UNIVERSIDADE FEDERAL DA BAHIA

ESCOLA POLITÉCNICA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

HENRIQUE SOUZA BAQUEIRO DOS SANTOS

PROJETO E IMPLEMENTAÇÃO DE UMA REDE NEURAL ARTIFICIAL *FEEDFORWARD* PARAMETRIZÁVEL EM FPGA

SALVADOR - BA

NOVEMBRO, 2019

HENRIQUE SOUZA BAQUEIRO DOS SANTOS

PROJETO E IMPLEMENTAÇÃO DE UMA REDE NEURAL ARTIFICIAL *FEEDFORWARD* PARAMETRIZÁVEL EM FPGA

Dissertação apresentada à Coordenação do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal da Bahia, em cumprimento às exigências para obtenção do grau de Mestre em Engenharia Elétrica.
Orientador: Wagner Luiz Alves de Oliveira

Salvador - BA

Novembro, 2019

Ficha catalográfica elaborada pelo Sistema Universitário de Bibliotecas (SIBI/UFBA),
com os dados fornecidos pelo(a) autor(a).

Souza Baqueiro dos Santos, Henrique
Projeto e implementação de uma rede neural artificial feedforward parametrizável em FPGA / Henrique Souza Baqueiro dos Santos. -- Salvador, 2019.
84 f. : il

Orientador: Wagner Luiz Alves de Oliveira.
Dissertação (Mestrado - Engenharia Elétrica) -- Universidade Federal da Bahia, Escola Politécnica da UFBA, 2019.

1. FPGA. 2. Inteligência Artificial. 3. Redes Neurais Artificiais. I. Luiz Alves de Oliveira, Wagner. II. Título.

Henrique Souza Baqueiro dos Santos

"PROJETO E IMPLEMENTAÇÃO DE UMA REDE NEURAL ARTIFICIAL FEEDFORWARD PARAMETRIZÁVEL EM FPGA"

Dissertação apresentada à Universidade Federal da Bahia, como parte das exigências do Programa de Pós-Graduação em Engenharia Elétrica, para a obtenção do título de *Mestre*.

APROVADA em: 28 de Novembro de 2019.

BANCA EXAMINADORA

Prof. Dr. Wagner Luiz Alves de Oliveira
Orientador - UFBA

Prof. Dr. Paulo César Machado de Abreu Farías
UFBA

Prof. Dr. Acbal Rucas Andrade Achy
UFRB

AGRADECIMENTOS

Gostaria de agradecer em primeiro lugar ao amigo Dórian Langbeck, que foi quem de fato sentou do meu lado e me ensinou Verilog, a pedra fundamental deste trabalho, lá em 2014 na época do CI Brasil. Também ao professor Gabriel Nazar por suas aulas brilhantes e aos instrutores Cézar Reinbrecht, Jerson Guex, Felipe Almeida, Maurício Carvalho, Henrique Fellini e Pedro Toledo.

Aos colegas do SENAI CIMATEC Cleber Almeida, Rodrigo Tutu, João Marcelo e Valmiro Rangel, pelos conhecimentos de FPGA e por me concederem um tempo no trabalho para que eu me dedicasse ao aprendizado de processamento digital de imagens e redes neurais.

Ao meu orientador Wagner Oliveira, por acreditar em mim, pela disponibilidade e pelo incessante apoio ao longo deste trabalho.

Ao professor Edson Santana, cuja tese de doutorado aguçou minha curiosidade sobre redes neurais durante a minha iniciação científica.

À Michael Nielsen e Andrew Ng, duas grandes mentes inspiradoras com a admirável habilidade de simplificar conhecimentos não raramente obscurecidos por uma notação matemática pesada e indigesta.

À Clara, primeiramente por sacrificar muito do nosso tempo para que eu pudesse conceber este trabalho, mas também pela coragem, força e apoio no período mais difícil de minha vida, em que por mais hiperbólico que possa soar, houve dias que eu não conseguia entender nem recordar nenhuma linha do que já havia desenvolvido.

Por fim, aos meus pais, pelo silêncio e colaboração que para eu conseguisse concluir esta etapa.

"Nós devemos fazer o nosso melhor.
Essa é a nossa responsabilidade humana sagrada.[1] *"*
(Einstein sagt, p.133, tradução nossa)

[1] Wir müssen unser Bestes tun. Das ist unsere heilige menschliche Verantwortung.

SANTOS, H.S.B. Projeto e Implementação de uma Rede Neural Artificial *Feedforward* parametrizável em FPGA. 84 f. il. color. Dissertação (Mestrado) – Programa de Pós-graduação em Engenharia Elétrica, Universidade Federal da Bahia. Salvador, 2019.

RESUMO

Uma Rede Neural Artificial (RNA) é uma rede paralela e distribuída formada por unidades de processamento não-lineares interconectadas em camadas. Este trabalho consiste no projeto e implementação em *hardware* da arquitetura de uma rede do tipo *feedforward* parametrizável. As descriçoes de *hardware* em *Verilog* são geradas por um *script* em Python. Nele, tem-se como parâmetros o número de *bits* na entrada, o número de camadas, o número de neurônios por camada, o número de *bits* para as partes inteira e fracionária na representação de ponto fixo e o número de *bits* para a representação da *look-up table* (LUT) da função de ativação. Essa parametrização permite que se consiga, ainda em tempo de simulação, uma configuração mínima de *hardware* que atenda a um determinado problema, economizando-se assim tanto tempo de desenvolvimento como recursos em *hardware*. Como aplicação, utilizou-se o *hardware* desenvolvido para descrever uma rede treinada *offline* com a base de dados MNIST e com isso classificar em tempo real imagens de algarismos provenientes de uma câmera. Dessa maneira, o trabalho mostra-se apropriado tanto para amparar pesquisas em áreas afins, que necessitem embarcar uma rede neural artificial, quanto para experimentar modificações na arquitetura do projeto, dada a disponibilidade do ambiente de verificação desenvolvido.

Palavras-chaves: FPGA. Inteligência Artificial. Redes Neurais Artificiais.

SANTOS, H.S.B. Project and Implementation of a Parametrizable Feedforward Artificial Neural Network in FPGA. 84 p. il. color. Thesis (Master) – Programa de Pós-graduação em Engenharia Elétrica, Universidade Federal da Bahia. Salvador, 2019.

ABSTRACT

An Artificial Neural Network (ANN) is a parallel and distributed network made of non-linear processing units arranged in layers.This work consists of the project and hardware implementation of the architecture of a parameterizable feedforward neural network. The Verilog hardware descriptions are generated by a Python script. The parameters are the number of input bits, the number of layers, the number of neurons per layer, the number of bits for the integer and fraction parts in fixed-point representation and the number of bits for the activation function look-up table (LUT) representation. This parameterization allows a minimum hardware configuration to be met at simulation time for a given problem, thus saving both development time and resources in hardware. As an application, a hardware was generated to describe a network which was trained offline with the MNIST database and which classifies images from a camera video streaming in real time. In this way, this work is appropriate both to support research in related areas that need to embed an artificial neural network, as well as to develop improvements in the architecture, given the availability of the developed verification environment.

