Trabalho de Graduação

Identificação de Sistema (ECG x Respiração) para Detecção de Batimentos Ectópicos de Maneira Automática Usando Rede Neural em um SoC

Gabriel Reves Vasques Tonussi

Brasília, Dezembro de 2020

ENGENHARIA
MECATRÔNICA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

Trabalho de Graduação

Identificação de Sistema (ECG x Respiração) para Detecção de Batimentos Ectópicos de Maneira Automática Usando Rede Neural em um SoC

Gabriel Reves Vasques Tonussi

Relatório submetido como requisito parcial de obtenção

de grau de Engenheiro de Controle e Automação

Banca Examinadora

Prof. Carlos Humberto Llanos Quintero, ENM/UnB
Orientador (TG1)

Prof. Renato Coral Sampaio, FGA/UnB
Orientador (TG2)

Prof. Jones Yudi Mori Alves da Silva, ENM/UnB
Examinador interno

Brasília, Dezembro de 2020

FICHA CATALOGRÁFICA

TONUSSI G. R. V., GABRIEL
Identificação de Sistema (ECG x Respiração) para Detecção de Batimentos Ectópicos de Maneira Automática Usando Rede Neural em um SoC

[Distrito Federal] 2020.

iix, 43p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2020). Trabalho de Graduação – Universidade de Brasília. Faculdade de Tecnologia.

1. Identificação de Sistemas 2. Batimentos Ectópicos
3. Rede Neural RBF 4. Hardware-Software Co-Design

I. Mecatrônica/FT/UnB

REFERÊNCIA BIBLIOGRÁFICA

TONUSSI G. R. V., GABRIEL, (2020). Identificação de Sistema (ECG x Respiração) para Detecção de Batimentos Ectópicos de Maneira Automática Usando Rede Neural em um SoC. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-no8, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, (43)p.

CESSÃO DE DIREITOS

AUTOR: Gabriel Reves Vasques Tonussi

TÍTULO DO TRABALHO DE GRADUAÇÃO: Identificação de Sistema (ECG x Respiração) para Detecção de Batimentos Ectópicos de Maneira Automática Usando Rede Neural em um SoC.

GRAU: Engenheiro ANO: 2020

Gabriel Reves Vasques Tonussi

SQS 203 Bloco H Apt. 605 -Asa Sul

70233080 Brasília– DF – Brasil.

Dedicatória

Aos meus pais, Ivanoé e Sandra.
Ao meu irmão, Eduardo.
A minha namorada, Milene.

Gabriel Reves Vasques Tonussi

Agradecimentos

Gostaria de agradecer, em primeiro lugar, aos meus orientadores. Ao professor Humberto Llanos e ao professor Renato Sampaio, por acreditaram na minha capacidade e me proporcionarem oportunidades de pesquisa na área de sistemas embarcados. Ambos foram um incentivo à minha dedicação diuturna ao projeto.

Agradeço, de forma especial,às seguintes pessoas:

À Milene, minha namorada, pela revisão de português e pelo carinho nos momentos difíceis.

Ao Zeca por ter sanado minhas dúvidas sobre programação e comunicação de dados.

Ao Bizzi pela ajuda nos cálculos matemáticos.

Aos meus colegas do curso, Kurosvski, Aldegundes, Eric e Peg, por acreditarem no meu potencial de futuro engenheiro.

Aos meus amigos próximos, Gustavo, Lúcio, Lucas, Anny, Ivan e Henrique, pelos momentos de lazer e de apoio emocional.

Aos meus pais, Ivanoé e Sandra, que apesar de rígidos, são sempre amorosos.

Aos meus avôs, Marcos e Reinaldo, pelo apoio incondicional.

Finalmente, ao meu irmão Eduardo por sempre me trazer docinhos enquanto eu trabalhava, que apesar de nunca serem muitos, significavam o mundo.

Gabriel Reves Vasques Tonussi

RESUMO

Este trabalho de graduação tem como objetivo a criação de redes neurais que possam ser treinadas e parametrizadas em FPGAs. Como base, está sendo utilizada uma ferramenta [1] desenvolvida e em vários trabalhos do LEIA (*Laboratory of Embedded Systems and Integrated Circuits Applications*) para a geração de redes RBF em VHDL que utiliza treinamento offline e que resulta em uma arquitetura em VHDL com parâmetros fixos. Esse trabalho evolui essa arquitetura para criar uma solução completa em hardware e software de uma rede neural RBF auto-treinada com parâmetros variáveis e comunicação com um computador em pipeline. Neste trabalho foi desenvolvido um sistema embarcado responsável por identificar processos de característica caixa-preta, ou seja, processos que não podem ser representados por equações matemáticas determinísticas. A solução foi testada e validada inicialmente em um sistema benchmark Wiener-Hammerstein e, posteriormente aplicada a um problema de detecção de batimentos ectópicos através da identificação do sistema biomédico: Respiração (entrada) x Batimento cardíaco (saída).

Palavras Chave: Identificação de sistema, ECG, Batimentos ectópicos, Rede neural RBF, FPGA, Co-Design, Hardware.

ABSTRACT

One of the embedded system projects developed by LEIA (Laboratory of Embedded Systems and Integrated Circuits Applications) [1], was a MATLAB tool capable of creating a RBF neural network in VHDL with fixed parameters and fixed inputs. In this graduation project, this tool was used to create a complete architecture in hardware and software of a self-trained neural network with variable parameters and communication with a computer in pipeline. In this work, an embedded system was developed responsible for identifying black-box processes, that is, processes that cant be represented by deterministic math equations. The chosen application for the project was ectopic heartbeat detection thought identification of the bio-medic system: Respiration (input) and Heartbeat (output).

Keywords: System Identification, ECG, Ectopic Heartbeat, RBF Neural Network, FPGA, Co-Design, Hardware.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Símbolos Gregos

ω	Pesos de cada neurônio
ϕ	Saída de cada neurônio
c	Centroides
$\hat{y}$	Saída estimada pela rede neural
y	Saída real
x	Entrada da rede neural
n	Número de neurônios
d	Tamanho da base de dados

Siglas

RBF	*Radial Basis Function*
ECG	Electrocardiograma
RESP	Respiração
HRV	*Heart Rate Variability*
HW/SW	*Hardware-Software*
SoC	*System on a Chip*
OLS	*Ordinary Least Squares*
FPGA	*Field-Programmable Gate Array*
IEEE	*Institute of Electrical and Electronics Engineers*

Capítulo 1

Introdução

O desenvolvimento de sistemas embarcados é essencial para o progresso da tecnologia moderna. Um sistema embarcado é definido como um dispositivo que foi desenvolvido para executar uma tarefa específica. A criação de um sistema embarcado pode ser feita utilizando um não processador convencional, isto ocorre quando os processadores já disponíveis no mercado não atendem os requisitos do projeto de maneira satisfatória. Esses requisitos podem ser: Um menor consumo de energia, menor tempo de processamento, limitações de espaço para o computador ou necessidade de diminuição de custo do projeto.

Uma estratégia importante entre os desenvolvedores de sistema embarcados é o uso de *Field Programmable Gate Arrays* (FPGAs) para criar e testas novas arquiteturas. A FPGA permite a prototipagem dessas arquiteturas através do uso de linguagens de programação de fácil acesso: o VHDL e o *Verilog*. Através da programação em VHDL é possível o projetista criar e simular arquiteturas complexas na FPGA sem a necessidade de trabalhar em nível de portas lógicas.

Como a FPGA permite a criação de praticamente qualquer tipo de sistema embarcado, uma técnica eficiente em reduzir o tempo de execução de tarefas é a implementação de vários núcleos de processamento paralelos. Ou seja, a FPGA possui uma capacidade excepcional para reduzir o tempo de processamento de problemas que podem ser paralelizados, implementação de redes neurais, por exemplo.

Uma rede neural é composta da união de diferentes neurônios que possuem uma mesma dinâmica mas possuem parâmetros diferentes. A quantidade de neurônios, a maneira que eles se relacionam, a quantidade de entradas de cada neurônio e a dinâmica de cada neurônio (também chamado de função de ativação) são decisões arbitrárias que devem ser escolhidas com atenção pelo projetista. Por outro lado, os valores das variáveis de cada neurônio devem ser determinados por treinamentos complexos e de alto custo computacional. Neste trabalho, foi selecionada a função de ativação gaussiana para os neurônios, caracterizando uma rede neural RBF (explicada em detalhes na Seção 2.3).

Para verificar a arquitetura desenvolvida, foi escolhido o problema de identificação de batimentos cardíacos ectópicos. Essa identificação é feita através da modelagem e leitura do sistema biomédico: Respiração (entrada) x Batimento cardíaco (saída). A necessidade de identificação de batimentos ectópicos é importante para garantir que as medidas das máquinas de eletrocardiograma (ECG) estão corretas, visto que, esses batimentos anormais afetam de maneira brusca as leituras cardíacas do paciente [5].

É importante notar que o sistema RESP x ECG não possui modelo determinístico, sendo necessária uma modelagem caixa-preta através de identificação por rede neural. É importante ressaltar que uma modelagem eficiente do sistema RESP x ECG permite a detecção de batimentos ectópicos através do estudo da correlação entre o valor estimado pela rede neural e o valor lido pelo dispositivo de ECG.

1.1 Objetivos

Nesse projeto, o objetivo principal foi desenvolver uma arquitetura de uma rede neural parametrizável auto-treinada, que, se comparada com uma implementação em um processador ARM, oferece um ganho de velocidade de processamento. Para isso, foram realizados os seguintes objetivos específicos:

- Desenvolver uma arquitetura eficiente em pipeline usando a estratégia de co-design *hardware/software* (FPGA+CPU).
- Desenvolver uma estratégia para a comunicação entre o SoC e o computador.
- Verificar a qualidade dos dados processados pela arquitetura projetada através do uso de um *benchmark*.
- Detectar de maneira precisa os batimentos ectópicos em um paciente real usando a rede neural RBF projetada.

1.2 Metodologia

Foram selecionados dois sistemas diferentes para se identificar, um *benchmark* e o sistema ECG *vs* Respiração.

1.2.1 Sistema *Benchmark* Wiener-Hammerstein

Foi selecionado o sistema Wiener-Hammerstein por possuir um comportamento extremamente não linear e por ser um *benchmark* famoso na área de identificação de sistemas [6]. O sistema WH é explicado em detalhes na Seção 4.

1.2.2 Sistema ECGxRESP

Para a elaboração do banco de dados biomédicos, foram coletados sinais de ECG e respiração de 3 pacientes diferentes (f2y05, f2o03 e f2o08). Os dados foram coletados dos pacientes em um período de duas horas de análise contínua [7].

Para a obtenção dos resultados, foi determinada uma sequência de tarefas essenciais:

1. Elaborar o design em VHDL do hardware a ser implementado;
2. Testar o design VHDL utilizando a ferramenta de *testbench* do software *Vivado 2019.4*;
3. Transformar o design VHDL desenvolvido em um dispositivo encapsulado AXI;
4. Testar a comunicação entre a placa SoC e o computador;
5. Elaborar o software de treinamento e controle da rede neural.
6. Treinar e executar a rede neural, utilizando os dados de entrada e saída do sistema;
7. Verificar a correlação entre os dados estimados pela rede e os dados reais;
8. Fazer o estudo do tempo de execução da rede neural projetada *versus* a rede neural executada puramente em software.

Para o caso do sistema ECGxRESP, será detectada a ocorrência de batimentos ectópicos através da análise da correlação, tendo em mente verificar a existência de falso positivo e falso negativo.

1.3 Estado da Arte

Foram estudados trabalhos importantes sobre identificação de sistemas com redes RBF, hardwares específicos para rede neural e detecção de batimentos ectópicos:

- Em [8], foi utilizada uma rede neural RBF para fazer a aproximação de um sistema genérico não linear;
- Em [1], foi feito a identificação de um sistema não linear utilizando uma rede neural RBF com parâmetros fixos em hardware. Nesse trabalho era necessário o treinamento prévio da rede neural em software antes da implementação em FPGA;
- Em [9], foi feito o controle de sistemas dinâmicos utilizando uma estratégia de identificação de sistema por rede neural RBF;
- Finalmente, em [5], foi feito um estudo da literatura para determinar quais métodos são os mais efetivos para detecção de batimentos ectópicos. Nesse trabalho, foi mostrado que a rede neural RBF é uma das melhores ferramentas para esse problema.

A contribuição deste trabalho é o desenvolvimento de um hardware embarcável novo, capaz de treinar e executar uma rede neural RBF parametrizável. Capaz também de identificar com prescrição uma gama de sistemas não lineares. Entre eles, o sistema Respiração x Batimento cardíaco, possibilitando assim a detecção de batimentos ectópicos.

Capítulo 2

Fundamentação Teórica

Nesse capítulo serão mostradas todas as ferramentas utilizadas para a execução do projeto. Foi estudado o funcionamento de redes neurais, sistemas embarcados em SoC e batimentos ectópicos.

2.1 Identificação de Sistemas

Identificar um sistema consiste em descobrir como um determinado processo pode ser modelado, ou seja, determinar as técnicas e equações matemáticas que são capazes de representar o processo analisado. Existem somente 3 tipos diferentes de modelos para sistemas [10]:

- **Modelo caixa branca:** A dinâmica do sistema é conhecida, mas os valores das variáveis precisam de ser determinados. Um exemplo desse tipo de modelo pode ser observado na Figura 2.1.

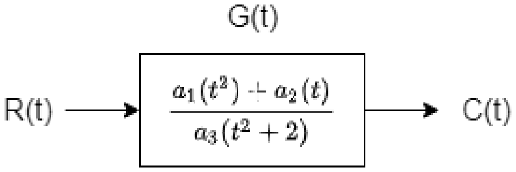


Figura 2.1: Modelo caixa branca: A dinâmica do sistema é bem conhecida.

- **Modelo caixa cinza:** Algumas informações a respeito da dinâmica do sistema são conhecidas, porém, não se sabe o modelo matemático do processo com exatidão. Apresentando a necessidade da inclusão de variáveis livres que devem ser estimadas. Um exemplo desse tipo de modelo pode ser observado na Figura 2.2.

