Universidade Federal
do Rio de Janeiro
Escola Politécnica

PROGRAMAÇÃO FUNCIONAL APLICADA AO GERENCIAMENTO DE SISTEMAS DE APRENDIZADO DE MÁQUINA

Paulo Oliveira Lenzi Valente

Projeto de Graduação apresentado ao Curso de Engenharia Eletrônica e de Computação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Flávio Luís de Mello

Rio de Janeiro
Agosto de 2019

PROGRAMAÇÃO FUNCIONAL APLICADA AO GERENCIAMENTO DE SISTEMAS DE APRENDIZADO DE MÁQUINA

Paulo Oliveira Lenzi Valente

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA ELETRÔNICA E DE COMPUTAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO ELETRÔNICO E DE COMPUTAÇÃO

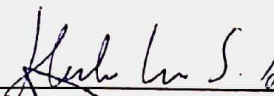
Autor:

Paulo Oliveira Lenzi Valente

Orientador:

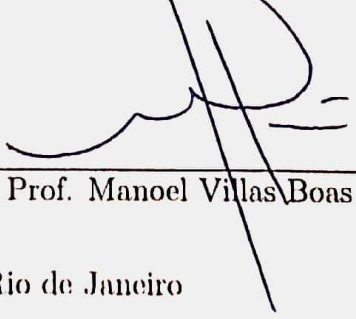
Prof. Flávio Luís de Mello, D. Sc.

Examinador:



Prof. Heraldo Luís Silveira de Almeida, DSc.

Examinador:



Prof. Manoel Villas Boas Junior, MSc.

Rio de Janeiro

Agosto de 2019

Declaração de Autoria e de Direitos

Eu, *Paulo Oliveira Lenzi Valente* CPF *168.508.797-31*, autor da monografia *Programação Funcional Aplicada ao Gerenciamento de Sistemas de Aprendizado de Máquina*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.
2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e idéias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.
3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.
4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.
5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.
6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.
7. Por ser verdade, firmo a presente declaração.

Paulo Valente

Paulo Oliveira Lenzi Valente

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Escola Politécnica - Departamento de Eletrônica e de Computação
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária
Rio de Janeiro - RJ CEP 21949-900

DEDICATÓRIA

À minha irmã, Helena, que apesar de estar conosco há pouco tempo, já é muito especial para mim.

AGRADECIMENTO

Dedico este trabalho ao povo brasileiro que contribuiu de forma significativa à minha formação e estada nesta Universidade. Este projeto é uma pequena forma de retribuir o investimento e confiança em mim depositados. Dedico também, à minha família – em especial, à minha mãe, Renata, ao meu pai, Guilherme, ao meu "paidrasto" Sergio Henrique, ao meu avô Marco Antônio Lenzi e minha madrasta Giselle, que acompanharam de perto este processo e me apoiaram incondicionalmente a todo instante. À minha namorada, Clara Menezes, sem a qual não teria conseguido finalizar este trabalho. Ao professor e amigo Luiz Wagner Biscainho, que me apresentou o curso de Engenharia Eletrônica e de Computação em 2012 e me acompanhou durante toda a jornada acadêmica. À Ana Beatriz e Alcio Braz, cujo acompanhamento foi imprescindível para minha formação acadêmica e pessoal. A todos os professores do Departamento de Eletrônica e de Computação, que ajudam a formar pesquisadores e profissionais excelentes. Aos meus amigos Adriano Fonseca, Arthur Cruz, Arthur Romeu, Arthur Trucco, Beatriz Mello, Bernardo Amorim, Bernardo Miranda, Camyla Romão, Carlos Lordelo, Carolina Margulies, Daniel Porto, Eduardo Alves, Fabiana Ferreira, Fábio Scelza, Flavia Orrico, Igor Quintanilha, Joana Luz, João Pedro Ramalho, João Victor Rezende, Lucas Lago, Lucas Maneschy, Luiz Carlos Penner, Maria Gabriella Felgas, Mariana Veloso, Marina Torres, Maurício Costa, Pedro Gil Couto, Pedro Lacerda, Rafael Oreiro, Rodrigo Jesus, Rodrigo Siqueira, Thomás Jagoda, Victor Barros e Vitor Cossetti, que são algumas das pessoas que me ajudaram a crescer durante os últimos cinco anos. Por fim, aos meus colegas de trabalho na Stone, com quem aprendo e cresço todos os dias.

RESUMO

Este trabalho trata da construção de um *framework* em programação funcional, voltado para o gerenciamento de sistemas de inteligência artificial. Para tal, se escolheu atacar uma frente de trabalho em Elixir e outra em Clojure. Como linguagens auxiliares, foram estudados C++, Java e Python. O projeto resultante deste estudo pode ser aplicado em sistemas de monitoramento em tempo real, como sistemas de câmera em prédios, sistemas de monitoramento de óleodutos e na segurança de fábricas.

Palavras-Chave: programação funcional, Elixir, aprendizado de máquina, sistemas concorrentes.

ABSTRACT

This project aims to build a functional programming framework that focuses on the management of artificial intelligence systems. There were two main work fronts: one in Elixir and the other in Clojure. C++, Java and Python were studied as auxiliary languages. The resulting project can be applied in real-time monitoring systems, such as CCTV systems, oil duct monitoring and factory security systems.

Key-words: functional programming, Elixir, machine learning, concurrent systems.

Sumário

1 Introdução **1**

1.1 Tema 1

1.2 Delimitação 1

1.3 Justificativa 2

1.4 Objetivos 2

1.5 Metodologia 2

1.6 Descrição 3

2 Fundamentação Teórica **4**

2.1 Paradigmas de Programação 4

2.1.1 Programação Orientada a Objetos 5

2.1.2 Paradigma Funcional 6

2.2 Máquinas Virtuais 6

2.3 *Machine Learning* 7

2.4 Clojure e a Plataforma Java 8

2.4.1 Lisp 8

2.4.2 JVM – *Java Virtual Machine* 9

2.4.3 Clojure 9

2.4.4 Bibliotecas Utilizadas em Clojure 10

2.5 Elixir e OTP 10

2.5.1 Erlang e OTP 11

2.5.2 NIF – *Native Implemented Function* 11

2.5.3 (*Erlang*) *Ports* 12

2.5.4 gRPC 12

2.5.5 Bibliotecas Utilizadas em Elixir 12

3 Avaliações **14**

3.1 Hardware utilizado 15

3.2 Experimentos Preliminares com Elixir 15

3.2.1 Elixir e C++ – NIFs 15

3.2.2 Elixir e C++ – gRPC 16

3.2.3 Elixir e Python – gRPC 16

3.2.4 Elixir e Python – *Ports* 16

3.2.5 Resultados e Análises 16

3.3 Experimentos com Classificação de Imagens 18

4 Produto Mínimo Viável **23**

4.1 Projeto do Produto Mínimo Viável 23

4.1.1 Sistema Gerenciador 23

4.1.2 Sistema Classificador 25

4.1.3 Sistema *Streamer* 26

4.2 Experimentos 26

5 Conclusão **30**

5.1 Trabalhos Futuros 30

Bibliografia **32**

Lista de Figuras

4.1 Funcionamento do Sistema Gerenciador 25
4.2 Tempo de Processamento do Lote (com desvio padrão) em Função do Número de *Streams* 28

Lista de Tabelas

3.1 Tempos de Execução (com desvio padrão) *vs* Quantidade de Dados Transferidos 18
3.2 Tempos de Classificação (com desvio padrão) *vs* Volume de Dados Processado 21
3.3 Tempos de Classificação em TensorFlow (com desvio padrão) *vs* Volume de Dados Processado 22

4.1 Atraso Acumulado em Função do Número de *Streams* 28
4.2 Quadros Perdidos em Função do Número de *Streams* 29

Capítulo 1

Introdução

1.1 Tema

Este trabalho trata da construção de sistemas de classificação de imagens em múltiplos *streams*, utilizando programação funcional [1] e aprendizado de máquina [2]. Neste sentido, o problema a ser resolvido é analisar diferentes *stacks* de tecnologia que utilizam linguagens funcionais para a construção de sistemas de gerenciamento sobre algum *backend* de aprendizado de máquina, este não necessariamente (no paradigma) funcional.

1.2 Delimitação

Os resultados deste trabalho podem ser aplicados a diversos sistemas de monitoramento de vídeo. Como casos de uso, se pode pensar em sistemas de segurança de vídeo. Em uma plataforma de petróleo, por exemplo, certas áreas têm acesso proibido para humanos. Então, seriam posicionadas camêras para monitorar tais áreas e, com um classificador de imagens apropriado, se poderia soar um alarme se alguma pessoa as adentrasse.