Key-words: FPGA. Artificial Intelligence. Artificial Neural Networks.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ANN	Artificial Neural Network (Rede Neural Artificial)
ASIC	Application Specific Integrated Circuit (Circuito Integrado de Aplicação Específica)
ASIP	Application Specific Instruction Processor (Processador de Instruções Específicas da Aplicação)
CNN	Convolutional Neural Network (Rede Neural Convolucional)
CRT	Cathode Ray Tube (Tubo de Raios Catódicos)
DAC	Digital to Analog Converter (Conversor Digital-Analógico)
DSP	Digital Signal Processor (Processador Digital de Sinais)
FIFO	First In, First Out (Fila - primeiro a entrar, primeiro a sair)
FPGA	Field-Programable Gate Array (Matriz de Portas Programável em Campo)
FSM	Finite State Machine (Máquina de Estados Finitos)
GPU	Graphics Processing Unit (Unidade de Processamento Gráfico)
IP	Intellectual Property (Propriedade Intelectual)
LUT	Look-Up Table (Tabela de Consulta)
MIF	Memory Initialization File (Arquivo de Inicialização de Memória)
MLP	Multilayer Perceptron (Perceptron Multicamada)
MNIST	Modified NIST (NIST modificado)
MSE	Mean Squared Error (Erro Quadrático Médio)
NIST	National Institute of Standards and Technology (Instituto Nacional de Padrões e Tecnologia)
OCR	Optical Character Recognition (Reconhecimento Óptico de Caracteres)
PLL	Phase Locked Loop (Malha de Captura de Fase)
RAM	Random Access Memory (Memória de Acesso Aleatório)

RNA	Rede Neural Artificial
RNN	Recurrent Neural Network (Rede Neural Recorrente)
ROM	Read Only Memory (Memória de Somente Leitura)
SCCB	Serial Camera Control Bus (Barramento de Controle de Câmera Serial)
VGA	Video Graphics Array (Matriz de Gráficos de Vídeo)

SUMÁRIO

1 INTRODUÇÃO

A Inteligência Artificial pode ser definida como a área do conhecimento que se dedica à pesquisa e desenvolvimento de máquinas inteligentes. A inteligência, por sua vez, não é um conceito de definição trivial, principalmente quando refere-se à máquinas. Contudo, ela está relacionada com a parte computacional da habilidade de alcançar objetivos no mundo. Variados graus de inteligência ocorrem em pessoas, alguns animais e também algumas máquinas (MCCARTHY, 2007).

A origem dessa área data o início da década de 50, quando alguns acontecimentos de destaque foram cunhados na história. É o caso do artigo "*Computing Machinery and Intelligence*", em que o cientista da computação britânico Alan Turing apresenta o atualmente famoso Teste de Turing (TURING, 1950), voltado para identificar a inteligência em máquinas por meio de perguntas e repostas. Também, ainda na mesma época e não menos importante, o escritor de ficção científica russo naturalizado americano Isaac Asimov apresenta as Três Leis da Robótica em seu livro "Eu, Robô". Apesar de possuírem uma abordagem mais filosófica, esses acontecimentos trataram de temas prevendo uma sociedade em que um dia os robôs estariam completamente inseridos entre humanos e as suas implicações.

Poucos anos antes, em 1947, o primeiro transistor foi construído pelos físicos norte-americanos William Shockley, John Bardeen e Walter Brattain na Bell Labs. Porém, a fabricação do primeiro transistor planar de silício comercial só se concretizou em 1954, pela Texas Instruments. Nascia então a área da Microeletrônica, com os esforços contínuos para a miniaturização dos componentes.

Em 1958, Frank Rosenblatt, inspirado em trabalhos prévios de Warren McCulloch e Walter Pitts, apresentou um modelo probabilístico para o armazenamento e organização de dados no cérebro, o qual ele chamou de Perceptron, o primeiro neurônio artificial (ROSENBLATT, 1958). A partir de então, começaram a surgir modelos matemáticos inspirados no cérebro biológico.

Nesse mesmo período em que a Microeletrônica começava a florescer, em 1969 Marvin Minsky e Seymour Papert publicaram o livro "Perceptrons" (MINSKY; PAPERT, 1988), no qual foi feita uma análise matemática profunda mostrando todas as limitações desse modelo que poderia ser considerado seriamente uma representação do cérebro. Esse livro causou grande alvoroço na comunidade científica e contou com várias edições. Com isso a Inteligência Artificial passou por um longo período de ceticismo e descrédito, com cortes nos financiamentos e cancelamentos de pesquisas, período que ficou conhecido como o inverno da Inteligência Artificial.

Paralelamente, em 1965, Gordon E. Moore, um dos fundadores e então diretor dos laboratórios de pesquisa e desenvolvimento da *Fairchild Semiconductor* (hoje Intel), escreveu um famoso artigo para a comemoração do 35º aniversário da revista *Electronics*. Nele, Moore observou que o número de componentes por chip aproximadamente dobrava a cada ano e previu, baseado nas condições tecnológicas da época, que essa tendência deveria continuar ao menos pelos próximos 10 anos (MOORE, 1965). Esta famosa previsão ficou conhecida como "Lei de Moore" e, surpreendentemente, manteve-se válida até por volta de 1998-2000 (MACK, 2011).

Esse contínuo crescimento exponencial do poder de computação ao longo dos anos fez com que passasse a ser possível desenvolver soluções para problemas que anteriormente somente humanos eram capazes de resolver, a exemplo da classificação de padrões. A classificação de padrões na visão computacional abriu portas para aplicações somente vistas antes em filmes de ficção científica, como veículos autônomos e diagnósticos precoces baseados em imagens médicas.

Apesar de os primeiros acontecimentos relevantes para a área da Inteligência Artificial darem-se no início dos anos 50, os avanços mais significativos até então aconteceram somente há pouco mais de uma década. Em primeiro lugar, em 2006 foi desenvolvido um conjunto de técnicas baseado em gradiente descendente estocástico e retropropagação, o qual possibilitou o treinamento de redes profundas muito maiores e com mais camadas do que as que vinham sendo apresentadas até então pela comunidade científica. Segundo, passou-se a utilizar unidades de processamento gráfico (GPUs) para acelerar o treinamento (em várias ordens de grandeza), o que possibilitou também um menor tempo de iteração entre as diferentes abordagens de treinamento (NIELSEN, 2015).

Para se ter uma ideia da importância da natureza do *hardware* na evolução dos trabalhos científicos, no ano de 1998, quando a base de dados MNIST (a qual é usada neste trabalho) foi apresentada pela primeira vez, levou-se semanas para treinar uma estação de trabalho que era estado da arte na época. Hoje, com uso de GPUs, consegue-se resultados de classificação substancialmente maiores em menos de uma hora (NIELSEN, 2015).