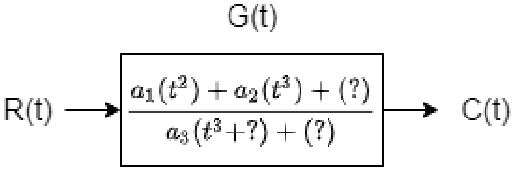


Figura 2.2: Modelo caixa cinza: A dinâmica do sistema é parcialmente conhecida.

- **Modelo caixa preta:** Nenhuma informação a respeito do modelo é conhecida, sendo necessária uma identificação usando somente os valores de entradas e saídas do sistema. Entre as técnicas disponíveis para identificar esse tipo de modelo, têm-se: análise por mínimos quadrados, análise em frequência por diagrama de bode e identificação de sistema por uso de rede neural. Essa última, é uma das únicas técnicas eficiente para modelar sistemas com difícil representação matemática (Figura 2.3).

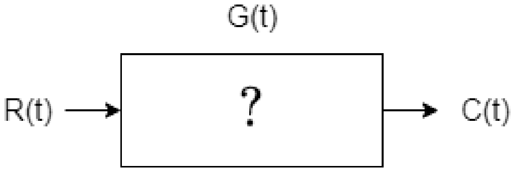


Figura 2.3: Modelo caixa preta: Nenhuma informação da dinâmica do sistema é conhecida.

2.2 Identificação de Sistemas Usando Redes Neurais

A identificação de sistemas usando rede neural é feita através da análise das entradas e saídas do processo, para depois, modelar os parâmetros necessários da execução da rede (pesos dos neurônios por exemplo). Um esquema simples de identificação é apresentado na Figura 2.4.

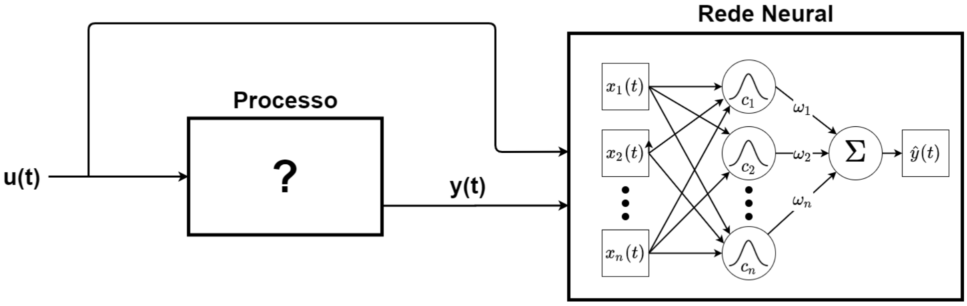


Figura 2.4: Projeto simplificado de identificação de sistema utilizando rede neural.

É importante lembrar que nem todo modelo pode ser identificado por uma função matemática determinística, seja pela falta de conhecimento sobre o sistema (modelo caixa preta) ou pela complexidade da dinâmica do processo.

Sabe-se que, por possuir as características de um aproximador universal ([11] e [8]), algumas redes neurais são capazes de modelar e identificar sistemas complexos. Podemos observar por exemplo, o uso de redes neurais RBF como ferramenta de detecção eficiente de batimentos em [12], ou seu uso para a identificação de sistemas mecânicos complexos. A identificação de um sistema mecânico complexo pode ser observada no trabalho [13], que interpreta o processo de um braço robótico com 3 graus de liberdade. Além do uso comprovado da RBF como uma ferramenta eficiente de identificação, outras vantagens desta rede são: baixo custo de implementação, alta velocidade de treinamento e boa tolerância a erros de entrada (*noise error*) [9].

É importante informar que para se avaliar as saídas da rede neural, deve ser feita a comparação com a saída real do sistema, atribuindo-se um coeficiente de correlação entre elas, representado pela através da Fórmula 2.7. Também deve ser notado que enquanto maior a complexidade do sistema, mais neurônios serão necessários para identificar com confiabilidade o processo.

2.3 Rede Neural RBF

Redes neurais são uma ferramenta cada vez mais usada no ramo da engenharia, visto que são capazes de resolver problemas não lineares, com modelagens do tipo caixa preta ou caixa cinza. As redes neurais são formadas por neurônios que são responsáveis por receber todas as entradas disponíveis e produzir uma saída de acordo com sua função de ativação.

A função de ativação de um neurônio, a quantidade de neurônios e a quantidade de camadas de neurônios devem ser escolhidas de acordo com sua aplicação. A função de ativação escolhida no trabalho foi a *Radial Basis Function*. Sabendo que para cada neurônio n existem m entradas; podemos representar a função de ativação pela Equação 2.2.

Nessa função são necessários 2 vetores de parâmetros diferentes:

- $x_{1:m}$: Todas as entradas x, de 1 até m;
- $c_{n,1:m}$: Os centroides para cada entrada de 1 até m do neurônio n;

$$\sigma_n = \sqrt{\frac{1}{2\delta_n}} \tag{2.1}$$

$$\phi_n = \exp \frac{(c_{n,1:m} - x_{1:m})^2}{2\sigma_n^2} \tag{2.2}$$

Na rede neural RBF também se determinam pesos para cada neurônio, pois, neurônios diferentes têm contribuições diferentes para o resultado final. Essa contribuição é determinada pelas variáveis ω que são os pesos da saída de cada neurônio. Uma vez obtida a saída multiplicada pelo peso de cada neurônio, a soma dessas saídas é o resultado final $\hat{y}$ observado na Equação 2.3. A rede neural RBF pode ser observada na Figura 2.5.

$$\hat{y} = \sum_{n=1}^{m} \omega_n \phi_n \tag{2.3}$$

Observando as equações 2.1, 2.2 e 2.3, percebe-se que dentro de cada neurônio, existem 2 variáveis a serem determinadas: um vetor de centroides (c) e um delta (δ). Já para saída de cada neurônio, deve-se determinar um peso (ω).

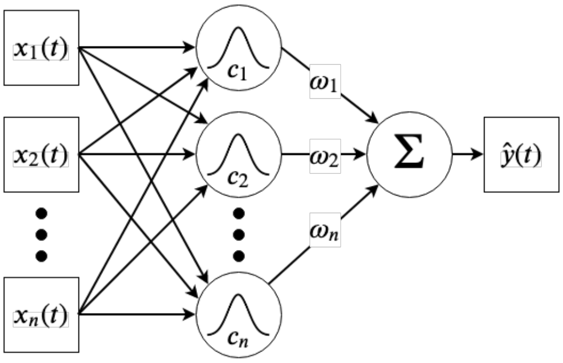


Figura 2.5: Rede neural RBF

2.4 Treinamento da Rede Neural RBF

Para o funcionamento correto da rede neural RBF, deve-se determinar os valores de c, ω e δ. Como utilizado no trabalho [13], foi atribuído à δ o valor fixo de 0.001. Já para valores de c e ω foram usados os treinamentos *k-means* e OLS respectivamente.

Os centroides são responsáveis por determinar os agrupamentos dos dados de entrada dos neurônios (*clusters*). Um exemplo de como dados de entrada da rede formam *clusters* pode ser observado na Figura 2.6 e na Figura 2.7. Esses centroides foram calculados através do algoritmo *k-means*, que será explicado na seção 2.4.1.

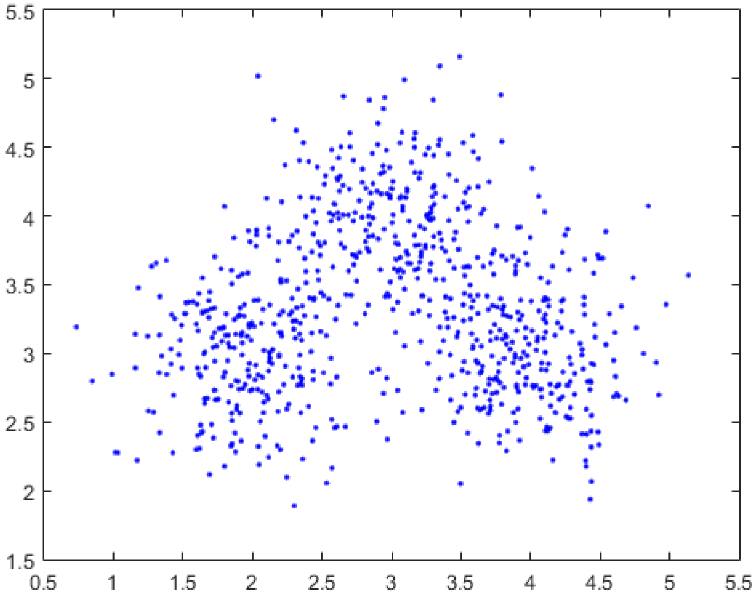


Figura 2.6: Dados de entrada da rede neural.

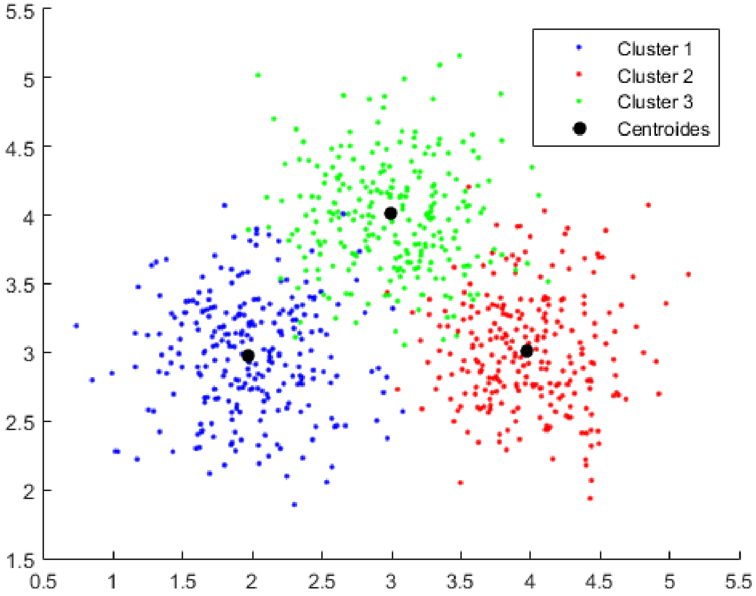


Figura 2.7: Separação dos dados de entrada da rede em grupos (clusters).

Para o cálculo dos pesos dos neurônios é necessário uma base de dados destinada ao treinamento da rede. Uma vez obtida essa base de dados e executado o cálculo das saídas dos neurônios, ficam conhecidos os valores de $\hat{y}$ e ϕ da Equação 2.3. Agora o problema se resume a calcular o sistema matricial super determinado 2.4.

$$\omega = \phi \backslash \hat{y}, \tag{2.4}$$

sendo ω uma matriz $n \times 1$, ϕ uma matriz $n \times d$ e $\hat{y}$ uma matriz $d \times 1$, com n representando a quantidade de neurônios e d representando o tamanho da base de dados de treinamento.

A Equação 2.4, por se tratar de um sistema super determinado, não é trivial de ser resolvida. Para tanto, será apresentado o algoritmo de cálculo matricial *OLS* para solução deste problema (2.4.2).

2.4.1 Treinamento k-means

O treinamento *k-means* tem como objetivo calcular a localização dos centroides e determinar qual o centroide de cada dado da entrada [14]. A execução do algoritmo é a seguinte:

1. O algoritmo é inicializado com centroides aleatórios;
2. Para cada valor de entrada da rede é assinalado um centroide. Primeiro é calculada a distância euclidiana dos centroides atuais para cada dado x, e depois escolhido o centroide que gera o menor resultado de acordo com a Equação 2.5.

$$min||x - c_j||^2 \tag{2.5}$$

3. Agora que todo dado possui seu respectivo *cluster*, os novos centroides são calculados. **Para cada *cluster*** G, será usada a Equação 2.6;

$$c_j\,novo = \{\frac{1}{(x_{1:i} - c_j)} \sum_{l=1}^{i} x_l \mid x, c_j \in G\} \tag{2.6}$$

   - c_j: Centróide do *cluster* G.
   - i: Número de dados pertencentes ao *cluster* G.
   - $\{x_{1:i}|\ x \in G\}$: Todas as entradas x, de 1 até i que fazem parte do *cluster* G.

4. Deve-se repetir os itens 2 e 3 até que o limite de iterações seja atingido ou o erro proveniente distâncias euclidianas (Equação 2.5) pare de diminuir.

2.4.2 Cálculo por Ordinary Least Squares (OLS)

O objetivo do algoritmo OLS é resolver sistemas super-determinados do tipo $A * x = B$. Segundo o livro [15], o algoritmo pode ser separado em 5 etapas diferentes:

1. Decomposição QR da matriz A em uma matriz ortogonal Q e uma matriz triangular superior R.
2. Inverter a matriz resultante da multiplicação $Q * Q'$.

3. Multiplicar a matriz resultante do item 2 com a matriz transposta Q'.

4. Multiplicar a matriz resultante do item 3 com a matriz B.

5. Resolver o novo sistema linear $A * x = B$ usando a matriz R (nova matriz A) e o resultado do item 4 (nova matriz B).

Algoritmo implementado em formato de função no *MATLAB*:

Algoritmo 2.1: Algoritmo OLS implementado em MATLAB.

```
function [SOL]=ols(A,B)

 [Q,R] = qr(A);
H=inv(Q*Q');
G=H*Q';
New_A=R;
New_B = G*B;
SOL = linsolve(New_A,New_B);

end
```

Para se utilizar o algoritmo OLS com o objetivo de calcular os pesos dos neurônios, deve-se obter a matriz A e a matriz B, sendo elas determinadas pelo resultado da Equação 2.2 (ϕ) e pelas saídas do sistema (y) respectivamente.

2.5 Validação e Correlação de Dados

Para validar a rede neural, deve-se calcular a correlação entre a saída obtida $\hat{y}$ e a saída real y [16]. Existem 2 correlações importantes que devem ser analisadas: A correlação a respeito do conjunto de dados usados para o treinamento ($corr\_e$) e a correlação a respeito de um novo conjunto de dados de um mesmo sistema que não foram usados em nenhuma etapa do treinamento da rede ($corr\_v$).