Outra aplicação similar seria o auxílio a porteiros e vigias, que precisam monitorar mosaicos de câmeras de segurança. Em shoppings, por exemplo, chegam a ser dezenas de *streams* distintos. Nesses casos, seria interessante que os vídeos contendo movimento fora do comum (i.e. pessoas se portando de forma agressiva) fossem destacados. Para isso, seria necessário um classificador mais especializado do

que o citado anteriormente.

1.3 Justificativa

O paradigma de programação funcional [1] tem ganhado cada vez mais espaço em sistemas distribuídos. Como ele tem como pilares não guardar estado e manter dados imutáveis, problemas clássicos como algoritmos de compartilhamento de recursos se tornam muito mais simples. Por exemplo, se torna muito mais difícil de se realizar uma leitura inconsistente em uma variável, visto que ela sempre está com um valor bem definido, e nunca em um estado intermediário.

Em paralelo, cada vez mais se tem utilizado algoritmos de inteligência artificial, principalmente a subárea de aprendizado de máquina, para as mais diversas aplicações. Neste campo, a subárea de *deep learning* tem ganhado cada vez mais força [3], apresentando resultados muitas vezes superiores aos de humanos [4, 5].

1.4 Objetivos

Se tem em vista a criação de um sistema concorrente de classificação de imagens e, portanto é necessário se decidir a tecnologia a ser utilizada. Para isso, o objetivo intermediário do trabalho é analisar diferentes *stacks* tecnológicos, que utilizam as linguagens funcionais Clojure [6] ou Elixir [7] , gerenciando um *backend* de aprendizado de máquina – C++ [8] ou Python [9], no caso de Elixir, ou Java [10], no caso de Clojure.

Após tal análise, o objetivo final do trabalho é a construção de um MVP (Produto Mínimo Viável, do inglês *Minimum Viable Product*) utilizando o *stack* escolhido na etapa anterior.

1.5 Metodologia

A primeira parte do trabalho realiza comparações com relação a custos de comunicação entre Elixir e C++ ou Python, com experimentos para medir o tempo para se obter uma resposta da linguagem de *backend*, sem processamento algum feito

nela. A partir desses resultados, se pode decidir sobre qual linguagem será utilizada para posterior desenvolvimento do sistema.

Já na segunda parte do trabalho, foram executados experimentos comparando o *stack* de Clojure com o de Elixir, em termos de velocidade de processamento em função da quantidade de dados a serem processados. Para isso, uma mesma rede neural foi construída nas duas plataformas, de modo que o algoritmo para análise executado fosse o mesmo. Nesta etapa, encontrou-se dificuldades para realizar experimentos utilizando processamento em GPU com Clojure e Java, enquanto que em Elixir e Python, foi possível transpor essa barreira.

Tendo isso em vista, a última etapa envolveu desenvolver um mínimo produto viável de um sistema de classificação de imagens em múltiplos *streams* de vídeo. O foco nesta etapa foi utilizar um único classificador central para processar os *streams*, em vez de um classificador para cada *stream*, de modo a se economizar em termos de tempo de máquina e, consequentemente, custo total do sistema.

1.6 Descrição

No capítulo 2, serão introduzidos os conceitos técnicos utilizados ao longo do trabalho. Isto inclui tanto os conceitos de computação utilizados quanto as bibliotecas de código e plataformas utilizadas.

O capítulo 3, por sua vez, desenvolve a análise das diferentes combinações de tecnologia para o projeto. São desenvolvidos experimentos para se analisar cada *stack*. Assim, culmina na escolha de qual será o conjunto utilizado no desenvolvimento do MVP.

Já no capitulo 4, são apresentados o processo de construção e estrutura do MVP. Aqui, também são desenvolvidos experimentos para se analisar o desempenho do sistema para classificação de imagens em múltiplos *streams* de vídeo, bem como são discutidos seus resultados.

Finalmente, o capítulo 5 contém a análise dos resultados do trabalho e propostas de melhorias para próximas iterações do sistema.

Capítulo 2

Fundamentação Teórica

Ambas as linguagens centrais do projeto, Clojure [6] e Elixir [7], são construídas sobre plataformas bem estabelecidas [11, 12, 13] (respectivamente, Java e Erlang). Contudo, cada uma traz uma sintaxe simplificada a seu modo. Clojure é um membro da família Lisp, que por definição tem sintaxe simples, e, por funcionar na JVM (*Java Virtual Machine*), pode invocar código Java nativamente. Por isso, foi escolhida como um dos objetos de estudo do trabalho.

Elixir, por sua vez, é uma linguagem da BEAM (*Bogdan's Erlang Abstract Machine*) e, portanto, apresenta as mesmas capacidades de resiliência a falhas, de distribuição e de processamento concorrente do Erlang [14, 15], além de poder invocar código Erlang nativamente. Contudo, para invocar códigos em outras linguagens, existem três métodos principais: `NIFs` (*Native Implemented Functions*) [16]; `Ports` [17] (que permitem a criação de um nó Erlang em outra linguagem, efetivamente criando um processo na BEAM) e; invocações por interoperabilidade (como estudo de caso, se utilizou o gRPC [18], por conta do suporte em Elixir [19]).

2.1 Paradigmas de Programação

Dentre os paradigmas de programação tradicionais, existem os paradigmas de orientação a objeto e de programação funcional. Nesta seção, se fará o contraponto entre os dois, no que se diz respeito à estrutura de código, às características de estado e de entrada-saída dos programas e às vantagens e desvantagens de suas respectivas aplicações.

2.1.1 Programação Orientada a Objetos

A orientação a objetos é, em geral, um dos paradigmas mais comuns no mercado. De acordo com o o GitHub, em 2018, as 10 linguagens mais populares foram, em ordem decrescente de popularidade: JavaScript, Java, Python, PHP, C++, C#, TypeScript, Shell, C e Ruby [20]. Destas, apenas Shell e C não apresentam características de orientação a objetos.

Uma linguagem é dita orientada a objetos quando apresenta o conceito de objetos, que são constructos de dados que também definem métodos específicos para operarem sobre os dados que contém [1]. Além disso, o paradigma é definido por muitos outros conceitos. Destes, é interessante para este trabalho discutir composição e herança.

Composição é um mecanismo que permite que objetos adquiram atributos (variáveis que representam seu estado) e métodos (funções, em um sentido amplo, que operam sobre seu estado) de outros objetos [1]. Herança, por sua vez, é um tipo de composição no qual existe uma hierarquia entre os objetos envolvidos, na qual o objeto que herda está sujeito a certas definições do objeto do qual herdou (i.e. quais parâmetros devem ser passados para certos métodos) [1].

A popularidade deste paradigma faz com que existam bibliotecas e *frameworks* para os mais diversos casos de uso. Outro ponto a favor da orientação a objetos é que se permite guardar o estado em memória, o que é favorável a certas aplicações. Por exemplo, um sistema gerenciador de fluxos de trabalho que pode guardar estado é razoavelmente simples de desenvolver, visto que cada ação tomada pode gerar a mutação diretamente no estado da aplicação. Em contraste, em um sistema sem estado, sempre se precisaria recalcular o estado a partir da série temporal de eventos que aconteceram [21].

Além disso, a capacidade de guardar estado fere a característica primária da programação funcional, no qual uma função sempre apresenta a mesma saída para a mesma entrada [22]. Isso torna o compartilhamento de recursos um dos pontos fracos do paradigma. Compartilhar recursos que guardam estado pode levar a problemas como a leitura inconsistente de um dado [23]. Isto, por sua vez, pode desencadear uma falha completa de um sistema. Então, se faz necessário aplicar algoritmos de compartilhamento, como variáveis de trava ou de vez, mutexes ou semáforos, de

modo a evitar que tais leituras inconsistentes ocorram [24].

2.1.2 Paradigma Funcional

O paradigma de programação funcional tem ganhado muita força em sistemas de processamento concorrente, devido às facilidades de compartilhamento de recursos que vêm com suas características chave. Além disso, muitas das linguagens que originalmente eram puramente orientadas a objeto, como Java e C++, tomaram emprestado funcionalidades de programação funcional (i.e. `map`, `reduce`; funções anônimas) [25, 26, 27]. Ainda nesse âmbito, duas linguagens funcionais Clojure e Scala – são implementadas sobre a plataforma Java.

Linguagens funcionais tem em seu núcleo a definição matemática de funções: para dado um conjunto de entradas, uma função sempre retorna a mesma saída [22]. Isso implica, também, que funções não guardam estado.

Tendo isso como base, as linguagens funcionais – de modo similar a orientação a objetos, onde todos os dados são, em última instância, objetos – definem que tudo são funções. Visto que todos os operandos são funções, é de se esperar que não guardem estado. Assim, é definido, também, que dados são imutáveis. Como sempre, certas linguagens relaxam uns ou outros aspectos da definição teórica do paradigma. Entretanto, os princípios básicos supracitados tendem a ser sempre mantidos.

