



UNIVERSIDADE FEDERAL DE SANTA CATARINA  
CENTRO TECNOLÓGICO  
DEPARTAMENTO DE ENGENHARIA ELÉTRICA E ELETRÔNICA

# Implementação em FPGA de um Sistema de *Keyword Spotting* Baseado em Redes Neurais Artificiais

Trabalho de Conclusão de Curso submetido ao curso de Engenharia Elétrica/Eletrônica da Universidade Federal de Santa Catarina como requisito para aprovação da disciplina EEL7890 - Trabalho de conclusão de Curso (TCC) / EEL7806 Projeto Final TCC.

Natan Votre

Orientador: Eduardo Luiz Ortiz Batista  
Co-orientador: Walter Antônio Gontijo

Florianópolis, 9 de julho de 2019.



Ficha de identificação da obra elaborada pelo autor,  
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Votre, Natan

Implementação em FPGA de um Sistema de Keyword Spotting Baseado em Redes Neurais Artificiais / Natan Votre ; orientador, Eduardo Ortiz Batista, coorientador, Walter Antônio Gontijo, 2019.

91 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia Eletrônica, Florianópolis, 2019.

Inclui referências.

1. Engenharia Eletrônica. 2. Dispositivos FPGA. 3. Processamento de Sinais. 4. Machine Learning. 5. Redes Neurais Profundas. I. Ortiz Batista, Eduardo. II. Gontijo, Walter Antônio. III. Universidade Federal de Santa Catarina. Graduação em Engenharia Eletrônica. IV. Título.



Natan Votre

**IMPLEMENTAÇÃO EM FPGA DE UM SISTEMA DE  
KEYWORD SPOTTING BASEADO EM REDES  
NEURAIIS ARTIFICIAIS**

Este Trabalho de Conclusão de Curso foi julgado adequado no contexto da disciplina EEL7890 - Trabalho de conclusão de Curso (TCC) / EEL7806 Projeto Final TCC, e aprovado em sua forma final pelo Departamento de Engenharia Elétrica e Eletrônica da Universidade Federal de Santa Catarina.

Florianópolis, 7 de julho de 2019.



Prof. Jefferson Luiz Brum Marques, PhD.  
Coordenador do Curso



Prof. Eduardo Luiz Ortiz Batista, PhD.  
Orientador

Universidade Federal de Santa Catarina

**Banca examinadora:**



Walter Antônio Gontijo, MSc.  
Co-orientador

Universidade Federal de Santa Catarina



Prof. Danilo Silva, PhD.

Universidade Federal de Santa Catarina



Eng. Edson Manoel da Silva

Universidade Federal de Santa Catarina



*Ao meu Senhor Jesus, que por todo amor se entregou.*



## **Agradecimentos**

Grato sou, meu Senhor Deus, por inebriar-me com vosso amor em todo o período dessa graduação. Me adornando neste amor, nunca me deixastes só. Não obstante tamanha bondade, encheu-me de companhias incríveis (namorada, família, verdadeiros irmãos, orientador, amigos) que mudaram meu caminho e hoje fazem parte da minha história. De gratidão enche-me o coração, pois vós realiza em verme tão vil aquilo que desejas. Agradeço Senhor por abençoar-me, conduzindo-me a desenvolver o presente trabalho.

À Thainá, pais e familiares, pelo apoio extensivo e amor filial.

Ao LINSE, por, além de boas conversas, ter proporcionado tamanho conhecimento na área de processamento de sinais e desenvolvimento em FPGAs.

Ao Grupo de Oração Universitário, que trouxe o sustento na fé, indispensável nesse período de minha vida.



## RESUMO

Este trabalho é dedicado à implementação em FPGA de um sistema de *acoustic keyword spotting* capaz de detectar palavras específicas na língua portuguesa em um sinal de áudio. O sistema implementado realiza o processamento em tempo real utilizando, como pré-processamento, a extração de coeficientes MFCCs, além de uma *convolutional neural network* (CNN) como classificador. Nesse contexto, são desenvolvidos um banco de fala na língua portuguesa, técnicas para melhorar a performance do *keyword spotting*, uma técnica de quantização dinâmica e um gerador automático de código para a CNN em FPGA. O sistema de *keyword spotting* proposto em FPGA mostrou-se equivalente à versão em *software*, obtendo um bom desempenho em termos de tempo de processamento, taxas de detecções corretas e de falsos positivos.

**Palavras-chave:** Detector de Palavras-chave. MFCCs. Redes Neurais Convolucionais. Redes Neurais Artificiais. Quantização de Redes Neurais. FPGA. Verilog.



## ABSTRACT

This work is dedicated to the implementation of an FPGA-based acoustic keyword spotting (KWS) system for the portuguese language. Such system performs real-time processing using MFCC extraction for pre-processing and a convolutional neural network (CNN) as the classifier. In this context, a portuguese speech database, techniques to improve KWS performance, a dynamic quantization technique, and an automatic code generator are developed. The obtained FPGA-based KWS system has achieved similar results as its software version, presenting very good performance in terms of both positive and false-alarm rates, along with extremely low processing times.

**Keywords:** Keyword Spotting. MFCCs. Convolutional Neural Networks. Artificial Neural Networks. Neural Networks Quantization. FPGA. verilog.



---

## Lista de Figuras

---

2.1	Modelo não linear de um neurônio. . . . .	7
2.2	Gráfico da função sigmóide. . . . .	8
2.3	Gráfico da função ReLU. . . . .	9
2.4	Ilustração das operações realizadas em uma <i>convolutional layer</i> , utilizando três filtros. Apenas o resultado para a convolução na posição (1,1) é apresentado. . . . .	12
2.5	Funcionamento de uma <i>pooling layer</i> pelo máximo e pela média para um filtro de tamanho $(2 \times 2)$ e passo 2. . . . .	13
2.6	Ilustração de uma CNN com um espectrograma de entrada, três <i>convolutional layers</i> (CL1, CL2 e CL3), uma <i>pooling layer</i> (PL1) e duas <i>fully connected layers</i> (FL1 e FL2), sendo que a última é a camada de saída. . . . .	14
2.7	Etapas da extração de MFCCs. . . . .	19
2.8	<i>Mel-Bank Filters</i> . $f_i = 400$ , $f_o = 4000$ , $N_f = 30$ , $f_s = 8000$ . . . . .	21
2.9	Matriz de MFCCs $\mathbf{V} \in \mathbb{R}^{m \times n}$ , onde $m = 30$ e $n = 15$ . . . . .	23
2.10	Gráfico da função de probabilidade de uma <i>keyword</i> através da suavização por média movel. . . . .	25
2.11	Um sistema <i>acoustic keyword spotting</i> . . . . .	27
2.12	<i>Acoustic keyword spotting</i> utilizando redes neurais. . . . .	27
3.1	Visão geral do projeto proposto neste trabalho. . . . .	32

3.2	Diagrama de blocos do <i>Framing</i> . . . . .	40
3.3	Diagrama de blocos do Extrator de MFCCs. . . . .	41
3.4	Diagrama de blocos da CNN implementada em FPGA. . . . .	44
3.5	Diagrama de blocos da Camada 1 da CNN implementada em FPGA. . . . .	45
3.6	Diagrama de blocos da Camada 8 da CNN implementada em FPGA. . . . .	46
3.7	Diagrama de blocos do <b>Conv</b> utilizado na CNN implementada em FPGA. . . . .	47
3.8	Diagrama de blocos do <b>Bnorm</b> utilizado na CNN implementada em FPGA. . . . .	48
3.9	Diagrama de blocos do <b>ReLU</b> utilizado na CNN implementada em FPGA. . . . .	49
3.10	Diagrama de blocos do <b>MaxPooling</b> utilizado na CNN implementada em FPGA. . . . .	49
3.11	Diagrama de blocos do <b>FullC</b> utilizado na CNN implementada em FPGA. . . . .	50
3.12	Diagrama de blocos da propagação das constantes de quantização de uma ANN genérica. . . . .	52
3.13	Diagrama de blocos do Decisor implementado em FPGA. . . . .	54
4.1	Comparação de um vetor de MFCCs extraído pelo desenvolvimento em <i>Python</i> e pela implementação em FPGA. . . . .	59

---

## Lista de Tabelas

---

3.1	Organização do <i>Speech Database</i> utilizado no trabalho. . . . .	33
3.2	Especificações dos diferentes cenários utilizados. . . . .	35
3.3	Tamanho por classe dos conjuntos de treinamento. . . . .	36
3.4	Acurácia dos conjuntos de teste pelos modelos salvos. . . . .	37
3.5	Escolha dos <i>thresholds</i> do Decisor para os cenários. . . . .	38
3.6	Resultado do KWS para os diferentes os cenários. . . . .	39
3.7	Implementação em FPGA da CNN baseada no cenário 5. . . . .	43
3.8	Valores de quantização utilizados na implementação da CNN em FPGA. . . . .	53
4.1	Resultado de síntese total FPGA. . . . .	56
4.2	Detalhamento da síntese dos blocos <i>Framing</i> e Extrator de MFCCs. Obs: *O número de ciclos do <i>FFT Calculator</i> não é computado no número de ciclos total. . . . .	57
4.3	Resultado de síntese para o bloco CNN. . . . .	58
4.4	Comparação das avaliações do KWS implementado em <i>Python</i> e em FPGA. . . . .	60



---

## Notação

---

<b>Símbolo</b>	<b>Descrição</b>
$\mathbb{N}$	Números Naturais.
$\mathbb{Z}$	Números Inteiros.
$\mathbb{Q}$	Números Racionais.
$\mathbb{R}$	Números Reais.



---

# Sumário

---

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Objetivos . . . . .	2
1.1.1	Objetivo geral . . . . .	2
1.1.2	Objetivos específicos . . . . .	2
1.1.3	Estrutura do Trabalho . . . . .	3
<b>2</b>	<b>Fundamentação Teórica</b>	<b>5</b>
2.1	<i>Artificial Neural Networks</i> . . . . .	5
2.1.1	Estrutura das ANNs . . . . .	6
2.1.2	Funções de Ativação . . . . .	7
2.1.3	<i>Multi-Layer Perceptrons</i> . . . . .	9
2.1.4	<i>Convolutional Neural Networks</i> . . . . .	13
2.2	Treinamento e Teste . . . . .	15
2.3	Treinamento em <i>batches</i> . . . . .	16
2.4	<i>Overfitting</i> e Validação . . . . .	16
2.5	Estratégias para Melhorar o Desempenho de Redes Neurais	17
2.5.1	<i>Data Augmentation</i> . . . . .	17
2.5.2	<i>Batch Normalization</i> . . . . .	17
2.6	Representando Sinais de Áudio Usando MFCCs . . . . .	18
2.6.1	Etapas da extração dos MFCCs . . . . .	19
2.6.2	Representação Matricial . . . . .	22

2.7	<i>Keyword Spotting</i> . . . . .	23
2.7.1	Métricas de Avaliação . . . . .	24
2.7.2	Tipos de KWS . . . . .	26
2.7.3	<i>Acoustic KWS</i> Utilizando ANNs . . . . .	26
2.8	FPGA . . . . .	28
2.9	Verilog . . . . .	28
2.10	Python . . . . .	29
2.11	Tensorflow . . . . .	29
<b>3</b>	<b>Desenvolvimento</b>	<b>31</b>
3.1	Visão Geral . . . . .	31
3.2	Desenvolvimento em <i>Python</i> . . . . .	32
3.2.1	<i>Speech Database</i> . . . . .	33
3.2.2	<i>Data Augmentation</i> . . . . .	34
3.2.3	Cenários . . . . .	34
3.2.4	Treinamento . . . . .	37
3.2.5	Avaliação . . . . .	38
3.3	Desenvolvimento para o FPGA . . . . .	39
3.3.1	<i>Framing</i> . . . . .	39
3.3.2	Extrator de MFCCs . . . . .	40
3.3.3	CNN . . . . .	43
3.3.4	Decisor . . . . .	53
<b>4</b>	<b>Resultados</b>	<b>55</b>
4.1	Resultados de Síntese . . . . .	55
4.1.1	Detalhamento dos Resultados de Síntese . . . . .	56
4.2	Comparação do KWS em FPGA e em <i>Python</i> . . . . .	58
4.3	Avaliação do KWS em FPGA . . . . .	59
4.4	Análise Tempo Real . . . . .	60
<b>5</b>	<b>Considerações Finais</b>	<b>61</b>
	Referências . . . . .	62

# CAPÍTULO 1

---

## Introdução

---

A aprendizagem de máquina (no inglês, *machine learning*) usando Redes Neurais Artificiais (*Artificial Neural Networks* - ANNs) vem despertando um crescente interesse da comunidade acadêmica nos últimos anos. Isso se deve, em boa medida, aos promissores resultados obtidos pelas Redes Neurais Profundas (*Deep Neural Networks* - DNNs) em diferentes aplicações, tais como tradução automática [1], colorização de imagens em tons de cinza [2], síntese de imagens [3], síntese de fala [4] e reconhecimento de fala [5–9]. Especificamente com respeito ao reconhecimento de fala, os resultados obtidos nos últimos anos utilizando DNNs [5–7] têm superado os obtidos com as técnicas baseadas em Modelos Ocultos de Markov (*Hidden Markov Models* - HMMs) [10–12], as quais até pouco tempo correspondiam ao estado da arte.

No contexto de reconhecimento de fala, uma tarefa que tem ganhado destaque é a detecção de palavras-chave (*keyword spotting* - KWS) [5–9], que é a detecção de uma palavra específica em um vocabulário extenso [6]. Há diferentes aplicações para sistemas de KWS, tais como assistentes virtuais [13], *smart homes* [14] e ferramentas de acessibilidade. Muitas dessas aplicações envolvem

sistemas embarcados, podendo então se beneficiar significativamente do uso de dispositivos *Field Programming Gate Array* (FPGA) devido ao melhor compromisso entre desempenho e consumo energético proporcionado por tais dispositivos [8]. Assim, o foco do presente trabalho está no desenvolvimento de um sistema de KWS em FPGA utilizando ANNs. Tal sistema deverá ser capaz de reconhecer diferentes comandos pré-treinados na língua portuguesa.

## 1.1 Objetivos

### 1.1.1 Objetivo geral

O objetivo central deste trabalho é utilizar um FPGA para realização de implementação eficiente de um sistema de *keyword spotting* baseado em ANNs.

### 1.1.2 Objetivos específicos

- Analisar algoritmos para a extração de características de sinais de áudio;
- Implementar os algoritmos analisados em *Python*;
- Avaliar arquiteturas de redes neurais aplicadas em *keyword spotting*;
- Implementar as arquiteturas avaliadas em *Python*;
- Analisar algoritmos de suavização (em inglês, *smoothing*) para melhorar as detecções dos comandos de voz;
- Utilizar conjuntos de dados (em inglês, *datasets*) disponíveis na *internet* para treinamento e validação das redes neurais;
- Elaborar um *dataset* em português com diversos locutores para o treinamento das redes neurais na língua portuguesa;
- Treinar a rede escolhida utilizando o *dataset* elaborado;
- Implementar em FPGA a rede escolhida e treinada para o *keyword spotting*;
- Avaliar o desempenho do sistema implementado.

**1.1.3 Estrutura do Trabalho**

O presente trabalho está organizado conforme descrito a seguir. No Capítulo 2 é apresentada uma visão geral da fundamentação teórica deste trabalho. No Capítulo 3 são detalhadas as técnicas e implementações utilizadas. No Capítulo 4, os experimentos realizados e os resultados obtidos são apresentados. Finalmente, no Capítulo 5 são apresentadas as considerações finais.



# CAPÍTULO 2

---

## Fundamentação Teórica

---

Neste capítulo é apresentada a fundamentação teórica para o desenvolvimento do presente trabalho. Inicialmente, são apresentadas noções gerais de ANNs, funções de ativação e tipos de camadas utilizadas. Na sequência, noções sobre extração de *features* e *keyword spotting* são apresentadas. Finalmente, são apresentadas as ferramentas utilizadas para o desenvolvimento e a avaliação do projeto proposto neste trabalho.

### 2.1 *Artificial Neural Networks*

ANNs são modelos computacionais de *machine learning* inspirados nas redes neurais biológicas que compõem o sistema nervoso humano. As primeiras abordagens baseadas em ANNs surgiram durante a década de 40, quando uma rede neural biológica simples foi modelada utilizando circuitos elétricos [15]. Ao longo das décadas de 40 a 70, surgiram diversos trabalhos científicos sobre ANNs, os quais deram origem ao conceito de *Perceptron* [16] e também identificaram suas principais limitações.

Por falta de equipamentos robustos para o treinamento, as ANNs

tiveram inicialmente pouca relevância prática, conseguindo resolver apenas problemas relativamente simples. Com o advento de novos algoritmos para o treinamento de redes neurais, como o algoritmo de *backpropagation* [17], a elaboração de maiores conjunto de dados e a utilização de novas funções de ativação, tornou-se possível o uso de ANNs com múltiplas camadas, ou DNNs, obtendo melhores resultados e permitindo o uso de ANNs em problemas mais complexos.

As ANNs em conjunto com técnicas de *machine learning* podem ser aplicadas em diversos problemas, tais como regressão [3], sistemas de controle [18], síntese e reconhecimento de fala [4–9], reconhecimento e classificação de objetos [19] e imagens [20], tradução automática [1], dentre outros. Muitos desses problemas só puderam ser solucionados com ANNs devido aos avanços promissores das DNNs. Tais redes ganharam popularidade apenas nos últimos anos, devido às técnicas de aprendizado profundo (em inglês, *deep learning*) desenvolvidas por volta de 2006 [21] e também ao amplo uso de *Graphics Processing Units* (GPUs) para um treinamento mais eficiente. Por se tratar de um assunto de pesquisa recente, muitos esforços de pesquisa tem sido realizados atualmente na área de *deep learning* [22].

### 2.1.1 Estrutura das ANNs

As ANNs são constituídas por neurônios artificiais com grande número de interconexões, com a finalidade de realizar uma determinada função. O neurônio, por sua vez, é considerado a unidade básica de uma rede neural. Estes possuem múltiplas entradas e, a partir de operações matemáticas relativamente simples, geram uma única saída [23]. O modelo básico de um neurônio  $k$  é apresentado na Figura 2.1. Tal modelo é composto por sinais de entrada  $x_{k1}, x_{k2}, x_{k3}, \dots, x_{km}$ , *bias*  $b_k$ , pesos sinápticos  $w_{k1}, w_{k2}, w_{k3}, \dots, w_{km}$ , junção de soma  $\sum \cdot$  e uma função de ativação  $\varphi(\cdot)$ . Os sinais de entrada são multiplicados pelos pesos sinápticos, e posteriormente somados a um sinal de *bias*, sendo então o resultado enviado para a função de ativação. Essa função corresponde a uma determinada não linearidade, a qual ativa ou desativa a saída do neurônio.