Para calcular a correlação entre os vetores $\hat{y}$ e y foi utilizada a Equação 2.7, que apresenta como resultado um valor entre 0 e 1, sendo 1 uma correlação perfeita. Ou seja, em um cenário ideal, espera-se 1 para as duas correlações citadas.

$$corr = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}} \tag{2.7}$$

- n: Tamanho do vetor x.

- $\sum x$: Soma de todos os elementos do vetor x.

- $\sum y$: Soma de todos os elementos do vetor y.

- $\sum xy$: Soma da multiplicação de cada elemento dos vetores x e y.
- $\sum x^2$: Soma da multiplicação de cada elemento do vetor x por ele mesmo.
- $\sum y^2$: Soma da multiplicação de cada elemento do vetor y por ele mesmo.

A correlação representa a "qualidade"do treinamento feito em relação ao sistema real, ou seja, ela é usada para verificar se novos treinamentos são necessários ou se o sistema real possuiu alguma mudança no processo.

2.6 Batimentos Ectópicos

Um batimento ectópico é qualquer batimento cardíaco que não seja esperado na leitura do eletrocardiograma (ECG) (Figura 2.8). Batimentos ectópicos são um tópico extensivamente estudado na área de sinais biomédicos devido seu efeito no cálculo da variabilidade cardíaca (HRV). O HRV é um indicador de como o tempo entre batimentos cardíacos do paciente varia, sendo ele determinado através do cálculo de desvio padrão. A identificação de batimentos ectópicos também é importante para determinar algum problema cardíaco, como níveis baixos de potássio no sangue, pouco fluxo de sangue no coração e crescimento não natural dos músculos cardíacos [17].

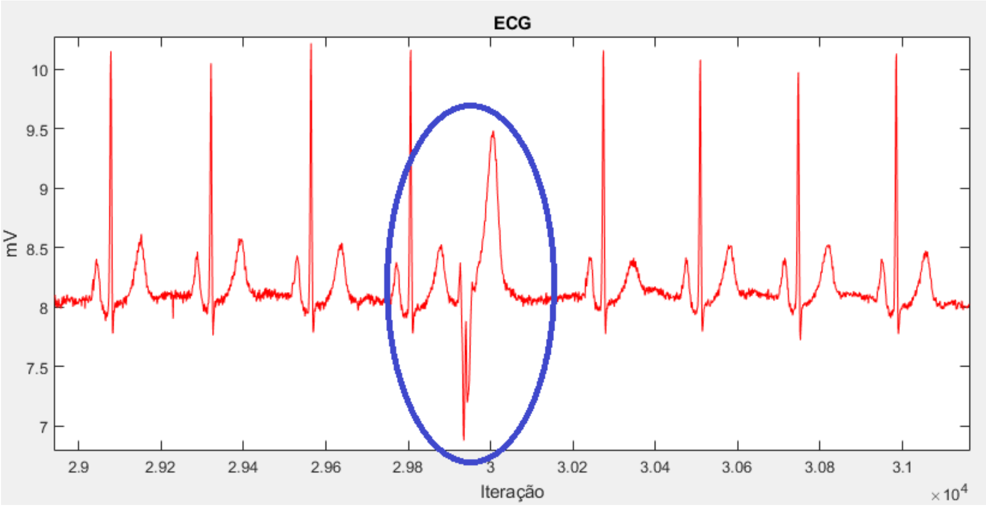


Figura 2.8: Batimento ectópico em paciente idoso.

Para a análise deste tipo de fenômeno, foi proposta a modelagem do sistema por caixa-preta, justificada por ser um processo biomédico com alto grau de complexidade [5] e [12]. Ou seja, como a decteção do batimento ectópico depende de uma identificação de um modelo caixa-preta (ECG x RESP), ela é um problema ideal para testar a capacidade de identificação da rede RBF projetada.

2.6.1 Base de dados Fantasia - PhysioBank ATM

Os dados usados para os sinais de respiração e de eletrocardiograma foram coletados do trabalho L, G.; LA, A.; L, G. Physiobank, physiotoolkit, and physionet [7]. O trabalho consiste em um site que agrupa diferentes bases de dados de pesquisas biomédicas e as disponibilizam gratuitamente. A base de dados utilizada no projeto foi inicialmente coletada por YENGAR, N. [18], que observou os sinais vitais de pacientes idosos para estudar como a saúde do coração se degrada com o tempo, causando um aumento da quantidade de batimentos ectópicos. Como os pacientes idosos estão mais sujeitos a batimentos ectópicos, a base de dados escolhida se mostrou ideal para o objetivo deste trabalho.

2.7 Sistema Embarcado do tipo SoC (System-on-a-chip)

Para a implementação da arquitetura do projeto foi usado um *System-on-a-Chip* (SoC), sendo o dispositivo escolhido o *chip* da *Xilinx XC7Z020-1CLG400C* que faz parte da placa ZYBO Z-20 (Figura 2.9) produzida pela *Digilent.* A placa possui uma FPGA, um processador ARM dual-core e cerca de 1 Gb de RAM. Mais especificações da placa podem ser observadas na Tabela 2.1.

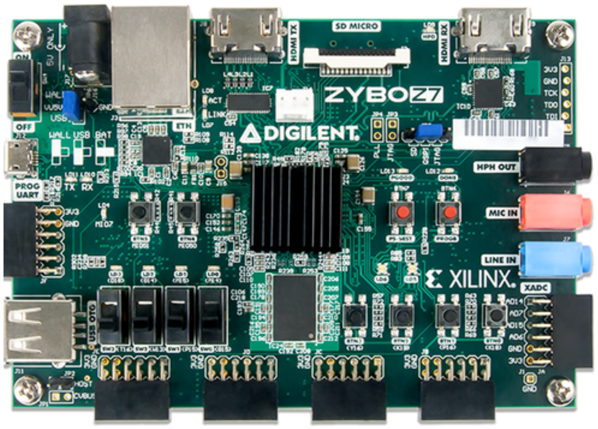


Figura 2.9: Placa zybo utilizada no projeto (foto retirada do site https://store.digilentinc.com/zybo-z7-zynq-7000-arm-fpga-soc-development-board/ [2]).

Tabela 2.1: Tabela de recursos disponíveis da placa ZYBO Z-20.

Chip-FPGA	XC7Z020-1CLG400C
Look-up Tables (LUTs)	53,200
Flip-flops	106,400
Block RAM	630 KB
I/O Disponíveis	40
HDMI CEC Support	TX and RX ports

Para a programação da placa foram usados 2 softwares diferentes produzidos pela Xilinx:

Vivado-WEBPACK para os códigos em VHDL e Xilinx-SDK para os códigos em C.

2.7.1 FPGA

A FPGA é um dispositivo de prototipagem de hardware, que foi inventado para desenvolver e testar arquiteturas antes de partir para o processo de produção. A FPGA é uma matriz de blocos lógicos que são ativados por uma memória flash programada pelo usuário (Figura 2.10), cada bloco lógico pode possuir *Flip-flops*, *Look-up Tables* (LUT) memória e *Digital Signal Processors* (DSP). Como a FPGA é um dispositivo complexo e difícil de se debugar, o designer também depende de *softwares* de simulação para testar a arquitetura. Esses softwares usam linguagens de descrição de hardware, *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) ou *Verilog*.

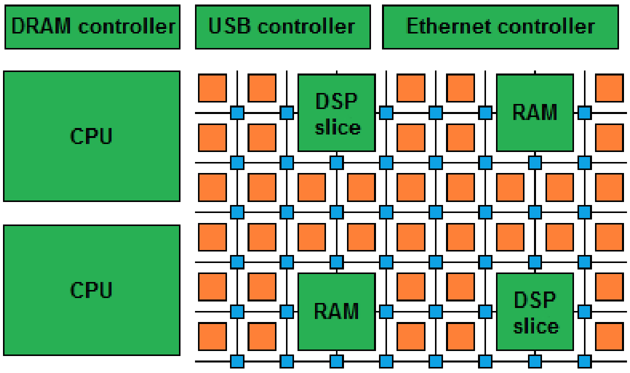


Figura 2.10: Arquitetura simplificada de uma FPGA [3].

2.7.2 Processador ZYNQ

O processador ZYNQ é composto por 2 núcleos *ARM Cortex 9* que são encapsulados por um módulo capaz de enviar e receber dados direto da FPGA no protocolo AXI (Figura 2.11). Isso permite uma comunicação de dados entre a FPGA e o processador de maneira robusta, rápida e relativamente simples.

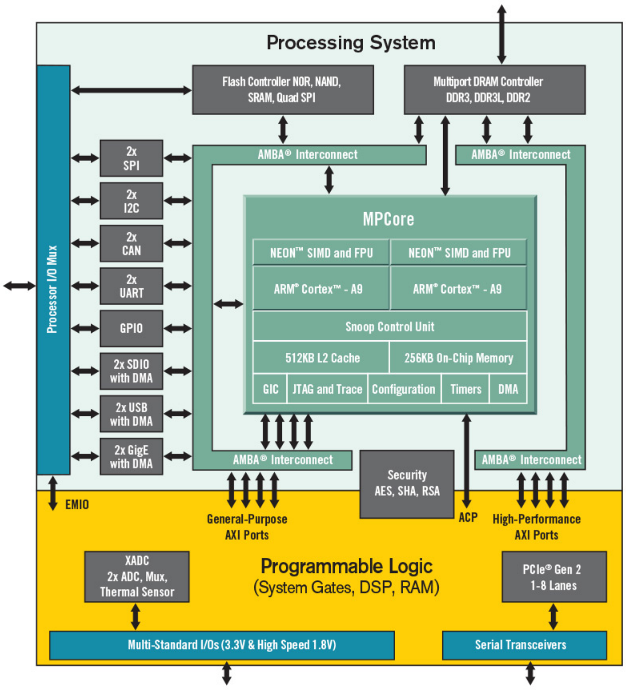


Figura 2.11: Processador ZYNQ-7000 (foto retirada do site https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html [4]).

Existem 2 estratégias básicas usadas para se trabalhar com o processador ZYNQ: Programar códigos que serão executados dentro de um sistema operacional (Linux por exemplo) ou programar em *bare metal*, ou seja, não utilizar nenhum sistema operacional. As vantagens de cada estratégia são:

- Programação em sistema operacional (SO): Maior facilidade de *debug* de códigos, disponibilidade de controle automático de alocação de memória, disponibilidade de *drivers* pré-implementados.
- Programação em sistema *bare metal*: Maior velocidade de execução de algoritmos, disponibilidade do processador em tempo integral (não é necessário executar mais nada além do código), maior quantidade de memória disponível (não é necessário alocar o SO).

Como o objetivo deste trabalho é melhorar a velocidade de um algoritmo, foi escolhida a estratégia em programação *bare metal*. Como o processador foi programado usando a linguagem de programação C sem um sistema operacional, foi necessário o desenvolvimento dos *drivers* de comunicação e a seleção manual dos tamanhos das pilhas de *stack* e *heap* da memória. Para otimizar o uso da memória do ZYNQ, todas as variáveis com mais de uma dimensão foram alocadas dinamicamente, ou seja, alocadas na pilha de *heap*.

2.7.3 Estratégia de co-design HW/SW

Para desenvolver a arquitetura do projeto, foi utilizada uma técnica chamada de co-design *hardware/software* (HW/SW). Essa técnica consiste em desenvolver o hardware e o software de uma projeto em conjunto, pensando em cumprir uma mesma tarefa específica.

O objetivo de se utilizar a estratégia co-design é separar a execução dos algoritmos do projeto de maneira a se aproveitar o melhor de cada dispositivo. Lembrando que a FPGA é muito eficiente na execução de cálculos paralelos, enquanto o processador convencional é muito eficiente no processamento de cálculos sequenciais. Sendo as limitações da FPGA: baixo ciclo de clock e dificuldade de implementação de algoritmos complexos.

Ou seja, para se desenvolver um sistema usando essa estratégia, é necessário separar as tarefas em 2 partes: as que devem ser executadas em paralelo na FPGA e as que devem ser executadas em série no processador convencional. A explicação de como foi feita essa divisão no projeto pode ser observada na Seção 3.2.

2.8 Software Dr. Memory

O software *Dr. Memory* é uma ferramenta para *Windows* ou *Linux* que permite a observação da maneira que a memória é alocada em algoritmos escritos na linguagem de programação C. O programa permite que, através do uso do arquivo executável gerado pelo compilador de C, seja possível de se observar como está sendo acessada a memória do computador rodando o algoritmo [19].

Como o processador ZYNQ utilizado não possui sistema operacional, essa ferramenta se mostrou essencial para garantir que os códigos a serem executados no SoC não possuíam nenhum problema de alocação de memória. Os principais problemas que foram descobertos e posteriormente corrigidos através do uso da ferramenta foram:

- Referenciar ponteiros vazios (*NULL reference*).
- Não alocar a quantidade certa de bytes a serem utilizados pelo *heap*.
- Não desalocar seções de memória que não estavam mais em uso.
- Liberar seções de memória que não estavam devidamente alocadas.
- *Overflow* das pilhas de *stack* e *heap*.

2.9 Comunicação de Dados

Para permitir a comunicação entre os diferentes componentes do projeto, foram utilizados 2 procolos diferentes de comunicação e 3 formatos diferentes de ponto flutuante. Um esquemático simplificado da comunicação entre os componentes principais pode ser observado na Figura 2.12.

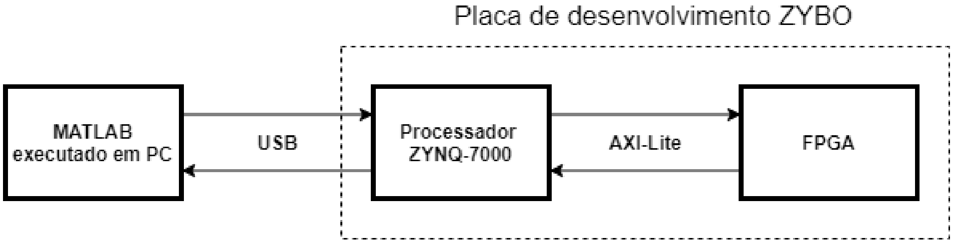


Figura 2.12: Procolos diferentes utilizados na comunicação entre componentes.