As exceções, em geral, são devido a interfaces com o mundo real, à geração de números aleatórios ou então, à plataforma onde a linguagem se desenvolve. Por exemplo, Clojure e Scala, por serem compilados para o mesmo *bytecode* que Java, podem invocar código Java nativamente. Essas invocações, então, podem conter estado. O mesmo acontece quando se invoca uma `JNI` (Java Native Interface) [28]. Similarmente, invocações de `NIF`s ou `Port`s, em Elixir ou Erlang também podem fazer com que tais garantias sejam quebradas.

2.2 Máquinas Virtuais

Uma máquina virtual é uma camada de abstração de software, que permite que um único computador seja hospedeiro de múltiplos sistemas operacionais dis-

tintos [24]. Neste trabalho, se observa duas aplicações distintas deste conceito, uma para linguagens de programação, e outra para computação na nuvem.

As linguagens Java e Erlang implementam máquinas virtuais, não para virtualizar hardware, mas para abstrair toda a plataforma hospedeira. Com isso, é possível que o mesmo código possa ser executado em diferentes plataformas, desde que haja uma implementação da respectiva máquinas virtual para a linguagem [29, 30]. Além disso, essa virtualização permite que haja uma etapa de compilação de código-fonte para a linguagem de máquina da máquina virtual, de modo que essa mesma linguagem de máquina possa ser reaproveitada, também, em diferentes plataformas [31, 32, 33].

Fora isso, se pode utilizar máquinas virtuais na nuvem. Um provedor de máquinas virtuais, como a AWS (*Amazon Web Services*, ver a seção 3.1 do capítulo 3), fornece diferentes configurações de máquina, de modo que o usuário possa utilizar a nuvem para atender a demandas específicas de CPU, RAM e GPU, sem precisar lidar diretamente com custos de manutenção de infraestrutura e hardware [34].

2.3 *Machine Learning*

O campo de *machine learning*, ou aprendizado de máquina, é uma forma de se implementar algoritmos de inteligência artificial. Diferentemente da programação convencional, em que o desenvolvedor fornece dados de entrada e a lógica para obter uma saída, em *machine learning*, o desenvolvedor fornece a entrada e a saída correspondente), de modo que o algoritmo possa inferir a lógica necessária para obter tal relação. Atualmente, a maioria dos algoritmos exige que seja fornecida grande quantidade de dados para inferência do modelo. Para este trabalho, se escolheu trabalhar com *deep learning*, que é uma classe dos algoritmos supracitados que emprega redes neurais com múltiplas camadas ocultas [35].

Redes neurais, por sua vez, são uma classe de algoritmos de aprendizado de máquina que realiza inferência a partir do conceito de neurônios interconectados. Neurônios, neste contexto, são nós em um grafo direcionado de cálculos. O valor de cada neurônio é o resultado da aplicação de uma função de ativação (que pode ser linear ou não) sobre uma combinação linear de todos os neurônios com arestas

que apontam para ele, com pesos representados em cada aresta [35]. Usualmente, se utiliza o conceito de camadas, para especificar um destes grafos, que tem uma quantidade pré-determinada pelo desenvolvedor de nós de entrada e de saída. Quando uma rede neural é composta de mais de duas camadas, as camadas intermediárias são chamadas de camadas ocultas [35].

2.4 Clojure e a Plataforma Java

Uma das frentes de trabalho neste projeto funcionou sobre a plataforma Java. Esta é construída sobre uma máquina virtual chamada de JVM (*Java Virtual Machine*), que foi originalmente concebida como motor para a linguagem orientada a objetos Java. Entretanto, ao longo do tempo surgiram outras linguagens que também funcionam sobre a mesma plataforma e que, por consequência de compartilharem a plataforma, podem invocar código Java como se invocaria um código nativo delas. Destas linguagens, se escolheu Clojure, por ser uma opção, a princípio, com razoável suporte para redes neurais.

2.4.1 Lisp

Linguagens do tipo Lisp apresentam uma estrutura em comum, onde praticamente tudo o que se faz é implementado com listas no formato `(funcao arg1 arg2 ...)`. Além disso, possuem como uma característica intrínseca a meta-programação – quando parte do código manipula código – devido, também a essa estrutura sintática simples. Esta estrutura também é a origem do nome Lisp, que é uma contração para List Processing [36].

A sintaxe de uma linguagem Lisp tem como elemento primário as expressões simbólicas – *S-expressions*. Estas, são compostas por símbolos atômicos, ou por uma lista de expressões simbólicas entre parêntesis (i.e. `(+ (* A B) (+ C D))` para simbolizar a expressão matemática (A * B) + (C + D)). Sua capacidade de meta-programação vem da possibilidade de converter uma *S-expression* em uma lista, através de uma sintaxe de *quoting*, que evita a compilação do código em sua declaração, retornando uma máquina de estados que pode ser manipulada como qualquer outra estrutura de dados.

2.4.2 JVM – *Java Virtual Machine*

A JVM é o motor da plataforma Java, e é definida por uma especificação detalhada, o que permite a existência de múltiplas implementações totalmente intercambiáveis [29]. Portanto, qualquer linguagem que possa ser compilada para sua linguagem de máquina pode ser utilizada nela. Neste projeto, ela é utilizada como substrato para a linguagem Clojure, um dos objetos de estudo.

A JVM oferece suporte às características da linguagem Java, como tipos de dados inteiros, booleanos e de ponto-flutuante, classes, objetos e métodos para invocação e instanciação de objetos. Além disso, é especificado um padrão para exceções em tempo de execução, que podem se propagar levando a uma parada completa do programa. Quando uma exceção ocorre, é comum que se veja um *stacktrace*, que simboliza a lista de chamadas de código que levou à ocorrência da exceção.

O programador interage com a JVM através de dois tipos de arquivos. O primeiro deles é o `.java`, que contém o código em texto plano, legível para humanos. O código (pré-)compilado, por sua vez, é composto de `bytecode` da JVM, e é uma forma de enviar para o usuário programas prontos para a execução.

2.4.3 Clojure

A linguagem Clojure é um Lisp e, portanto é uma linguagem funcional. Seu principal diferencial é que funciona sobre a JVM. Assim, consegue invocar código Java como código nativo, o que permite que se utilize bibliotecas Java em programas escritos em Clojure [6]. Além disso, é uma linguagem pensada para a criação de aplicações de processamento concorrente, seguindo um modelo de concorrência por threads [37].

Esse modelo de concorrência é alimentado pelo sistema transacional de memória em software [37], que permite que se mude estado no programa mas, quando isso acontece, o estado se mantenha consistente. Assim, não é necessário que o programador precise pensar em travas e em outros mecanismos de compartilhamento de recursos. Outra característica que Clojure apresenta é o polimorfismo a tempo de execução para funções. Isto significa que uma mesma declaração de função pode ser direcionada a um corpo de execução diferente, de acordo com os argumentos

passados (i.e. `(fn ([:a :b] "ab") ([x y] (str x y)))`).

2.4.4 Bibliotecas Utilizadas em Clojure

Além da biblioteca padrão da linguagem, algumas bibliotecas foram utilizadas para o desenvolvimento do projeto. Três delas, destacadas abaixo, lidam com redes neurais e com *benchmarking* de código.

2.4.4.1 Criterium

A biblioteca Criterium [38], é uma biblioteca de código aberto escrita em Clojure, aplicada para coleta de estatísticas sobre o tempo de execução de um programa ou função. Portanto, é utilizada para a extração de parâmetros de comparação entre diferentes implementações de código.

2.4.4.2 Cortex

Cortex [39] é uma biblioteca de redes neurais, que fornece uma API similar à do Keras/TensorFlow em Python [40]. É implementada totalmente em Clojure, o que faz com que não haja dependências com a linguagem Java diretamente.

2.4.4.3 TensorFlow (Java)

TensorFlow é uma biblioteca originalmente escrita em Python [41]. Contudo, existem APIs para múltiplas linguagens, inclusive Java. Apesar de não ter uma API tão vasta quanto à original, é uma alternativa para a utilização de redes neurais em Clojure, visto que utiliza chamadas Java de código nativo [28] e, portanto, pode atingir uma eficência similar à de código de mais baixo nível. Além disso, se pode importar modelos treinados TensorFlow a partir de outra linguagem, como Python. Assim, o treinamento e construção do modelo pode ser feito em Python, e apenas a classificação, em Java.

2.5 Elixir e OTP

Elixir é uma linguagem de programação brasileira, criada por José Valim, e lançada em 2011 [7]. Sua sintaxe e filosofia pegam muito emprestado de Ruby.