Em termos matemáticos, é possível descrever um neurônio de índice

$k$  utilizando as seguintes equações [23]:

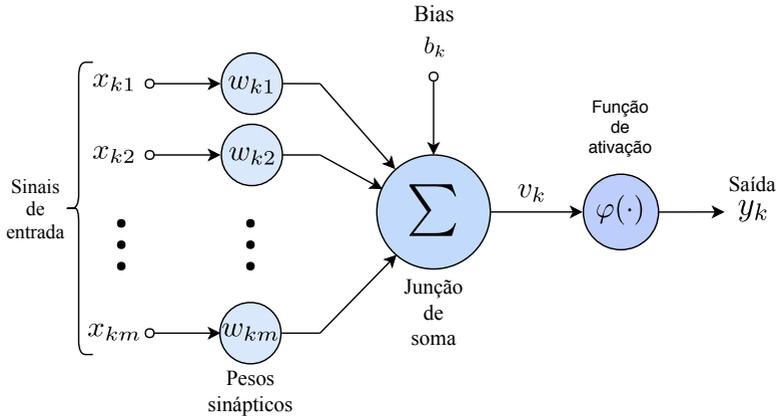
$$v_k = \mathbf{w}_k^T \mathbf{x}_k + b_k \quad (2.1)$$

e

$$y_k = \varphi(v_k) \quad (2.2)$$

onde  $\mathbf{x}_k = [x_{k1}, x_{k2}, x_{k3}, \dots, x_{km}]^T \in \mathbb{R}^m$  é o vetor que contém os sinais de entrada,  $\mathbf{w}_k = [w_{k1}, w_{k2}, w_{k3}, \dots, w_{km}]^T \in \mathbb{R}^m$  o vetor de pesos que contém os pesos sinápticos,  $b_k$  o *bias*,  $\varphi(\cdot)$  a função de ativação e  $y_k$  a saída do neurônio.

Figura 2.1: Modelo não linear de um neurônio.



Fonte: do Autor.

O valor dos elementos do vetor de pesos  $\mathbf{w}_k$  e de  $b_k$  são calculados a partir de um processo prévio de treinamento, utilizando um *dataset* adequado.

### 2.1.2 Funções de Ativação

A escolha de uma função de ativação  $\varphi(\cdot)$  de um determinado neurônio  $k$  influencia diretamente no treinamento e também no desempenho de uma rede neural [24]. Nesta seção, são abordadas as funções mais importantes para a implementação apresentada neste trabalho.

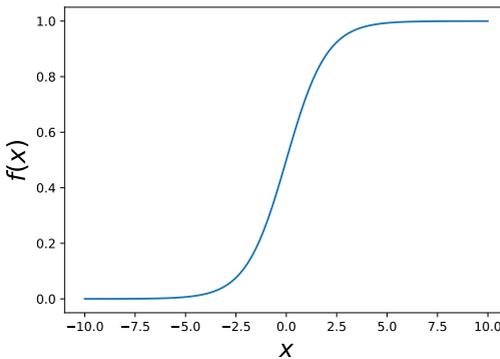
### 2.1.2.1 Função Sigmóide

A função sigmóide foi por muito tempo a principal função de ativação utilizada em redes neurais [24]. Esta função pode ser representada como

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.3)$$

possuindo a saída limitada entre  $[0, 1]$ . Neste caso, podemos interpretar que, quando a saída de um neurônio é 0, ele está totalmente desativado e, quando a saída é 1, está totalmente ativado. A Figura 2.2 mostra o gráfico da função sigmóide.

Figura 2.2: Gráfico da função sigmóide.



Fonte: do Autor.

Com a evolução dos algoritmos de treinamento, como o *backpropagation*, percebeu-se uma limitação no treinamento de ANNs que utilizavam a função de ativação sigmóide. Tal limitação é conhecida como *vanishing gradient problem* e pode ser observada quando o valor da sigmóide é muito baixo ou muito alto, tornando o valor da sua derivada próximo de zero e, conseqüentemente, dificultando o treinamento de ANNs com muitas camadas [24].

### 2.1.2.2 Função ReLU

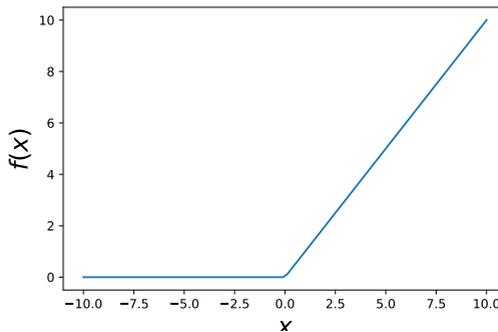
Visando resolver o *vanishing gradient problem* gerado no uso da função sigmóide, a função de ativação *Rectified Linear Unit* (ReLU)

e suas variações começaram a ser utilizadas [24]. A ReLU pode ser representada como

$$f(x) = \max(0, x) \quad (2.4)$$

possuindo a saída limitada entre  $[0, \infty)$ . Neste caso, podemos interpretar que quando a saída de um neurônio é 0, ele está totalmente desativado. Diferentemente da função simóide (Seção 2.1.2.1), esta função não possui limitação de ativação (eixo positivo da função), permitindo que um determinado neurônio possa sempre estar mais ou menos ativado em relação a outro [24]. A Figura 2.3 mostra o gráfico da função ReLU.

Figura 2.3: Gráfico da função ReLU.



Fonte: do Autor.

### 2.1.3 Multi-Layer Perceptrons

A organização de vários *perceptrons* conectados em camadas distintas ficou conhecida como *Multi-Layer Perceptron* (MLP) [24]. As redes baseadas em MLPs são divididas em diferentes camadas, as quais são classificadas em camada de entrada, camadas ocultas e camada de saída. Quando as MLPs são utilizadas com mais de uma camada oculta, são chamadas de DNNs. Na literatura, são encontrados diversos tipos de camadas ocultas utilizadas em DNNs [24] e, nesta seção, são apresentados os tipos utilizados na implementação apresentada neste trabalho.

### 2.1.3.1 Fully Connected Layer

A camada totalmente conectada (em inglês, *fully connected layer*) foi o primeiro tipo de camada oculta utilizado. Nela, cada elemento do vetor de entrada se conecta a todos os neurônios presentes na camada [24]. Assim, a saída de um neurônio pode ser representada por (2.1), sendo que  $\mathbf{x}_k$  é o próprio vetor de entrada  $\mathbf{x}$  da camada, o qual é comum para todos os neurônios. A saída de uma *fully connected layer* pode ser descrita através das seguintes equações:

$$\mathbf{v} = \mathbf{W}\mathbf{x} + \mathbf{b} \quad (2.5)$$

e

$$\mathbf{y} = \Phi(\mathbf{v}) \quad (2.6)$$

onde  $\mathbf{x} \in \mathbb{R}^m$  é o vetor de entrada da camada,  $\mathbf{W} \in \mathbb{R}^{k \times m}$  a matriz de pesos,  $\mathbf{b} \in \mathbb{R}^k$  o vetor de *bias*,  $\Phi$  a função de ativação de valores de vetor e  $\mathbf{y} \in \mathbb{R}^k$  o vetor de saída. A matriz de pesos contém, em suas linhas, os vetores de pesos dos neurônios da camada. Na  $i$ -ésima linha da matriz de pesos é encontrado o vetor de pesos do  $i$ -ésimo neurônio da camada. Já  $\mathbf{b}$  contém, em seu  $i$ -ésimo elemento, o *bias* do  $i$ -ésimo neurônio.

A *fully connected layer*, por ter muitas conexões, possui importantes limitações práticas. Para camadas que recebem vetores com muitos elementos, a aplicação se torna inviável devido ao tamanho excessivo da matriz  $\mathbf{W}$ , o que conseqüentemente eleva muito o número de parâmetros da ANN. Por exemplo, uma *fully connected layer* com 1000 entradas e apenas 100 neurônios, necessita de 101000 parâmetros, contando com os parâmetros contidos no vetor  $\mathbf{b}$ . Por conta dessa limitação, as *fully connected layers* são mais utilizadas em camadas com um número reduzido de entradas, o que é tipicamente o caso nas últimas camadas de uma rede neural.

### 2.1.3.2 Convolutional Layer

Um dos tipos de camada que permite contornar algumas das limitações da *fully connect layer* é a camada convolucional (em inglês, *convolutional layer*) [24]. Da mesma forma que a *fully connect layer*,

a *convolutional layer* também atua na extração de atributos dos dados de entrada. A principal diferença entre ambas está no fato que a *convolutional layer* considera a disposição espacial dos dados de entrada, através do cálculo da convolução [25]. Com isso, o número de parâmetros necessários em uma camada reduz-se drasticamente. Sua entrada consiste em um tensor  $x \in \mathbb{R}^{C \times H \times W}$ , onde  $C$  é o número de canais,  $H$  a altura e  $W$  a largura. A camada também possui filtros, utilizados na convolução, definidos como  $h_i \in \mathbb{R}^{C \times H_c \times W_c}$ , os quais possuem um termo de *bias*.

A convolução para valores tri-dimensionais pode ser representada pela equação

$$v[i, j, k] = \sum_l \sum_m \sum_n x[i-l, j-m, k-n] h[l, m, n]. \quad (2.7)$$

Na operação de convolução, a saída pode possuir três formatos: *full*, *same* e *valid*. Suponha que a convolução entre os tensores  $d \in \mathbb{R}^{C_d}$  e  $g \in \mathbb{R}^{C_g}$ , onde  $C_d > C_g$ , é dada por  $y = d * g$ . Em uma convolução com formato *full*, a saída será  $y \in \mathbb{R}^{C_d + C_g - 1}$ , já com formato *same* a saída será  $y \in \mathbb{R}^{C_d}$  e, por fim, se o formato for *valid*, a saída será  $y \in \mathbb{R}^{C_d - C_g + 1}$ .

Em uma *convolutional layer*, a operação de convolução tipicamente utiliza, para cada filtro, o formato *valid* na dimensão  $C$ . Portanto, a saída na dimensão  $C$  possuirá tamanho 1 em cada operação de convolução. Já nas outras dimensões ( $H$  e  $W$ ) é possível escolher entre os três formatos existentes. Normalmente é utilizado o formato *valid* em todas as dimensões, auxiliando na diminuição de atributos ao longo das camadas. Em todas as operações de convolução em uma *convolutional layer*, a saída é adicionada com um sinal de *bias* antes de passar pela função de ativação.

O cálculo da *convolutional layer* pode ser descrito através das seguintes equações:

$$v = x * \{h_1, h_2, ..h_n\} \quad (2.8)$$

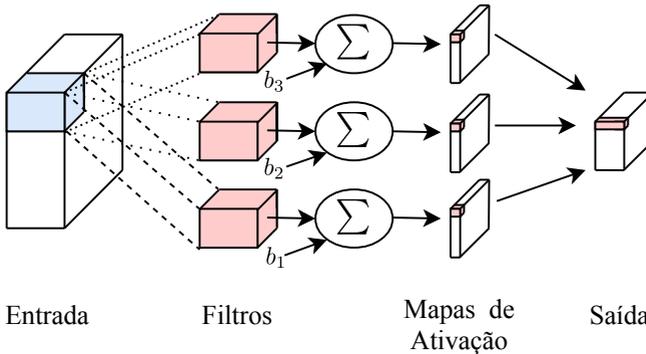
$$y = \Phi(v + b) \quad (2.9)$$

onde o operador  $*$  representa a convolução entre o tensor de entrada

$x \in \mathbb{R}^{C \times H_x \times W_x}$  e os filtros  $h_1, h_2, \dots, h_n \in \mathbb{R}^{C \times H_h \times W_h}$ ,  $v \in \mathbb{R}^{n \times H_v \times W_v}$  representa o resultado das convoluções,  $\Phi(\cdot)$  é a função de ativação de valores de tensor, também chamado de mapa de ativação,  $b \in \mathbb{R}^{n \times H_v \times W_v}$  é o tensor *bias*, e  $y \in \mathbb{R}^{n \times H_v \times W_v}$  é a saída da *convolutional layer*. As dimensões  $H_v$  e  $W_v$  são dependentes do formato de convolução utilizado.

A Figura 2.4 ilustra as operações realizadas por uma *convolutional layer* de três filtros. Observa-se na Figura 2.4 as operações de uma *convolutional layer* na posição (1,1): convolução entre a entrada e os filtros, adição do *bias* à saída da convolução, aplicação dos mapas de ativação e a saída da *convolutional layer*.

Figura 2.4: Ilustração das operações realizadas em uma *convolutional layer*, utilizando três filtros. Apenas o resultado para a convolução na posição (1,1) é apresentado.



Fonte: do Autor.

### 2.1.3.3 Pooling Layer

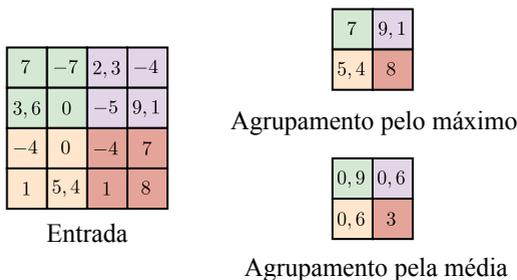
A camada de *pooling* (em inglês, *pooling layer*) é tipicamente utilizada para reduzir a quantidade de parâmetros da rede neural. Isso ocorre pois esta camada reduz as dimensões do tensor de entrada [24]. Tal processamento é realizado através de agrupamentos de elementos de um tensor, retornando apenas um elemento por grupo. É interessante o uso da *pooling layer* quando existem redundâncias entre elementos vizinhos em um tensor, comprimindo as informações contidas no tensor de entrada.

Esta camada é composta por um único filtro  $h \in \mathbb{R}^{H_p \times W_p}$ , que utiliza uma função fixa e não parametrizada, aplicada no tensor de entrada em forma de agrupamentos.

As funções comumente utilizadas em uma *pooling layer* são: função de máximo (em inglês, *max-pooling*) e função de média (em inglês, *average-pooling*). Normalmente o valor do passo e o tamanho do filtro são iguais, de modo que não haja sobreposição entre os agrupamentos. O passo mais utilizado nas *pooling layers* é de 2, em que, considerando um agrupamento bi-dimensional, é aplicado uma compressão de 4 : 1 nos elementos do tensor.

A Figura 2.5 ilustra os cálculos realizados em duas *pooling layers*, uma utilizando a função *max-pooling* e a outra a função *average-pooling*, ambos com o passo e o tamanho do filtro igual a dois.

Figura 2.5: Funcionamento de uma *pooling layer* pelo máximo e pela média para um filtro de tamanho  $(2 \times 2)$  e passo 2.



Fonte: do Autor.

### 2.1.4 Convolutional Neural Networks

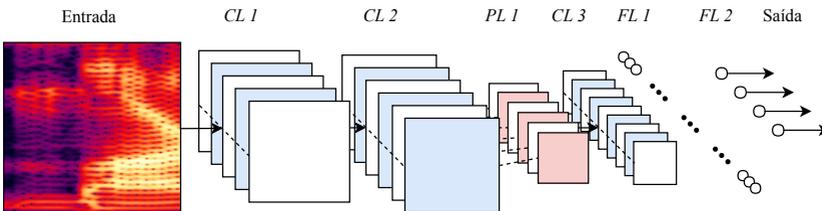
As Redes Neurais Convolucionais (*Convolutional Neural Networks* - CNNs) consistem em redes neurais em que o tipo principal de camada utilizado é *convolutional layer*. Este tipo de rede foi idealizado visando o processamento de imagens, particularmente em aplicações de classificação, reconhecimento e síntese de imagens. Porém, CNNs também são utilizadas em processamento de fala [9] [26] [27], utilizando algoritmos de pré-processamento que dispõem os parâmetros espacialmente.

A arquitetura de uma CNN é inspirada no córtex visual [28], onde cada neurônio é responsável por apenas uma fração do campo de visão, sendo que ao final todo campo visual é tratado. Assim, a CNN utiliza filtros para analisar pequenos grupos de vizinhança, extraindo as características necessárias. Somente estas características extraídas são enviadas para as próximas camadas, justificando assim que uma CNN seja chamada de *feature extractor*.

Usualmente, as CNNs são compostas pelas *convolutional layers* (Seção 2.1.3.2), *pooling layers* (Seção 2.1.3.3) e *fully connected layers* (Seção 2.1.3.1) [24]. As *convolutional layers* são as mais utilizadas em uma CNN e são responsáveis por extrair os atributos das entradas nas primeiras camadas. Já as *pooling layers* são utilizadas entre as *convolutional layers*, com a função de comprimir os tensores de entrada. Elas são inseridas entre camadas em que há uma determinada redundância entre os elementos vizinhos do tensor de entrada. Por fim, as *fully connected layers* são utilizadas nas últimas camadas de uma CNN, na qual o número de elementos de entrada são reduzidos. Estas camadas também são utilizadas na camada de saída de uma CNN, a fim de realizar a classificação propriamente dita.

A Figura 2.6 ilustra uma CNN utilizando as camadas citadas anteriormente.

Figura 2.6: Ilustração de uma CNN com um espectrograma de entrada, três *convolutional layers* (CL1, CL2 e CL3), uma *pooling layer* (PL1) e duas *fully connected layers* (FL1 e FL2), sendo que a última é a camada de saída.



Fonte: do Autor.

A CNN apresentada na Figura 2.6 possui em sua entrada um espectrograma de um sinal de áudio. Em sua composição de camadas são encontradas três *Convolutional Layers* (CLs), uma *Pooling Layer* (PL) entre a CL 2 e a CL 3 e, por último, duas *Fully connected Layers* (FLs) utilizadas para a classificação.

### 2.1.4.1 Aplicações

As CNNs podem ser utilizadas em diversas aplicações, desde processamento de imagens até processamento de áudio e fala. Dentre tais aplicações, encontram-se o reconhecimento de objetos em imagens [19], detecção de faces [29], identificação de locutor em áudio [30] e detecção de palavras-chave [9].