Existem na literatura outras abordagens para a implementação em *hardware* de redes neurais artificiais, cada uma com suas vantagens e desvantagens intrínsecas às decisões de projeto. Os circuitos integrados de aplicação específica (ASICs) possuem as vantagens da alta performance e eficiência energética. Porém, as suas grandes desvantagens são a absoluta falta de flexibilidade em se adaptar à diferentes modelos, o alto tempo de desenvolvimento e o elevadíssimo custo de produção (WANG et al., 2016). Aceleradores neurais existentes podem ser classificados na categoria de processador de instruções de aplicação específica (ASIP) e ainda não possuem flexibilidade o suficiente para dar suporte à diferentes arquiteturas de diferentes áreas, as quais podem diferir bastante em se tratando de parâmetros e fluxo de dados (SU et al., 2017). As implementações em processadores de

sinais digitais (DSPs) são sequenciais e portanto não preservam a arquitetura paralela de neurônios em uma dada camada. Por fim, as matrizes de portas programáveis em campo (FPGAs) são o tipo de *hardware* mais adequado para a implementação de redes neurais, dado que preservam a arquitetura paralela dos neurônios em uma camada, o tempo e custo de implementação são baixos e podem ser reconfiguradas pelo usuário (HIMAVATHI; ANITHA; MUTHURAMALINGAM, 2007).

Arquiteturas de redes neurais profundas podem conter um número muito grande de neurônios. Goodfellow, Bengio e Courville (2016) mostram que, desde a introdução das unidades ocultas, as redes neurais artificiais dobraram de tamanho aproximadamente a cada 2,4 anos. A rede GoogLeNet, em 2015, já continha neurônios da ordem de 10^6. Assim, do ponto de vista da descrição de hardware, torna-se impraticável para o projetista digital conseguir realizar a implementação sem algum tipo de ferramenta que auxilie a codificação.

Este trabalho apresenta o projeto e a implementação de uma rede neural artificial do tipo *feedforward* parametrizável em FPGA, mostrando no seu decorrer as vantagens e desvantagens de cada decisão de projeto. O *script* gerador da descrição de *hardware* desenvolvido para o projeto proposto é parametrizado e pode ser utilizado para gerar redes customizadas para trabalhos futuros.

1.1 OBJETIVOS

O objetivo geral deste trabalho é desenvolver uma ferramenta que gere a descrição de hardware em *Verilog* para uma rede neural artificial do tipo *feedforward* com parâmetros variáveis (número de *bits* na entrada, número de camadas, número de neurônios por camada, número de *bits* para as partes inteira e fracionária na representação de ponto fixo e número de *bits* para a representação da *look-up table* (LUT) da função de ativação). Como aplicação sugerida tem-se o problema do reconhecimento óptico de caracteres: uma rede neural artificial cuja descrição de *hardware* é gerada pelo *script* proposto por este trabalho é utilizada para classificar imagens de algarismos de zero a nove.

Os objetivos específicos são:

- Apresentar uma visão geral sobre redes neurais artificiais;
- Projetar a arquitetura de uma rede neural artificial *feedforward* em *hardware*, apresentando os motivos de cada decisão de projeto e contrapondo com outras possíveis abordagens;
- Utilizar o *hardware* desenvolvido para uma aplicação, desenvolvendo os módulos adicionais que porventura sejam necessários;

- Treinar a rede em *software* por meio de um computador (treinamento *offline*) para obter os pesos e polarizações e o posterior carregamento dos mesmos em *hardware*;
- Publicar em artigos e/ou revistas nas correspondentes áreas do conhecimento.

1.2 ORGANIZAÇÃO DO TRABALHO

O Capítulo 2 apresenta a metodologia seguida, contendo a descrição de todas as etapas de desenvolvimento, bem como os materiais e *softwares* utilizados. O Capítulo 3 apresenta a fundamentação teórica necessária para o desenvolvimento deste trabalho: a estrutura do neurônio, tipos funções de ativação, a notação utilizada, topologias de redes, tipos de função custo e a descrição dos algoritmos de retropropagação e de gradiente descendente. O Capítulo 4 apresenta pontos relevantes a serem considerados quando da implementação de uma rede neural em FPGA, como tipos de treinamento, os sistemas de representação de dados e formas de se implementar a função de ativação. O Capítulo 5 apresenta com detalhes a arquitetura proposta e as decisões de projeto. O Capítulo 6 apresenta a aplicação proposta, a qual é a utilização de uma câmera para prover dados de entrada para a rede neural desenvolvida e com isso classificar em tempo real imagens de algarismos. Ele também contém informações sobre a base de dados utilizada e o projeto dos módulos adicionais necessários para se trabalhar com a câmera. O Capítulo 7 contém os resultados e discussão e por fim, o Capítulo 8 conclui o trabalho e apresenta sugestões de trabalhos futuros.

2 METODOLOGIA

O desenvolvimento da pesquisa foi dividido em seis etapas:

1. Revisão Bibliográfica;
2. Modelagem do Sistema;
3. Projeto, Codificação e Verificação da Microarquitetura;
4. Prototipagem em FPGA;
5. Desenvolvimento do *Script* Gerador da descrição de *hardware*;
6. Documentação e Análise de Resultados.

A revisão bibliográfica engloba a pesquisa e leitura de artigos, livros e periódicos com o objetivo de determinar o estado da arte e ampliar os conhecimentos sobre redes neurais artificiais e suas possíveis implementações em FPGA.

A modelagem do sistema em alto nível tem como objetivo realizar o treinamento e produzir os parâmetros de configuração da rede (pesos e polarizações) para uma dada aplicação.

O projeto da microarquitetura consiste na elaboração da organização da arquitetura, incluindo todas unidades funcionais, suas interconexões e controle. É nesta etapa que pondera-se os prós e contras de cada forma de implementação. Uma boa revisão bibliográfica é fundamental para que nesta etapa estejam claras as possibilidades de caminhos que podem ser seguidos e quais devem ser evitadas. A codificação compreende a transcrição da representação do esquemático do circuito desenvolvido em linhas de código que representem a sua descrição em *hardware*. Já a verificação baseia-se na utilização de *testbenches* para garantir a funcionalidade do projeto e/ou de partes dele. Em um fluxo normal de projeto, a codificação e verificação ocorrem em paralelo.

A prototipagem em FPGA é importante para confirmar que a codificação da descrição do *hardware* é sintetizável, ou seja, que é possível traduzir a descrição em portas lógicas (em rigor, em uma dada configuração de um conjunto de elementos lógicos interconectados, no caso de FPGAs).

O desenvolvimento de um *script* gerador de descrição de *hardware* é bastante útil para a automatização do processo de codificação, principalmente no caso de arquiteturas muito grandes mas que tenham trechos de códigos que se repetem. Ainda, ele possibilita a

parametrização do design em um nível maior que a linguagem Verilog ou SystemVerilog por contar com bibliotecas matemáticas e o poder de manipular strings.

Por fim, a documentação e análise de resultados concluem o trabalho e delineam as suas contribuições.

Os materiais e *softwares* utilizados ao longo da pesquisa foram:

Mendeley Desktop versão 1.19.3 - *Software* produzido pela editora de literatura científica Elsevier para gerenciar artigos científicos e fazer fichamentos.

Anaconda3 versão 1.8.7 - Distribuição *free* e *open source* das linguagens de programação Python e R que visa simplificar o gerenciamento e a implantação de pacotes.