Entretanto, Elixir funciona sobre a BEAM – a máquina virtual da linguagem Erlang. Portanto, se pode invocar código Erlang nativamente a partir de Elixir, bem como Clojure pode invocar código Java.

2.5.1 Erlang e OTP

Erlang é uma linguagem de programação funcional, criada na Ericsson na década de 1980 [42]. Seu intuito é fornecer como funcionalidade *built-in* um modelo de programação concorrente com tolerância a falhas. O modelo de concorrência Erlang é através de uma variação do modelo de atores [43], no qual os programas são construídos como processos na BEAM. Os processos Erlang podem, também, ser supervisionados. Um supervisor é um processo cuja única função é monitorar os processos conectados a ele, de modo que um *crash* em um processo filho não se propague não-intencionalmente por todo o sistema. Assim, um processo supervisionado pode ser reiniciado sem quebrar todo o sistema [44]. Por ser construída na BEAM, Elixir também apresenta essas mesmas características e funcionalidades. OTP (*Open Telecom Platform*, Plataforma Aberta de Telecomunicações), por sua vez, é um conjunto de bibliotecas Erlang que é voltado para a construção de software aplicado a telecomunicações [45]. Certos padrões para a construção de um processo *long-running* são abstraídos pela OTP, de modo que parte do código *boilerplate* seja eliminada.

2.5.2 NIF – *Native Implemented Function*

Uma NIF é uma função que pode ser invocada a partir da BEAM, mas cuja implementação é feita em uma linguagem externa à BEAM. Por exemplo, uma função que necessita de mais alto desempenho ou que necessita de uma funcionalidade específica de outra linguagem, pode ser conectada à BEAM através das bibliotecas fornecidas para se implementar uma NIF [16]. Entretanto, uma NIF que levantar uma exceção do sistema destrói todo o processo da BEAM. Isso elimina uma das principais vantagens de se utilizar uma linguagem BEAM. Por isso, deve-se minimizar o tempo que um programa passa executando uma NIF e, sempre que possível, utilizar programação defensiva, até mesmo com blocos `try-catch` para evitar que tais exceções derrubem toda a máquina virtual.

2.5.3 (*Erlang*) *Ports*

Outra forma de se conectar à BEAM a partir de um programa externo é utilizando *ports* [46]. Um *port* é um programa que se conecta à BEAM de modo que ela o enxergue como um processo nativo. A comunicação de interoperabilidade se dá por meio de `stdin` e `stdout`, então nesse caso pode haver um *overhead* de comunicação a ser vencido. Além disso, em contraste a NIFs, um *crash* dentro de um *port* pode ser monitorado por um supervisor Erlang. Portanto, em sistemas que precisam apresentar resiliência a falhar, a utilização de um *port* pode ser mais interessante que uma NIF.

2.5.4 gRPC

A terceira forma de interoperabilidade com a BEAM utilizada neste trabalho foi o gRPC (*gRPC Remote Procedure Calls*), um padrão de comunicação HTTP para implementação de RPC (chamadas de procedimento remotas) [18]. O padrão gRPC utiliza *Protocol Buffers* [47] para definir as interfaces de comunicação e o conteúdo das mensagens trocadas entre os sistemas envolvidos. Sua principal vantagem é que exitem implementações para muitas linguagens, e que, dadas as especificações da interface, existem programas que geram uma aplicação-cliente ou uma aplicação-servidor básicas automaticamente.

2.5.5 Bibliotecas Utilizadas em Elixir

2.5.5.1 ErlPort

A biblioteca ErlPort [48] apresenta uma interface de alto nível para a implementação de ports em Erlang. Para este projeto, foi utilizada para implementar a comunicação entre Elixir e Python.

2.5.5.2 TensorFlow (Python)

Como dito anteriormente, em 2.4.4.3, a principal implementação de TensorFlow é feita em Python. Visto que não há implementação de TensorFlow na BEAM, se utilizou a versão de Python por conta do amplo suporte online [41]. Além disso,

fora da aplicação direta com Elixir, se utilizou a versão Python de TensorFlow para criação e treinamento de uma rede neural (ver o capítulo 3).

Capítulo 3

Avaliações

Para a construção de um sistema de software, é necessário que se decida o conjunto de tecnologias a ser utilizado. Por isso, este capítulo é dedicado a comparações de desempenho ou de *overhead* de interoperabilidade entre diferentes combinações entre linguagens de programação.

O projeto apresenta certas limitações, decorrentes do fato de visar ao processamento de vídeo em tempo real. A primeira delas está relacionada à taxas de transferência de dados, mas como o escopo está limitado a casos de uso em rede local, isto não é um limitante. A segunda, que motivou os experimentos descritos neste capítulo, é o tempo máximo de processamento para cada quadro. Vídeos são comumente gravados a taxas de 24 ou 30 FPS (quadros por segundo, do Inglês *frames per second*). Quanto maior a taxa, menor o tempo disponível para processar cada quadro.

Para tratar vídeos a uma taxa de 30 FPS, é necessário que cada quadro seja processado em $\frac{1}{30}$ segundo, ou $33,3$ milissegundos. Os testes a seguir visam estudar diferentes tecnologias e como elas se comportam de acordo com essa limitação. Foi apenas considerado o tempo dentro do sistema, pois se pode considerar o atraso de comunicação com o mundo externo constante, que resulta em um atraso absoluto no vídeo transmitido. Se o atraso absoluto total for pequeno o suficiente, de acordo com critérios a determinar pela aplicação, ele não é problemático.

Para se utilizar diferentes linguagens de programação em um sistema, é necessário que haja uma forma estabelecida de interoperabilidade entre as duas. Para o caso de Clojure e Java, a interoperabilidade é nativa, devido a ambas serem com-

piladas para o *bytecode* da JVM e a Clojure oferecer sintaxe para executar código Java. Já no caso de Elixir, foram escolhidas três formas distintas: NIFs, *Ports*, e gRPC (ver as seções 2.5.2, 2.5.3 e 2.5.4).

3.1 Hardware utilizado

Devido a falta de disponibilidade de uma GPU localmente, foi provisionada uma máquina virtual na AWS (*Amazon Web Services*), serviço de computação em nuvem da Amazon. As configurações da máquina virtual utilizada são:

- Processador: 8 CPUs Intel Xeon E5-2686 v4 (Broadwell) virtuais, com *clock* de 2.7 GHz.
- Memória RAM: 61 GiB
- GPU: Tesla v100, com 16GB de memória dedicada,

3.2 Experimentos Preliminares com Elixir

Os primeiros experimentos executados tiveram em vista a avaliação do *overhead* de comunicação entre Elixir e C++, através de NIFs e de gRPC, e entre Elixir e Python, através de Ports. Para isso, se mediu o tempo para se enviar quantidades de bytes de Elixir para a outra linguagem, e para receber um dado de tamanho fixo como resposta.

3.2.1 Elixir e C++ – NIFs

Essa instância do experimento consiste em utilizar a linguagem Elixir como parte fundamental da aplicação, tendo uma função implementada em baixo nível por C++. Isso é necessário porque Elixir funciona sobre a máquina virtual Erlang, a BEAM e, além disso, não é uma linguagem bem preparada para cálculos de ponto flutuante.

Portanto, o processamento pesado poderia ser feito em C++, através de NIFs (*Native Implemented Functions*, ver 2.5.2), interface fornecida pela BEAM para

implementar funções de baixo nível. Contudo, isso incorre em riscos para a BEAM, visto que exceções levantadas em uma NIF quebram a BEAM por completo.

Isto acontece porque quando as exceções acontecem dentro da BEAM, é ela quem decide como lidar com a exceção. Em geral, isso leva à morte do processo dentro do sistema virtualizado. De modo similar, uma exceção que acontece no sistema hospedeiro faz com que o processo que a causou seja morto pelo sistema operacional. Assim, é criada uma situação em que um processo dentro da BEAM pode levar à morte da máquina virtual como um todo.

3.2.2 Elixir e C++ – gRPC

Outra alternativa analisada foi combinar Elixir e C++ através de gRPC (*gRPC Remote Procedure Calls*, ver 2.5.4). A vantagem em relação a NIFs seria evitar os riscos apresentados por possíveis exceções. Contudo, é de se esperar que o *overhead* de comunicação seja maior, visto que gRPC funciona em cima de HTTP.

3.2.3 Elixir e Python – gRPC

A fim de poder usufruir das bibliotecas de aprendizado de máquina em Python, também se estudou a comunicação através de gRPC entre Elixir e Python. Valem também as vantagens e desvantagens comentadas com relação a gRPC em C++.