Em [19] é utilizado o modelo de CNN para a classificação de objetos em uma imagem. O *dataset* utilizado neste trabalho foi o VOC 2007 (*Pascal Visual Object Challenge 2007*) [31], obtendo resultados satisfatórios.

Em [29], CNNs são utilizadas para a detecção de faces em imagens. Tal aplicação superou modelos que utilizavam MLPs convencionais, diminuindo a taxa de erro de 40% para 3,8%.

Já em [30] é proposto um novo método utilizando *3D Convolutional Neural Networks* (3D-CNNs) para a identificação de locutor em sinais de áudio independente de texto. Neste trabalho, é utilizado como pré-processamento o a extração *Mel-frequency energy coefficients* (MFCE). Esta extração dispõe os parâmetros de forma espacial, auxiliando nos cálculos das camadas convolucionais das 3D-CNNs.

Por último, em [9] são utilizados Redes Convolucionais Recorrentes para a detecção de palavras-chave. Neste trabalho é apresentado um ganho relativo de 27-44% na taxa de falsa rejeição (em inglês, *false reject rate*) comparado com DNNs. Tais comparações foram realizadas observando o número de parâmetros e multiplicações realizadas, mostrando que as CNNs utilizam muito menos parâmetros e multiplicações.

## 2.2 Treinamento e Teste

Para um treinamento adequado de uma ANN, é necessário utilizar métricas de desempenho específicas ao problema em questão. Em tarefas de classificação, que são as consideradas neste trabalho, emprega-se geralmente como métrica de desempenho a acurácia do modelo treinado [24]. Para analisar a acurácia das ANNs, é considerado um *dataset* diferente do utilizado no treinamento, denominado conjunto de teste. Em tarefas de reconhecimento de fala, geralmente além de considerar o conjunto de teste diferente do conjunto de treinamento,

estes *datasets* são divididos em locutores. Assim, os locutores utilizados para a elaboração do conjunto de treinamento não são empregados no teste. A avaliação no conjunto de teste permite determinar o desempenho da ANN no uso de um caso real, isto é, aplicando a ANN em situações diferentes das estipuladas no treinamento.

### 2.3 Treinamento em *batches*

Para acelerar o processo de treinamento de uma ANN, é possível dividir o conjunto de treinamento em diversos sub-conjuntos, denominados *mini-batches* [24]. Com a utilização do algoritmo *backpropagation*, todos os exemplos de um *mini-batch* são processados em paralelo durante o treinamento, produzindo um conjunto de saída utilizado para atualizar os pesos da ANN. Portanto, durante o processamento dos exemplos de um mesmo *mini-batch*, os pesos da ANN permanecem fixos e são atualizados após o processamento do *mini-batch*. Após todo o conjunto de treinamento ser processado (processamento de 1 época), é realizada novamente a divisão em *mini-batches* e processados tais sub-conjuntos em uma nova época. No fim do treinamento, os pesos das ANNs não são mais alterados, gerando um modelo correspondente.

### 2.4 *Overfitting* e Validação

Considerando a divisão do *dataset* em treinamento e teste, é desejado de uma ANN obter um bom desempenho principalmente no conjunto de teste. O *overfitting* ocorre quando a ANN se especializa demais no conjunto de treinamento, perdendo a capacidade de generalização e obtendo um resultado insatisfatório na avaliação do conjunto de teste. Para minimizar o *overfitting* de um treinamento, utiliza-se um conjunto de validação, diferente dos conjuntos de treinamento e teste, para avaliar o desempenho da rede durante o treinamento [24]. Em tarefas de reconhecimento de fala, portanto, são utilizados 3 *datasets* com informações de locutores distintos para realizar o treinamento (conjunto de treinamento), para avaliar o *overfitting* durante o treinamento (conjunto de validação) e a generalização da ANN (conjunto de teste).

## 2.5 Estratégias para Melhorar o Desempenho de Redes Neurais

Nesta seção são apresentadas algumas estratégias para a melhoria do desempenho de ANNs. As estratégias consideradas neste trabalho são: *data augmentation* e *batch normalization*.

### 2.5.1 Data Augmentation

Geralmente, a quantidade de dados do *dataset* não é suficientemente grande para um treinamento eficiente de DNNs. Para solucionar este problema, uma técnica considerada é o *data augmentation*. Tal técnica consiste na elaboração de dados artificiais a partir do *dataset* original. O *data augmentation* é utilizado em muitas tarefas de *machine learning*, tais como detecção de objetos [32] e reconhecimento de fala [33, 34].

Em problemas de classificação, deseja-se manter os dados sintetizados na mesma classe dos dados originais. Assim, o *data augmentation* é realizado a partir de pares  $(x, y)$  do *dataset* original, realizando apenas transformações nas entradas  $x$ . Em *datasets* de áudio, pode-se utilizar as transformações de *time shift*, *pitch shift* e *time stretch*. O *time shift* realiza apenas pequenos deslocamentos temporais, dados por um valor máximo  $d_1$ , no trecho de áudio correspondente a uma determinada classe [34]. O *pitch shift*, por sua vez, realiza um deslocamento em frequência do trecho de áudio, tornando-o mais grave ou mais agudo [34]. Já o *time stretch* realiza a compressão ou expansão do trecho de áudio, mantendo o espectro de frequências [33].

### 2.5.2 Batch Normalization

A tarefa de treinar uma DNN para gerar um modelo adequado é complicada. Isso se deve às alterações nas distribuições das entradas de cada camada a cada passo do treinamento. Para um modelo com muitas camadas ocultas, tais alterações influenciam expressivamente as saídas das camadas mais profundas. Esse fenômeno diminui a velocidade de treinamento e é chamado de *internal covariate shift*. A técnica *batch normalization* reduz a ocorrência de tal fenômeno, realizando a normalização das saídas das camadas escolhidas a cada *mini-batch* do treinamento [35].

Durante o treinamento, são calculados 4 parâmetros a cada normalização. Tais parâmetros são utilizados para a realizar a normalização durante a predição, através das seguintes equações:

$$\bar{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2}} \quad (2.10)$$

$$y_i = \gamma \bar{x}_i + \beta \quad (2.11)$$

onde  $\beta$ ,  $\gamma$ ,  $\mu_B$  e  $\sigma_B$  são os parâmetros do *batch normalization*,  $x_i$  é a entrada e  $y_i$  é a saída. O *batch normalization* é comumente utilizado antes da função de ativação (Seção 2.1.2), portanto, interno às camadas do modelo. Quando utilizados em *convolutional layers* (Seção 2.1.3.2), são calculados os parâmetros para a saída de cada filtro  $h_i$ . Portanto, para uma *convolutional layer* com  $n$  filtros, são utilizados  $n$  parâmetros  $\beta$ ,  $\gamma$ ,  $\mu_B$  e  $\sigma_B$ .

Manipulando (2.10) e (2.11), tem-se

$$y_i = Ax_i + B, \text{ onde} \quad (2.12)$$

$$A = \frac{\gamma}{\sqrt{\sigma_B^2}} \text{ e} \quad (2.13)$$

$$B = -\gamma \frac{\mu_B}{\sqrt{\sigma_B^2}} + \beta. \quad (2.14)$$

Utilizando (2.12), pode-se realizar a operação de *batch normalization* utilizando apenas uma única multiplicação e soma. Nota-se que, em uma *convolutional layer*, considerando (2.12), são utilizados  $n$  parâmetros  $A$  e  $B$ .

## 2.6 Representando Sinais de Áudio Usando MFCCs

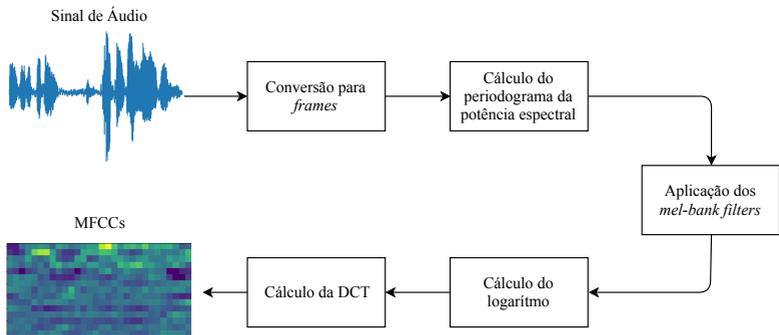
Em aplicações de reconhecimento de fala, a extração de atributos de um sinal de áudio pode ser realizada utilizando *Mel-Frequency Cepstral Coefficients* (MFCCs). Os MFCCs são utilizados para descrever a função de transferência do trato vocal [36]. A extração dos MFCCs foi desenvolvida de forma a modelar a percepção do ouvido humano e permite uma representação mais compacta do sinal de áudio, podendo

reduzir o número de amostras de um *frame* de 1024 para um intervalo entre 15 a 40 amostras [37]. Tal compactação conserva as informações de timbre e de envelope espectral em poucas amostras [38]. Em aplicações de ANNs em reconhecimento de fala, os MFCCs podem ser utilizados como pré-processamento realizando a extração dos atributos do sinal de entrada. Assim, a etapa de processamento, que neste trabalho é uma CNN, é simplificada.

### 2.6.1 Etapas da extração dos MFCCs

A extração dos MFCCs é realizada nas seguintes etapas: Conversão para *Frames*, Cálculo do Periodograma da Potência Espectral, Filtragem com *Mel-Bank Filters*, Cálculo do Logarítmo e Cálculo da *Discrete Cosine Transform* (DCT). Tal sequência de etapas é ilustrada na Figura 2.7.

Figura 2.7: Etapas da extração de MFCCs.



Fonte: do Autor.

#### 2.6.1.1 Conversão para *Frames*

Nesta etapa, o sinal de áudio é dividido em *frames*. Como os coeficientes MFCCs são do tipo *short-term spectral-based features*, os *frames* devem ser de pequena duração, tipicamente 20ms [36, 39]. Geralmente, para a obtenção dos *frames*, são utilizadas as técnicas de janelamento (por exemplo, usando a janela *Hanning*) e de sobreposição (50%). Os *frames* obtidos são enviados para o Cálculo do Espectro de Frequências.

### 2.6.1.2 Cálculo do Periodograma da Potência Espectral

Nesta etapa, é calculada a potência espectral de cada *frame* obtido anteriormente, resultando em um periodograma estimado na potência espectral. As equações para o cálculo da potência espectral são [40]:

$$\hat{y}[k] = \sum_{j=0}^{N-1} y[j] e^{-\frac{2\pi ijk}{N}} \quad (2.15)$$

$$P[k] = \begin{cases} N^{-2} |\hat{y}[0]|^2 & \text{para } k = 0 \\ N^{-2} |\hat{y}[k] + \hat{y}[N - k]|^2 & \text{para } 0 < k < \frac{N}{2} \\ N^{-2} |\hat{y}[\frac{N}{2}]|^2 & \text{para } k = \frac{N}{2} \end{cases} \quad (2.16)$$

onde  $N$  é o número de *bins*,  $\hat{y}[k]$  é o resultado do cálculo da *Discrete Fourier Transform* (DFT) do sinal discreto  $y[j]$  e  $P[k]$  é a energia do espectro de frequências de  $y[j]$ .

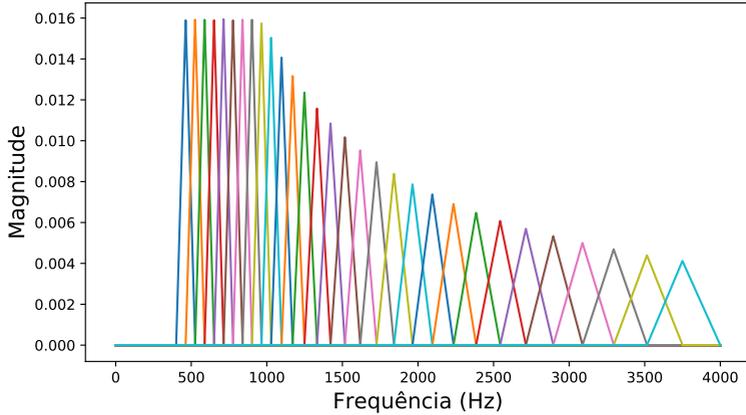
### 2.6.1.3 Filtragem com *Mel-Bank Filters*

Nesta etapa, o espectro de frequências é filtrado por um número determinado de filtros passa-banda (geralmente 40 filtros), espaçados na escala *mel*. Tal escala pode ser representada através da equação [41]:

$$f_{mel} = 2595 \log_{10} \left( 1 + \frac{f_{Hz}}{700} \right) \quad (2.17)$$

onde  $f_{Hz}$  é a frequência na escala linear e  $f_{mel}$  é a frequência na escala *mel*. A Figura 2.8 mostra um conjunto de filtros *mel* (em inglês, *mel-bank filters*) que pode ser utilizado nessa etapa. Conforme ilustrado nessa figura, os filtros utilizados são triangulares com ganhos e espaçamentos diferentes, de forma a aproximar a resposta em frequência com a do ouvido humano [41]. Os ganhos dos filtros são dispostos de modo que as baixas frequências possuam mais peso do que as altas frequências. Já o espaçamento dos filtros é obtido a partir de (2.17), dando maior ênfase para variações em baixas frequências, aproximando-se da percepção auditiva humana.

Os *mel-bank filters* possuem alguns parâmetros que podem ser escolhidos pelo projetista, tais como a  $f_i$  (frequência inicial),  $f_o$

Figura 2.8: *Mel-Bank Filters*.  $f_i = 400$ ,  $f_o = 4000$ ,  $N_f = 30$ ,  $f_s = 8000$ .

Fonte: do Autor.

(frequência final) e  $N_f$  (número de filtros do banco). A Figura 2.8 mostra um *mel-bank* com os parâmetros  $f_i = 400$ ,  $f_o = 4000$  e  $N_f = 30$ .

O cálculo realizado nesta etapa pode ser descrito através das seguintes equações:

$$\mathbf{y}^{(i)} = \mathbf{M}\mathbf{x}^{(i)} \quad (2.18)$$

$$\mathbf{M} = \begin{bmatrix} \mathbf{m}_1^T \\ \mathbf{m}_2^T \\ \dots \\ \mathbf{m}_{N_f}^T \end{bmatrix} \quad (2.19)$$

onde  $\mathbf{x}^{(i)} \in \mathbb{R}^{N \times 1}$  é o vetor de entrada que corresponde ao espectro de frequências do  $i$ -ésimo *frame*,  $N$  é o número de amostras do espectro de frequências,  $\mathbf{m}_1, \mathbf{m}_2 \dots \mathbf{m}_{N_f} \in \mathbb{R}^{N \times 1}$  são os *Mel-bank filters*,  $\mathbf{M} \in \mathbb{R}^{N_f \times N}$  é a matriz de filtros e  $\mathbf{y}^{(i)} \in \mathbb{R}^{N_f}$  é o  $i$ -ésimo vetor de saída.

### 2.6.1.4 Cálculo do Logaritmo

Nesta etapa é calculado o logaritmo do vetor de saída obtido anteriormente. Da mesma forma que a etapa anterior, o cálculo do logaritmo é realizado para aproximar a resposta com a do ouvido humano [41]. O ser humano consegue perceber sinais de baixa intensidade, tais como um susurro ou um murmurinho, mas também identifica sinais de alta intensidade, tais como o som de um foguete ou de um avião. Utilizando o logaritmo, é realizada uma compressão na amplitude aproximando valores de baixa e alta amplitude.

### 2.6.1.5 Cálculo da DCT

Finalmente, nesta etapa é realizado o cálculo da DCT, que é utilizada para converter o espectro log-Mel para o domínio *quefrequency*, obtendo os coeficientes *cepstrais* (inverso de *spectrais*), ou seja, os MFCCs. O cálculo da DCT pode ser descrito através das equações [42]:

$$X[f] = \frac{1}{2} C[f] \sum_{n=0}^{N-1} x[n] \cos\left(\frac{(2n-1)f\pi}{2N}\right) \quad (2.20)$$

$$C[f] = \begin{cases} \frac{1}{\sqrt{N}} & \text{para } f = 0 \\ \sqrt{\frac{2}{N}} & \text{para } f > 0 \end{cases} \quad (2.21)$$

onde  $X[f]$  é o resultado do cálculo da DCT do sinal de entrada  $x[n]$  e  $N$  é o tamanho de  $x[n]$ .

A DCT é muito utilizada na compressão de dados, pois a maior parte da informação contida na entrada é transferida para os primeiros elementos do vetor de saída [43]. Desse modo, são comumente utilizados apenas os 13 primeiros coeficientes obtidos na DCT para compor os MFCCs.

## 2.6.2 Representação Matricial

Para o  $i$ -ésimo *frame* obtido na etapa de Conversão para *Frames* é extraído um vetor de MFCCs  $\mathbf{v}^i \in \mathbb{R}^n$ , onde  $n$  é o número de coeficientes obtidos no Cálculo da DCT. Cada vetor  $\mathbf{v}^i$  corresponde a um trecho de áudio de curto período de tempo (tipicamente 20ms). Para analisar

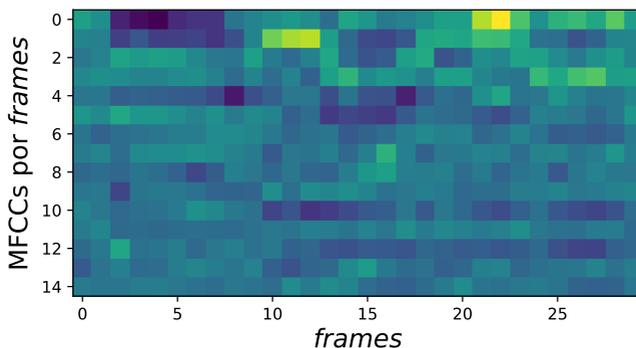
conteúdos de fala com trechos maiores, pode-se utilizar uma matriz de MFCCs

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}^1 & \mathbf{v}^2 & \mathbf{v}^3 & \dots & \mathbf{v}^m \end{bmatrix} \quad (2.22)$$

onde  $\mathbf{V} \in \mathbb{R}^{n \times m}$ ,  $\mathbf{v}^1 \dots \mathbf{v}^m \in \mathbb{R}^{n \times 1}$  são os vetores de MFCCs,  $m$  é o número de vetores utilizados e  $n$  é o tamanho dos vetores de MFCCs.

Para aplicações de reconhecimento de fala utilizando CNNs e MFCCs como *features*, os coeficientes MFCCs são dispostos em matrizes, as quais são tratadas como imagens. A Figura 2.9 apresenta uma matriz de MFCCs  $\mathbf{V} \in \mathbb{R}^{m \times n}$ , onde  $m = 30$  e  $n = 15$ , extraída de um sinal de áudio.