- Ambiente Python 2.7 com os pacotes bitstring, numpy e matplotlib.
- Spyder versão 3.2.8

GNU Octave versão 4.2.2 - Linguagem de alto-nível destinada principalmente a cálculos numéricos e que possui uma alta compatibilidade com Matlab.

ModelSim - INTEL FPGA STARTER EDITION versão 10.5b - Ambiente de simulação para linguagens de descrição de *hardware*.

Quartus Prime Lite Edition versão 18.0.0 build 614 - *Software* de projeto para dispositivos lógicos programáveis. É responsável pela compilação do projeto (análise e síntese, *placement* e roteamento, geração dos arquivos de programação, análise de *timing* e escritor de *netlist*) e gravação no dispositivo alvo.

Kit de desenvolvimento Altera DE2-115 - FPGA Cyclone IV de alta densidade (114.480 elementos lógicos, 3.888 Kbits de memória embarcada, 266 multiplicadores 18x18 embarcados, 4 PLLs de propósito geral e 528 entradas/saídas para o usuário).

Câmera OV7670 - Câmera VGA de baixo custo que provê imagens 8-bit em diferentes formatos.

3 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados a estrutura de um neurônio, as principais funções de ativação, a notação utilizada ao longo do trabalho, as topologias das redes *feedforward*, convolucionais e recorrentes, o conceito de função custo e o algoritmo de treinamento gradiente descendente utilizando retropropagação.

3.1 ESTRUTURA DO NEURÔNIO

A estrutura de um neurônio de múltiplas entradas é mostrada na Figura 1. Sejam p_1, p_2 ... p_n as entradas do sistema, w_1, w_2 ... w_n os pesos correspondentes, "b" a polarização e $f(z)$ a função de ativação/excitação. O processamento realizado pelo neurônio é descrito pelas equações 3.2 e 3.1.

$$z = \sum_{i=1}^{n} p_i w_i + b \tag{3.1}$$

$$a = f(z) \tag{3.2}$$

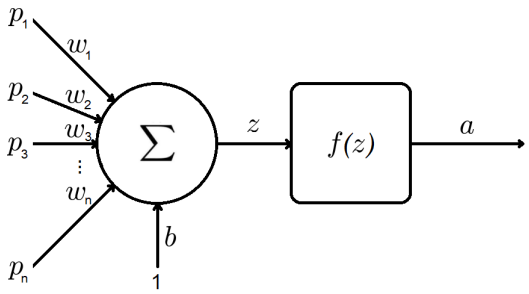


Figura 1 – Estrutura básica de um neurônio de múltiplas entradas.

A função de ativação $f(z)$ representa o mapeamento da saída ou ativação "a" de um neurônio em relação ao sinal de suas entradas ponderadas "z". Ela é responsável por inserir não-linearidades na rede e com isso possibilitar que os neurônios aprendam modelos mais complexos.

3.2 FUNÇÕES DE ATIVAÇÃO

Existem diversos tipos de função de ativação reportados na literatura. A lista apresentada nesta seção não tem a intenção de ser exaustiva, porém engloba todas as referências utilizadas.

- **Degrau** - O Perceptron, um tipo de neurônio artificial desenvolvido por Frank Rosenblatt na década de 50, modelava um neurônio disparando caso as suas entradas ponderadas ultrapassassem um determinado valor de limiar (o qual pode ser demonstrado matematicamente como sendo o negativo do valor da polarização b). Do contrário, o Perceptron não disparava. Portanto, a função de ativação de um Perceptron pode ser feita com uma função degrau (em rigor, para um Perceptron $f(z) = 0$ para $z = 0$):

$$f(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases} \tag{3.3}$$

- **Linear** - A função de ativação linear é utilizada em neurônios de Modelos de Regressão Linear, um modelo cuja performance é conhecida por ser bastante limitada. A não-linearidade da função de ativação é importante pois permite que a rede mapeie a função que se deseja sem que a(s) saída(s) tenha(m) que ser necessariamente uma combinação linear das entradas. Por consequência, a função de ativação linear não costuma ser utilizada para todos os neurônios.

$$f(z) = Cz \tag{3.4}$$

  onde C é uma constante, geralmente igual a 1.

- **Sigmóide** - A função Sigmóide é a escolha clássica de muitos trabalhos pois a sua derivada tem um custo computacional baixo: $f'(z) = f(z)(1 - f(z))$.

$$f(z) = \frac{1}{1 + e^{-z}} \tag{3.5}$$

- **Sigmóide *Hard*** - A função Sigmóide *Hard* é a aproximação da função Sigmóide por três retas.

$$f(z) = \begin{cases} 0 & z < -2,5 \\ 0,2z + 0,5 & -2,5 \leq z \leq 2,5 \\ 1 & z > 2,5 \end{cases} \tag{3.6}$$

- **Tanh** - A função Tangente Hiperbólica tem o mesmo formato da função Sigmóide, sendo na verdade uma versão escalada dela. Enquanto que a Sigmóide varia no eixo y de 0 a 1, a Tangente Hiperbólica varia de -1 a 1.

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3.7}$$

- **Exponencial** - A função Exponencial tem características bastante desejáveis para uma função de ativação, como uma derivada de custo computacional baixo: $f'(z) = ze^z$. Seu maior problema é que quanto maior a abcissa, maior sua inclinação. Esse

aspecto faz com que se mescle a parte negativa da Exponencial com a parte positiva de outras funções, criando-se novas funções de ativação como a ELU.

$$f(z) = e^z \tag{3.8}$$

- **ReLU** - A função ReLU é linear com inclinação positiva para abcissas positivas e zero para negativas.

$$f(z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases} \tag{3.9}$$

- ***Leaked* ReLU** - A função *Leaked* ReLU é igual à Linear e ReLU para números positivos. Para números negativos ela continua retilínea, porém com uma inclinação muito pequena ($\varepsilon = 0,01$ por exemplo) visando garantir que nessa região a derivada não seja zero, como acontece na ReLU.

$$f(z) = \begin{cases} \varepsilon z & z < 0 \quad (\varepsilon << 1) \\ z & z \geq 0 \end{cases} \tag{3.10}$$

- **ELU** - A função ELU é linear para números positivos e exponencial para números negativos. O valor de $-\alpha$ determina a assíntota em $-\infty$.

$$f(z) = \begin{cases} \alpha(e^z - 1) & z < 0 \\ z & z \geq 0 \end{cases} \tag{3.11}$$

- **SELU** - A função SELU é a ELU escalada por uma constante C.

$$f(z) = \begin{cases} C\alpha(e^z - 1) & z < 0 \\ Cz & z \geq 0 \end{cases} \tag{3.12}$$

- ***Softplus*** - A função *Softplus* é uma versão da ReLU com curvas mais suaves e portanto sem a descontinuidade na origem.

$$f(z) = log(e^z + 1) \tag{3.13}$$

- ***Softsign*** - A função *Softsign* tem um um formato semelhante à Tanh, porém com curvaturas mais suaves.

$$f(z) = \frac{z}{|z| + 1} \tag{3.14}$$

A Figura 2 reúne o gráfico de cada uma das funções apresentadas anteriormente.