2.9.1 Formato dos Dados em Cada Etapa

Para garantir uma transferência eficiente de dados entre as diferentes partes da arquitetura, foram usadas algumas representações diferentes de ponto flutuante:

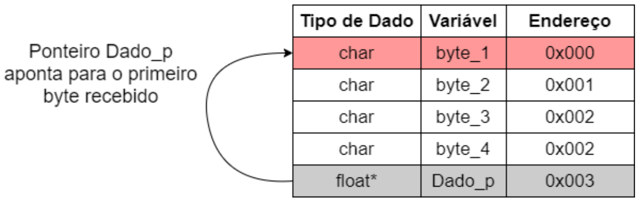
1. 8 bits de expoente 23 bits de mantissa (IEE-754 *single*): Usado no protocolo AXI-Lite para a comunicação entre o processador ZYNQ e FPGA. Também é utilizado na comunicação entre o processador ZYNQ e o MATLAB (separado em pacotes de 4 bytes cada).
2. 11 bits de expoente 52 bits de mantissa (IEE-754 *double*): Usado para os cálculos internos do processador ZYNQ.
3. 8 bits de expoente 55 bits de mantissa: Usado nos módulos internos da FPGA: Somadores, multiplicadores, etc. Esse formato não convencional de representação de ponto flutuante foi escolhido por 2 motivos: Garantir uma maior precisão dos dados calculados (3 bits a mais de mantissa) e facilitar a conversão de números do formato do item 3 para o formato do item 1. Deve-se perceber, que para fazer essa conversão, basta descartar os últimos 32 bits de cada número, estratégia que seria impossível de se adotar caso estivesse usando o formato padrão *double.*

2.9.2 Comunicação entre MATLAB e SoC (Serial USB)

A comunicação entre o MATLAB e o SoC foi desenvolvida com o objetivo de facilitar o *debug* e análise dos dados a serem processados. O MATLAB é responsável por armazenar as bases de dados de entrada e saída do sistema, possibilitando a criação de gráficos e tabelas com facilidade.

A comunicação entre o computador rodando o software *MATLAB* e o SoC foi feita através de um cabo USB usando o protocolo *serial.* No *MATLAB* os dados de entrada são armazenados em variáveis tipo *float* (*Floating Point IEEE-754*) e enviados em blocos de 4 *bytes* para o SoC. Por causa de uma limitação dos *drivers* do processador ZYNQ, ele só é capaz de receber pacotes de 1 byte de cada vez, necessitando chamar a função de receber dados 4 vezes para cada envio do *MATLAB.*

Para reconstruir os dados recebidos pelo SoC em formato *float*, foi usada uma manipulação de ponteiros. Primeiro, são recebidos os 4 *bytes* separadamente, armazenando-os em variáveis tipo *char*, depois, é apontado um ponteiro do tipo *float** para o primeiro dos *bytes* que foram recebidos (Figura 2.13).



Ponteiro Dado_p aponta para o primeiro byte recebido

Tipo de Dado	Variável	Endereço
char	byte_1	0x000
char	byte_2	0x001
char	byte_3	0x002
char	byte_4	0x002
float*	Dado_p	0x003

Figura 2.13: Reconstrução de dado tipo *float* a partir de dados tipo *char*.

2.9.2.1 Protocolo Serial USB

O protocolo USB é atualmente o protocolo serial mais utilizado no mundo com cerca de mais de 7 bilhões de dispositivos [20]. Esse protocolo utiliza de somente 2 canais diferentes para executar a comunicação. Como USB é um protocolo serial, ele envia as informações de maneira sequencial, gerando um atraso maior de comunicação se comparado com a transferência de dados paralela (Figuras 2.14 e 2.15).

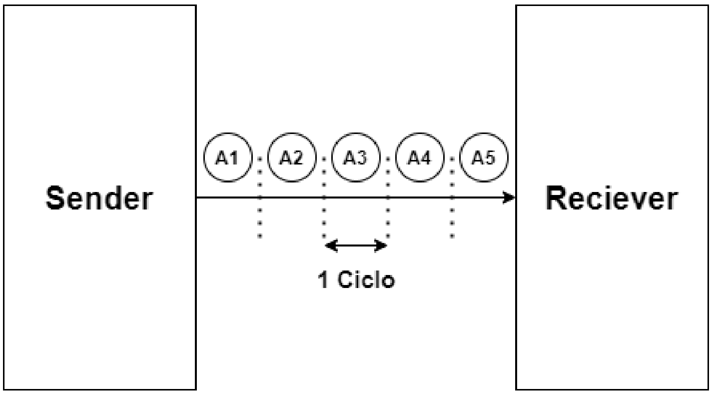


Figura 2.14: Exemplo de comunicação serial.

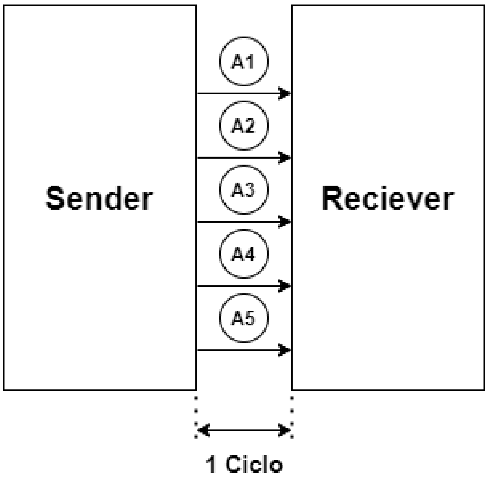


Figura 2.15: Exemplo de comunicação paralela.

A grande vantagem do protocolo USB é a baixa quantidade de canais necessários (um data + e um data -) e a confiabilidade da transferência de cada dado, pois, o protocolo utiliza de pacotes *ACK* para verificar se os dados completaram o caminho emissor->receptor. Além disso, para garantir uma transferência mais robusta, o emissor (*sender*) deve sempre aguardar o receptor (*reciever*) emitir um sinal avisando que está pronto para receber.

2.9.3 Comunicação entre ZYNQ e FPGA (AXI-LITE)

Pode-se observar que na Figura 2.11 que a comunicação entre o processador ZYNQ e a FPGA é feita somente através de portas do tipo AXI. Para ser possível a transmissão de dados do processador para os módulos na FPGA, foi necessário a criação de um *wrapper* que converte as portas de entrada e saída do módulo de mais alto nível da FPGA em portas tipo AXI (Figura 2.16), transformando o módulo *Top Level* da arquitetura FPGA em um bloco AXI "escravo".

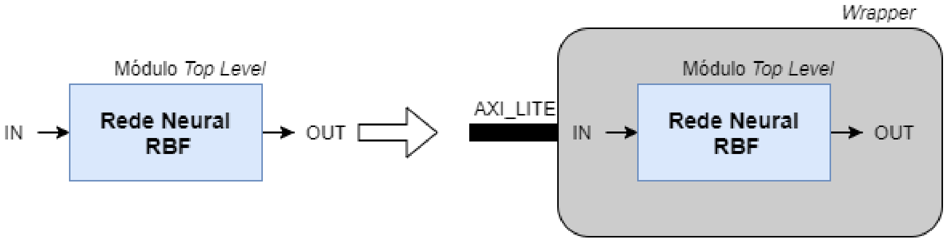


Figura 2.16: Criação de *wrapper* "escravo"para comunicação AXI.

2.9.3.1 Protocolo AXI-Lite

O protocolo AXI-Lite é uma das variações do protocolo de comunicação AXI4 [21]. A comunicação do protocolo AXI-Lite acontece de maneira chaveada, ou seja, somente um dado do módulo principal é recebido e enviado para cada ciclo de clock. Esse controle é feito por um multiplexador interno do *wrapper*, que controla qual sinal de saída e entrada está sendo manipulado por vez (Figura 2.17).

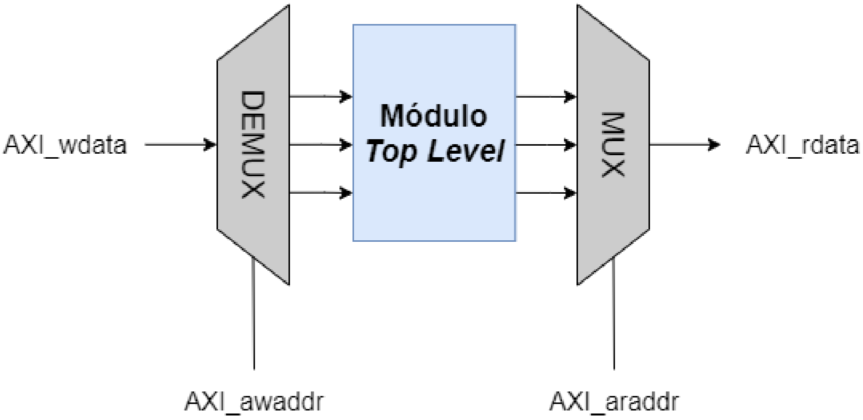


Figura 2.17: Multiplexação de dados AXI.

Para a garantir que nenhum dado é perdido durante a comunicação, esse protocolo utiliza de diversas *flags* diferentes (Figura 2.18).

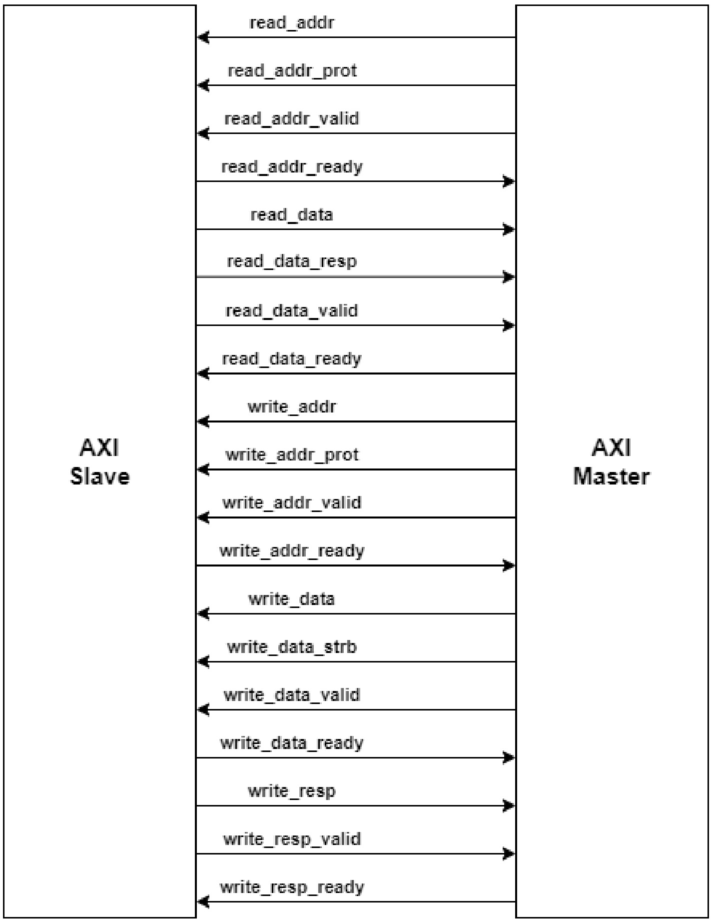


Figura 2.18: Comunicação AXI entre "mestre"e "escravo".

O funcionamento de cada uma dessas *flags* é descrito abaixo:

- **read_addr_prot**: Sinal de 3 bits responsável por indicar a prioridade e o grau de segurança da leitura no módulo "escravo".
- **read_addr_valid**: Sinal binário responsável por alertar o módulo "escravo"que o módulo "mestre"precisa enviar um endereço de leitura.
- **read_addr_ready**: Sinal binário que indica que o módulo "escravo"está pronto para guardar o endereço de uma leitura.
- **read_data_resp**: Sinal binário que indica se a informação de leitura foi enviada com sucesso para o módulo "mestre".
- **read_data_valid**: Sinal binário responsável por alertar o módulo "escravo"que o módulo "mestre"precisa receber uma informação de leitura.
- **read_data_ready**: Sinal binário que indica que o módulo "escravo"está pronto para enviar a informação de uma leitura.
- **write_addr_prot**: Sinal de 3 bits responsável por indicar a prioridade e o grau de segurança da escrita no módulo "escravo".
- **write_addr_valid**: Sinal binário responsável por alertar o módulo "escravo"que o módulo "mestre"precisa guardar um endereço de escrita.
- **write_addr_ready**: Sinal binário que indica que o módulo "escravo"está pronto para ler o endereço de uma escrita.
- **write_data_str**: Sinal responsável por indicar a máscara do endereço de memória do módulo "slave".
- **write_data_valid**: Sinal binário responsável por alertar o módulo "escravo"que o módulo "mestre"precisa gravar uma informação.
- **write_data_ready**: Sinal binário que indica que o módulo "escravo"está pronto para escrever uma informação.
- **write_resp**: Sinal binário que indica se a escrita no módulo "escravo"foi feita com sucesso.
- **write_resp_valid**: Sinal binário responsável por alertar o módulo "escravo"que o módulo "mestre"precisa verificar se a gravação foi feita com sucesso.
- **write_resp_ready**: Sinal binário que indica que o módulo "escravo"está pronto para enviar **write_resp**.

Apesar do protocolo AXI-Lite permitir diferentes estratégias de transferência de dados, ele não permite a comunicação por *AXI-burst*. Isso significa que a transferência de uma grande quantidade de dados em um mesmo *handshake* é impossível, sendo necessário um *handshake* para cada dado transferido.

O protocolo AXI-Lite foi selecionado para o projeto pelos seguintes motivos:

- Simplicidade na implementação do *wrapper*.
- Menor *overhead* de comunicação se comparado com o protocolo AXI-Stream.
- Existência de módulos prontos de fácil utilização para interconexão de dispositivos AXI-Lite no software *Vivado*.

Capítulo 3

Arquitetura do Sistema Embarcado

Nesse capítulo será apresentado como foi feita a arquitetura do sistema SoC para a execução da rede neural RBF auto-treinada. Primeiro, serão apresentados os módulos individuais em VHDL, para depois, apresentar o código usado no processador ARM. Também será apresentado neste capítulo como ocorre o processo de transmissão de dados em *pipeline*.