Figura 2.9: Matriz de MFCCs  $\mathbf{V} \in \mathbb{R}^{m \times n}$ , onde  $m = 30$  e  $n = 15$ .



Fonte: do Autor.

## 2.7 Keyword Spotting

*Keyword spotting* (KWS) é uma técnica de identificação de determinadas palavras-chave (em inglês, *keywords*) em um trecho de áudio [8]. Essa técnica é comumente utilizada como *front-end* de sistemas *Automatic Speech Recognition* (ASR). Por exemplo, um dispositivo eletrônico portátil necessita de economia de energia. Deseja-se então que tal dispositivo, que utiliza um sistema ASR, seja acionado apenas nos momentos desejados pelo usuário. Portanto, utiliza-

se KWS para "acordar" o dispositivo com apenas algumas *keywords*, economizando energia [44].

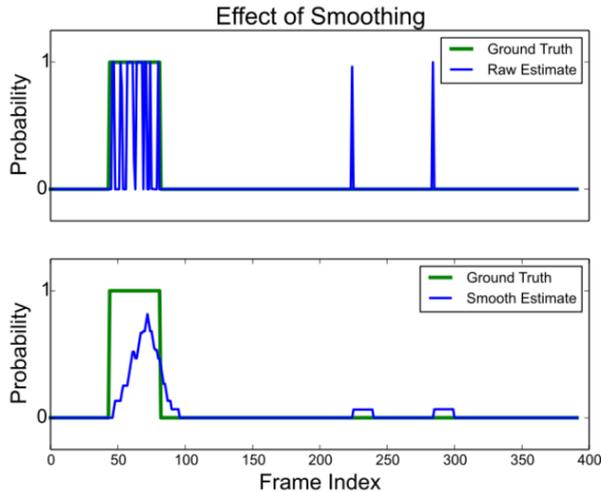
Para utilizar um banco de dados que contém arquivos de fala em um sistema de KWS, deve-se pesquisar termos ou palavras específicas (*keywords*) no respectivo banco. O resultado de tal pesquisa apresenta uma lista de arquivos com ponteiros que indicam a posição das *keywords* encontradas [44]. Posteriormente, esse banco de dados é utilizado no reconhecimento das *keywords* para uma entrada de fala qualquer. Diferentemente de sistemas ASR, um sistema de KWS não necessita da transcrição integral do banco de fala, pois sua principal tarefa é a identificação rápida das *keywords* nos trechos de fala.

### 2.7.1 Métricas de Avaliação

Os resultados de KWS podem conter palavras reconhecidas corretamente e incorretamente. Os reconhecimentos incorretos são definidos como falsos alarmes ou falsos positivos. O *tradeoff* entre reconhecimentos corretos e falsos alarmes é uma métrica de avaliação importante [44]. A distribuição de falsos alarmes e reconhecimentos corretos é medida através de uma função de limiar de distância, determinando o ponto ótimo de trabalho. Esta função, por sua vez, pode ser calculada através da probabilidade de ocorrência de uma *keyword* em cada trecho de áudio. A Figura 2.10 apresenta o gráfico da probabilidade de ocorrência de uma *keyword* [8].

A Figura 2.10 mostra o cálculo da probabilidade da ocorrência de uma *keyword* através do efeito *smoothing* realizado por um filtro média movel. A partir dessa probabilidade é realizado o cálculo da função de limiar de distância, a qual é utilizada para determinar o ponto ótimo de trabalho de acordo com os requisitos do usuário. Assim, dependendo do limiar escolhido, o KWS pode ser mais ou menos permissivo a erros, bem como ser mais ou menos robusto no reconhecimento correto das *keywords*. Por exemplo, com a utilização de um limiar próximo dos falsos positivos na função de distância, o sistema de KWS identifica com maior facilidade as *keywords*, porém se torna mais permissivo a falsas detecções. Já com a utilização de um limiar mais próximo dos reconhecimentos corretos, o sistema de KWS identifica com menor frequência as *keywords*, porém se torna mais robusto a falsas detecções. A partir das detecções realizadas corretamente e das falsas detecções

Figura 2.10: Gráfico da função de probabilidade de uma *keyword* através da suavização por média movel.



Fonte: [8].

são calculadas as taxas de detecções e falsos positivos.

Usualmente, as taxas de detecções e falsos positivos são as principais métricas de avaliação de um sistema de KWS e refletem diretamente nos resultados obtidos [44]. Outra métrica que pode ser utilizada é a taxa de detecções perdidas, em que detecções perdidas são as ocorrências de *keywords* na entrada de fala que não são detectadas. Essa métrica auxilia na avaliação da taxa de detecção do KWS.

Uma importante especificação de uma aplicação que utiliza KWS é seu tempo de resposta [44], que é altamente dependente de segmentos de mercado, podendo variar de aplicação de tempo real, alguns minutos até muitas horas. Em algumas aplicações tais como sistemas *offline* de busca, o tempo de resposta não é muito relevante mas deseja-se a melhor taxa de detecção possível. Em outras aplicações onde deseja-se uma resposta em tempo real, são aceitáveis resultados menos precisos, com taxas de falsos alarmes mais elevadas.

## 2.7.2 Tipos de KWS

Os métodos normalmente utilizados para KWS são [45]: *KWS-based Large Vocabulary Continuous Speech Recognition (LVCSR)*, *Phonetic-Search KWS* e *Acoustic KWS*.

### 2.7.2.1 KWS-based LVCSR

O método *KWS-based LVCSR* utiliza um sistema LVCSR para realizar primeiramente a transcrição textual de um trecho de fala. Posteriormente, é realizada uma busca no texto obtido, identificando as *keywords* e os trechos de fala correspondentes [45].

### 2.7.2.2 Phonetic-Search KWS

Este método realiza uma transcrição fonética de um trecho de fala. Na sequência, é realizada uma busca nesses fonemas identificando os trechos de áudio correspondentes às *keywords* [46].

### 2.7.2.3 Acoustic KWS

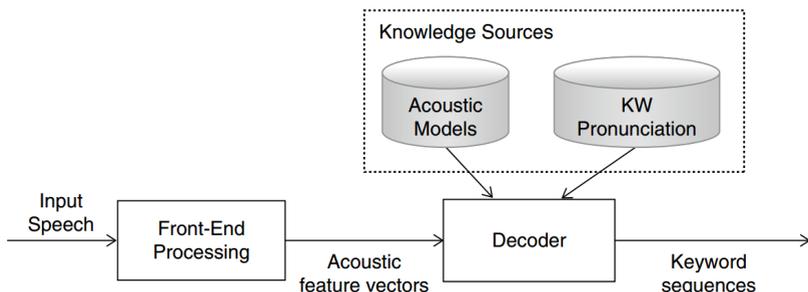
O *Acoustic KWS* não envolve a transcrição da fala e é restrito ao reconhecimento de apenas algumas poucas *keywords* [45]. Neste trabalho, o método *Acoustic KWS* é o utilizado para a implementação do *keyword spotting*, visto que este é o mais simples dos métodos apresentados e tem sua implementação facilitada em FPGA.

Um sistema de *Acoustic KWS* pode ser implementado conforme o diagrama de blocos ilustrado na Figura 2.11.

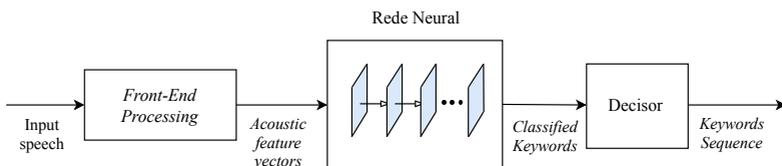
Conforme ilustrado na Figura 2.11, o *Acoustic KWS* é composto pelos blocos *Front-End Processing*, *Knowledge Sources* e *Decoder*. O bloco *Front-End Processing* realiza um pré-processamento na entrada de fala, obtendo vetores de atributos acústicos. Já o bloco *Decoder*, com o auxílio das *Knowledge Sources*, classifica os vetores de atributos acústicos, obtendo na saída a sequência das *keywords* identificadas.

## 2.7.3 Acoustic KWS Utilizando ANNs

A aplicação de redes neurais em *Acoustic KWS* pode ser representada através do diagrama de blocos ilustrado na Figura 2.12.

Figura 2.11: Um sistema *acoustic keyword spotting*.

Fonte: [44].

Figura 2.12: *Acoustic keyword spotting* utilizando redes neurais.

Fonte: do Autor.

Conforme apresentado na Figura 2.12, tal aplicação é composta pelos blocos *Front-End Processing*, Rede Neural e Decisor. O bloco *Front-End Processing* possui a mesma função do bloco apresentado na Figura 2.11, obtendo os vetores de atributos acústicos. O bloco Rede Neural realiza a classificação dos vetores de atributos acústicos, obtendo na saída a sequência de classificação das *keywords*. Já o bloco Decisor realiza a suavização da sequência de classificação e, através de *thresholds*, calcula a função de limiar de distância, obtendo a sequência de *keywords* identificadas. Comparando as Figuras 2.11 e 2.12, nota-se que a junção dos blocos Rede Neural e Decisor (Figura 2.12) possui a mesma função que a junção dos blocos *Decoder* e *Knowledge Sources* (Figura 2.11). Assim, os blocos Rede Neural e Decisor realizam o *Decoder* através das operações matemáticas de cada camada (Seção 2.1.3) e da função de limiar de distância, e as informações contidas no bloco *Knowledge Sources* são distribuídas e fixadas em cada peso da

Rede Neural, através de um treinamento prévio.

### 2.7.3.1 Aplicações

Em [7, 8] são implementados sistemas de *Acoustic KWSs* que podem ser representados pelo diagrama de blocos mostrado na Figura 2.12. Em [7], a Rede Neural utilizada é do tipo CNN e o bloco *Front-End Processing* é um extrator de *Log Mel-Filterbank Energy features* (LMFE). Já em [8], a Rede Neural utilizada é do tipo MLP convencional e o bloco *Front-End Processing* é um extrator de MFCCs (Seção 2.6).

## 2.8 FPGA

O FPGA é um dispositivo semicondutor composto por *configurable logic blocks* (CLBs) conectados por meio de chaves e matrizes de conexão (em inglês, *switch matrices*) [47]. As *switch matrices* também podem estabelecer conexões entre os blocos I/O e os CLBs. O FPGA é configurado por um arquivo de *bitstream* que pode ser gerado utilizando códigos descritos em *Very high speed integrated circuit Hardware Description Language* (VHDL) e Verilog.

Os FPGAs, quando comparados com outros dispositivos (DSPs, CPUs, GPUs), possuem um elevado desempenho no consumo de energia e tempo de processamento, sendo até 15 vezes mais rápidos e consumindo até 61 vezes menos energia em comparação com as GPUs [48]. Em contrapartida, a duração no desenvolvimento de implementações em dispositivos FPGAs podem ser até 12 vezes mais longa [49].

O uso de FPGAs para a implementação de ANNs vem apresentando grande interesse das grandes empresas de tecnologia, tais como Intel e a Microsoft. Por exemplo, o projeto *brainwave* da Microsoft utiliza FPGAs para a implementação de *Neural Processing Units* (NPU) para realizar a inferência usando DNNs, realizando tarefas de regressão e classificação com alta performance [50].

## 2.9 Verilog

Verilog é uma linguagem de descrição de *hardware* (em inglês, *hardware description language*) utilizada para descrever circuitos eletrônicos e

sistemas [51]. Esta linguagem é mais utilizada na área de projeto e verificação de circuitos digitais em *Application Specific Integrated Circuits* (ASICs) e FPGAs. Um projeto em Verilog permite, através de unidades lógicas e registradores, implementar qualquer algoritmo, podendo então se beneficiar do paralelismo e de técnicas de *pipeline*.

## 2.10 Python

*Python* é uma linguagem de programação interpretada, orientada a objetos e de alto nível, com semântica dinâmica [52]. Tal linguagem possui uma considerável documentação e é cada vez mais utilizada. O *Python* possui diversas bibliotecas, tais como: bibliotecas de manipulação de arquivos, processamento de sinais, manipulação de tabelas, operações matriciais, leitura de diversos tipos de arquivos, desenvolvimento de ANNs (treinamento e teste), etc. Atualmente, a linguagem *Python* é a mais utilizada em trabalhos de pesquisa envolvendo inteligência artificial e ANNs, devido a ampla documentação, facilidade e aderência dos usuários a esta linguagem.

## 2.11 Tensorflow

*TensorFlow* é uma biblioteca *open source*, desenvolvida pela empresa Google e utilizada em problemas de *machine learning* [53]. O *tensorflow* vem se tornando a biblioteca padrão para desenvolvimento em *deep learning* e inteligência artificial. Uma das principais características de tal biblioteca é a capacidade de facilmente gerar um produto ou serviço a partir do modelo preditivo treinado, eliminando a necessidade de reimplementação o modelo [54]. A biblioteca *TensorFlow* oferece suporte para programação em C++, Java e *Python*.



# CAPÍTULO 3

---

## Desenvolvimento

---

Neste capítulo, o desenvolvimento do presente trabalho é apresentado. Inicialmente a visão geral do projeto proposto é apresentada, mostrando o desenvolvimento na linguagem *Python* e a implementação em FPGA. Na sequência, são detalhadas as etapas de desenvolvimento em *Python* que consistem na elaboração do banco de fala, escolha dos parâmetros de rede e dos MFCCs, implementação de cenários, treinamento e avaliação. Finalmente, é apresentada a implementação em FPGA dos blocos *Framing*, Extrator de MFCCs, CNN e Decisor, além das técnicas de quantização dinâmica e de geração automática de código propostas.

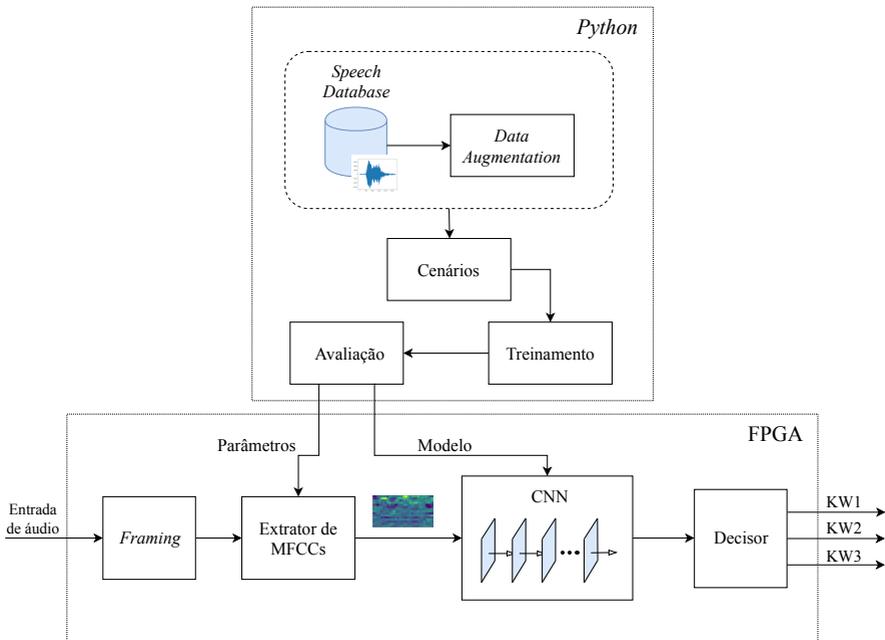
### 3.1 Visão Geral

Como mencionado anteriormente, este trabalho é dedicado à implementação em FPGA de um sistema de *Acoustic* KWS a partir de um banco de fala em português. A topologia utilizada para *Acoustic* KWS é apresentada na Seção 2.7.3, sendo o *front-end processor* um extrator de MFCCs e a rede neural do tipo CNN. A Figura 3.1 mostra um diagrama de blocos que ilustra o processo de desenvolvimento realizado neste trabalho. Conforme mostrado nessa

figura, o desenvolvimento é dividido em duas partes: uma envolvendo a programação em *Python* e a outra a implementação em FPGA. Na primeira parte, são obtidos os parâmetros do extrator de MFCCs e o modelo da CNN. Já na segunda parte, é realizada a implementação em FPGA do KWS.

O algoritmo em *Python* é composto pelos seguintes blocos: *Speech Database*, *Data Augmentation*, Cenários, Treinamento e Avaliação. Já a implementação em FPGA do sistema de *Acoustic KWS* é composta pelos seguintes blocos: *Framing*, Extrator de MFCCs, CNN e Decisor.

Figura 3.1: Visão geral do projeto proposto neste trabalho.



Fonte: do Autor.

## 3.2 Desenvolvimento em *Python*

Esta seção é dedicada à primeira parte do desenvolvimento do trabalho, realizada usando programação em *Python*.

### 3.2.1 *Speech Database*

O *Speech Database* (veja Figura 3.1) é um banco de dados reduzido composto por um conjunto de arquivos de áudio, tendo ele sido desenvolvido para a aplicação na língua portuguesa. Este banco de dados pode ser utilizado como *knowledge source* para *Acoustic KWS* (Seção 2.7.2.3). A Tabela 3.1 mostra a organização do *Speech Database* utilizado.

Tabela 3.1: Organização do *Speech Database* utilizado no trabalho.

	ID	OOV	KW
Sinais com Locutor	1	✓	
	2		✓
	3	✓	✓
	⋮	⋮	⋮
	$N_l-2$	✓	✓
	$N_l-1$	✓	✓
	$N_l$		✓
Ruídos sem Locutores	1	✓	
	2	✓	
	3	✓	
	⋮	⋮	⋮
	$N_n-1$	✓	
	$N_n$	✓	

Fonte: do Autor.

Conforme mostrado na Tabela 3.1, o banco de dados é dividido em arquivos de áudio com e sem locutor. Os locutores são identificados através de um ID de 1 a  $N_l$  e os arquivos sem locutor de 1 a  $N_n$ . O banco de dados possui um total de 72 locutores ( $N_l = 72$ ) correspondendo a 140 minutos de áudio. Já o conjunto sem locutor (diferentes tipos de ruídos) é composto por 961 arquivos ( $N_n = 961$ ), correspondendo a 400 minutos de áudio.