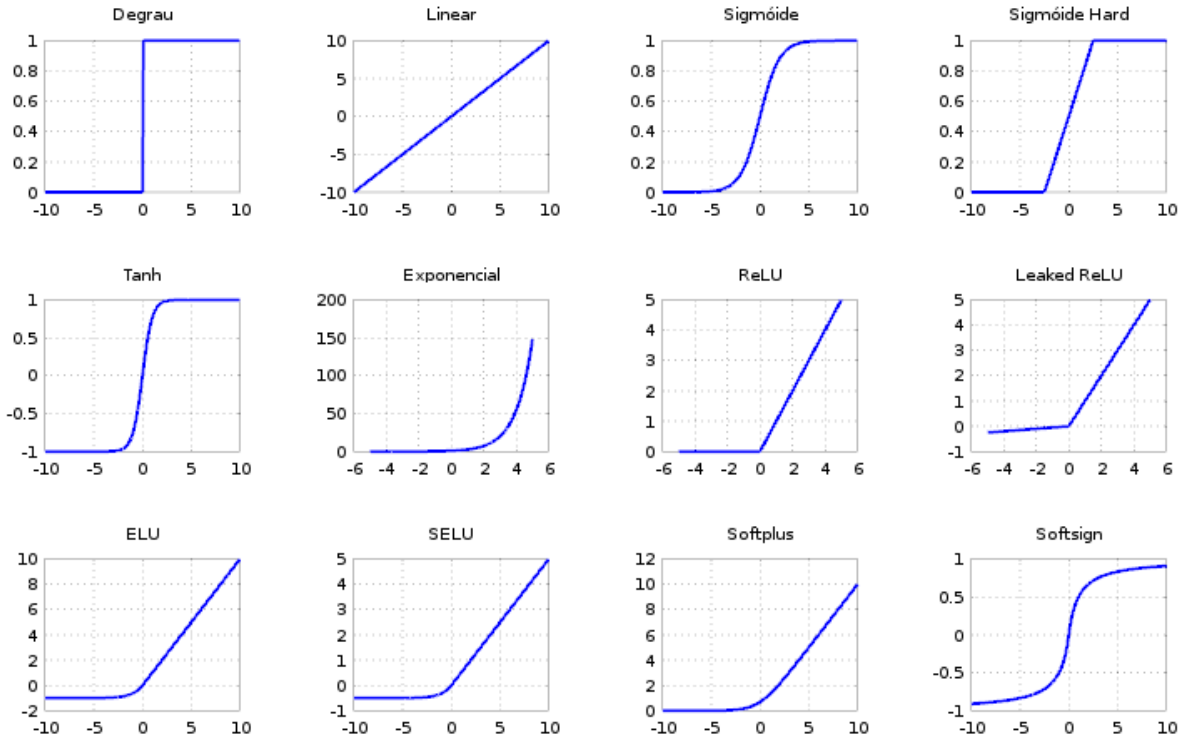


Figura 2 – Principais funções de ativação.

3.3 NOTAÇÃO

A notação utilizada ao longo deste trabalho é baseada no livro *"Neural Networks and Deep Learning"* (NIELSEN, 2015) e exemplificada com detalhes na Figura 3.

- $\boldsymbol{b}$ - Polarização. Exceto pelos neurônios de entrada, todos neurônios possuem a sua respectiva polarização. O índice superior indica a camada, enquanto que o inferior indica a qual neurônio daquela camada ele pertence. Por exemplo, b_3^2 indica a polarização do terceiro neurônio da segunda camada.

- $\boldsymbol{w}$ - Peso. Os pesos ligam a saída de um neurônio à entrada de outro. Em uma arquitetura *feedforward*, essas ligações só acontecem entre neurônios de camadas adjacentes. O índice superior indica a camada de destino, enquanto que os índices inferiores indicam o neurônio de destino na camada de destino l, e o neurônio de origem na camada $l-1$. Na Figura 3, w_{24}^3 indica o peso que vai para o segundo neurônio da terceira camada, advindo do quarto neurônio da camada imediatamente anterior.

- $\boldsymbol{z}$ - Entradas ponderadas. É o valor resultante da soma de todas as entradas do neurônio multiplicadas pelos respectivos pesos e adicionando-se a polarização. O índice superior indica a camada, enquanto que o inferior indica a qual neurônio

daquela camada ela pertence. Por exemplo, z_2^3 indica a ativação do segundo neurônio da terceira camada.

- ***a*** - Ativação. É o valor resultante da aplicação da função de ativação em uma entrada ponderada. O índice superior indica a camada, enquanto que o inferior indica a qual neurônio daquela camada ela pertence. Por exemplo, a_2^2 indica a ativação do segundo neurônio da segunda camada.

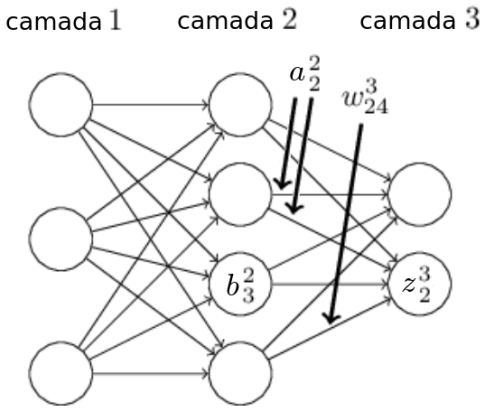


Figura 3 – Notação para os pesos, polarizações, entradas ponderadas e ativações.

3.4 TOPOLOGIAS

A topologia refere-se a como os neurônios são conectados entre si. Existe um número muito grande de topologias publicadas, cada uma com suas peculiaridades. Nesta seção serão vistos três grandes grupos que cobrem os conceitos mais importantes. São eles: redes feedforward, convolucionais e recorrentes.

3.4.1 Redes Neurais Feedforward

Redes *feedforward* servem de base para muitas outras topologias e por isso são o exemplo clássico dentre modelos de aprendizagem profunda. O objetivo de uma rede *feedforward* é fazer a aproximação de alguma função f^*. Para um classificador tendo $y = f^*(x)$, a qual mapeia uma entrada x a uma categoria y, uma rede *feedforward* define o mapeamento $y = f(x; \theta)$ e aprende o valor dos parâmetros θ que resultam na melhor aproximação da função f^* (GOODFELLOW; BENGIO; COURVILLE, 2016).

Chama-se *feedforward* porque nesse tipo de rede não há conexões de saídas de neurônios para eles mesmos ou para algum outro neurônio de alguma camada anterior. Toda a informação flui sempre da entrada para a saída, sem a possibilidade de *loops* ao longo do caminho. Redes *feedforward* também são comumente chamadas de MLP (*Multilayer Perceptron*), apesar de seus neurônios poderem ser de outros tipos (ou seja, com outras funções de ativação), não necessariamente Perceptrons.

Uma rede *feedforward* de quatro camadas é apresentada na Figura 4. A camada mais à esquerda é chamada de camada de entrada. A camada de entrada consiste simplesmente nos valores de entrada na rede e portanto seus neurônios não possuem polarizações. As setas para a camada seguinte representam os pesos. A camada mais à direita é chamada camada de saída. Todas as outras camadas entre a de entrada e de saída são chamadas de camadas ocultas. Diz-se ser completamente conectada quando para qualquer neurônio da rede existem conexões para todos os neurônios da camada seguinte.