3.1 Sistema Geral

A distribuição de tarefas necessárias para a execução do projeto é feita entre 3 componentes diferentes:

- **O computador**: Responsável por armazenar os dados de entrada e saída da rede.
- **A FPGA**: Responsável por fazer uma das etapas do treinamento da rede e por processar a rede neural.
- **O processador ARM**: Responsável por fazer a etapa final do treinamento da rede, pela comunicação entre os componentes e pela análise da correlação entre resultados esperados e obtidos. O processador também é responsável por coordenar futuros treinamentos.

Uma ilustração representando a distribuição de tarefas entre esses componentes pode ser observada na Figura 3.1.

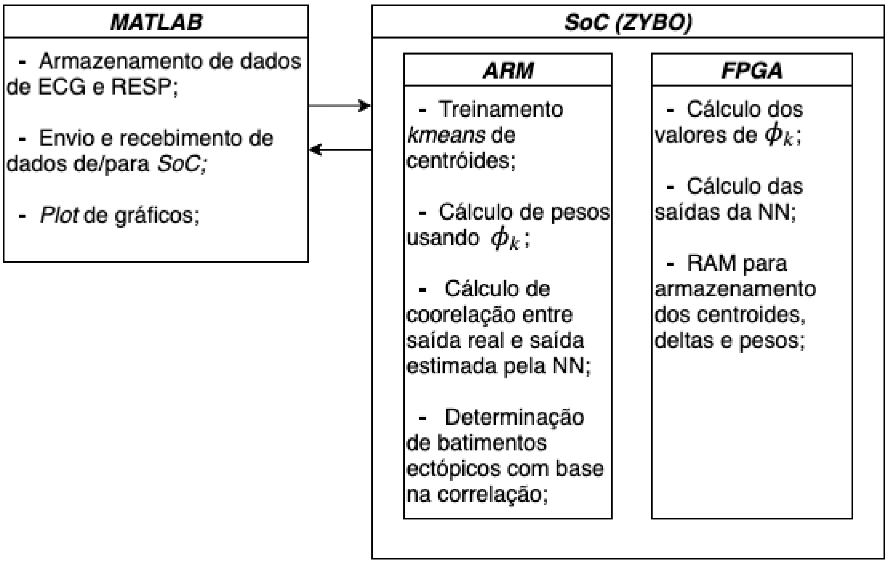


Figura 3.1: Distribuição de tarefas para cada componente do sistema.

3.2 Design em FPGA

Com objetivo de melhorar a arquitetura de rede neural desenvolvida por [1], foram criados diversos módulos e modificações que permitiram uma parametrização da rede, um re-treinamento em tempo real (quando determinado necessário pelo processador) e uma estratégia de execução em pipeline (transferência de dados e execução simultânea). Ou seja, com a nova arquitetura desenvolvida nesse projeto, é possível de se modificar em tempo de execução os valores dos pesos, centroides e deltas de cada neurônio. Esquemáticos da arquitetura inicial dos neurônios e da rede neural desenvolvida por [1] podem ser observados nas Figuras 3.2 e 3.3.

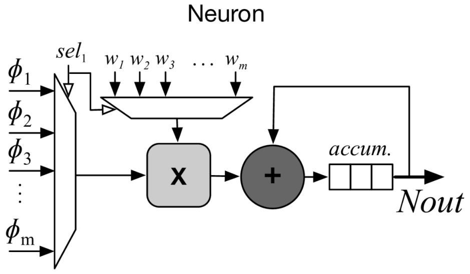


Figura 3.2: Esquemático do design de neurônio elaborado por [1].

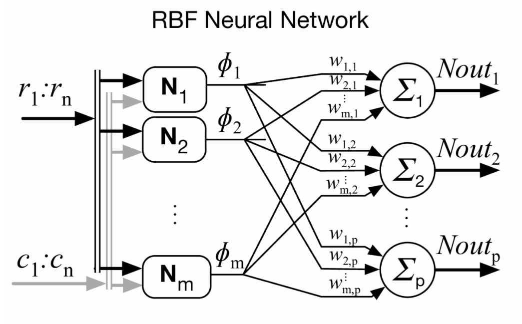


Figura 3.3: Esquemático do design da rede neural RBF elaborada por [1].

Para se fazer o design em FPGA, uma das etapas foi determinar quais partes da rede neural deveriam ser executadas em paralelo e quais partes deveriam ser executadas sequencialmente. Como pode-se observar na Figura 2.5, existe uma oportunidade de paralelização no cálculo dos resultados dos neurônios. Isto ocorre porque todos os neurônios da rede recebem as mesmas entradas (somente uma camada).

O design da arquitetura completa foi feita e testada no software *VIVADO 2018.3*.

3.2.1 Módulos VHDL

Para a melhor organização da arquitetura, o projeto foi dividido em módulos que funcionam de maneira independente e síncrona (com um mesmo sinal de clock). Nesta seção será explicado o funcionamento de cada módulo e como eles se relacionam.

3.2.1.1 Módulo Rede Neural

O módulo de rede neural é responsável por comportar a máquina de estados principal e todos os sub-módulos necessários para a execução da rede (Figura 3.4). Sendo estes:

- Neurônios (3.2.1.2).
- Memórias (3.2.1.3).
- Multiplicadores em Ponto Flutuante.
- Somadores em Ponto Flutuante.

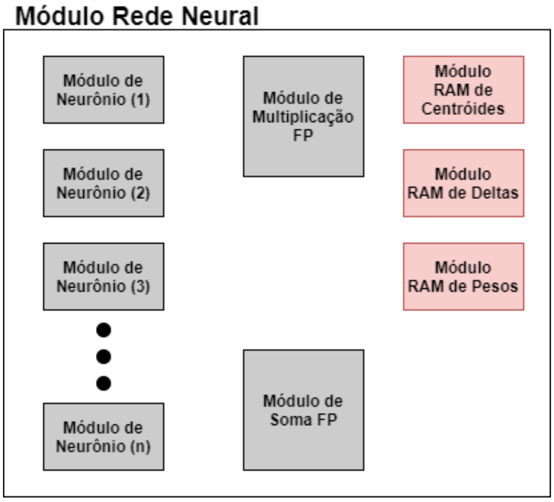


Figura 3.4: Sub-módulos usados dentro do módulo principal de rede neural.

Uma versão simplificada da máquina de estados que é executada no módulo de rede neural pode ser observada na Figura 3.5.

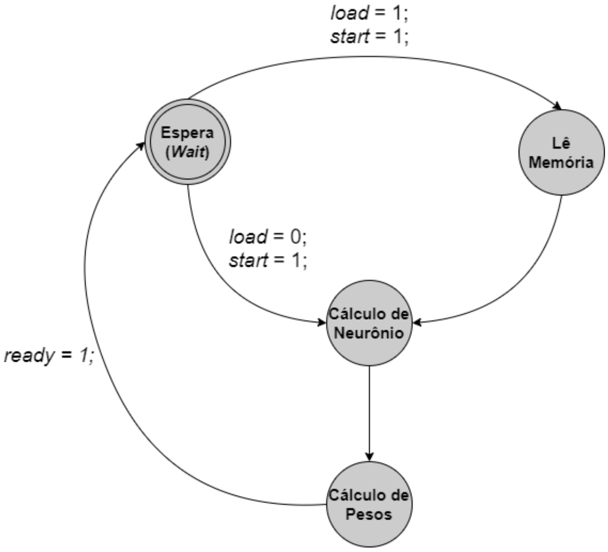


Figura 3.5: Estados simplificados do módulo de rede Neural.

3.2.1.2 Módulo Neurônio

O módulo de neurônio usado foi desenvolvido pelos professores do laboratório LEIA-UnB [1]. Este módulo é capaz de calcular a função de ativação exponencial apresentada em 2.2. Essa função é calculada através da expansão de Taylor da exponencial em $x = 0$, que é apresentada na Equação 3.1.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{3.1}$$

Ou seja, primeiro calcula-se a subtração $c_m - x_m$ usando o módulo de subtração em ponto flutuante; Em seguida, é calculada a potência de 2; E por fim, é calculado um número finito de termos da Equação 3.1 usando os mesmos módulos de multiplicação e soma.

3.2.1.3 Módulos de Memória

Para permitir que os valores de c, ω e δ pudessem ser alterados durante a execução da rede neural, foi necessária a criação de módulos de memória que se comportam como registradores intermediários em uma arquitetura de *pipeline*. Foram feitos 3 módulos de memória diferentes de mesma arquitetura. Cada módulo possui 4 sinais de entrada e 1 sinal de saída, a função de cada sinal é apresentado na Tabela 3.1.

Tabela 3.1: Sinais pertencentes ao módulo de memória.

Nome do Sinal	Função
we	Controla se o dado armazenado em data_in deve ser escrito na memória.
wadd	Endereço de escrita.
radd	Endereço de leitura.
data_in	Dado de entrada para ser escrito na memória.
data_out	Dado lido da memória.

Os tempos de leitura ou escrita de um dado é de exatamente 1 ciclo de *clock*.

3.2.2 Encapsulamento AXI

O processador utilizado na placa ZYBO só é capaz de se comunicar com a FPGA através de um barramento do tipo AXI. Isso gera a necessidade de transformar os sinais originais do módulo de rede neural em sinais para o barramento AXI, essa transformação é feita através da ferramenta automática de criação de *wrapper* do software *VIVADO*. A maneira que o encapsulamento ocorre pode ser observado na Seção 2.9.3 deste documento.

3.3 Código para processador ZYNQ

Como foi apresentado anteriormente, o processador ZYNQ é responsável por uma parte importante do treinamento da rede neural. Esse treinamento é divido em 2 partes: Cálculo de centroides (através do algoritmo *k-means*) e cálculo de pesos (através do do algoritmo OLS). O processador também é responsável pela comunicação com o MATLAB, pela análise de correlação de resultados e por executar a máquina de estados principal do projeto.

A máquina de estados do processador quando se é feito somente um treinamento da rede pode ser observada na Figura 3.6.

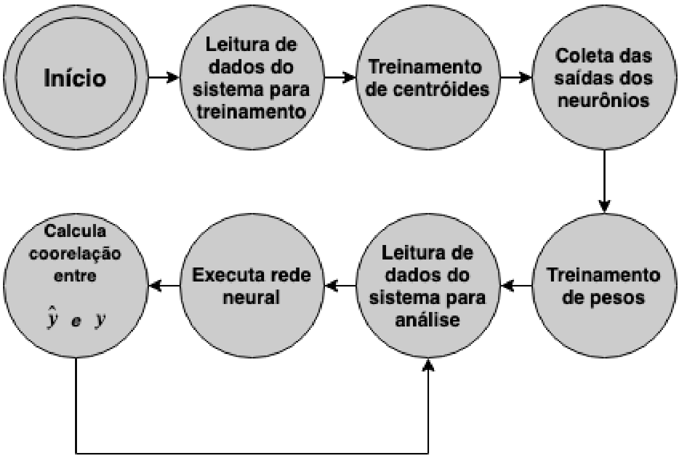


Figura 3.6: Estados do processador ZYNQ.

Para garantir a qualidade da alocação dinâmica de memória dos códigos desenvolvidos, foi usado o software *Dr. Memory* apresentado na Seção 2.8.

3.3.1 Treinamento de Centroides

O treinamento de centroides dos neurônios da rede é feito através do algoritmo *k-means* apresentado na seção 2.4.1 (o código *k-means* em linguagem C é original de [14]). O objetivo desse treinamento é calibrar as variáveis da função de ativação para entrada de cada neurônio. Ou seja, para o caso de uma rede neural com 3 entradas e 5 neurônios, deve-se calcular um total de 15 centroides diferentes (5*3). O algoritmo foi implementado em C no processador ARM integrado da FPGA.

3.3.2 Coleta de Dados para Treinamento de Pesos

Para treinar a rede neural, é necessário determinar a saída individual de cada neurônio (ϕ). Na arquitetura projetada isto é feito usando a rede já mapeada na FPGA, oferecendo um treinamento mais rápido se comparado com o cálculo puramente em C. Também foi utilizada uma arquitetura em *pipeline* para melhorar ainda mais o tempo de execução, explicada nas Figuras 3.7, 3.8 e 3.9.

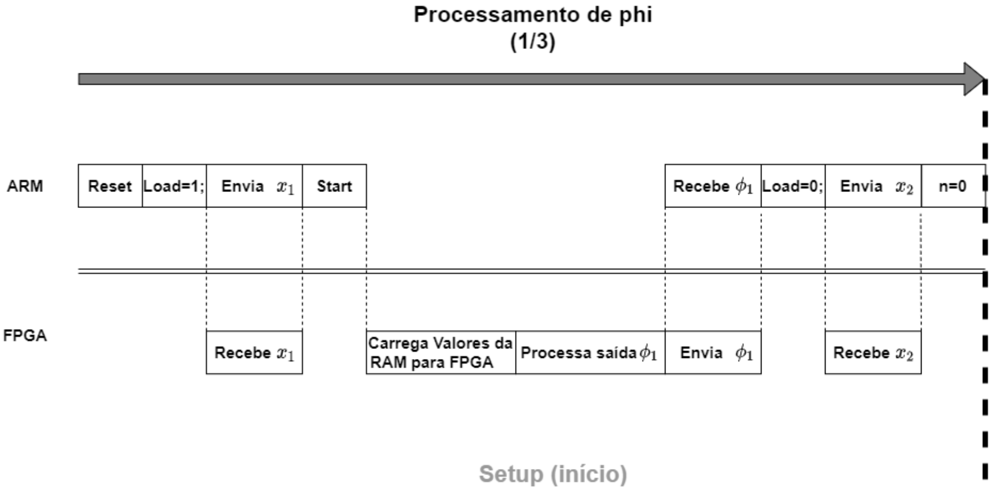


Figura 3.7: Comunicação em *pipeline* de ARM e FPGA para cálculo de ϕ parte (1/3).