3.2.4 Elixir e Python – *Ports*

A quarta e última alternativa foi a comunicação com Python via Ports (ver 2.5.3). Assim como gRPC, *Ports* não apresentam riscos à estabilidade da BEAM e, como NIFs, são uma forma de invocar código externo como se fosse nativo. Isso se dá através da criação de um processo BEAM na linguagem Python, através da biblioteca ErlPort.

3.2.5 Resultados e Análises

A Tabela 3.1 mostra os resultados dos experimentos descritos anteriormente. Se pode perceber que o desempenho entre NIFs e *Ports* é da mesma ordem de

grandeza, enquanto o de gRPC com C++ passa a ser melhor com uma quantidade de dados de 4kB, nos experimentos. O desempenho de gRPC com Python, por sua vez, passa a ser melhor que o de NIFs com 5 kB e que o de *Ports* com 7 kB. Em todos os casos, o desempenho de gRPC com C++ apresenta entre 0.3 ms a 0.6 ms de vantagem em relação ao de gRPC com Python.

Enquanto NIFs podem incorrer em riscos graves para a estabilidade da BEAM, gRPC faz com que seja necessária a manutenção de dois processos distintos no computador hospedeiro. Com isso, não se pode utilizar da capacidade de auto-regeneração de processos da BEAM. Este quesito, entretanto, seria respeitado por *Ports*. Entretanto, para volumes maiores, as duas modalidades de gRPC (em C++ e em Python) se equiparam, a menos de uma diferença constante da ordem de 0.5 ms. Assim, se decidiu utilizar Elixir e Python, com *gRPC* como método de interoperabilidade. Deste modo, se pode usufruir das bibliotecas de aprendizado de máquina e a facilidade de desenvolvimento em Python, mantendo a estabilidade (pois evita exceções em funções NIF) e a escalabilidade da BEAM (pois mantém um baixo overhead de comunicação e não trava os processos BEAM).

Tabela 3.1: Tempos de Execução (com desvio padrão) *vs* Quantidade de Dados Transferidos

Quantidade de Dados	Elixir e C++		Elixir e Python	
	NIFs (*ms*)	gRPC (*ms*)	gRPC (*ms*)	*Erlang Ports* (*ms*)
1 kB	$\mathbf{0,159 \pm 0,116}$	0.593 ± 1.774	0.898 ± 1.803	$0,204 \pm 0,008$
2 kB	$\mathbf{0,282 \pm 0,086}$	0.362 ± 0.039	0.637 ± 0.049	$0,299 \pm 0,007$
4 kB	$0,554 \pm 0,146$	$\mathbf{0.360 \pm 0.023}$	0.646 ± 0.055	$0,578 \pm 0,038$
5 kB	$0,765 \pm 0,211$	$\mathbf{0.375 \pm 0.033}$	0.647 ± 0.052	$0,606 \pm 0,008$
7 kB	$0,945 \pm 0,261$	$\mathbf{0.378 \pm 0.029}$	0.657 ± 0.052	$0,923 \pm 0,101$
8 kB	$1,220 \pm 0,344$	$\mathbf{0.401 \pm 0.046}$	0.664 ± 0.053	$0,858 \pm 0,013$
10 kB	$1,448 \pm 0,309$	$\mathbf{0.417 \pm 0.028}$	0.678 ± 0.052	$1,147 \pm 0,106$
20 kB	$2,814 \pm 0,786$	$\mathbf{0.498 \pm 0.046}$	0.737 ± 0.057	$2,450 \pm 0,259$
28 kB	$3,397 \pm 0,685$	$\mathbf{0.542 \pm 0.124}$	0.781 ± 0.058	$3,449 \pm 0,270$
40 kB	$4,679 \pm 0,910$	$\mathbf{0.613 \pm 0.067}$	1.161 ± 2.239	$3,749 \pm 0,063$
80 kB	$8,886 \pm 0,653$	$\mathbf{0.846 \pm 0.078}$	1.429 ± 2.155	$10,067 \pm 0,813$
120 kB	$13,653 \pm 1,133$	$\mathbf{1.087 \pm 0.077}$	1.376 ± 0.066	$12,343 \pm 0,218$
160 kB	$19,406 \pm 1,542$	$\mathbf{1.309 \pm 0.084}$	1.650 ± 0.116	$18,010 \pm 1,176$
200 kB	$25,271 \pm 2,293$	$\mathbf{1.571 \pm 0.110}$	1.906 ± 0.140	$25,631 \pm 2,316$
240 kB	$29,609 \pm 3,733$	$\mathbf{2.020 \pm 0.092}$	2.596 ± 0.114	$31,087 \pm 2,804$
280 kB	$39,624 \pm 4,086$	$\mathbf{1.952 \pm 0.099}$	2.527 ± 0.123	$29,623 \pm 0,465$
320 kB	$45,826 \pm 4,092$	$\mathbf{2.102 \pm 0.104}$	2.757 ± 0.128	$40,813 \pm 3,701$

3.3 Experimentos com Classificação de Imagens

Esta seção trata da análise de desempenho para classificação de imagens. Como definido na seção anterior, uma das tecnologias analisadas é a combinação entre Elixir e Python via Ports. Por outro lado, se escolheu Clojure como alternativa, visto que pode utilizar código Java nativamente e, com isso, se elimina a necessidade de se medir o *overhead* de comunicação entre Clojure e Java.

Ambos os experimentos utilizam as mesmas imagens para classificação. Não

será avaliado o desempenho das redes, apenas o tempo de execução. Além disso, a classificação é dada através de uma rede neural. A topologia foi retirada de um dos arquivos de exemplo da biblioteca Cortex [49], e apresenta a seguinte sequência de camadas:

- Camada Aplainadora
- Camada Convolucional 2D
  - Filtros: 20, 5 x 5
  - Passo: 1 vertical e 1 horizontal
  - MaxPooling 2D
    - Tamanho do pool: 2 x 2
    - Passo: 2 vertical e 2 horizontal
  - Descarte de 10%.
  - Ativação ReLU
- Camada Convolucional 2D
  - Filtros: 50, 5 x 5
  - Passo: 1 vertical e 1 horizontal
  - MaxPooling 2D
    - Tamanho do pool: 2 x 2
    - Passo: 2 vertical e 2 horizontal
- Camada de Normalização em Lote, com *momentum* 0.9 e *epsilon* 0.0001
- Camada Aplainadora
- Camada Densa
  - Neurônios: 1000,
  - Descarte de 50%.
  - Ativação ReLU com *threshold* de 0.0001 e inclinação negativa de 0.9
- Camada Densa

– Neurônios: 2,

– Ativação Softmax

Os parâmetros da rede descrita acima são dados de acordo com os nomes definidos na documentação de TensorFlow, versão 1.13. As camadas Aplainadora, Convolucional 2D e de Normalização em Lote são nomeadas lá, respectivamente, como *Flatten*, *Conv2D* e *BatchNormalization*.

Os experimentos foram executados de diferentes formas: Clojure utilizando a biblioteca Cortex, sobre CPU; Clojure com TensorFlow de Java, sobre CPU; Elixir e Python com TensorFlow, sobre CPU; Elixir e Python com TensorFlow, sobre GPU. Em todos os casos, os experimentos foram alimentados com imagens de 25px por 25px e de 50px por 50px, e em cada um dos casos, em lotes de 1, 2, 5, 7, e 10 imagens. Os tamanhos das imagens incluem os headers PNG do arquivo, devido a haver pré-processamento dos arquivos.

A Tabela 3.2 apresenta a comparação entre as bibliotecas Cortex e TensorFlow. Os valores em negrito apontam as linhas provindas de um lote de imagens de 50px por 50px. Se pode observar que conforme o número de imagens aumenta, o *speed-up* diminui. Também se observa que conforme o número de imagens aumenta, há uma queda inicial do tempo de execução, o que pode estar correlacionado com algum overhead de *setup* inicial do experimento.

Entretanto, em termos absolutos, enquanto a biblioteca Cortex não consegue atender os requisitos (33ms de processamento por quadro) para se processar 30 quadros por segundo, TensorFlow consegue atendê-los em todas as linhas da tabela. Além disso, como será observado na Tabela 3.3, o desempenho pode melhorar através do processamento via GPU.

