Tabela 4: Objetivos específicos e seus resultados

Objetivo específico	**Resultado**
1. Estudar e aplicar conceitos de sistemas distribuídos na BEAM VM	Sistema implementado com Elixir e BEAM VM, utilizando GenServers, distribuição via Libcluster, etc.
2. Investigar a aplicação de aprendizado de máquinas em sistemas distribuídos	Treinamento de modelos simples distribuído com sucesso entre nós da rede.
3. Investigar bibliotecas de aprendizado de máquinas na linguagem Elixir	Uso de Nx e Axon para criação e treino de modelos.
4. Distribuir processos de aprendizado de máquinas em uma rede computacional	Com resultados insatisfatórios. Protótipo operando em ambiente com múltiplos nós, com execução de partes do treinamento de forma distribuída.

Fonte: Autor

Os objetivos de 1 a 3 focaram na pesquisa, estudo, exploração e aprendizado das tecnologias propostas, e isso foi bem sucedido.

O objetivo 4 apresentou resultados insatisfatórios. Embora a distribuição dos processos tenha sido bem sucedida, a mesma é injustificada. Com os modelos de teste, que apresentam conjuntos de dados de treinamento pequenos, não existe nenhuma vantagem na distribuição do sistema via paralelização de dados, isto porque quando reduzido o número de iteraçẽos por dados (estes também reduzidos), a acurácia do modelo se torna inaceitável. Quando aumentado a quantidade de iterações, a carga de trabalho se torna igual à que seria se o modelo fosse treinado localmente.

Por fim, o objetivo geral, desenvolver um sistema distribuído que utilize a capacidade computacional de uma rede para o aprendizado de máquinas, também foi alcançado.

4.3 LIMITAÇÕES ENCONTRADAS

As limitações de performance podem ser resolvidas com um certo nível de engenharia. Pontos cruciais como a ingestão de dados e a distribuição desses através de mensageria não devem ser ignorados.

Uma grande limitação, aliás, é justamente o uso de mensageria para a transferência de dados e estados. Uma das grandes vantagens da biblioteca Nx é utilizar LazyContainer para guardar seus tensores, o que significa que os mesmos são apenas referências em memória passadas de processo à processo. Quando precisa ser passado através de um nó, os tensores precisam ser materializados em memória. Um ponto focal da engenharia para otimização de pŕojetos de aprendizado distribuído é este, tendo várias possíveis soluções.

Outra grande limitação é o próprio ecossistema Elixir. No presente momento as evoluções estão em baixa e o estado atual está muito aquém de outros ecossistemas. As novidades não param no cenário de aprendizado de máquina, e ferramentas similares às propostas da aplicação apresentada são lançadas, como o Kubeflow (Kubeflow, 2025), que instrumentaliza a ferramenta kubernets para o uso com algoritmos de aprendizado de máquinas.

5 CONCLUSÕES

O trabalho alcançou o objetivo de desenvolver um sistema distribuído aliado a algoritmos de aprendizado de máquinas utilizando Elixir e suas tecnologias. Dotado de limitações significativas nos resultados de desempenho, como detalhado no Capítulo 4, propôs uma arquitetura funcional para a utilização de múltiplos modelos em um sistema.

Mesmo com um ainda limitado ecossistema de ferramentas para trabalho com aprendizado de máquinas em Elixir, muito pode ser feito se aproveitando da vasta tecnologia já implementada para Elixir, especialmente tratando-se de concorrência e tolerância a falhas. Contudo, conforme demonstrado nos testes, atendendo aos requisitos atuais, existem limitações na aplicação. Melhorias de otimização podem ser apontadas, como um melhor balanceamento de carga durante o treinamento, e com algoritmos de distribuição que lidem melhor com conjuntos de dados de treinamento pequenos.

Um melhor meio de interagir com o treinamento de novos modelos, e até modelos de aprendizado contínuo podem ser explorados também. Um último ponto a destacar seria a maneira de armazenar o módulo, podendo servir-se de métodos distribuídos como tabelas de hash distribuídas, por exemplo.

Com um caráter exploratório, a aplicação pode seguir por rumos distintos e evoluir de forma contínua, visando atender melhor os requisitos propostos e adquirir novos em seu desenvolvimento.

6 BIBLIOGRAFIA

BAJI, Toru. **GPU: the Biggest Key Processor for AI and Parallel Processing**. 2017

BERGMANN, Dave. **The most important AI trends in 2024**. 2023. Disponível em: https://www.ibm.com/think/insights/artificial-intelligence-trends Acesso em 1 nov. 2024

BHARDWAJ, Sushil & Jain, Leena & Jain, Sandeep. **Cloud Computing: A Study Of Infrastructure As A Service (IaaS)**. 2010

BITTORRENT. 2025. Disponível em: https://bittorrent.org/beps/bep_0003.html. Acessado em 3 jun. 2025

DEISENROTH, Marc Peter & Faisal, A. Aldo & Ong, Cheng Soon. **Mathematics for Machine Learning**. 2020

ELIXIR. 2025. Disponível em: https://elixir-lang.org/docs.html. Acessado em 1 nov. 2024

JURIC, Saša. **Elixir in Action**. 2016

KAUFMAN, Dora & Santaella, Lucia. **O papel dos algoritmos de inteligência artificial nas redes sociais**. 2020

KHAN, Asif. **Key Characteristics of a Container Orchestration Platform to Enable a Modern Application**. 2017

KUBEFLOW, 2025. disponivel em: https://www.kubeflow.org/. Acessado em 3 jan. 2025

MCQUATE, Sarah. **Q&A: UW researcher discusses just how much energy ChatGPT uses**. 2023. Disponível em:

https://www.washington.edu/news/2023/07/27/how-much-energy-does-chatgpt-use/. Acessado em 1 nov. 2024

MITTAL, S., & Vetter, J. S. **A Survey of Methods for Analyzing and Improving GPU Energy Efficiency**. 2014

MOORE, Susan. **What We Can Do With Machine Learning**. Acessado em: https://www.gartner.com/smarterwithgartner/what-we-can-do-with-machine-learning. 2016

RODRÍGUEZ-HAROA, Fernando & Freitag, Felix & Navarro, Leandro & Hernández-Sánchez, Efraín & Farías-Mendoza, Nicandro & Guerrero-Ibáñez, Juan Antonio & González-Potes, Apolinar. **A Summary Of Virtualization Techniques**. 2012

STAMFORD, Conn. **Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly $600 Billion in 2023**. 2023. Disponível em: https://www.gartner.com/en/newsroom/press-releases/2023-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023. Acessado em 1 nov. 2024

STEEN, Marten van. & Tanenbaum, Andrew S. **A brief introduction to distributed systems**. 2016

STENMAN, Erik. **The BEAM Book: Understanding the Erlang Runtime System.** 2025

THOKE, Virendra Dilip. **Theory Of Distributed Computing And Parallel Processing With Its Applications, Advantages And Disadvantages**. 2014

VERBRAEKEN, Joost & Wolting, Matthijs & Katzy, Jonathan & Kloppenburg, Jeroen & Verbelen, Tim & Rellermeyer, Jan S. **A Survey on Distributed Machine Learning**. 2020