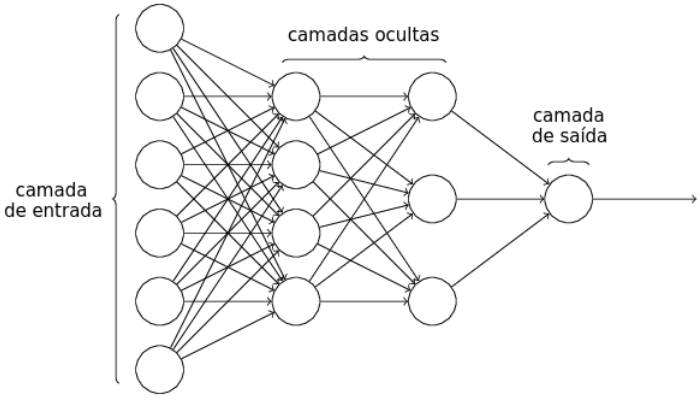


Figura 4 – Exemplo de rede *feedforward*.

3.4.2 Redes Neurais Convolucionais

As redes neurais convolucionais (CNN) exploram a bidimensionalidade do vetor de entrada e por isso são bastante apropriadas para a classificação de imagens. Esse tipo de topologia é baseado em três novos conceitos: campos receptivos locais, parâmetros compartilhados e *pooling*.

Em uma CNN, um dado neurônio da primeira camada oculta não está completamente conectado com todos os neurônios de entrada, mas somente com uma pequena região dela, como mostrado na Figura 5, à qual dá-se o nome de campo receptivo local. Cada uma dessas conexões aprende um peso e o neurônio oculto em si aprende uma polarização.

A Figura 6 exemplifica a construção de parte da primeira camada oculta para um vetor de entrada de 28x28 *pixels* e campos receptivos locais de 5x5 *pixels* separados uns dos outros por 1 *pixel*. Ao mover-se a área do campo receptivo local sobre o vetor de entrada sempre 1 *pixel* da direita para esquerda e de cima para baixo, o resultado será um mapa de características de 24x24 neurônios.

Dentre um mesmo mapa de características, todos os neurônios aprendem os mesmos pesos e polarizações, isto é, cada neurônio oculto é responsável por identificar exatamente a mesma característica em diferentes partes do vetor de entrada. Isso reduz drasticamente o número de parâmetros a serem aprendidos - no exemplo, 25 pesos e 1 polarização compartilhados. Caso cada campo receptivo local possuísse seu próprio conjunto de pesos

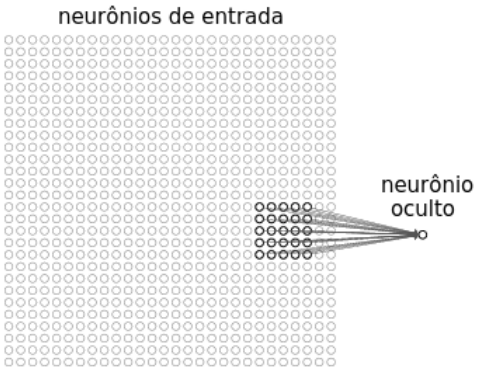


Figura 5 – Campo receptivo local em uma CNN.

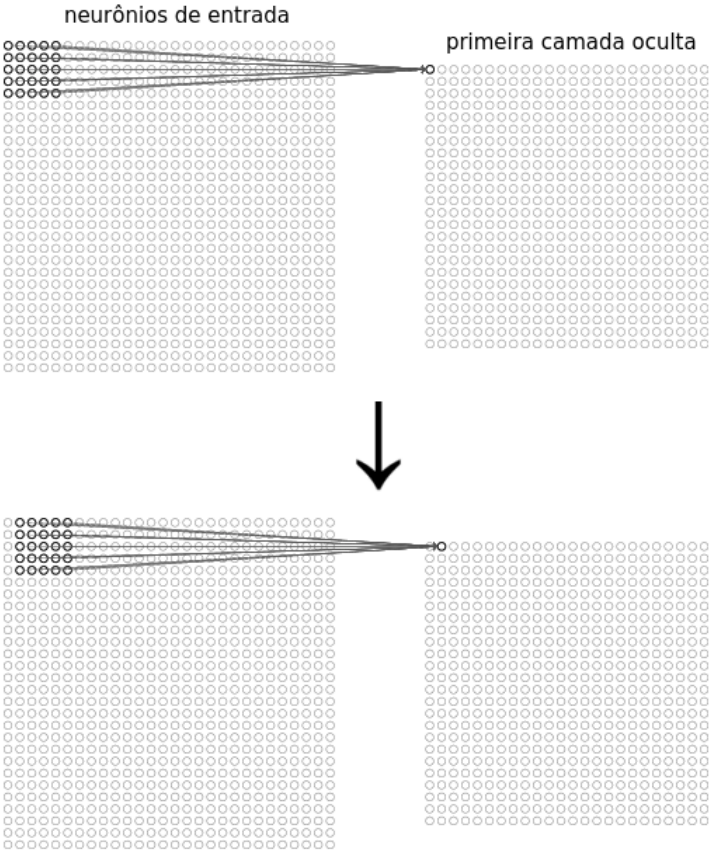


Figura 6 – Mapeamento dos neurônios ocultos para os respectivos campos receptivos locais.

e polarizações, teriam-se $25 * 24 * 24 = 14400$ pesos e $24 * 24 = 576$ polarizações. Os pesos e polarizações compartilhados são muitas vezes também chamados de *kernel* ou filtro.

Como todos os neurônios dentre um mesmo mapa de características compartilham a mesma polarização e os mesmos pesos, cada mapa de características é capaz de identificar somente um tipo de característica. Consequentemente, para que uma CNN seja útil são necessários múltiplos mapas por camada. A Figura 7 apresenta o exemplo de uma camada oculta constituída por três mapas de características. À essa camada dá-se o nome de camada de convolução.

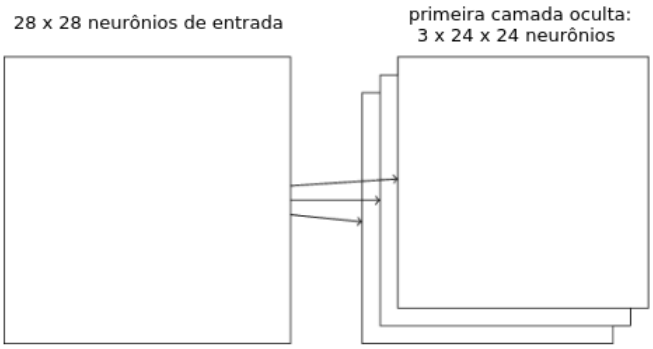


Figura 7 – Exemplo de uma camada oculta constituída por três mapas de características.