Tabela 3.2: Tempos de Classificação (com desvio padrão) *vs* Volume de Dados Processado

Dimensões da Imagem	Tamanho do Lote	Volume de Dados	Clojure		*Speed-up* ($\frac{t_{Cortex}}{t_{TensorFlow}}$)
			Cortex – CPU (ms)	TensorFlow – CPU (ms)	
25 x 25	1	1 kB	$39,10 \pm 2,06$	$0,83 \pm 0,16$	$47,1$
25 x 25	2	2 kB	$28,22 \pm 1,20$	$1,10 \pm 0,97$	$25,6$
25 x 25	5	5 kB	$21,79 \pm 1,68$	$1,79 \pm 0,32$	$12,17$
25 x 25	7	7 kB	$17,89 \pm 1,29$	$2,47 \pm 0,54$	$7,24$
25 x 25	10	10 kB	$19,54 \pm 1,12$	$3,10 \pm 0,36$	$6,3$
50 x 50	1	4 kB	$93,00 \pm 4,78$	$2,44 \pm 0,25$	$38,1$
50 x 50	2	8 kB	$70,99 \pm 3,68$	$4,23 \pm 0,18$	$16,7$
50 x 50	5	20 kB	$59,34 \pm 3,38$	$7,75 \pm 0,31$	$7,6$
50 x 50	7	28 kB	$50,37 \pm 2,46$	$10,75 \pm 0,54$	$4,6$
50 x 50	10	40 kB	$58,33 \pm 3,05$	$15,50 \pm 0,55$	$3,7$

A Tabela 3.3 apresenta os resultados para o mesmo experimento que a tabela anterior – classificação de imagens através da rede neural descrita no começo da seção 3.3. Se pode perceber uma diferença de desempenho muito grande entre o experimento executado em Clojure e o experimento executado em Python. Esta diferença, que é maior para volumes de dados menores, pode ser atribuída ao *overhead* de comunicação imposto pelo uso de gRPC nos experimentos em Python.

Entretanto, ao se executar os experimentos, uma barreira importante foi atingida no uso de Clojure. O suporte de TensorFlow para Clojure é precário, de modo que não foi possível executar o experimento utilizando GPU.

Com relação aos experimentos em Python, se pode perceber que, apesar do *overhead* de comunicação, o desempenho para com GPU apresenta vantagens para maiores volumes de dados (a partir de 20 kB na tabela). O desempenho em CPU apresenta desvantagem em todas as instâncias, mesmo se descontarmos um atraso médio de comunicação de cerca de 1 ms (devido aos resultados obtidos na Tabela 3.1).

Tabela 3.3: Tempos de Classificação em TensorFlow (com desvio padrão) *vs* Volume de Dados Processado

Dimensões da Imagem	Tamanho do Lote	Volume de Dados	Clojure	Elixir/Python	
			CPU (ms)	CPU (ms)	GPU (ms)
25 x 25	1	1 kB	$0,833 \pm 0,016$	3.957 ± 0.173	$4,693 \pm 0,635$
25 x 25	2	2 kB	$1,102 \pm 0,096$	4.255 ± 0.046	$4,781 \pm 0,459$
25 x 25	5	5 kB	$1,790 \pm 0,320$	5.026 ± 0.066	$5,215 \pm 0,428$
25 x 25	7	7 kB	$2,466 \pm 0,540$	5.429 ± 0.074	$5,625 \pm 0,580$
25 x 25	10	10 kB	$3,102 \pm 0,355$	6.066 ± 0.111	$6,179 \pm 0,696$
50 x 50	1	4 kB	4.823 ± 0.115	$9,353 \pm 1,444$	$4,845 \pm 0,676$
50 x 50	2	8 kB	5.096 ± 0.076	$12,028 \pm 1,674$	$4,995 \pm 0,483$
50 x 50	5	20 kB	6.630 ± 0.113	$19,321 \pm 4,101$	$5,709 \pm 0,575$
50 x 50	7	28 kB	7.672 ± 0.078	$21,637 \pm 2,261$	$6,176 \pm 0,671$
50 x 50	10	40 kB	9.277 ± 0.103	$29,937 \pm 7,548$	$6,879 \pm 0,805$

Capítulo 4

Produto Mínimo Viável

Levando em conta os dados obtidos nos experimentos do capítulo 3 e as dificuldades encontradas em sua respectiva execução, se escolheu criar o produto mínimo viável (MVP) do projeto em Elixir, utilizando gRPC como plataforma de comunicação com Python (para se classificar em TensorFlow).

Para isso, será criado um sistema auxiliar, também escrito em Elixir, que servirá como um *streamer* substituto para câmeras de segurança.

A primeira seção do capítulo descreverá em detalhes a implementação e o modelo de classificação utilizado, enquanto os trechos seguintes descreverão os experimentos e os resultados obtidos.

4.1 Projeto do Produto Mínimo Viável

Para a implementação do sistema, se escolheu a divisão em três subsistemas que serão executados em paralelo (podendo, inclusive, ser executados em computadores distintos). Os três sistemas foram denominados: o (Sistema) Gerenciador; o *Streamer*; e o Classificador. A comunicação entre estes sistemas se dá através do protocolo gRPC [18]. O código de interface, tanto para o cliente quanto para o servidor, é gerado automaticamente a partir de uma especificação Protobuf [47]

4.1.1 Sistema Gerenciador

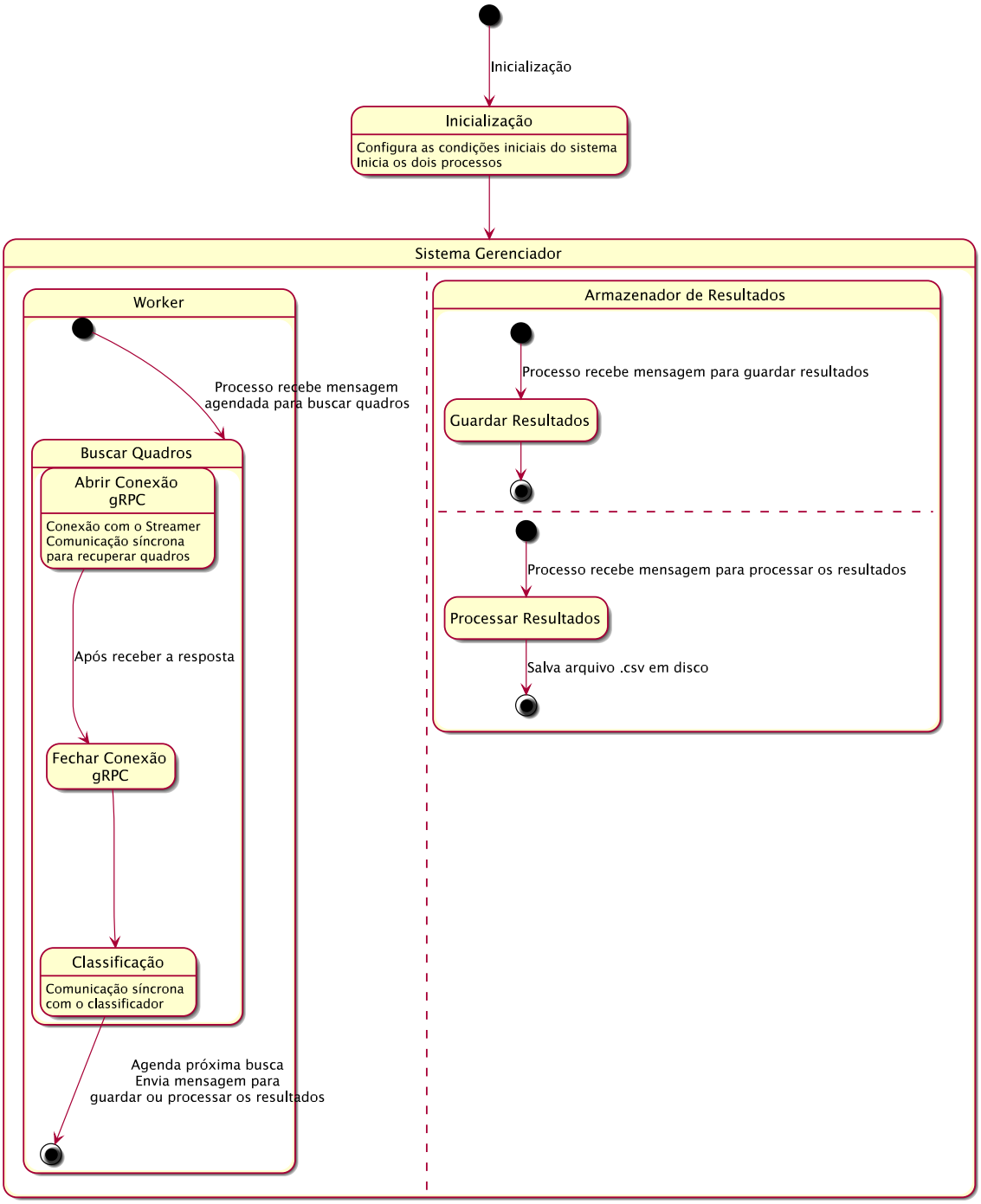
Este é o subsistema que comanda os outros. Para fins dos experimentos descritos a seguir, na seção 4.2, o sistema é implementado em Elixir e é composto

por dois processos BEAM principais, implementados via GenServer: o *Worker* e o Armazenador de Resultados. O primeiro é o processo principal do diagrama na Figura 4.1. Seu papel é recuperar os próximos quadros de cada *stream* do *Streamer* e enviá-los em lote para o Classificador. Após a classificação, os resultados são enviados para o Armazenador de Resultados.