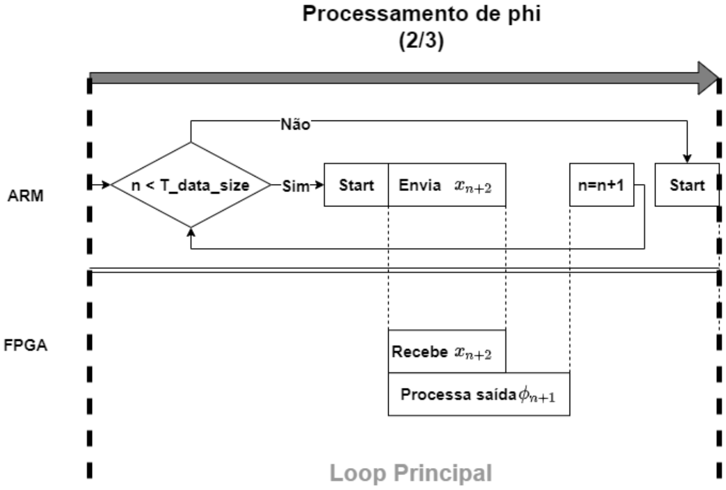


Figura 3.8: Comunicação em *pipeline* de ARM e FPGA para cálculo de ϕ parte (2/3).

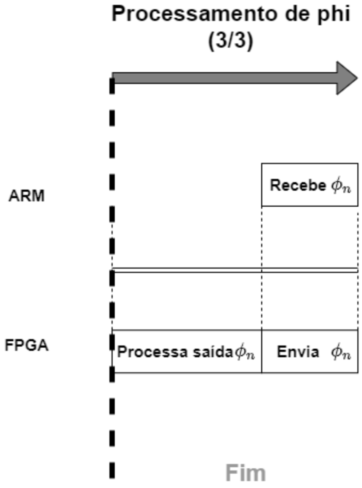


Figura 3.9: Comunicação em *pipeline* de ARM e FPGA para cálculo de ϕ parte (3/3).

Como é possível de se observar na Figura 3.8, as entradas x_{n+1} são transferidas para a FPGA ao mesmo tempo que as saídas da rede ϕ_n são processadas. Isso permite mitigar o atraso gerado pela comunicação via protocolo AXI entre o ARM e a FPGA.

3.3.3 Treinamento de Pesos usando Algoritmo OLS

Os pesos (w) representam a relevância de cada neurônio para o resultado final. Esse peso é calculado através do sistema super-determinado $\phi * w = y$, sendo ϕ as saídas individuais de cada neurônio e y as saídas esperadas para a rede. O algoritmo utilizado para solucionar este sistema foi o OLS (explicado na seção 2.4.2). A maneira que o código foi implementado no ARM pode ser observada no algoritmo 3.1.

Algoritmo 3.1: Algoritmo OLS em termos das funções implementadas em C.

```
void OLS (mat x, mat x_B, mat M_final,int neuron_number,int datasize){

    mat R, Q,Q_t,M1,M2,M1_inv;
        int i,j;

        //Decomposicao QR usando metodo Householder
        householder(x, &R, &Q);
        Q_t = matrix_copy_dynamic(datasize, Q->v, datasize);

        //Transposicao de matriz
        matrix_transpose(Q_t);

        //Multiplicacao de matrizes
        M1 = matrix_mul(Q, Q_t);
        matrix_delete(Q);

        //Inversao de matriz diagonal
        M1_inv = matrix_new(1, datasize);
        for(i=0;i<datasize;i++){
                        M1_inv->v[0][i]=1/(M1->v[i][i]);
        }
        for(i=0;i<datasize;i++){
                for(j=0;j<datasize;j++){
                        M1->v[i][j]=M1_inv->v[0][i]*Q_t->v[i][j];
                }
        }
        matrix_delete(Q_t);

        //Multiplicacao de matrizes
        M2 = matrix_new(datasize, 1);
        const_matrix_mul(M1,x_B,M2);
        matrix_delete(M1);

```

```
        //Calculo para a solucao de A*x=B quando A e uma matriz triangular superior
        triangular_linsolve(R,M2,neuron_number,M_final);

        matrix_delete(R);
        matrix_delete(M1_inv);
        matrix_delete(M2);
}
```

A função *householder* em linguagem C é original de [22].

3.3.4 Execução da Rede Neural Treinada

Uma vez treinada a rede, ou seja, determinados os centroides e pesos, é iniciado o processamento da saída da rede para cada conjunto de entradas ($\hat{y}$). Da mesma maneira que foi observado na determinação dos valores ϕ, este processamento também é executado em *pipeline*, e pode ser observado nas Figuras 3.10, 3.11 e 3.12.

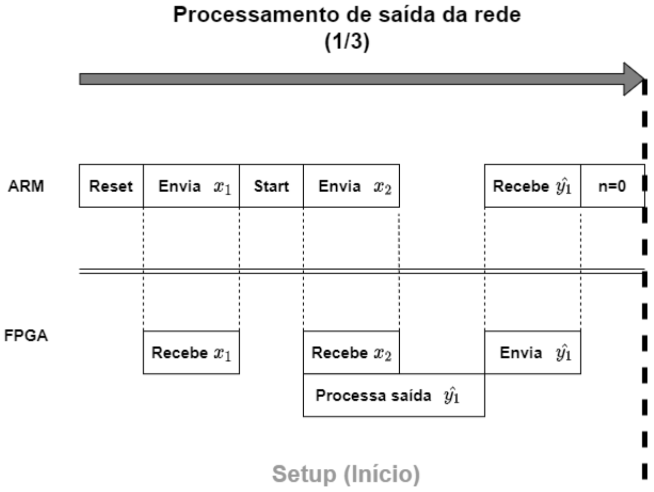


Figura 3.10: Comunicação em *pipeline* de ARM e FPGA para cálculo de $\hat{y}$ parte (1/3).

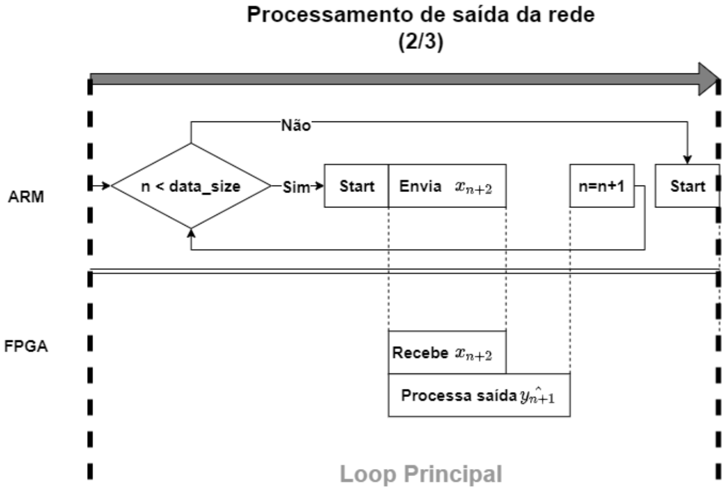


Figura 3.11: Comunicação em *pipeline* de ARM e FPGA para cálculo de $\hat{y}$ parte (2/3).

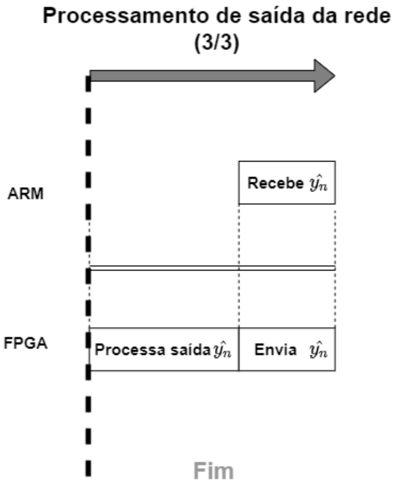


Figura 3.12: Comunicação em *pipeline* de ARM e FPGA para cálculo de $\hat{y}$ parte (3/3).

Sabendo da estratégia utilizada no cálculo de ϕ, $\hat{y}$ é calculado ao mesmo tempo em que as entradas x_{n+1} são recebidas pela rede; Mitigando novamente os *delays* causados pela comunicação AXI.

Capítulo 4

Resultados

4.1 Estudo de Caso - Resultado por MATLAB vs Resultado por SoC

Para estudar melhor a capacidade da arquitetura de identificar um sistema caixa-preta não linear, foi selecionado um *Benchmark* conhecido; O sistema escolhido foi o *Wiener-Hammerstein System* [23]. Esse sistema é conhecido por possuir um alto grau de não linearidade e por possuir uma dinâmica complexa para ser identificada (a relação entrada-saída do sistema pode ser observada na Figura 4.1). Esse sistema pode ser representado por um diagrama de blocos simples composto de 3 blocos diferentes (Figura 4.2).

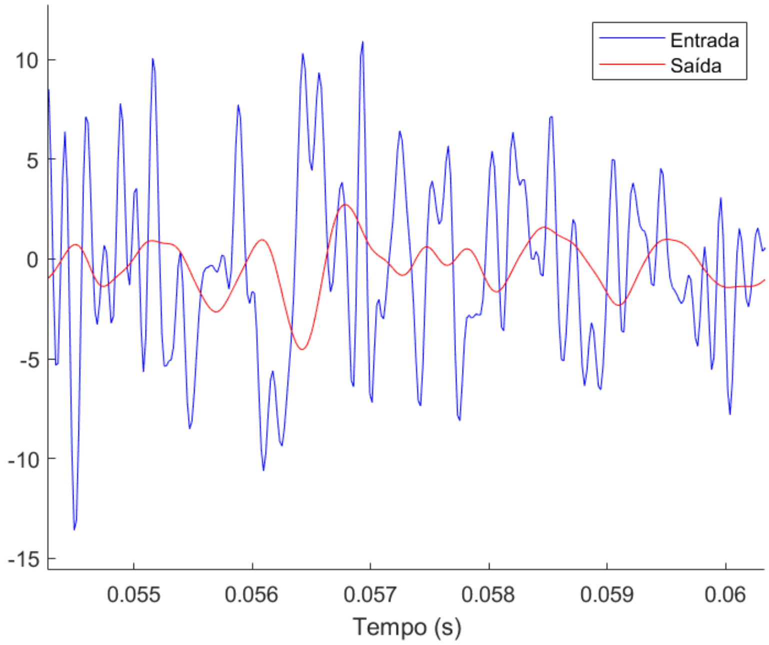


Figura 4.1: Relação não linear entre a entrada e saída do sistema WH.

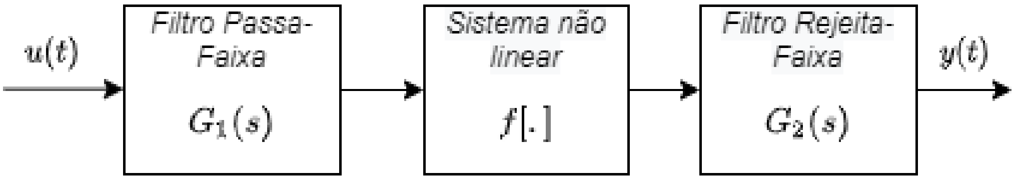


Figura 4.2: Diagrama de blocos representando o sistema *Wiener-Hammerstein*.

- $G_1(s)$: Filtro Chebyshev passa-faixa com *ripple* de 0,5dB e frequência de corte de 4.4kHz.
- $f[.]$: Circuito elétrico não linear representado pela Figura 4.3.

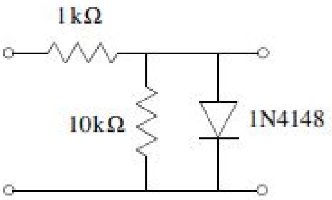


Figura 4.3: Circuito elétrico não linear usado no sistema *Wiener-Hammerstein* ($f[.]$).

- $G_2(s)$: Filtro Chebyshev rejeita-faixa com atenuação de 40dB iniciando em 5kHz.

Uma vez coletados os dados de entrada e saída do *benchmark* através do site *Nonlinear Benchmarks* [6], eles foram utilizados para treinar e validar a rede neural apresentada neste projeto. Esta validação foi feita através da análise das correlações entre as saídas estimadas das redes ($\hat{y}$) e as saídas analíticas do sistema (y) em 2 processadores diferentes (CPU executando MATLAB vs Arquitetura SoC).

Tabela 4.1: Tabela para comparação de correlações entre saída da rede e saída analítica.

	SoC (ARM+FPGA)	*MATLAB*
corr_e	0,9858	0,9646
corr_v	0,9904	0,9678

- **corr_e**: Correlação entre a saída estimada pela rede e a saída analítica para os dados utilizados no treinamento.
- **corr_v**: Correlação entre a saída estimada pela rede e a saída analítica para os dados de validação. Ou seja, dados provenientes do mesmo sistema, mas com valores diferentes dos dados utilizados para o treinamento da rede.

Como pode ser observado na Tabela 4.1 as duas redes neurais se apresentaram eficientes para estimar as saídas do sistema, com a rede implementada na SoC apresentando os melhores resultados (correlação mais próxima de 1).

Com o objetivo de descobrir o motivo da melhoria da correlação na arquitetura projetada, foram feitos dois projetos em *VHDL* para um teste em simulação. Um projeto utilizando a mantissa de 52 bits (*double ieee*) para os cálculos em FP e um projeto utilizando a mantissa de 55 bits. Foi verificado que o projeto em 52 bits possui a mesma correlação do MATLAB, enquanto projeto em 55 bits possuiu a mesma correlação da arquitetura proposta. Ou seja, foi provado que precisão superior da SoC é justificada pela maior mantissa na representação do ponto flutuante: 55 bits para SoC *vs* 52 bits para o *MATLAB*.

4.2 Resultados após análise de ECG

Para verificar que a rede neural RBF é capaz de detectar os batimentos ectópicos com eficiência, diversos conjuntos de dados de pacientes diferentes foram utilizados:

- Paciente **f2y05**: 680 segundos de leitura de ECG com amostragem de 250Hz, onde ocorreram 2 batimentos ectópicos: um em **120 segundos** outro em **221 segundos**.
- Paciente **f2o03**: 680 segundos de leitura de ECG com amostragem de 250Hz, onde ocorreram 2 batimentos ectópicos: um em **500 segundos** outro em **625 segundos**.
- Paciente **f2o08**: 680 segundos de leitura de ECG com amostragem de 250Hz, onde ocorreram 12 batimentos ectópicos (caso extremo): em **30 segundos**, **47,5 segundos**, **125,8 segundos**, **130,5 segundos**, **169,7 segundos**, **531 segundos**, **535,6 segundos**, **579,2 segundos**, **594,4 segundos**, **629,6 segundos**, **638,4 segundos** e **671,6 segundos**.