A operação dos neurônios continua a mesma: sua saída é a multiplicação das entradas pelo respectivos pesos, somada com a polarização. Para um neurônio na linha j e coluna k do mapa de características do exemplo, sua saída é dada pela expressão 3.15, onde $f(z)$ é a função de ativação escolhida, b é o valor da polarização compartilhada, $w_{l,m}$ é o vetor 5x5 de pesos compartilhados e $a_{x,y}$ é a ativação de entrada na posição (x, y). Por conta dessa expressão, camadas desse tipo são chamadas de camadas de convolução.

$$a_{j,k} = f\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}\right) \tag{3.15}$$

Camadas do tipo *pooling* são usadas logo após camadas de convolução. Elas são usadas para simplificar a informação na saída da camada de convolução, gerando novos mapas de características, porém dessa vez condensados. Um exemplo comum de camada do tipo *pooling* é a *max-pooling*, mostrada na Figura 8, em que a unidade de saída tem como valor a ativação de entrada mais alta. Outro tipo comum de *pooling* é o do tipo L2, em que ao invés de escolher a ativação mais alta, calcula-se a raiz quadrada da soma dos quadrados dos valores das ativações de entrada.

As últimas camadas, mostradas na Figura 9, geralmente são camadas completamente conectadas com a camada anterior, como nas redes *feedforward*. Para problemas de classificação pode-se usar na saída uma camada do tipo *softmax* para ter-se uma distribuição de probabilidade com a probabilidade de cada saída, ou simplesmente uma camada do tipo

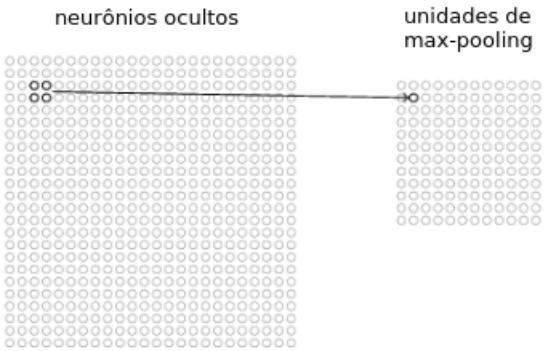


Figura 8 – Unidades de *max-pooling* para uma área de 2x2.

hardmax, onde somente tem-se "1" na saída do neurônio com o maior sinal de entradas ponderadas, enquanto que tem-se "0" na saída de todos os outros neurônios. Essas funções de ativação serão apresentadas na subseção 5.2.3.

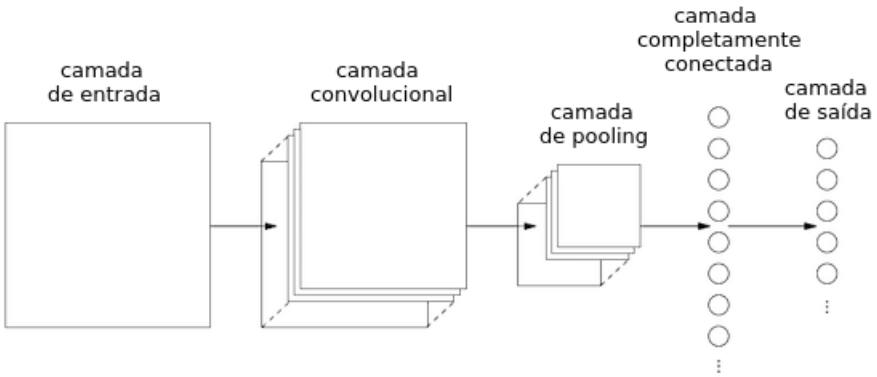


Figura 9 – Exemplo de uma CNN simples.

3.4.3 Redes Neurais Recorrentes

Redes neurais recorrentes (RNN) são redes em que *loops* são permitidos. A ideia por trás destes modelos é ter neurônios que disparem por um determinado período, antes de se tornarem quiescentes. Esse disparo pode estimular outros neurônios, os quais podem disparar instantes depois e assim sucessivamente, em cascata. Dessa forma, consegue-se que a influência da ativação do neurônio na rede seja prolongada, simulando de certa maneira o comportamento de uma memória.

Existem um grande número de topologias de redes neurais publicadas e a apresentação de cada uma delas foge do escopo deste trabalho. Contudo, a Figura 10 apresenta, a título de exemplo, uma extensa lista das possibilidades mais conhecidas na literatura e os seus respectivos tipos de neurônios.

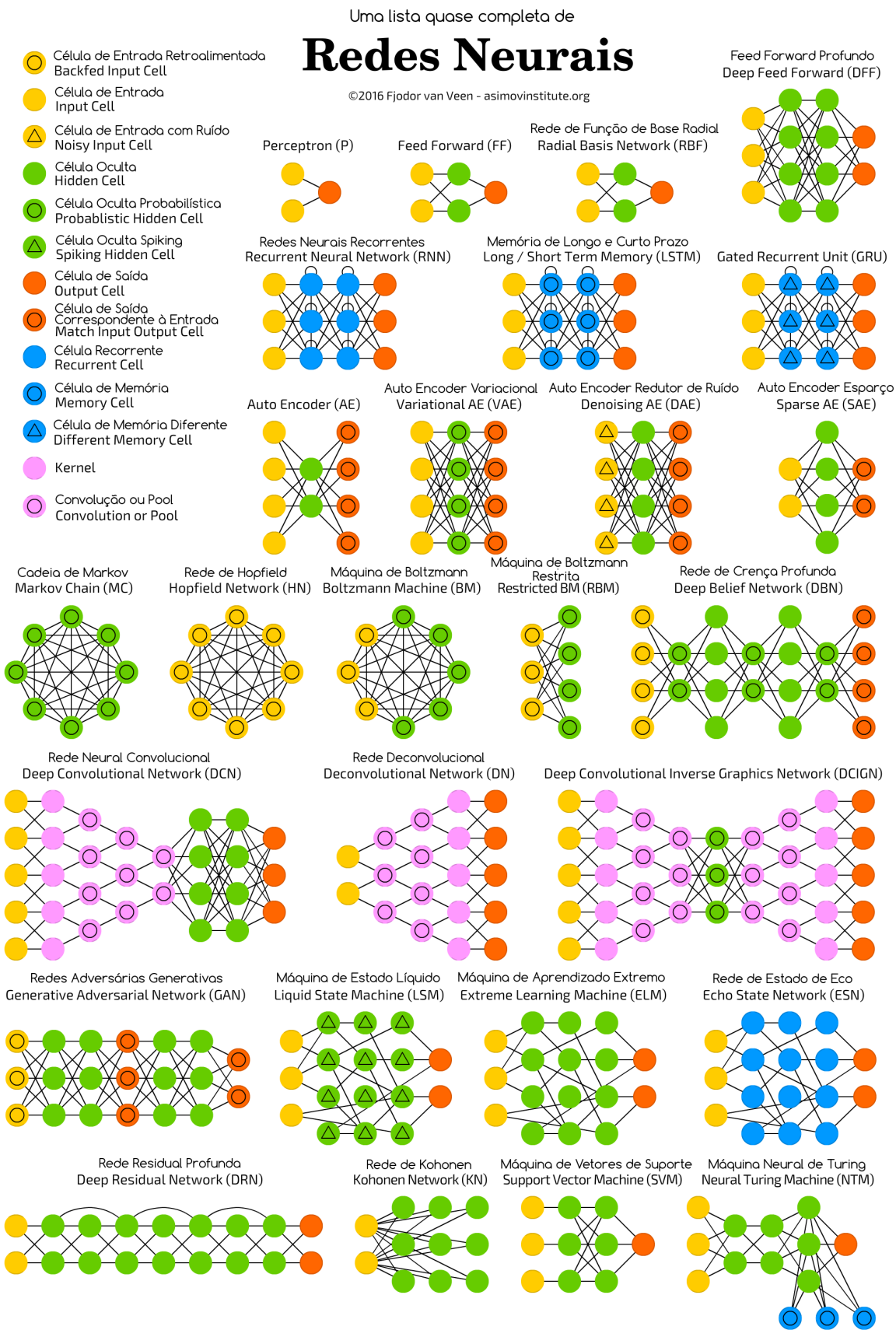


Figura 10 – Diferentes topologias de redes neurais artificiais.
Fonte: https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464 (tradução nossa)