O Armazenador de Resultados, por sua vez, apenas guarda em memória toda a sequência de resultados obtida da classificação, incluindo o tempo que se levou para processar cada lote. Além disso, após o *Worker* atingir sua condição de parada, ele processa os resultados e grava arquivos em formato CSV (*Comma-Separated Values*) para análise posterior.

A Figura 4.1, a seguir, descreve em alto nível a máquina de estados obedecida por cada um destes processos.

Figura 4.1: Funcionamento do Sistema Gerenciador



4.1.2 Sistema Classificador

Este sistema é implementado em Python, e tem como único propósito, receber mensagens gRPC contendo o lote de imagens, processá-las e retornar o resultado do processamento, também via gRPC. A classificação de imagens é realizada através de *deep learning*, em TensorFlow.

4.1.3 Sistema *Streamer*

Finalmente, o *Streamer* é também implementado em Elixir. Durante a inicialização, ele carrega em memória todos os quadros, separados por *stream*, a partir do banco de dados. O conteúdo dos arquivos que contem cada quadro é persistido no banco de dados durante o setup inicial do projeto, de modo a facilitar sua manipulação através da biblioteca de ORM Ecto.

4.2 Experimentos

Os experimentos descritos a seguir utilizam como vídeo a ser processado parte do *dataset* CAVIAR [50]. Os vídeos utilizados são apresentados a uma taxa de 25 quadros por segundo, com dimensões de 384 px por 288 px. O modelo de classificação utilizado foi o SSD MobileNET V1, retirado de [51].

A primeira série de experimentos realizada tem como objetivo medir a taxa de processamento, em lotes de quadros por segundo, do sistema, bem como sua variação conforme a quantidade de *streams* monitorados aumenta. Cada série executada inclui 1, 2 e 3 *streams*, respectivamente.

O experimento consiste em executar o sistema, consumindo o(s) *stream(s)* em questão até o final de um deles. Cada iteração inicia um contador antes de abrir a primeira conexão gRPC e finaliza o contador após persistir os resultados no Armazenador de Resultados. Deste modo, é possível medir o tempo necessário para executar toda a sequência de obtenção e processamento de um lote de quadros.

Os resultados da primeira série são apresentados de duas formas. A primeira mostra a sequência de tempos de processamento obtida e a taxa média de processamento. É desejável que esta seja maior do que a taxa de amostragem original dos *streams*. A segunda apresentação é uma tabela que compara o tempo corrido após o processamento do último quadro com seu *timestamp* (i.e. o quadro de número 25 foi processado após 2 segundos, mas deveria ter sido processado após 1 segundo, com um atraso acumulado de 1 segundo). Neste caso, o ideal é que o tempo após o processamento seja no máximo igual ao *timestamp*.

Para a segunda série de experimentos, o sistema incluirá a possibilidade de perder quadros do *stream*, implementada através de uma configuração no *Streamer*.

Como resultado, será calculada uma tabela que relaciona a quantidade de *streams* com o número de quadros perdidos. Idealmente, é desejável se perder zero quadros.

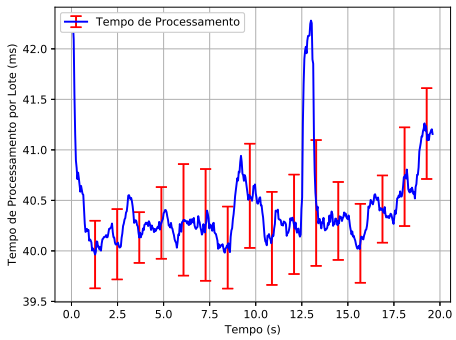
A Figura 4.2 apresenta os gráficos de tempo de processamento médio para cada lote, ao longo do tempo. As escalas dos gráficos foram ajustadas para mostrar em mais detalhes as variações dinâmicas em cada um deles. É interessante observar que apenas na Figura 4.2a o requesito de 25 quadros por segundo, o que se traduz em um tempo de processamento por quadro de 40 ms é próximo de ser atingido. Além disso, devido ao processamento se dar em lotes, o crescimento do tempo de processamento em função do número de *streams* não é linear.

Já na Tabela 4.1, são mostrados os atrasos acumulados ao final do processamento do *stream*. Como nesta bateria de experimentos o *stream* é processado sem perda de quadros, conforme o tempo de processamento por lote aumenta, cresce o atraso cumulativo.

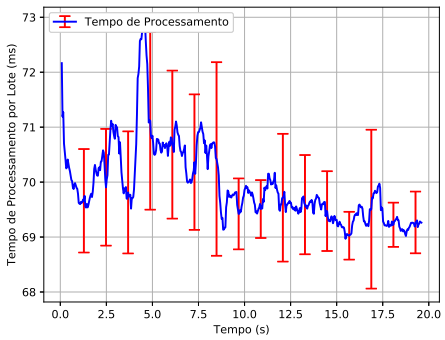
Finalmente, a Tabela 4.2 apresenta os resultados da segunda bateria de experimentos. Nela, estão enumeradas as quantidades de quadros perdidos por cada *stream* (todos os *streams* perdem a mesma quantidade de quadros). Ao lado, também estão listadas as taxas de processamento em lotes por segundo, onde cada lote é composto de tantos quadros quanto se tem *streams*.

Vale ressaltar que há discrepância com a primeira bateria de testes, visto que a taxa calculada de lotes por segundo para 1 *stream* é de 15,3, enquanto que o atraso acumulado calculado na bateria anterior não condiz com essa taxa. Essa discrepância pode ser explicada por algum fator extrínseco ao software, como escalonamento de vCPUs na AWS. Por ter sido o primeiro experimento a ser executado, é possível que ainda não houvesse a mesma quantidade de vCPUs reservados para a máquina virtual.

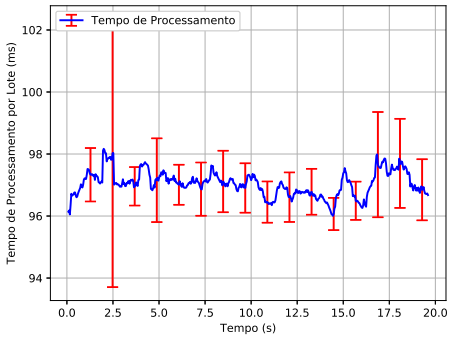
Figura 4.2: Tempo de Processamento do Lote (com desvio padrão) em Função do Número de *Streams*



(a) 1 *stream*



(b) 2 *streams*



(c) 3 *streams*

Tabela 4.1: Atraso Acumulado em Função do Número de *Streams*

Número de *streams*	Atraso Acumulado (s)
1	$0,15$
2	$14,38$
3	$27,81$

Tabela 4.2: Quadros Perdidos em Função do Número de *Streams*

Número de *streams*	Quadros Perdidos por cada *Stream*	Lotes por Segundo
1	185	$15,3$
2	246	$12,25$
3	326	$8,25$

Capítulo 5

Conclusão

Os experimentos no final do Capítulo 4 mostram que o MVP não atendeu às expectativas iniciais do projeto, de processar múltiplos *streams* à mesma taxa de amostragem original dos vídeos. Entretanto, é importante ressaltar que é possível utilizá-lo em aplicações que não necessitem do processamento de todos os quadros presentes, como se pode perceber nos experimentos supracitados. Neles, se obteve uma taxa de processamento de 8 quadros por segundo para cada *stream*, o que atende aplicações que necessitem de uma taxa igual ou mais lenta que essa.

Ademais, o projeto resultou em um sistema modular, que é configurável tanto em termos de qual classificador utilizar, quanto em qual linguagem essa classificação será implementada. Resultados intermediários como os observados no Capítulo 3 sugerem que o gargalo do projeto está no subsistema classificador. Então, ao se levar em conta o fato de que a comunicação com o classificador é realizada através de gRPC, é de se pensar que seja possível implementar a classificação em uma linguagem mais rápida que Python.

O código-fonte do MVP se encontra no GitHub [52].

5.1 Trabalhos Futuros

Os resultados do trabalho sugerem que ainda há múltiplos caminhos a serem explorados. Dentre eles:

- Utilizar mais de uma cópia do *backend* de classificação, de modo a atingir a taxa de quadros desejada.