Para estudar a arquitetura projetada, foi utilizado o mesmo conjunto de dados para a execução no *MATLAB* e na SoC:

4.2.1 Executando ECG do paciente f2y05 em *MATLAB*

Após treinar a rede neural no *MATLAB* com 1100 dados iniciais, 173000 dados de ECG foram analisados. Ao usar um coeficiente de correlação com janela móvel de 170 dados foi obtido o resultado da Figura 4.4.

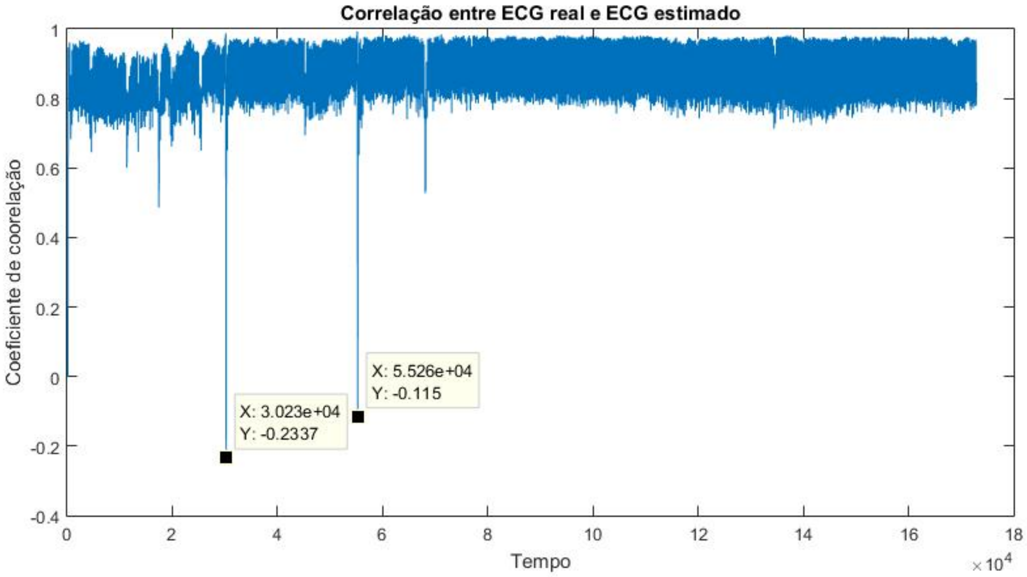


Figura 4.4: Coeficiente de correlação entre os dados de ECG estimados e os dados de ECG reais obtido pelo MATLAB do paciente f2y05.

Como pode ser observado na Figura 4.4, existem 2 pontos com coeficiente de correlação consideravelmente menor. Ao analisar os pontos $3,023 * 10^4$ e $5,526 * 10^4$ no gráfico de ECG, podemos confirmar que os batimentos ectópicos foram detectados (Figura 4.5 e Figura 4.6).

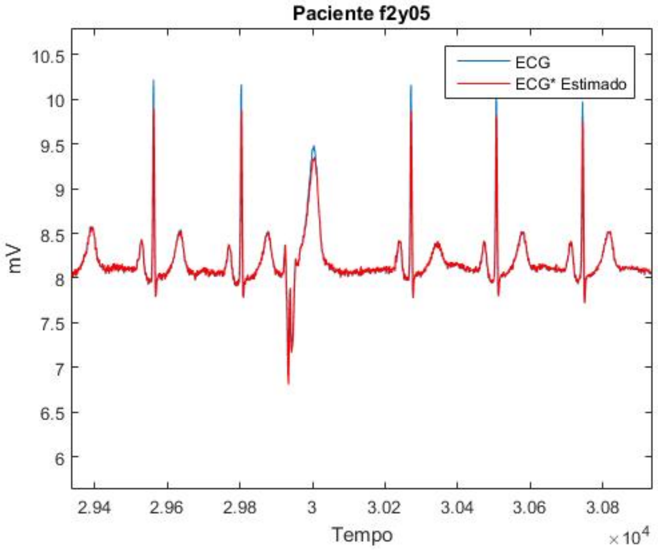


Figura 4.5: Gráfico de ECG estimado *vs* ECG real com batimento ectópico observável em $3 * 10^4$.

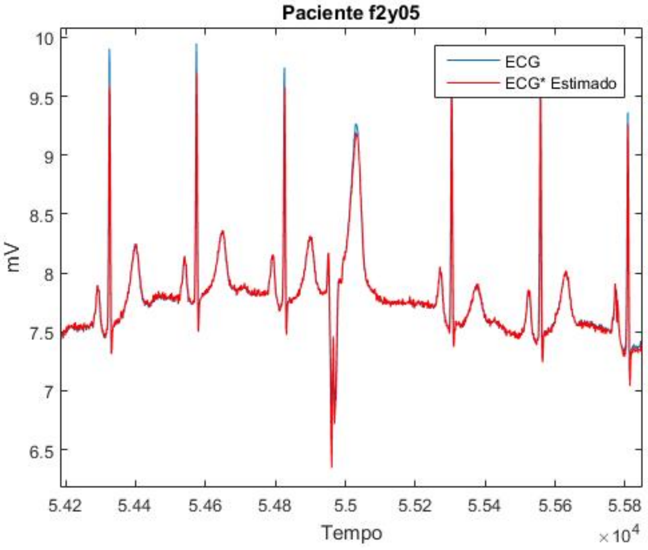


Figura 4.6: Gráfico de ECG estimado *vs* ECG real com batimento ectópico observável em $5,5 * 10^4$.

4.2.2 Executando ECG do paciente f2y05 em SoC

Ao treinar a rede neural com os mesmos 1100 dados iniciais e analisar os mesmos 173000 dados utilizados na execução pelo MATLAB, foi obtido o resultado da Figura 4.7.

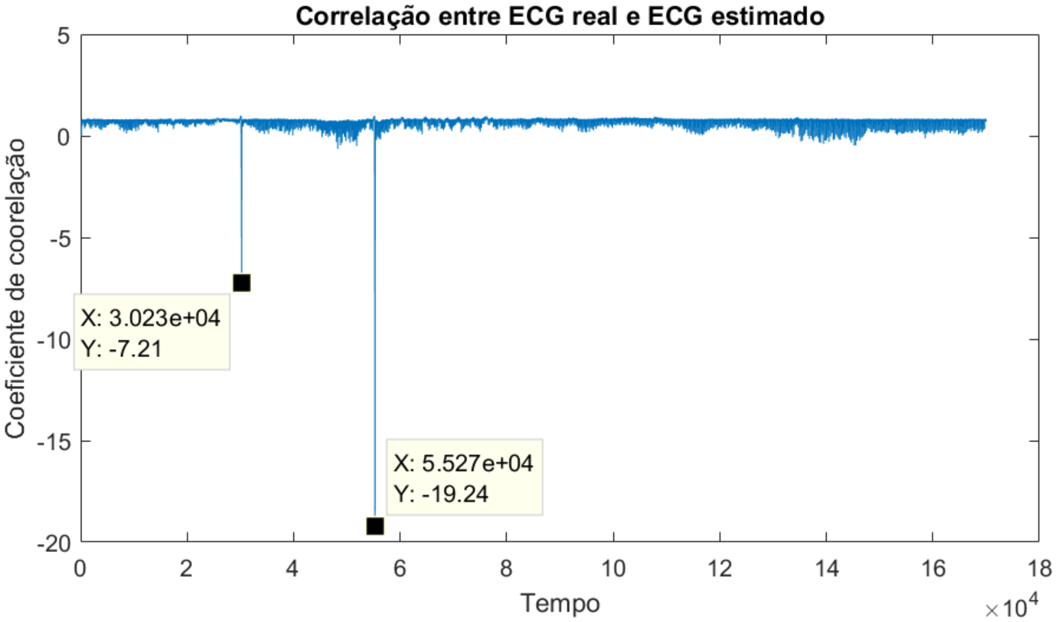


Figura 4.7: Coeficiente de correlação entre os dados de ECG estimados e os dados de ECG reais obtido pelo SoC do paciente f2y05.

Dividindo pela frequência de amostragem de 250Hz, os pontos encontrados na Figura 4.7 foram em **120,92 segundos** e **221,8 segundos**.

Pode-se observar que existe uma diferença entre os valores finais de correlação dos batimentos ectópicos (quando comparado com os resultados em MATLAB). Essa difrença é proveniente da maior prescisão da rede neural (55 bits de mantissa vs 52 bts do MATLAB). Porém, os batimentos irregulares foram detectados com o mesmo sucesso nas duas plataformas.

4.2.3 Executando ECG do paciente f2o03 em SoC

Usando 1100 dados para treinamento e 173000 dados para execução, foi obtido o resultado da Figura 4.8.

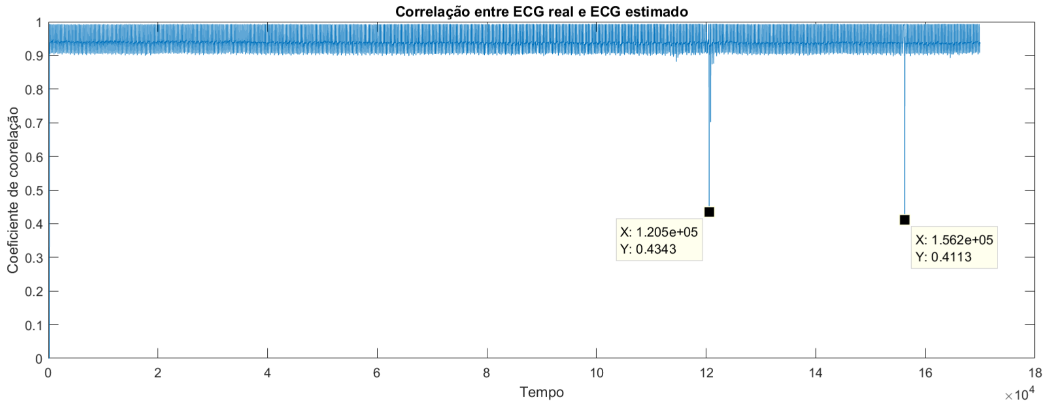


Figura 4.8: Coeficiente de correlação entre os dados de ECG estimados e os dados de ECG reais obtido pelo SoC do paciente f2o03.

Dividindo pela frequência de amostragem de 250Hz, os pontos encontrados na Figura 4.8 foram em **482 segundos** e **624,8 segundos**.

4.2.4 Executando ECG do paciente f2o08 em SoC

Usando 1100 dados para treinamento e 173000 dados para execução, foi obtido o resultado da Figura 4.9.

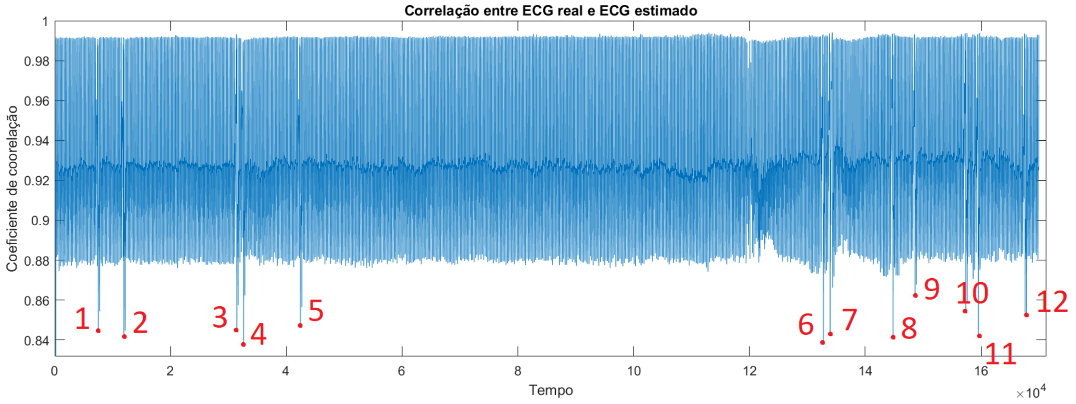


Figura 4.9: Coeficiente de correlação entre os dados de ECG estimados e os dados de ECG reais obtido pelo SoC do paciente f2o08.

Os pontos de baixa correlação encontrados na Figura 4.9 estão apresentados na Tabela 4.2.

Tabela 4.2: Tabela mostrando a localização dos pontos apresentados na Figura 4.9.

Ponto	Localização no eixo x	Localização no eixo x em segundos
1	7523	30,092
2	11960	47,84
3	31470	125,88
4	32620	130,48
5	42440	169,76
6	132800	531,2
7	133900	535,6
8	144800	579,2
9	148600	594,4
10	157400	629,6
12	159600	638,4
13	167900	671,6

Os pontos 1 e 2 podem ser observados na Figura 4.10 e na Figura 4.11 respectivamente.

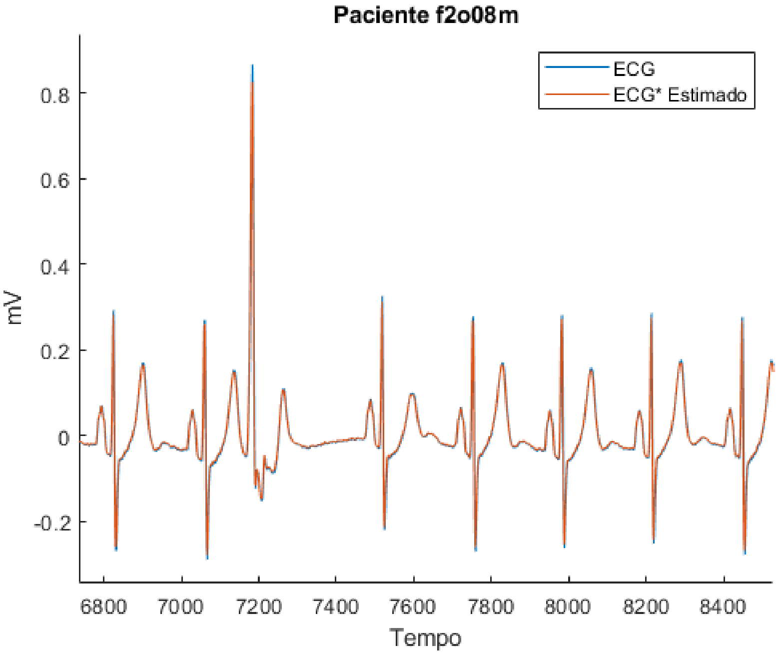


Figura 4.10: Gráfico de ECG estimado *vs* ECG real com batimento ectópico observável em $7,4{*}10^4$.