Os arquivos de áudio são classificados em *Out Of Vocabulary* (OOV) ou *Key Word* (KW). Os arquivos da classe OOV não possuem ocorrências de *keywords*. Já os arquivos da classe KW contém pelo

menos uma ocorrência de todas as *keywords*. Os arquivos sem locutor são compostos por sons ambientes (ruído murmurinho, escritório, etc.) e pertencem apenas à classe OOV. Já os arquivos com locutor possuem apenas um locutor e podem pertencer tanto à classe OOV quanto à KW. Os arquivos com locutores da classe OOV possuem trechos de voz com diferentes assuntos. Essa divisão é realizada para facilitar o processamento dos arquivos de áudio. Os arquivos de ruído foram extraídos do banco de dados MUSAN [55]. O banco de dados desenvolvido no presente trabalho (Tabela 3.1) está disponível em [56].

Para a utilização do *Speech Database* em *Acoustic KWS*, inicialmente, deve-se definir as *keywords* desejadas. Após tal definição, é realizada a procura nos arquivos da classe KW, obtendo uma lista de ponteiros indicando a localização das ocorrências das *keywords*. Para elaborar o *dataset* com amostras que não são *keywords*, é gerado um conjunto de  $N_{OOV_p}$  ponteiros localizados nos arquivos pertencentes a classe OOV. Tais ponteiros são posicionados aleatoriamente, desejando uma generalização dos arquivos OOVs escolhidos.

### 3.2.2 Data Augmentation

Como o *Speech Database* é considerado um banco de dados reduzido, utilizou-se da técnica de *data augmentation* para expandir tal banco de dados. Os tipos de transformações de fala utilizados no presente trabalho são apresentados na Seção 2.5.1.

### 3.2.3 Cenários

Para avaliar a influência do *data augmentation* e do acréscimo de arquivos de ruído no treinamento, foram elaborados 5 cenários extraíndo diferentes *datasets* dos *Knowledge Sources*. A Tabela 3.2 mostra os diferentes cenários com suas especificações.

As colunas *Add-Noise*, *Noise Features* e *Data Augmentation* representam, respectivamente, a adição de ruído ambiente nas amostras de *keywords*, a utilização dos arquivos de ruído do *Speech Database* (Tabela 3.1) e, por último, o uso de amostras obtidas pela técnica de *data augmentation* com as transformações de *pitch shift* e *time stretch*. A *Signal Noise Ratio* (SNR) entre as amostras de *keywords* e o ruído adicionado é de, no mínimo, 25 dB. Na técnica de *data augmentation*

Tabela 3.2: Especificações dos diferentes cenários utilizados.

	ID	Add-Noise	Noise Features	Data Augmentation
Cenários	1			
	2	✓		
	3	✓	✓	
	4	✓		✓
	5	✓	✓	✓

Fonte: do Autor.

utilizada, as variações tonais consideradas nas transformações de *pitch shift* são de -1 a 1. Já nas transformações de *time stretch*, foram consideradas as variações de expansão de 0.9 a 1.1.

As *keywords* consideradas nos cenários são: "Sim Máquina", "Abre" e "Fecha". Portanto, as classes utilizadas nas CNNs são: 0="OOV", 1="Sim Máquina", 2="Abre", 3="Fecha".

### 3.2.3.1 Parâmetros do Extrator de MFCCs

Os parâmetros utilizados na extração dos MFCCs foram apresentados na Seção 2.6. Os valores atribuídos a esses parâmetros estão de acordo com as recomendações encontradas na literatura [36–42] e foram obtidos empiricamente. Para os cenários considerados, foram utilizados os mesmos valores dos parâmetros, os quais são organizados nas etapas apresentadas na Seção 2.6.1.

Na etapa de Conversão para *Frames* (Seção 2.6.1.1), os parâmetros utilizados são:  $f_s = 8$  kHz,  $N_{f_r} = 512$ ,  $N_{hop} = 256$  e  $window = \text{Hanning}$ , onde  $f_s$  é a taxa de amostragem do sinal de entrada,  $N_{f_r}$  é o número de amostras utilizadas em cada *frame*,  $N_{hop}$  é o número de amostras por passo e  $window$  é a janela utilizada. Portanto, a duração dos *frames* é de  $\frac{N_{f_r}}{f_s} = 64$  ms e a sobreposição é de  $\frac{N_{hop}}{N_{f_r}} = 50\%$ . Na etapa do Cálculo do Periodograma da Potência Espectral (Seção 2.6.1.2), o número de *bins* da FFT é  $N_{FFT} = 512$ . Na etapa de Filtragem com *Mel-Bank Filters* (Seção 2.6.1.3), os parâmetros utilizados são:  $f_i = 100$ ,  $f_o = 4000$  e  $N_f = 64$ . Na etapa do Cálculo da DCT (Seção 2.6.1.5), são utilizados os coeficientes DCT de 3 a 17. Por último, considerando a representação matricial (Seção 2.6.2), são

utilizados 35 *frames* MFCCs ( $m = 35$ ). Portanto, as matrizes de MFCCs obtidas pelo extrator de MFCCs correspondem a *frames* de duração de  $\frac{N_{hop} \cdot m}{f_s} = 1.12$  s.

### 3.2.3.2 Criação dos *Datasets*

Após a extração dos MFCCs, são elaborados os conjuntos de treinamento, teste e validação. Conforme apresentado na Seção 2.4, os locutores utilizados no conjunto de treinamento não são utilizados nos conjuntos de teste e validação. Para cada cenário (Tabela 3.2), são elaborados 4 diferentes trios de *datasets* (treinamento, teste e validação), sendo que, para cada trio, os locutores são separados de forma aleatória. Tal estratégia é realizada para auxiliar na generalização do KWS. A proporção entre os 3 conjuntos é dada por 80% para treinamento, 10% para teste e 10% para validação.

Em problemas de classificação, recomenda-se que os *datasets* sejam balanceados, ou seja, o número de observações de ocorrência é o mesmo para todas as classes do *dataset*. Para balancear os *datasets* considerados em todos os cenários, é utilizada a técnica de *data augmentation* com a transformação *time shift*, utilizando um fator de  $d_1 = 0.4$  (Seção 2.5.1). Tal transformação permite retirar a influência do deslocamento do trecho de fala na classificação, sem alterar os parâmetros da voz.

A Tabela 3.3 mostra o tamanho dos conjuntos de treinamento utilizados para cada cenário.

Tabela 3.3: Tamanho por classe dos conjuntos de treinamento.

	ID	Total	OOV	KW 1	KW 2	KW 3
Cenários	1	34278	8808	8490	8490	8490
	2	34278	8808	8490	8490	8490
	3	35228	9758	8490	8490	8490
	4	49290	13380	12180	12180	12180
	5	50816	14276	12180	12180	12180

Fonte: do Autor.

Observa-se, na Tabela 3.3, que o conjunto de treinamento dos Cenários 1 e 2 é do mesmo tamanho, já que a diferença entre esses

cenários é a adição de ruído nas amostras de *keywords*. Já para o Cenário 5 observa-se que o tamanho do conjunto de treinamento é de 50816 amostras, devido ao acréscimo de amostras de ruído e de *data augmentation*.

### 3.2.4 Treinamento

A ferramenta utilizada para elaborar, treinar e avaliar as CNNs é o *TensorFlow* (Seção 2.11). Para realizar o treinamento, são elaboradas 8 estruturas de CNN distintas escolhidas empiricamente. Tais estruturas são apresentadas em [56]. Para cada estrutura de CNN e em cada cenário, são treinados 20 modelos de CNN, 5 para cada trio de *datasets*. Os modelos são treinados em 150 épocas e o treinamento é realizado conforme apresentado na Seção 2.4. Para cada treinamento, é analisada a acurácia no conjunto de validação. O modelo referente à época que obteve a maior acurácia é salvo. Posteriormente, esses modelos são utilizados para inferir sobre o conjunto de teste. A Tabela 3.4 mostra a acurácia do melhor modelo de cada cenário, na inferência do conjunto de teste.

Tabela 3.4: Acurácia dos conjuntos de teste pelos modelos salvos.

	ID	Acurácia	Estrutura CNN
Cenários	1	94.68%	8
	2	96.93%	7
	3	98.37%	8
	4	96.73%	8
	5	97.72%	8

Fonte: do Autor.

Observa-se, na Tabela 3.4, que a estrutura de CNN que obteve os melhores resultados é a 8. Os cenários que obtiveram a maior acurácia foram o 3 e o 5. Portanto, conclui-se que a adição de amostras de ruído no *dataset* e a adição de ruído nas amostras de KW levaram a um aprimoramento do resultado da CNN.

### 3.2.5 Avaliação

Após obter os melhores modelos de CNN para cada cenário, são avaliados os resultados na execução do KWS. Conforme apresentado na Figura 2.12, a saída da Rede Neural é enviada para o bloco Decisor. Tal bloco realiza a suavização da sequência de classificações e, através de *thresholds*, realiza a função de limiar de distância, obtendo a sequência de *keywords* identificadas. O número de *thresholds* utilizados pelo Decisor é equivalente ao número de *keywords* que o KWS é capaz de identificar. Portanto, neste trabalho o bloco Decisor possui 3 *thresholds*.

A escolha dos *thresholds* no Decisor afeta diretamente no desempenho do sistema de KWS, podendo variar a taxa de falsos alarmes e de reconhecimentos corretos. Neste trabalho, é utilizado um algoritmo *brute-force search* [57] para determinar os melhores *thresholds* para o Decisor. Deseja-se obter um KWS que possua a taxa de reconhecimentos corretos equivalente para cada *keyword*. Portanto, o *score* utilizado neste algoritmo é a média geométrica entre as acurácias de cada classe. A Tabela 3.5 mostra os *thresholds* e os *scores* obtidos pelo algoritmo *brute-force search*.

Para realizar a avaliação, são utilizados os arquivos de áudio do *Speech Database* (Tabela 3.1) e avaliadas as ocorrências de *keywords*, a partir dos *thresholds* escolhidos. Também é avaliado o número de falsos positivos nos arquivos OOV com e sem locutor. A Tabela 3.6 mostra os resultados do KWS para cada cenário, considerando as taxas de reconhecimentos corretos das *keywords* e de falsos positivos em arquivos de fala e de ruído.

Tabela 3.5: Escolha dos *thresholds* do Decisor para os cenários.

	ID	Thresholds	Score
Cenários	1	(0.6, 0.5, 0.6)	0.958
	2	(0.5, 0.4, 0.3)	0.959
	3	(0.6, 0.3, 0.3)	0.971
	4	(0.5, 0.2, 0.4)	0.969
	5	(0.5, 0.4, 0.3)	0.974

Fonte: do Autor.

Observa-se na Tabela 3.5 que o Cenário 5 foi o que levou ao maior

		Taxa Detecção Correta			Taxa Falso Positivo		
		ID	KW 1	KW 2	KW 3	Voz	Ruído
Cenários	1	95.77%	99.72%	98.87%	10.58%	56.02%	
	2	97.18%	97.18%	96.90%	7.22%	69.53%	
	3	97.46%	99.15%	96.62%	4.62%	1.40%	
	4	98.03%	99.72%	96.90%	6.64%	71.36%	
	5	98.31%	99.15%	97.75%	5.31%	1.73%	

Tabela 3.6: Resultado do KWS para os diferentes os cenários.

valor de *score*. Portanto, mesmo que o treinamento neste cenário tenha sido realizado utilizando amostras sintetizadas pelo *data augmentation*, quando avaliado com apenas os arquivos do *Speech Database* (Tabela 3.1), ele levou ao melhor resultado. Observa-se na Tabela 3.6 que péssimos resultados na taxa de falsos positivos em arquivos de ruído foram obtidos para os Cenários 1, 2 e 4. Tal fato ocorre pela falta de amostras de ruído (*Noise Features*, veja Tabela 3.2) nos *datasets* durante o treinamento. Os melhores resultados nas taxas de detecções corretas foram obtidos para o Cenário 5 quando comparados com os do Cenário 3, ao custo de um resultado ligeiramente pior nas taxas de falsos positivos.

A partir dos resultados obtidos, são escolhidos os parâmetros e o modelo referente ao Cenário 5 para a implementação em FPGA.

### 3.3 Desenvolvimento para o FPGA

Nesta seção, é detalhada a implementação dos blocos do sistema de KWS em FPGA, mostrada na Seção 3.1. Nesta implementação, é considerado o Cenário 5 mostrado na Seção 3.2.3. A implementação em FPGA é dividida nos blocos: *Framing*, Extrator de MFCCs, CNN e Decisor. Tais blocos são sincronizados por um mesmo sinal de *clock*, o qual é a referência de tempo na avaliação dos blocos.

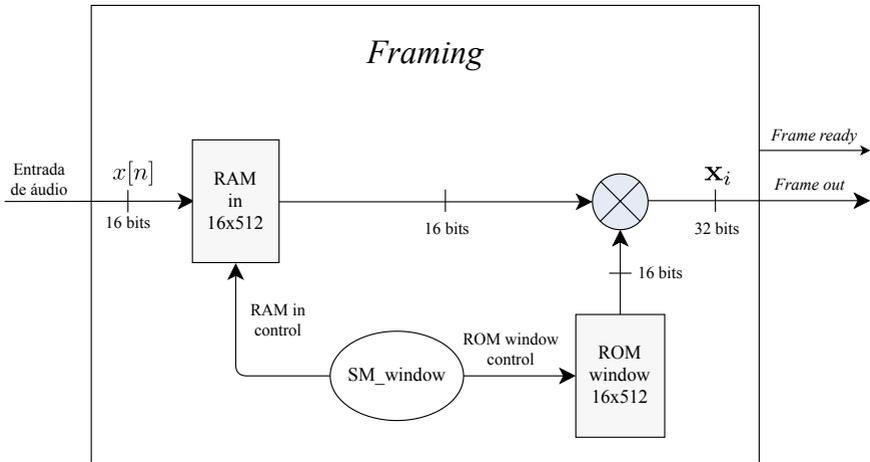
#### 3.3.1 Framing

*Framing* é o bloco de entrada da implementação em FPGA do sistema de KWS. Tal bloco é responsável por separar as amostras de entrada

em *frames* ( $N_{fr} = 512$  amostras, passo de  $N_{hop} = 256$  amostras) e realizar o janelamento. Conforme apresentado na Seção 3.2.3.1, a taxa de amostragem  $f_s$  do sinal de entrada é de 8 kHz e a janela é do tipo *hanning*.

A Figura 3.2 mostra o diagrama de blocos do *Framing*. Observe nessa figura que o *Framing* é composto por uma memória RAM de tamanho  $16 \times 512$  (512 elementos de 16 bits), uma memória ROM de  $16 \times 512$  e por uma máquina de estados *SM\_window*. A entrada de áudio  $x[n]$  é de 16 bits e a saída  $\mathbf{x}_i$  de 32 bits. Na memória ROM são armazenados os coeficientes da janela *Hanning* de  $N_{fr}$  amostras. Para cada *frame*, são armazenadas  $N_{hop}$  amostras na memória RAM para realizar o janelamento. A saída  $\mathbf{x}_i$  corresponde a um vetor de  $N_{fr}$  elementos. Após obter um *frame*, um *flag* (*Frame ready*) sinaliza para o próximo bloco (Extrator de MFCCs) que há um vetor disponível para processamento.

Figura 3.2: Diagrama de blocos do *Framing*.



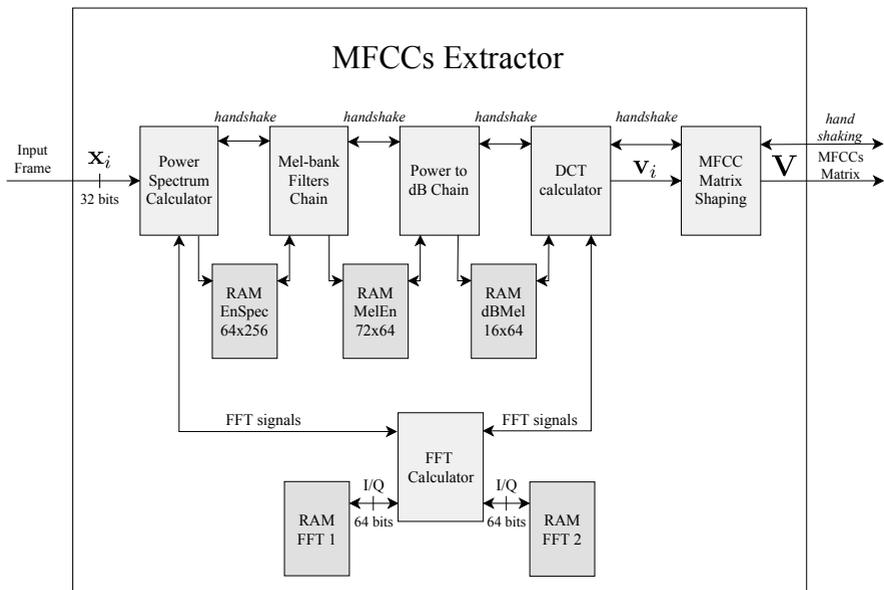
Fonte: do Autor.

### 3.3.2 Extrator de MFCCs

O bloco Extrator de MFCCs é responsável pelo cálculo dos coeficientes MFCCs dos frames  $\mathbf{x}_i$  e disponibilizá-los na forma matricial ( $\mathbf{V}$ ) para o bloco CNN. A Figura 3.3 mostra o diagrama de blocos do Extrator de

MFCCs. Observa-se nesta figura que o Extrator de MFCCs é composto pelos seguintes blocos: *Power Spectrum Calculator*, *FFT Calculator*, *Mel-bank Filters Chain*, *Power to dB Chain*, *DCT calculator* e *MFCC Matrix Shaping*.

Figura 3.3: Diagrama de blocos do Extrator de MFCCs.



Fonte: do Autor.

### 3.3.2.1 *FFT Calculator*

O bloco *FFT Calculator* utilizado neste trabalho foi desenvolvido em [58] e configurado para 512 *bins*. Esse bloco utiliza duas memórias RAM dual-port de tamanho  $64 \times 512$  para armazenar as etapas da FFT. As amostras armazenadas nessas RAMs são do tipo complexo ( $\mathbb{C}$ ), em que as partes real e a imaginária são de 32 bits.

O bloco *FFT Calculator* foi desenvolvido utilizando técnicas de paralelismo e de *pipeline*, permitindo o cálculo de FFTs de 512 *bins* utilizando um *clock* de frequência acima de 300 MHz. Tais técnicas também permitiram realizar o cálculo da FFT em apenas 2523 ciclos de *clock*.