- Permitir que múltiplos processos *workers* interajam de forma independente com classificadores distintos. Assim, seria possível que o sistema monitorasse aspectos distintos de uma mesma fonte, ou mesmo tipos de fontes distintos (i.e. um processo classificaria texto enquanto o outro classificaria imagem)
- Explorar outros protocolos de interoperabilidade, como o Apache Thrift[53]
- Utilizar TensorFlow.js [54] para realizar a classificação diretamente na camada de visualização Web (através de Phoenix [55])

Finalmente, por ser implementado primariamente em Elixir, é possível de se paralelizar o sistema gerenciador ainda mais, inclusive o distribuindo em múltiplas máquinas hospedeiras, de modo que a capacidade computacional seja expandida. Além disso, também é possível que o próprio sistema forneça visualização e interação através de uma interface Web, que é um dos pontos fortes da linguagem. Portanto, apesar de não atender as expectativas iniciais, a estrutura de software utilizada é uma prova de conceito que tem seu lugar em aplicações práticas.

Referências Bibliográficas

[1] WEBBER, A. B., *Modern Programming Languages: A Practical Introduction*. 2 ed. Franklin, Beedle & Assoc., 2010.

[2] RUSSELL, S. J., NORVIG, P., *Artificial Intelligence: A Modern Approach, 3rd Edition*, Global Edition. 3 ed. Pearson Education, 2016.

[3] DAR, P., "Popular Machine Learning Applications and Use Cases in our Daily Life", https://www.analyticsvidhya.com/blog/2019/07/ultimate-list-popular-machine-learning-use-cases, 2019, (Acesso em 05 Agosto 2019).

[4] CHENNUPATI, S., *Hierarchical Decomposition of Large Deep Networks*. Ph.D. dissertation, Rochester Institute of Technology, 05 2016.

[5] KAGGLE, "Histopathologic Cancer Detection", https://www.kaggle.com/c/histopathologic-cancer-detection/overview/description, 2019, (Acesso em 05 Agosto 2019).

[6] "Clojure", https://clojure.org, (Acesso em 05 Agosto 2019).

[7] "Elixir", https://elixir-lang.org, (Acesso em 05 Agosto 2019).

[8] "C++", http://www.cplusplus.org/, (Acesso em 05 Agosto 2019).

[9] "Python", https://python.org, (Acesso em 05 Agosto 2019).

[10] "Java", https://www.java.com, (Acesso em 05 Agosto 2019).

[11] "Github Java Repositories", https://github.com/search?q=java&type=Repositories, (Acesso em 05 Agosto 2019).

[12] "Github Erlang Repositories", https://github.com/search?q=erlang&type=Repositories, (Acesso em 05 Agosto 2019).

[13] “Github Elixir Repositories”, https://github.com/search?q=elixir&type=Repositories, (Acesso em 05 Agosto 2019).

[14] “Erlang – Errors”, http://erlang.org/doc/reference_manual/errors.html, (Acesso em 05 Agosto 2019).

[15] “Erlang – Processes”, http://erlang.org/doc/reference_manual/processes.html, (Acesso em 05 Agosto 2019).

[16] “Erlang – NIFs”, http://erlang.org/doc/tutorial/nif.html, (Acesso em 05 Agosto 2019).

[17] “Erlang – Ports”, http://erlang.org/doc/reference_manual/ports.html, (Acesso em 05 Agosto 2019).

[18] “gRPC”, https://grpc.io, (Acesso em 05 Agosto 2019).

[19] “Elixir gRPC”, https://github.com/elixir-grpc/grpc, (Acesso em 05 Agosto 2019).

[20] “Projects | The State of the Octoverse”, https://octoverse.github.com/projects.html, (Acesso em 06 Agosto 2019).

[21] “Event Sourcing”, https://martinfowler.com/eaaDev/EventSourcing.html, (Acesso em 10 de Agosto 2019).

[22] FORD, N., *Functional Thinking*. 1 ed. O’Reilly Media, Inc, 2014.

[23] PAPATHOMAS, M., “Concurrency Issues in Object-Oriented Programming Languages”, 1989.

[24] TANNENBAUM, A. S., BOS, H., *Sistemas Operacionais Modernos*. 4 ed. Vrije Universiteit, 2016.

[25] “Java Functional Programming”, http://tutorials.jenkov.com/java-functional-programming/index.html, (Acesso em 10 de Agosto 2019).

[26] “<functional> – C++ Reference”, http://www.cplusplus.com/reference/functional/, (Acesso em 10 de Agosto 2019).

[27] “Functional Programming HOWTO”, https://docs.python.org/3/howto/functional.html, (Acesso em 10 de Agosto 2019).

[28] “Java Native Interface Specification”, https://docs.oracle.com/en/java/javase/12/docs/specs/jni (Acesso em 10 Agosto 2019).

[29] “The Java Virtual Machine Specification”, https://docs.oracle.com/javase/specs/jvms/se12/html/index.html, (Acesso em 09 Agosto 2019).

[30] “The Erlang BEAM Virtual Machine Specification”, http://www.cs-lab.org/historical_beam_instruction_set.html, (Acesso em 10 de Agosto 2019).

[31] “Erlang – compile”, http://erlang.org/doc/man/compile.html, (Acesso em 10 de Agosto 2019).

[32] “javac”, https://docs.oracle.com/en/java/javase/12/tools/javac.html#GUID-AEEC9F07-CB49-4E96-8BC7-BCC2C7F725C9, (Acesso em 10 de Agosto 2019).

[33] “Compile Python source files”, https://docs.python.org/3/library/py_compile.html, (Acesso em 10 de Agosto 2019).

[34] “AWS Cloud Solutions”, https://aws.amazon.com/solutions/?nc2=h_m2&solutions-all.sort-by=item.additionalFields.sortDate&solutions-all.sort-order=desc, (Acesso em 10 de Agosto 2019).

[35] BENGIO, Y., COURVILLE, A., GOODFELLOW, I. J., *Deep learning: adaptive computation and machine learning*, Adaptive Computation and Machine Learning series. The MIT Press, 2016.

[36] MCCARTHY, J., ABRAHAMS, P. W., EDWARDS, D. J., *et al.*, *Lisp 1.5 Programmer's Manual*. 2 ed. The MIT Press, 1985.

[37] “Clojure – Concurrent Programming”, https://clojure.org/about/concurrent_programming, (Acesso em 10 de Agosto 2019).

[38] “Criterium”, https://github.com/hugoduncan/criterium, (Acesso em 09 Agosto 2019).

[39] “Cortex (Biblioteca em Clojure)”, https://github.com/originrose/cortex, (Acesso em 10 de Agosto 2019).

[40] “TensorFlow - Keras”, https://www.tensorflow.org/api_docs/python/tf/keras, (Acesso em 10 de Agosto 2019).

[41] “TensorFlow”, https://www.tensorflow.org/api_docs/python/tf, (Acesso em 10 Agosto 2019).

[42] “History of Erlang”, https://www.erlang.org/course/history, (Acesso em 12 de Agosto 2019).

[43] HEWITT, C., “Actor Model of Computation: Scalable Robust Information Systems”, https://pdfs.semanticscholar.org/7626/93415b205b075639fad6670b16e9f72d14cb.pdf.

[44] “Erlang – supervisor”, http://erlang.org/doc/man/supervisor.html, (Acesso em 12 de Agosto 2019).

[45] “Erlang/OTP 22.0”, http://erlang.org/doc/, (Acesso em 12 de Agosto 2019).

[46] “Erlang – Ports and Port Drivers”, http://erlang.org/doc/reference_manual/ports.html, (Acesso em 12 de Agosto 2019).

[47] “Protocol Buffers | Google Developers”, https://developers.google.com/protocol-buffers/, (Acesso em 12 de Agosto 2019).

[48] “ErlPort”, http://erlport.org/, (Acesso em 12 de Agosto 2019).

[49] “Treinamento para o *dataset* Cats and Dogs”, https://github.com/originrose/cortex/blob/master/examples/catsdogs-classification/src/catsdogs/training.clj, (Acesso em 10 de Setembro 2019).

[50] “EC Funded CAVIAR project/IST 2001 37540”, http://homepages.inf.ed.ac.uk/rbf/CAVIAR/, (Acesso em 29 de Agosto 2019).

[51] “Model Zoo - TensorFlow SSD Mobilenet V1”, https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection
(Acesso em 10 de Setembro 2019).

[52] “GRPClassify”, https://github.com/polvalente/grpclassify/tree/v1.0.0, (Acesso em 29 de Agosto 2019).

[53] “Apache Thrift”, https://thrift.apache.org/, (Acesso em 29 de Agosto 2019).

[54] “TensorFlow.js”, https://www.tensorflow.org/js, (Acesso em 29 de Agosto 2019).

[55] “Phoenix Framework”, https://phoenixframework.org/, (Acesso em 29 de Agosto 2019).