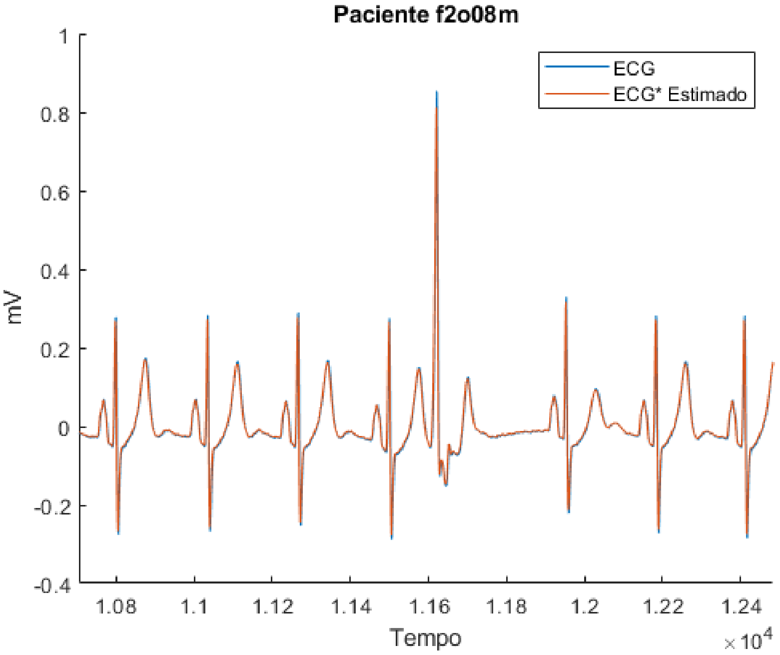


Figura 4.11: Gráfico de ECG estimado *vs* ECG real com batimento ectópico observável em $1,18 * 10^5$.

A partir dos resultados obtidos, pode-se observar que todos os batimentos ectópicos foram encontrados através da análise do gráfico de correlação.

4.2.5 Velocidade de execução no caso do WH

Para poder estudar a velocidade de execução da arquitetura desenvolvida, foi feito um novo projeto puramente em C que não utiliza da FPGA para calcular as saídas da rede neural.

Utilizando o *benchmark Wiener-Hammerstein* apresentado em 4.1, os dois projetos (com FPGA e sem FPGA) foram submetidos ao seguinte teste:

1. É feito um treinamento da rede neural com 1300 dados.
2. É calculada a saída da rede treinada para 9700 dados diferentes.

Os tempos necessários para cada um dos 2 projetos calcularem os mesmos 9700 dados podem ser vistos na na Tabela 4.3.

Tabela 4.3: Tabela comparando os tempos de execução da arquitetura projetada *vs* processador ARM para Benchmark.

Tempo de Execução (ms)	Para o Treinamento	Para o Cálculo das saídas
Em SoC (ARM+FPGA)	1472364	32,67
Em ARM Puro	1471921	74,46

4.2.6 Velocidade de execução no caso do ECG

Também foi feita uma análise comparativa entre o código de identificação de ectópicos executando no processador ARM *vs* o código executando na arquitetura proposta (ARM+FPGA co-design). A análise pode ser observada na Tabela 4.4.

Tabela 4.4: Tabela comparando os tempos de execução da arquitetura projetada *vs* processador ARM para ECG.

Tempo de Execução (ms)	Para o Treinamento	Para o Cálculo das saídas
Em SoC (ARM+FPGA)	893079	582
Em ARM Puro	892844	1337

Nos 2 casos escolhidos foi possível de se observar um aumento de 228% no tempo de execução quando removida a estratégia de *Co-design HW/SW* (ARM executando sozinho). Deixando evidente a importância da FPGA para acelerar o processamento da rede neural.

Apesar do tempo de execução da rede neural ter sido muito mais rápido na arquitetura proposta, os tempos necessários para o treinamento da rede não possuíram diferença considerável. Isso ocorre por causa da falta de otimização do algoritmo de fatorização QR. Esse algoritmo é umas das partes fundamentais para os cálculos dos pesos que, apesar de ser a parte mais custosa do treinamento, sempre é executado puramente em software (ARM).

Capítulo 5

Discussões sobre o Trabalho

5.1 Desafios do Trabalho

5.1.1 Tamanho limitado de RAM na SoC

A SoC utilizada possui 900 MB de memória disponível, que devem ser distribuídos entre *stack* e o *heap* de maneira manual com o objetivo de a evitar o *overflow*. Em diversos testes diferentes na placa, a pilha *heap* não possuía espaço suficiente para alocar todas as matrizes necessárias para o treinamento da rede neural. Isso ocorre por causa da necessidade de alocar matrizes nxm no treinamento da rede, com n e m assumindo valores nas casas dos milhares. Esse problema foi resolvido através da diminuição da pilha de *stack* para somente 30 MB e da liberação de espaços de memória que não estavam sendo imediatamente utilizados. Ou seja, somente era alocado em memória aquilo que estava sendo processado: dados que já tinham sido utilizados eram desalocados no meio do código.

5.1.2 Diferenças de Resultados entre as Simulações e as Implementações

Antes de toda implementação, era feita uma simulação do sistema no software *Vivado*. Porém, foi descoberto que os dados e atrasos calculados pela simulação nem sempre representavam fielmente os dados e atrasos do sistema implementado. Uma vez que essa diferença foi observada, as simulações em software não foram mais utilizadas para estimar saídas e atrasos.

5.1.3 Execução do Código em *Bare-Metal*

Como o projeto foi desenvolvido em um processador ARM em *bare-metal*, realizar o *debug* dos erros de programação do processador era extremamente difícil. Para solucionar esse problema, foi desenvolvido um código *mocking*, responsável por imitar o comportamento da arquitetura. Ou seja, todo o projeto (incluindo as partes implementadas em FPGA) foi implementado e simulado em C para facilitar a identificação de possíveis erros de programação (foi usado o software *devCpp*).

5.2 Limitações do Trabalho

Apesar da arquitetura projetada possuir uma execução mais rápida que um computador convencional, a etapa de treinamento é mais rápida quando feita em um processador externo ao SoC. Isso ocorre por causa da dificuldade da otimização do algoritmo sequencial de decomposição QR de uma matriz muito grande. Por esse algoritmo possuir características sequenciais e não paralelas, processadores convencionais conseguem executar essa seção de código mais rapidamente por possuírem um clock maior que o processador ZYNQ (333 MHz).

Outra limitação do trabalho é a existência do atraso gerado pela comunicação AXI-Lite (de 7 a 15 ciclos de clock) entre a placa FPGA e o processador ARM. Sendo importante ressaltar que esse atraso só existe na coleta de resultados da rede implementada na FPGA, visto que o atraso relacionado ao envio de entradas da rede é eliminado pelo *pipeline*. Também deve ser levado em consideração o atraso relacionado a comunicação USB entre o computador e a placa estimada em 14400 bytes por segundo (uma vez iniciado o processo de transferência).

Capítulo 6

Conclusões

Neste projeto foi apresentada uma arquitetura eficiente para a implementação de uma rede neural RBF auto-treinada com capacidade de identificar sistemas do tipo caixa-preta. A arquitetura projetada foi feita em um SoC composto de um processador ZYNQ e uma FPGA que se comunicam ao mesmo tempo em que ocorre o processamento das saídas da rede neural. O projeto foi validado utilizando com 2 sistemas diferentes: Um *Benchmark Wiener-Hammerstein* e o sistema biomédico ECG x Respiração. A arquitetura se mostrou competente em identificar os sistemas propostos, apresentando um ganho de velocidade de cerca de 228% quando comparado com um processador ARM comum. A correlação entre as saídas estimadas pela rede neural e as saídas analíticas foi de 96,78% para o caso do *benchmark*, provando que a rede foi capaz de identificar o sistema com sucesso.

Para o caso do processo ECGxRESP, a rede também obteve sucesso na estimação de saídas, possibilitando assim a detecção de batimentos cardíacos irregulares ao comparar a saída estimada com a saída analítica. Ou seja, foi apresentada uma arquitetura eficiente em SoC, que pode ser usada como ferramenta de detecção de batimentos ectópicos ou para outra aplicações de identificação de sistemas, visto que a arquitetura proposta é parametrizável e modular.

6.1 Trabalhos Futuros

- Otimizar a velocidade de execução do algoritmo de decomposição QR em ARM através da troca do algoritmo de decomposição de Gram–Schmidt para um algoritmo de decomposição por *householder*. Talvez implementar o algoritmo *householder* na FPGA.
- Melhorar o código de determinação de centroides iniciais convertendo o código *k-means* em sua versão mais avançada *k-means++*.
- Desenvolver o algoritmo de re-treinamento automático da rede neural caso o treinamento antigo se mostrar insatisfatório para representar o sistema (se adaptar às mudanças de dinâmica do sistema).
- Testar novas aplicações para a rede neural desenvolvida.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] SAMPAIO, R. et al. Nonlinear model predictive control hardware implementation with custom-precision floating point operations. *24th Mediterranean Conference on Control and Automation (MED)*, p. pp. 135–140, June 2016.

[2] DIGILENT. *Foto Zybo.* 2017. `https://store.digilentinc.com/zybo-z7-zynq-7000-arm-fpga-soc-development-board/`. Online; Acesso: 2020-01-12.

[3] KREININ, Y. *How FPGAs work, and why you'll buy one.* 2013. `https://www.embeddedrelated.com/showarticle/195.php`. Online; Acesso: 2020-01-12.

[4] XILINX. *Foto Processador ZYNQ-7000.* 2017. `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`. Online; Acesso: 2020-01-12.

[5] NABIL, D.; Bereksi Reguig, F. Ectopic beats detection and correction methods: A review. *Biomedical Signal Processing and Control*, v. 18, p. 228 – 244, 2015. ISSN 1746-8094. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1746809415000105>.

[6] TECHNOLOGY, E. U. of. *Non Linear Benchmarks.* 2019. `https://sites.google.com/view/nonlinear-benchmark/`. Online; Acesso: 2020-01-12.

[7] AL, G.; LA, A.; L, G. Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals. *Circulation. 2000;101(23):E215-E220*, 2000.

[8] PENG, H. et al. Nonlinear system identification using radial basis function-based signal-dependent arx model. *IFAC Proceedings Volumes*, v. 34, n. 6, p. 675 – 680, 2001. ISSN 1474-6670. 5th IFAC Symposium on Nonlinear Control Systems 2001, St Petersburg, Russia, 4-6 July 2001. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1474667017352564>.

[9] Yu, H. et al. Advantages of radial basis function networks for dynamic system design. *IEEE Transactions on Industrial Electronics*, v. 58, n. 12, p. 5438–5450, 2011.

[10] KALMYKOV, L. *Black, grey and black box definitions.* 2020. `https://www.biodiverseperspectives.com/2014/10/07/the-dark-side-of-theoretical-ecology/`. Online; Acesso: 2020-01-12.

[11] Park, J.; Sandberg, I. W. Universal approximation using radial-basis-function networks. *Neural Computation*, v. 3, n. 2, p. 246–257, 1991.

[12] MATEO, J. et al. An efficient method for ecg beat classification and correction of ectopic beats. *Computers & Electrical Engineering*, v. 53, p. 219 – 229, 2016. ISSN 0045-7906. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0045790615004450>.

[13] AYALA, H. V. H.; COELHO, L. dos S. Multiobjective cuckoo search applied to radial basis function neural networks training for system identification. *IFAC Proceedings Volumes*, v. 47, n. 3, p. 2539 – 2544, 2014. ISSN 1474-6670. 19th IFAC World Congress. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1474667016419919>.

[14] Rong Zhang; Rudnicky, A. I. A large scale clustering scheme for kernel k-means. In: *Object recognition supported by user interaction for service robots*. [S.l.: s.n.], 2002. v. 4, p. 289–292 vol.4.

[15] FROST, J. *Regression Analysis: An Intuitive Guide*. 1. ed. [S.l.: s.n.], 2019.

[16] Zhang, L. F.; Zhu, Q. M.; Longden, A. A correlation-test-based validation procedure for identified neural networks. *IEEE Transactions on Neural Networks*, v. 20, n. 1, p. 1–13, 2009.

[17] Jenna Fletcher; Gerhard Whitworth. *What to know about ectopic heartbeats*. 2020. `https://www.medicalnewstoday.com/articles/323202#causes`. Online; Acesso: 2020-01-12.

[18] IYENGAR, N. et al. Age-related alterations in the fractal scaling of cardiac interbeat interval dynamics. *Am J Physiol. 1996;271(4 Pt 2):R1078-R1084*, Oct 1996.

[19] Bruening, D.; Zhao, Q. Practical memory checking with dr. memory. In: *International Symposium on Code Generation and Optimization (CGO 2011)*. [S.l.: s.n.], 2011. p. 213–223.

[20] COMPANY, F. *The unlikely origins of USB*. 2019. `https://www.fastcompany.com/3060705/an-oral-history-of-the-usb`. Online; Acesso: 2020-01-12.

[21] ARM. *AMBA® AXI™ and ACE™ Protocol Specification*. 2011. `http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf`.

[22] MOL, M. *QR Decomposition*. 2020. `https://rosettacode.org/wiki/QR_decomposition#C`. Online; Acesso: 2020-01-12.

[23] SCHOUKENS, J.; SUYKENS, J.; LJUNG, L. Wiener-hammerstein benchmark. *15th IFAC Symposium on System Identification (SYSID 2009)*, July 2009.

ANEXOS

I. LINK PARA ACESSO AOS CÓDIGOS

Os códigos utilizados no projeto estão disponíveis em GIT:

Figura I.1: Código QR para acesso ao GIT.

https://gitlab.com/leiaunb/trabalhos-finais-de-gradua-o/tg-gabriel-reves.git