### 3.3.2.2 *Power Spectrum Calculator*

Este bloco é responsável por calcular a potência do espectro (Seção 2.6.1.2) do vetor  $\mathbf{x}_i$ . Tal bloco utiliza o *FFT Calculator* para obter os *bins* da FFT. De posse dos *bins*, é calculada a potência do espectro estimada do *frame*, gerando 256 valores de 64 bits, que são salvos na *RAM EnSpec*.

### 3.3.2.3 *Mel-bank Filters Chain*

Este bloco realiza a operação de filtragem do vetor armazenado na *RAM EnSpec* com os coeficientes dos *Mel-bank filters*. A filtragem é realizada utilizando o cálculo matricial apresentado na Seção 2.6.1.3. A matriz de filtros *mel* é armazenada em uma ROM de tamanho  $8 \times 16384$  (matriz de  $256 \times 64$  com resolução de 8 bits). A saída do bloco é um vetor de 64 elementos de 72 bits, que é salvo na *RAM MelEn*.

### 3.3.2.4 *Power to dB Chain*

Neste bloco é realizado o cálculo do logaritmo sobre os valores armazenados na *RAM MelEn*. Inicialmente, é realizado o cálculo em ponto fixo do logaritmo na base 2. Posteriormente, tal valor é multiplicado por uma constante, produzindo um resultado em dB com resolução de 16 bits que é salvo na memória *RAM dBMel*.

### 3.3.2.5 *DCT calculator*

Este bloco realiza o cálculo da DCT usando o *FFT calculator*. Tal estratégia foi proposta em [59], que realiza um embaralhamento nos elementos do vetor de entrada da FFT, permitindo que o bloco *FFT calculator* realize a DCT de forma rápida. Após o cálculo da DCT, os coeficientes MFCCs de 3 a 17 (vetor contendo 15 elementos) são enviados, com resolução de 16 bits, ao bloco *MFCC Matrix Shaping*.

### 3.3.2.6 *MFCC Matrix Shaping*

Este bloco é responsável por disponibilizar os vetores de MFCCs na forma matricial (Seção 2.6.2). O bloco possui uma memória RAM de tamanho  $16 \times 525$  (35 vetores de 15 elementos com resolução de 16 bits) utilizada para armazenar os últimos 35 vetores de MFCCs gerados pelo

*DCT calculator.* Para cada vetor de MFCCs escrito nesta memória, um *flag (Matrix ready)* sinaliza ao próximo bloco (CNN) que há uma matriz de MFCCs disponível para processamento.

### 3.3.3 CNN

Conforme descrito na Seção 3.3, a implementação do bloco CNN em FPGA utiliza o Cenário 5, apresentado na Seção 3.2.3. A CNN é implementada em ponto fixo com resolução de 16 bits, ou seja, as entradas e saídas de todas as camadas possuem essa resolução.

A Tabela 3.7 descreve as camadas utilizadas na CNN, os blocos que compõem cada camada e os formatos de entrada e saída correspondentes aos blocos.

Tabela 3.7: Implementação em FPGA da CNN baseada no cenário 5.

Camada	Bloco	Formato de Entrada	Formato de Saída
1	Conv1	15x35x1	11x31x4
	Bnorm1	11x31x4	11x31x4
	ReLU1	11x31x4	11x31x4
2	Conv2	11x31x4	8x27x6
	ReLU2	8x27x6	8x27x6
3	Conv3	8x27x6	7x23x8
	ReLU3	7x23x8	7x23x8
4	Conv4	7x23x8	6x19x10
	ReLU4	6x19x10	6x19x10
5	Conv5	6x19x10	6x16x12
	ReLU5	6x16x12	6x16x12
6	MaxPooling	6x16x12	3x8x12
7	Conv6	3x8x12	2x6x14
	ReLU6	2x6x14	2x6x14
8	FullC1	168	100
	ReLU7	100	100
9	FullC2	100	4

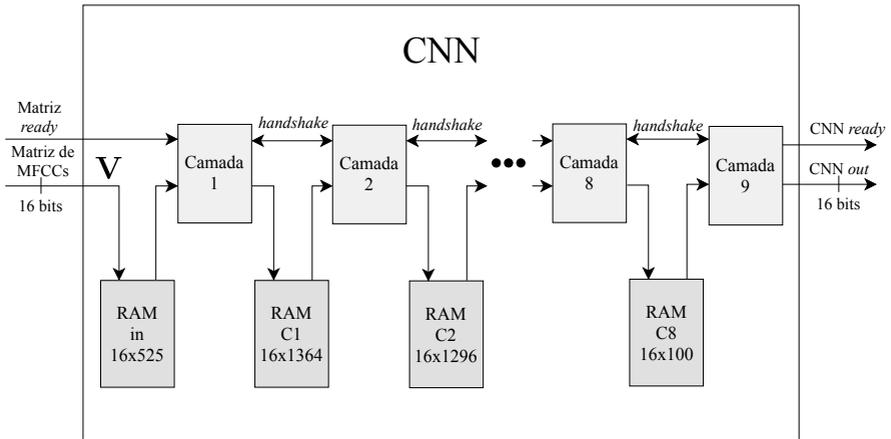
Fonte: do Autor.

Conforme apresentado na Tabela 3.7, a CNN possui 9 camadas que são compostas pelos blocos do tipo **Conv**, **Bnorm**, **ReLU**, **MaxPooling** e **FullC**. Tais blocos implementam, respectivamente, os cálculos da *convolutional layer* (Seção 2.1.3.2), *batch normalization* (Seção 2.5.2), ReLU (Seção 2.1.2.2), *pooling layer* (Seção 2.1.3.3) e *fully connected layer* (Seção 2.1.3.1).

Conforme apresentado na Tabela 3.7, o formato de entrada da CNN é  $15 \times 35 \times 1$ , que corresponde ao formato da matriz de MFCCs gerada no bloco Extrator de MFCCs. Já o formato do bloco de saída da CNN (**FullC2**) é 4, que corresponde ao número de classes da CNN.

A Figura 3.4 mostra o diagrama de blocos da implementação da CNN em FPGA.

Figura 3.4: Diagrama de blocos da CNN implementada em FPGA.



Fonte: do Autor.

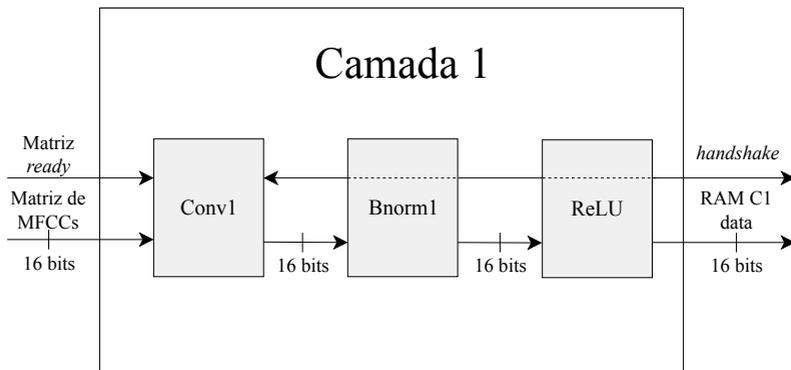
Observa-se na Figura 3.4 que há uma memória RAM na saída de cada camada, exceto na última. A Matriz de MFCCs (**V**), de formato  $15 \times 35$  e resolução de 16 bits, é armazenada em uma memória RAM de  $16 \times 525$ . As camadas se comunicam através de um *handshake* e armazenam os resultados em suas respectivas memórias RAM.

Neste trabalho, os blocos da CNN são implementados separadamente, ou seja, os cálculos são realizados em diferentes blocos. Além disso, as camadas da CNN são organizadas de forma sequencial. Portanto, tais camadas podem realizar o processamento

em paralelo, desde que suas memórias de entrada e saída estejam disponíveis. Assim, camadas consecutivas (e.g., Camadas 1 e 2) não podem realizar o processamento simultaneamente, dado que há uma memória compartilhada entre tais camadas. Por exemplo, em um instante  $t$ , enquanto a Camada 3, que utiliza as memórias RAM C2 e RAM C3, realiza o processamento de uma matriz  $\mathbf{V}_k$ , a Camada 1, que utiliza as memórias RAM  $in$  e RAM C1, pode realizar o processamento da matriz  $\mathbf{V}_{k+1}$ . Considerando o processamento em paralelo nas camadas da CNN, a restrição de processamento é dada pelo tempo necessário para processar as duas camadas consecutivas de maior complexidade computacional.

A Figura 3.5 mostra o diagrama de blocos da Camada 1 da CNN.

Figura 3.5: Diagrama de blocos da Camada 1 da CNN implementada em FPGA.



Fonte: do Autor.

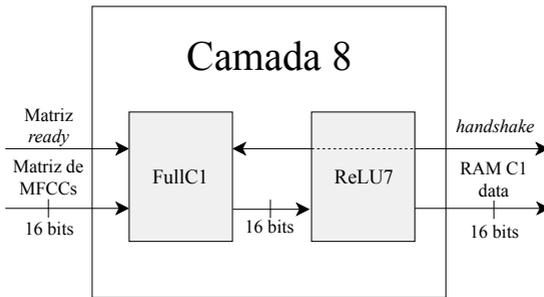
Observa-se, na Figura 3.5, que os blocos que compõem a Camada 1 são: **Conv1**, **Bnorm1** e **ReLU**. O bloco **Conv1** lê as amostras (elementos da matriz  $\mathbf{V}$ ) da RAM de entrada (RAM  $in$   $16 \times 525$ ) e realiza os cálculos da *convolutional layer* (Seção 2.1.3.2). Os resultados do bloco **Conv1** são enviados ao **Bnorm1** que realiza os cálculos do *batch normalization*. Conforme apresentado na Seção 2.5.2, o *batch normalization* é comumente utilizado antes da função de ativação. Já o bloco **ReLU** realiza a função de ativação e envia os resultados para a memória de saída (RAM C1  $16 \times 1364$ ). A sinalização *handshake* é realizada pelo bloco **Conv1**, dado que este necessita das amostras

disponíveis nas memórias.

As Camadas 2, 3, 4, 5 e 7 utilizam basicamente os mesmos blocos (**Conv** e **ReLU**) que a Camada 1, diferindo apenas nos formatos de entrada e saída e no número de filtros utilizados em cada camada. A Camada 6 é o bloco **MaxPooling** e as Camadas 8 e 9 utilizam os blocos **FullC** e **ReLU**.

A Figura 3.6 mostra o diagrama de blocos da Camada 8.

Figura 3.6: Diagrama de blocos da Camada 8 da CNN implementada em FPGA.



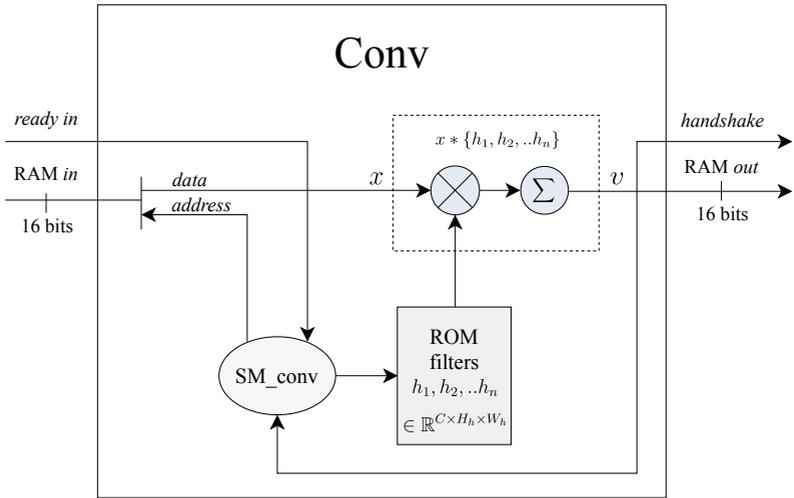
Fonte: do Autor.

Observa-se, na Figura 3.6, que o funcionamento da Camada 8 é similar ao da Camada 1. Os blocos que compõem a Camada 8 são: **FullC1** e **ReLU7**. O bloco **FullC1** realiza as operações da *fully connected layer* (Seção 2.1.3.1), enviando um vetor de saída (100 elementos) ao bloco **ReLU7**, que realiza o cálculo da função de ativação e envia os resultados para a memória de saída (RAM C8  $16 \times 100$ ). De forma similar a Camada 1, a sinalização *handshake* é realizada pelo bloco **FullC1**, que é o bloco que necessita das amostras armazenadas em memória.

### 3.3.3.1 Bloco Conv

Conforme descrito na Seção 3.3.3, o bloco do tipo **Conv** realiza os cálculos da *convolutional layer* apresentados na Seção 2.1.3.2, com os formatos de entrada e saída e número de filtros parametrizados. A Figura 3.7 mostra o diagrama de blocos do **Conv**. Conforme mostrado nessa figura, tal bloco possui uma memória ROM utilizada para armazenar os coeficientes dos filtros  $h_1, h_2, \dots, h_n$ .

Figura 3.7: Diagrama de blocos do **Conv** utilizado na CNN implementada em FPGA.



Fonte: do Autor.

O bloco **Conv** necessita de uma RAM de entrada para realizar os cálculos da convolução. A máquina de estados *SM\_conv* controla a leitura dos dados das memórias *RAM in* e *ROM filters* utilizados para realizar os cálculos de convolução. As amostras de saída do bloco correspondem aos elementos do tensor  $v \in \mathbb{R}^{n \times H_v \times W_v}$ , definido em (2.8), e possuem resolução de 16 bits.

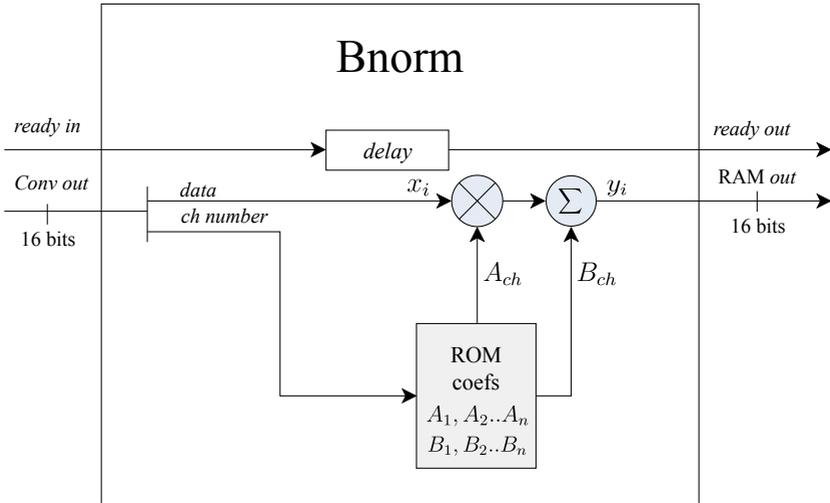
Todos os blocos **Conv** utilizados neste trabalho utilizam o formato *valid* em todas as dimensões. Após finalizados os cálculos da convolução, um *flag* (*handshake*) sinaliza ao próximo bloco que um tensor  $v$  está disponível.

### 3.3.3.2 Bloco Bnorm

O bloco **Bnorm** é utilizado para calcular o *batch normalization* apresentado na Seção 2.5.2. Diferentemente do bloco **Conv**, o **Bnorm** não necessita de memória RAM de entrada. A Figura 3.8 mostra o diagrama de blocos da implementação do **Bnorm**. Observa-se nessa figura que o bloco **Bnorm** realiza os cálculos do *batch normalization* de acordo com (2.12). A *flag* (*ready in*) enviada pelo bloco anterior

(**Conv1**) é sincronizada com o sinal  $y_i$ , através de um *delay*.

Figura 3.8: Diagrama de blocos do **Bnorm** utilizado na CNN implementada em FPGA.



Fonte: do Autor.

### 3.3.3.3 Bloco ReLU

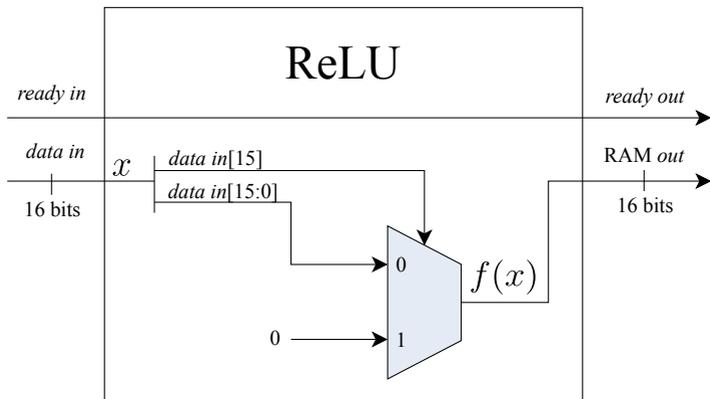
O bloco **ReLU** realiza o cálculo da função de ativação apresentado na Seção 2.1.2.2. Tal função pode ser representada por (2.4), em que a saída  $f(x)$  é o máximo entre 0 e a entrada  $x$ . Portanto, o bloco **ReLU** analisa o bit de sinal da amostra de entrada e realiza a escolha entre 0 e a amostra.

A Figura 3.9 mostra o diagrama de blocos da **ReLU** implementada.

### 3.3.3.4 Bloco MaxPooling

O bloco **MaxPooling** (Camada 7) realiza as operações da *pooling layer* descritas na Seção 2.1.3.3. A Figura 3.10 mostra o diagrama de blocos do **MaxPooling** implementado em FPGA. Observa-se nessa figura que o **MaxPooling** é composto por uma máquina de estados *SM\_pool* e pelo bloco *max function*. A *SM\_pool* controla a leitura da memória

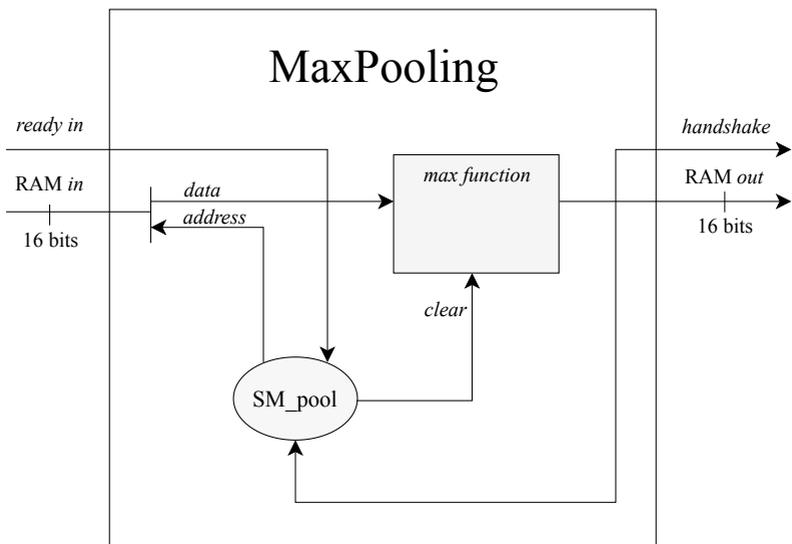
Figura 3.9: Diagrama de blocos do **ReLU** utilizado na CNN implementada em FPGA.



Fonte: do Autor.

RAM *in* e o fluxo de dados do *max function*. Tal bloco tem a função de armazenar o maior valor da entrada *data*.

Figura 3.10: Diagrama de blocos do **MaxPooling** utilizado na CNN implementada em FPGA.



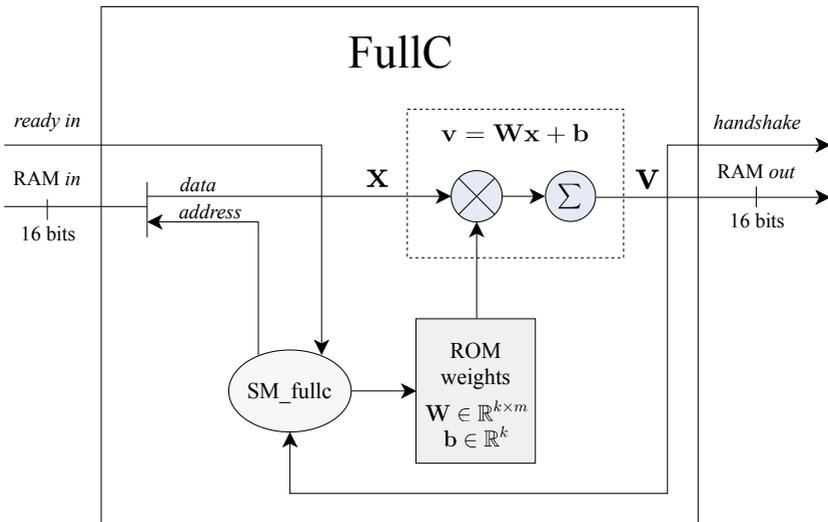
Fonte: do Autor.

Neste trabalho, o valor escolhido do passo da **MaxPooling** é 2 e do tamanho do filtro também é 2. Conforme apresentado na Seção 2.1.3.3, não há sobreposição entre os agrupamentos realizados pelo **MaxPooling** e a compressão nos elementos do tensor de entrada é de 4 : 1.

### 3.3.3.5 Bloco FullC

O bloco **FullC** implementa a *fully connected layer* apresentada na Seção 2.1.3.1 e é utilizado nas Camadas 8 e 9. A Figura 3.11 mostra o diagrama de blocos do **FullC** implementado em FPGA.

Figura 3.11: Diagrama de blocos do **FullC** utilizado na CNN implementada em FPGA.



Fonte: do Autor.

Observa-se na Figura 3.11 que o bloco **FullC** realiza a operação matricial descrita em (2.5), em que a matriz  $W \in \mathbb{R}^{k \times m}$  e o vetor  $b \in \mathbb{R}^k$  são armazenados na memória *ROM weights* com resolução de 16 bits. A saída do bloco corresponde ao vetor  $v \in \mathbb{R}^k$  e possui resolução de 16 bits.

A máquina de estados *SM\_fullc* controla a leitura dos dados das memórias *RAM in* e *ROM weights*, utilizados para realizar o cálculo

matricial. Finalizado o cálculo, um *flag* (*handshake*) sinaliza ao próximo bloco que um vetor  $\mathbf{v}$  está disponível.

### 3.3.3.6 Quantização Dinâmica

Durante o treinamento de uma ANN, os valores dos pesos e consequentemente das saídas das camadas são alterados (Seção 2.3). Deste modo, é uma tarefa difícil inferir a priori os alcances (no inglês, *ranges*) de tais valores. Com isso, a implementação de uma ANN que utiliza representação em ponto fixo é comprometida, impossibilitando os cálculos de erro de quantização e de probabilidade de *overflow*.

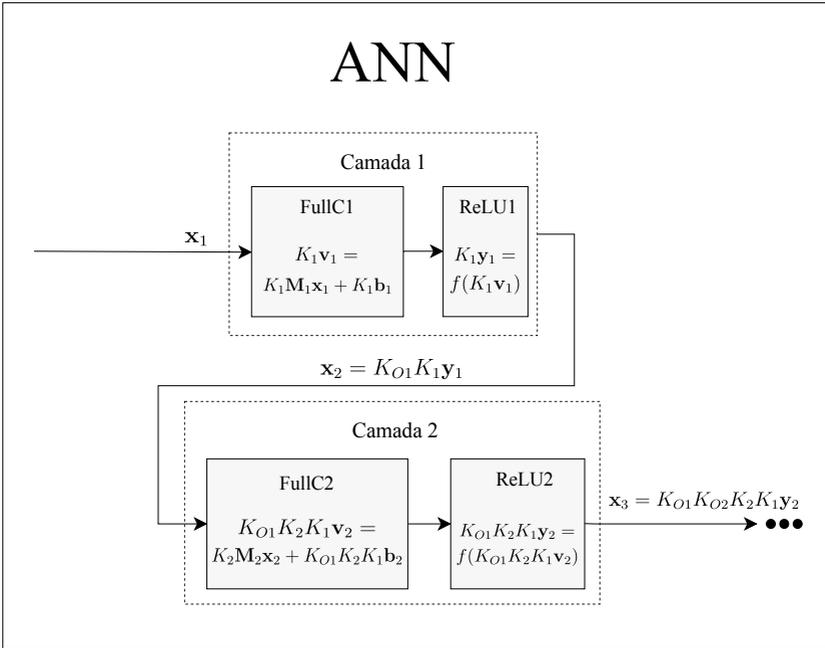
Um método de analisar tais *ranges* é através de uma inferência da ANN, utilizando um conjunto de dados conhecido (preferencialmente o conjunto de validação). Com tal inferência, pode-se avaliar a distribuição de valores e o *range* das saídas de cada camada. De posse dos *ranges*, é possível definir valores de quantização para cada camada de modo que sejam diminuídos o erro de quantização e a probabilidade de *overflow*. A CNN implementada neste trabalho utiliza tal método para a quantização das saídas e dos pesos de cada camada.

Os blocos que possuem pesos armazenados em memória ROM são: **Conv**, **Bnorm** e **FullC**. Portanto, apenas tais blocos serão analisados para avaliar a quantização dos pesos e de suas saídas. Os blocos restantes (**MaxPooling** e **ReLU**) utilizam as quantizações definidas pelos blocos anteriores.

A Figura 3.12 mostra o diagrama de blocos da propagação da quantização de uma ANN genérica, utilizando os blocos **FullC** e **ReLU**. Os termos  $K_{O_i}$  e  $K_i$  são, respectivamente, as constantes de multiplicação utilizadas para a quantização da saída e dos pesos da camada  $i$ . Observa-se nessa figura que as quantizações da camada 1 são propagadas para a camada 2.

Para escolher os valores das constantes  $K_{O_i}$  e  $K_i$  é utilizado o seguinte procedimento. Inicialmente, é analisado o *range* dos valores dos pesos de cada camada, pois tais valores são fixados após o treinamento da ANN, conforme apresentado na Seção 2.3. Na sequência, é realizada a quantização de tais valores, definindo as constantes  $K_i$ . Posteriormente, é efetuada a inferência da CNN (a partir do conjunto de validação), realizando a quantização das saídas das camadas, definindo as constantes  $K_{O_i}$ . Finalmente, é realizado

Figura 3.12: Diagrama de blocos da propagação das constantes de quantização de uma ANN genérica.



Fonte: do Autor.

um ajuste na quantização dos sinais de *bias*  $b_i$ . É considerado tal ajuste devido à propagação das constantes  $K_{O_i}$  e  $K_i$ , a qual pode ser observada no sinal  $b_2$  na Figura 3.12.

A Tabela 3.8 mostra os valores das constantes  $K_i$  e  $K_{O_i}$ , utilizadas na quantização das camadas da CNN, e o ajuste realizado no sinal de *bias*. Observa-se nessa tabela que os blocos **ReLU** e **MaxPooling** não utilizam valores de quantização, dado que estes não possuem pesos. É atribuído às constantes  $K_{O_i}$  valores na potência de 2, em que as multiplicações são realizadas através de deslocamento.

### 3.3.3.7 Gerador automático de código

O gerador automático de código desenvolvido para este trabalho permite produzir códigos fonte em Verilog de ANNs de forma automática, a partir dos modelos obtidos na etapa de treinamento com

Tabela 3.8: Valores de quantização utilizados na implementação da CNN em FPGA.

Layer	Block	$K_i$	$K_{O_i}$	Propagation <b>b</b>
1	Conv1	27128	$2^{-1}$	362
	Bnorm1	70398	$2^1$	149
	ReLU1			
2	Conv2	13368	$2^2$	643
	ReLU2			
3	Conv3	11079	$2^1$	1050
	ReLU3			
4	Conv4	10010	$2^1$	710
	ReLU4			
5	Conv5	9087	$2^1$	434
	ReLU5			
6	MaxPooling			
7	Conv6	8141	$2^3$	240
	ReLU6			
8	FullC1	11895	$2^1$	478
	ReLU7			
9	FullC2	16694	$2^0$	347

Fonte: do Autor.

o *TensorFlow*. Tal gerador é responsável por instanciar os blocos de memória RAM, **Conv**, **FullC**, **MaxPooling** e **ReLU**, fazer a conexão entre tais blocos, realizar a quantização dinâmica e, finalmente, gerar os arquivos de inicialização das memórias ROM.

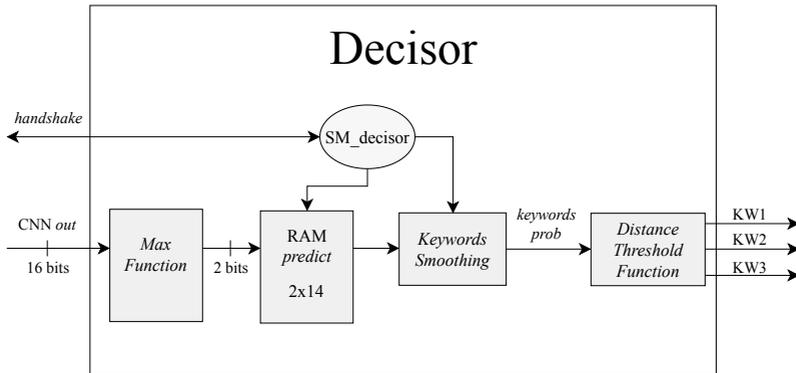
O código em Verilog gerado segue a topologia mostrada na Figura 3.4, bem como o padrão de interconexões entre as camadas e as memórias RAM. Em [56] é apresentado o código obtido em Verilog para o Cenário 5 mostrado na Tabela 3.7.

### 3.3.4 Decisor

O bloco Decisor é responsável por obter a sequência de *keywords* identificadas a partir das classificações realizadas pelo bloco CNN

(Seção 2.7.3). A Figura 3.13 mostra o diagrama de blocos do Decisor implementado em FPGA.

Figura 3.13: Diagrama de blocos do Decisor implementado em FPGA.



Fonte: do Autor.

Observa-se na Figura 3.13 que o Decisor é composto pelos blocos *Max Function*, *Keywords Smoothing* e *Distance Threshold Function*, pela máquina de estados *SM\_decisor* e pela memória *RAM predict*. O bloco *Max Function* recebe as previsões realizadas pela CNN e as classifica pelo maior valor, obtendo na saída os valores correspondente às classes (0="OOV", 1="Sim Máquina", 2="Abre", 3="Fecha"). Tais valores são salvos na memória *RAM predict* que armazena as últimas 14 classificações. Tal valor foi obtido a partir de resultados experimentais. Na sequência, o bloco *Keywords Smoothing* realiza a suavização das classificações para cada *keyword*, obtendo as probabilidades de ocorrência. Finalmente, no bloco *Distance Threshold Function*, são analisadas as probabilidades e, através dos *thresholds* definidos (Tabela 3.5), a *keyword* identificada é obtida.

# CAPÍTULO 4

---

## Resultados

---

Neste capítulo são apresentados os resultados obtidos na implementação em FPGA do sistema de KWS proposto. Inicialmente, é definido o FPGA alvo e apresentados os resultados de síntese. Na sequência, é apresentada a avaliação do KWS implementado em FPGA, comparando com a realizada no desenvolvimento em *Python*. Finalmente, são discutidos os resultados da avaliação.

### 4.1 Resultados de Síntese

A síntese do código em FPGA do sistema de KWS é realizada pela ferramenta Quartus Prime da Intel [60] e as simulações pelo ModelSim [61]. Os resultados da síntese são apresentados na Tabela 4.1. Conforme mostrado nessa tabela, o dispositivo FPGA alvo é o EP4CE115F29C7 da família Cyclone IV E. Considerando esse dispositivo, a implementação do sistema de KWS utilizou 711 kbits de memória (18% da capacidade total), 78 multiplicadores de 9 bits embarcados, 3423 registradores e 4% do total de elementos lógicos. A frequência máxima  $f_{max}$  obtida para sinal de *clock* é de 262 MHz.

Tabela 4.1: Resultado de síntese total FPGA.

Família	Cyclone IV E
Dispositivo	EP4CE115F29C7
Total de elementos lógicos	5,023 / 114,480 (4%)
Total de registradores	3423
Total de bits de memória	711,474 / 3,981,312 (18%)
Multiplicadores de 9 bits	78 / 532 (15%)
Frequência máxima ( <i>speed grade 3</i> )	262 MHz

Fonte: do Autor.

#### 4.1.1 Detalhamento dos Resultados de Síntese

A Tabela 4.2 detalha o número de bits de memória e o número de ciclos de *clock* por *frame* nos blocos *Framing* e Extrator de MFCCs. Conforme mostrado nessa tabela, é utilizado em tais blocos um total de 270 kbits de memória e 24904 ciclos de *clock*.

Observa-se ainda na Tabela 4.2 que o bloco *Mel-Bank Filters* é o que necessita de mais recursos de memória e tempo de processamento na etapa de extração de MFCCs. Tal restrição deve-se à necessidade de carregar os coeficientes do banco de filtros em memória ROM e também aos cálculos matriciais, descritos na Seção 2.6.1.3. Já o bloco *FFT Calculator* necessita de 2523 ciclos de *clock* e é utilizado pelos blocos *Energy Spectrum Calculator* e *DCT Calculator*. Deve-se observar na Tabela 4.2 que o número de ciclos do *FFT Calculator* não é considerado no número total de ciclos de *clock*, dado que tal bloco é utilizado pelo *Energy Spectrum Calculator* e *DCT Calculator*.

Tabela 4.2: Detalhamento da síntese dos blocos *Framing* e Extrator de MFCCs. Obs: \*O número de ciclos do *FFT Calculator* não é computado no número de ciclos total.

		Ciclos por <i>frame</i>	Quantidade de Memória (bits)
Framing	-	518	24604
Extrator de MFCCs	<i>Energy Spectrum Calculator</i>	2531	16488
	<i>Mel-Bank Filters Chain</i>	16385	136184
	<i>Power to dB Chain</i>	2666	1472
	<i>DCT Calculator</i>	2800	0
	<i>FFT Calculator</i>	2523*	74603
	<i>MFCC Matrix Shaping</i>	4	17044
	Total	24904	270395

Fonte: do Autor.

A Tabela 4.3 detalha a quantidade de memória e o número de ciclos de *clock* utilizados nas camadas da CNN. Observa-se, nessa tabela, que a Camada 2 (composta por um bloco **Conv** e uma **ReLU**) é a que necessita de um maior número de ciclos de *clock*, correspondendo a 106282 ciclos. Já a Camada 8 (bloco **FullC** e **ReLU**) utiliza a maior quantidade de memória, devido ao número de pesos  $((168 + 1) \times 100)$  utilizados no bloco **FullC**. A Camada 8 utiliza cerca de 272 kbits de memória, ou seja, aproximadamente 9 vezes mais memória que a Camada 4 (camada convolucional).

Considerando o processamento em paralelo nas camadas da CNN, a restrição de processamento é dada pelo tempo necessário para processar as duas camadas consecutivas de maior complexidade computacional (Seção 3.3.3). Portanto, de acordo com a Tabela 4.3, o número de ciclos é dado pela soma das camadas 2 e 3 ( $106282 + 79866 = 186148$ ). A quantidade total de memória utilizada na CNN é 441 kbits. Nota-se que somente a Camada 8 utiliza mais da metade do total de memória

Tabela 4.3: Resultado de síntese para o bloco CNN.

Camada CNN	Ciclos por frame	Quantidade de memória (bits)
1	36838	23616
2	106282	28512
3	79866	28416
4	93490	31200
5	48394	26304
6	1184	4608
7	12442	19160
8	16908	272735
9	423	6464
Total	186148	441015

Fonte: do Autor.

da CNN.

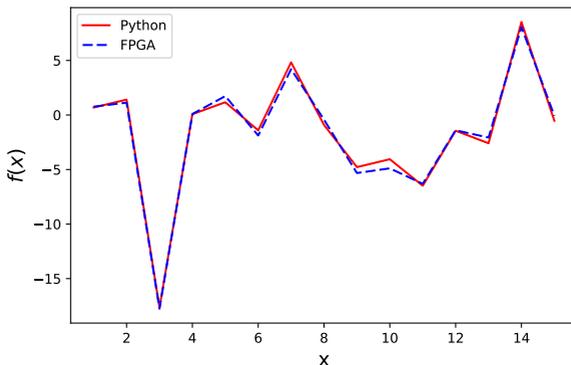
Finalmente, o bloco Decisor utiliza apenas 60 bits de memória e realiza seu processamento em apenas 57 ciclos de *clock*.

## 4.2 Comparação do KWS em FPGA e em *Python*

A Figura 4.1 mostra a comparação de um vetor de MFCCs, obtido em *Python* e pela implementação em FPGA. Observa-se nessa figura que o vetor de MFCCs obtido pelo FPGA possui uma pequena diferença em relação ao do *Python*. Tal diferença é justificada pelo erro de quantização presente na implementação em FPGA, que é devido principalmente ao bloco *Power to dB Chain*. Conforme apresentado na Seção 3.3.2.4, o bloco *Power to dB Chain* realiza o cálculo do logaritmo na base 2, que é a principal fonte de erro de quantização no KWS.

Em testes de simulação, foi analisado o *mean square error* (MSE) entre os vetores obtidos em FPGA e em *Python*. O MSE calculado nos vetores de entrada do bloco *Power to dB Chain* é de  $2.11 \cdot 10^{-3}$ . Já na saída desse bloco o MSE obtido é de  $6.20 \cdot 10^{-1}$ , mostrando realmente que tal bloco é a principal fonte de erros.

Figura 4.1: Comparação de um vetor de MFCCs extraído pelo desenvolvimento em *Python* e pela implementação em FPGA.



Fonte: do Autor.

Foram elaborados testes para verificar o erro de quantização do bloco CNN. Foi observado nesses testes que o erro de quantização do bloco pode ser desconsiderado, devido à técnica de quantização dinâmica proposta neste trabalho (Seção 3.3.3.6). O MSE calculado na saída do bloco CNN é de  $3.09 \cdot 10^{-2}$ .

### 4.3 Avaliação do KWS em FPGA

A avaliação do KWS em FPGA foi realizada considerando o kit de desenvolvimento DE10-Standard [62], que permite processar arquivos de áudio utilizando o sistema operacional Ubuntu 16.04. O KWS implementado no FPGA recebe os arquivos de áudio do *Speech Database* (Seção 3.2.1), produzindo os resultados em FPGA. Tais resultados são salvos em arquivos para permitir a avaliação das taxas de detecções corretas e de falsos positivos. Os *thresholds* utilizados no KWS são os escolhidos na Seção 3.2.5 para o Cenário 5 (Tabela 3.5).

A Tabela 4.4 apresenta os resultados de desempenho do KWS em *Python* e em FPGA. Conforme mostrado nessa tabela, as taxas de detecções corretas são equivalentes, porém os resultados em FPGA são um pouco inferiores. Contudo, na avaliação das taxas de falsos positivos, o resultado em FPGA possui um melhor desempenho para

sinais de voz e inferior para sinais de ruído.

Tais resultados mostram que a implementação em FPGA é equivalente ao desenvolvido em *Python*, permitindo a integração em um dispositivo FPGA de um sistema de KWS.

Tabela 4.4: Comparação das avaliações do KWS implementado em *Python* e em FPGA.

		Python	FPGA
Taxa Detecção Correta	KW1	98.31%	96.90%
	KW2	99.15%	97.18%
	KW3	97.75%	96.62%
Taxa Falso Positivo	Voz	5.31%	3.75%
	Ruído	2.12%	4.60%

Fonte: do Autor.

#### 4.4 Análise Tempo Real

Considerando que os blocos *Framing*, Extrator de MFCCs, CNN e Decisor realizam o processamento em paralelo, a restrição de tempo de processamento do sistema de KWS é dada apenas pelo bloco CNN, que é o que necessita de um maior número de ciclos de *clock*. Sabendo que o número de ciclos utilizados na CNN é 186148, o tempo de processamento do KWS é dado por  $\frac{186148}{f_{clk}}$ , onde  $f_{clk}$  é a frequência de *clock* do KWS.

O período mínimo necessário para o processamento em tempo real do KWS é dado por  $\frac{N_{hop}}{f_s} = \frac{256}{8000} = 32$  ms. Portanto, a frequência mínima para a operação em tempo real do KWS é de  $(f_{min} = 186148 \frac{f_s}{N_{hop}}) = 5.9$  MHz.

Utilizando o sistema de KWS implementado com um *clock* de frequência  $f_{max}$  (Tabela 4.1), pode-se realizar o processamento do KWS a uma taxa de 45 vezes o tempo real. Portanto, utilizando uma multiplexação do KWS implementado, é possível realizar o processamento de 45 canais em tempo real.

# CAPÍTULO 5

---

## Considerações Finais

---

Este trabalho foi dedicado à implementação em FPGA de um sistema de KWS com capacidade de detectar palavras específicas na língua portuguesa. Foram analisados os principais tipos de KWS e foi escolhido para a implementação o *Acoustic* KWS. Utilizou-se, como pré-processamento, a extração de MFCCs e, como Decoder, uma CNN. O sistema de KWS proposto foi desenvolvido em *Python* e posteriormente implementado em FPGA.

Inicialmente, no desenvolvimento em *Python*, foi analisada uma técnica de extração de MFCCs e foram discutidos os principais tipos de camadas de ANNs, ressaltando a utilização em CNNs. Na sequência, foi elaborado um banco de fala em português possuindo arquivos de fala e de ruído. O banco de fala foi utilizado para a elaboração dos *datasets* para o treinamento, teste e avaliação da CNN. Posteriormente, foram propostas a utilização das técnicas de *data augmentation* e a adição de ruídos visando melhorar a performance do KWS. Em seguida, foram elaborados cenários com diferentes parâmetros para avaliar as técnicas utilizadas. O cenário que obteve o melhor desempenho foi considerado como base na implementação em FPGA. Finalmente, foram implementados, em FPGA, um extrator de MFCCs e as camadas

da CNN. Foi proposto um gerador automático de código e uma técnica de quantização dinâmica para auxiliar na performance e na implementação da CNN em FPGA.

A implementação em FPGA ocupou cerca de 18% do dispositivo alvo (EP4CE115F29C7), obtendo uma frequência máxima de 262 MHz. Os resultados em FPGA do sistema de KWS proposto se mostraram bastante próximos aos do desenvolvimento em *Python*, obtendo taxas de detecções corretas de até 97.18% e taxas de falsos positivos de até 4.6%. O sistema de KWS proposto obteve um ótimo desempenho e pode ser utilizado com uma frequência de *clock* de no mínimo 5.9 MHz para a detecção em tempo real. Utilizando a frequência máxima obtida, é possível realizar a detecção em até 45 canais simultaneamente em tempo real.

---

## Bibliografia

---

- [1] D. Bahdanau, K. Cho e Y. Bengio, “Neural machine translation by jointly learning to align and translate”, *arXiv preprint arXiv:1409.0473*, 2014.
- [2] J. Hwang e Y. Zhou, “Image Colorization with Deep Convolutional Neural Networks”, Stanford University, Tech. Rep., 2016. Available: <http://cs231n.stanford.edu/reports2016/219Report.pdf>, rel. técn.
- [3] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele e H. Lee, “Generative adversarial text to image synthesis”, *arXiv preprint arXiv:1605.05396*, 2016.
- [4] R. M. e Rui Seara, “Um Sistema TTS Baseado em Redes Neurais Profundas Usando Parametros Síncronos de Pitch”, *SIMPOSIO BRASILEIRO DE TELECOMUNICAÇÕES E PROCESSAMENTO DE SINAIS*, 2017.
- [5] K. Hwang, M. Lee e W. Sung, “Online Keyword Spotting with a Character-Level Recurrent Neural Network”, *CoRR*, v. abs/1512.08903, 2015. arXiv: 1512.08903. endereço: <http://arxiv.org/abs/1512.08903>.

- [6] C. T. Lengerich e A. Y. Hannun, “An End-to-End Architecture for Keyword Spotting and Voice Activity Detection”, *CoRR*, v. abs/1611.09405, 2016. arXiv: 1611.09405. endereço: <http://arxiv.org/abs/1611.09405>.
- [7] G. Chen, C. Parada e G. Heigold, “Small-footprint keyword spotting using deep neural networks”, em *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, mai. de 2014, pp. 4087–4091. DOI: 10.1109/ICASSP.2014.6854370.
- [8] M. Shah, J. Wang, D. Blaauw, D. Sylvester, H. S. Kim e C. Chakrabarti, “A fixed-point neural network for keyword detection on resource constrained hardware”, em *2015 IEEE Workshop on Signal Processing Systems (SiPS)*, out. de 2015, pp. 1–6. DOI: 10.1109/SiPS.2015.7345026.
- [9] S. Ö. Arik, M. Kliegl, R. Child, J. Hestness, A. Gibiansky, C. Fougner, R. Prenger e A. Coates, “Convolutional Recurrent Neural Networks for Small-Footprint Keyword Spotting”, *CoRR*, v. abs/1703.05390, 2017. arXiv: 1703.05390. endereço: <http://arxiv.org/abs/1703.05390>.
- [10] R. C. Rose e D. B. Paul, “A hidden Markov model based keyword recognition system”, em *International Conference on Acoustics, Speech, and Signal Processing*, abr. de 1990, 129–132 vol.1. DOI: 10.1109/ICASSP.1990.115555.
- [11] J. R. Rohlicek, W. Russell, S. Roukos e H. Gish, “Continuous hidden Markov modeling for speaker-independent word spotting”, em *International Conference on Acoustics, Speech, and Signal Processing*, mai. de 1989, 627–630 vol.1. DOI: 10.1109/ICASSP.1989.266505.
- [12] J. G. Wilpon, L. G. Miller e P. Modi, “Improvements and applications for key word recognition using hidden Markov modeling techniques”, em *[Proceedings] ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing*, abr. de 1991, 309–312 vol.1. DOI: 10.1109/ICASSP.1991.150338.

- [13] E. F. Damasceno, J. B. D. J. José Remo Ferreira Brega, F. M. Ramos e L. F. B. Lopes, “ATENDENTE VIRTUAL: UMA ABORDAGEM DO USO DE REALIDADE VIRTUAL E RECURSOS DE FALA”, 2007.
- [14] *Google Store*, [https://store.google.com/?srp=/category/home\\_entertainment](https://store.google.com/?srp=/category/home_entertainment), Acessado: 2018-06-21.
- [15] W. S. McCulloch e W. Pitts, “A Logical Calculus of the Ideas Immanent in Nervous Activity”, *The bulletin of mathematical biophysics*, v. 5, n. 4, pp. 115–133, 1943.
- [16] F. ROSENBLATT, “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”, *Psychological Review*, pp. 65–386, 1958.
- [17] P. J. Werbos., “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences”, *Harvard University*, 1974.
- [18] K. J. Hunt, D Sbarbaro, R Żbikowski e P. J. Gawthrop, “Neural networks for control systems—a survey”, *Automatica*, v. 28, n. 6, pp. 1083–1112, 1992.
- [19] C. Szegedy, A. Toshev e D. Erhan, “Deep Neural Networks for Object Detection”, em *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, sér. NIPS’13, Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 2553–2561. endereço: <http://dl.acm.org/citation.cfm?id=2999792.2999897>.
- [20] P. de Andrade Kovalski, “Implementação de Redes Neurais Profunda para Reconhecimento de Ações em Vídeo”, *Universidade Federal do Rio de Janeiro*, 2018.
- [21] G. E. Hinton, S. Osindero e Y.-W. Teh, “A Fast Learning Algorithm for Deep Belief Nets”, *Neural Comput.*, pp. 1527–1554, jul. de 2006. endereço: <http://dx.doi.org/10.1162/neco.2006.18.7.1527>.
- [22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever e R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, *J. Mach. Learn. Res.*, v. 15, pp. 1929–1958, jan. de 2014. endereço: <http://dl.acm.org/citation.cfm?id=2627435.2670313>.

- [23] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2<sup>a</sup> ed., sér. 10. Prentice Hall, 2005.
- [24] I. Goodfellow, Y. Bengio e A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [25] A. Krizhevsky, I. Sutskever e G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, em *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou e K. Q. Weinberger, ed., Curran Associates, Inc., 2012, pp. 1097–1105. endereço: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [26] Y. Gong e C. Poellabauer, “How do deep convolutional neural networks learn from raw audio waveforms?”, *International Conference on Machine Learning*, 2018.
- [27] L. Wyse, “Audio spectrogram representations for processing with Convolutional Neural Networks”, *National University of Singapore*, 2017.
- [28] M. D. Zeiler e R. Fergus, “Visualizing and Understanding Convolutional Networks.”, 2013. endereço: <http://dblp.uni-trier.de/db/journals/corr/corr1311.html#ZeilerF13>.
- [29] S. Lawrence, C. L. Giles, A. C. Tsoi e A. D. Back, “Face recognition: a convolutional neural-network approach”, *IEEE Transactions on Neural Networks*, v. 8, n. 1, pp. 98–113, jan. de 1997, ISSN: 1045-9227. DOI: 10.1109/72.554195.
- [30] A. Torfi, N. M. Nasrabadi e J. M. Dawson, “Text-Independent Speaker Verification Using 3D Convolutional Neural Networks”, 2017. endereço: <http://arxiv.org/abs/1705.09422>.
- [31] M. Everingham, L. Gool, C. K. Williams, J. Winn e A. Zisserman, “The Pascal Visual Object Classes (VOC) Challenge”, *Int. J. Comput. Vision*, v. 88, n. 2, pp. 303–338, jun. de 2010. endereço: <http://dx.doi.org/10.1007/s11263-009-0275-4>.
- [32] J. Redmon e A. Farhadi, “YOLOv3: An Incremental Improvement”, v. abs/1804.02767, 2018. endereço: <http://arxiv.org/abs/1804.02767>.

- [33] T. Ko, V. Peddinti, D. Povey e S. Khudanpur, “Audio augmentation for speech recognition”, em *INTERSPEECH*, 2015.
- [34] J. Salamon e J. P. Bello, “Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification”, 2016. endereço: <http://arxiv.org/abs/1608.04363>.
- [35] S. Ioffe e C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, 2015. endereço: <http://arxiv.org/abs/1502.03167>.
- [36] J. R. Deller Jr., J. G. Proakis e J. H. Hansen, *Discrete Time Processing of Speech Signals*, 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1993, ISBN: 0023283017.
- [37] W. Brent, *Physical and Perceptual Aspects of Percussive Timbre*. Proquest, Umi Dissertation Publishing, 2011, ISBN: 9781243757500. endereço: <https://books.google.com.br/books?id=Us4vLgEACAAJ>.
- [38] L. Rabiner, L. Rabiner e B. Juang, *Fundamentals of Speech Recognition*, sér. Prentice-Hall Signal Processing Series: Advanced monographs. PTR Prentice Hall, 1993. endereço: <https://books.google.com.br/books?id=XEVqQgAACAAJ>.
- [39] B. Logan, “Mel Frequency Cepstral Coefficients for Music Modeling”, em *ISMIR*, 2000.
- [40] B. P. Lathi, *Linear Systems and Signals*, 2nd. New York, NY, USA: Oxford University Press, Inc., 2009.
- [41] V. Tiwari, “MFCC and its applications in speaker recognition”, *International Journal on Emerging Technologies*, 2009.
- [42] S. A. Khayam, *The Discrete Cosine Transform (DCT): Theory and Application*. Department of electrical & computing engineering, 2003.
- [43] S.-C. B. Lo, H. Li e M. T. Freedman, “Optimization of wavelet decomposition for image compression and feature preservation”, *IEEE Transactions on Medical Imaging*, v. 22, pp. 1141–1151, 2003.

- [44] E. T. M. G. Ami Moyal Vered Aharonson, *Phonetic Search Methods for Large Speech Databases*, 1st. Tel-Aviv, Israel: Springer, 2013.
- [45] I. Szöke, P. Schwarz, P. Matějka e M. Karafiát, “Comparison of keyword spotting approaches for informal continuous speech”, em *In Proceedings Eurospeech*, 2005.
- [46] A. Amir, A. Efrat e S. Srinivasan, “Advances in Phonetic Word Spotting”, em *Proceedings of the Tenth International Conference on Information and Knowledge Management*, sér. CIKM '01, Atlanta, Georgia, USA: ACM, 2001, pp. 580–582. endereço: <http://doi.acm.org/10.1145/502585.502697>.
- [47] Xilinx, *What is an FPGA*, <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>, Acessado: 2019-03-28.
- [48] D. B. Thomas, L. Howes e W. Luk, “A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation”, em *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, sér. FPGA '09, Monterey, California, USA: ACM, 2009, pp. 63–72. endereço: <http://doi.acm.org/10.1145/1508128.1508139>.
- [49] J. Chase, B. Nelson, J. Bodily, Z. Wei e L. Dah-Jye, “Real-Time Optical Flow Calculations on FPGA and GPU Architectures: A Comparison Study”, mai. de 2008, pp. 173 –182.
- [50] Microsoft, *Project Brainwave*, <https://www.microsoft.com/en-us/research/project/project-brainwave>, Acessado: 2019-03-28.
- [51] S. Brown e Z. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, 2<sup>a</sup> ed. New York, NY, USA: McGraw-Hill, Inc., 2008.
- [52] P. S. Foundation, *What is Python? Executive Summary*, <https://www.python.org/doc/essays/blurb>, Acessado: 2019-03-28.
- [53] techopedia, *What does TensorFlow mean?*, <https://www.techopedia.com/definition/32862/tensorflow>, Acessado: 2019-04-25.

- [54] D. S. Academy, *O que é o tensorflow machine intelligence platform?*, <http://datascienceacademy.com.br/blog/o-que-e-o-tensorflow-machine-intelligence-platform/>, Acessado: 2019-04-25.
- [55] D. Snyder, G. Chen e D. Povey, *MUSAN: A Music, Speech, and Noise Corpus*, arXiv:1510.08484v1, 2015. eprint: 1510.08484.
- [56] N. Votre, *Repositório do TCC*, <https://gitlab.com/natan.votre/repositorio-tcc>, Acessado: 2019-05-21.
- [57] R. Stephens, *Essential Algorithms: A Practical Approach to Computer Algorithms*, 1st. Wiley Publishing, 2013.
- [58] D. Zanco, N. Votre e W. Gontijo, “Contribuições ao Algoritmo A-SPADE Visando a Implementação de um Declipper em FPGA”, set. de 2017.
- [59] H. Malvar, “Fast computation of the discrete cosine transform and the discrete Hartley transform”, *Acoustics, Speech and Signal Processing, IEEE Transactions on*, v. ASSP-35, pp. 1484 –1485, nov. de 1987.
- [60] Intel, *Quartus Prime*, <https://www.intel.com/content/www/us/en/programmable/downloads/download-center.html>, Acessado: 2019-05-11.
- [61] I. FPGA, *ModelSim*, <https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/model-sim.html>, Acessado: 2019-05-11.
- [62] TerasIC, *DE10-Standard Development Kit*, <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1081>, Acessado: 2019-05-21.