3.5 FUNÇÃO CUSTO

Para que se saiba se o treinamento está indo bem ou mal é preciso que se haja algum tipo de métrica. Considerando um problema de treinamento supervisionado, é preciso que de alguma forma consiga-se medir o quão próximas as saídas atuais estão das saídas desejadas.

A forma mais intuitiva de se fazer isso seria variar os pesos e polarizações de forma a se tentar maximizar diretamente o número de saídas corretas. Todavia, o número de saídas corretas não é um função suave e contínua dos pesos e polarizações. Como consequência, pode ocorrer que para uma pequena variação desses parâmetros não se observe nenhuma mudança no número de saídas corretas, tornando bastante difícil escolher uma estratégia para melhorar a performance da rede.

Para superar esse problema, escolhe-se uma função contínua, e portanto diferenciável, que determine o comportamento que se quer para a rede. Essa função, a qual podemos querer minimizar ou maximizar, é chamada função objetivo ou critério. Quando a minimizamos, ela também pode ser chamada de função custo, função perda ou função erro. A maioria dos problemas de otimização de aprendizado profundo são postos como uma minimização de $f(x)$. A maximização pode ser alcançada com um problema de minimização minimizando-se $-f(x)$ (GOODFELLOW; BENGIO; COURVILLE, 2016).

- **Erro Quadrático Médio (mean squared error - MSE)**

$$C(w,b) = \frac{1}{2n}\sum_x \|y-a\|^2 \tag{3.16}$$

- **Entropia-Cruzada (cross-entropy)**

$$C(w,b) = \frac{-1}{n}\sum_x [yln(a) + (1-y)ln(1-a)] \tag{3.17}$$

- **Verossimilhança logarítmica (log-likelihood)**

$$C(w,b) = \frac{-1}{n}\sum_x ln(a_y^L) \tag{3.18}$$

onde w representa todos os pesos da rede, b todas as polarizações, n o número total de exemplares de treinamento, a é o vetor de saída (ativações) da rede quando o exemplar x é posto na entrada, y é o vetor de saída dos valores esperados da rede quando o exemplar x é posto na entrada e o somatório é sobre todos os exemplares de treinamento. Naturalmente os valores das ativações $a(x,w,b)$ são funções do exemplar apresentado, dos pesos e das polarizações. Da mesma maneira, o vetor de saída dos valores esperados da rede $y(x)$ é função do exemplar apresentado. Essas dependências foram suprimidas das equações 3.16, 3.17 e 3.18 com o único intuito de simplificar a notação.

3.6 RETROPROPAGAÇÃO

Retropropagação é um algoritmo utilizado para calcular os gradientes da função custo em relação aos pesos e polarizações. Apesar de ter sido apresentado originalmente na década de 70, foi no artigo *"Learning representations by back-propagating errors"* publicado em 1986 por David Rumelhart, Geoffrey Hinton e Ronald Williams que ele foi devidamente apreciado pela comunidade científica (NIELSEN, 2015). Neste artigo, os autores mostraram que a retropropagação funcionava muito mais rápido que os outros algorítmos da época, tornando possível redes neurais resolverem problemas que eram até então insolucionáveis.

Para que se possa usar a retropropagação, a função custo deve cumprir dois requisitos:

1. Deve poder ser escrita como uma média de funções custo para exemplares individuais de treinamento x, ou seja:

$$C = \frac{1}{n} \sum_x C_x \tag{3.19}$$

   onde n é o número total de exemplares de treinamento. O motivo deste requisito é que a retropropagação calcula $\partial C_x/\partial w$ e $\partial C_x/\partial b$ para exemplares individuais. Caso esse requisito seja cumprido, é possível recuperar $\partial C/\partial w$ e $\partial C/\partial b$ a partir de $\partial C_x/\partial w$ e $\partial C_x/\partial b$.

2. Deve poder ser escrita como uma função das saídas da rede, ou seja, das ativações dos neurônios da última camada L.

$$C = C(a^L) \tag{3.20}$$

O algoritmo é regido por quatro equações. A equação 3.21 calcula os erros δ_j^L para cada neurônio j da última camada L a partir das derivadas parciais da função custo em relação às ativações destes neurônios a_j^L e da derivada f' da função de ativação em relação às entradas ponderadas z_j^L destes mesmos neurônios.

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} f'(z_j^L) \tag{3.21}$$

De posse dos erros dos neurônios da última camada, é possível achar os erros dos neurônios de todas as camadas anteriores, uma a uma, utilizando a equação 3.22 tantas vezes quanto forem necessárias. Determina-se o erro δ^l da camada l a partir do erro da

camada seguinte δ^{l+1}, da transposta dos pesos da camada seguinte $(w^{l+1})^T$ e da derivada da função de ativação f' em relação às entradas ponderadas na camada l.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^l) \tag{3.22}$$

O operador $\odot$ é utilizado para representar o produto de Hadamard, cuja operação multiplica dois vetores elemento a elemento e retorna um vetor com a mesma dimensão das entradas. A equação 3.23 exemplifica essa operação em duas matrizes 3x3.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \odot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & a_{13}b_{13} \\ a_{21}b_{21} & a_{22}b_{22} & a_{23}b_{23} \\ a_{31}b_{31} & a_{32}b_{32} & a_{33}b_{33} \end{bmatrix} \tag{3.23}$$

Por fim, alcança-se o objetivo do algoritmo de retropropagação, o qual é achar os gradientes da função custo em relação aos pesos e polarizações, utilizando-se as equações 3.24 e 3.25.

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{3.24}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{3.25}$$

3.7 GRADIENTE DESCENDENTE

Gradiente Descendente é um algoritmo utilizado para problemas de minimização. Em se tratando de treinamento de redes neurais, é bastante comum encontrar trabalhos que utilizem ele ou adaptações dele. O seu objetivo é achar um conjunto de pesos e polarizações que façam com que o custo seja o menor possível.

Como o próprio nome sugere, a forma como ele minimiza a função custo é atualizando os parâmetros no sentido contrário do vetor gradiente, ou seja, das derivadas parciais da função custo em relação aos parâmetros que se quer otimizar. Os pesos e polarizações são, portanto, atualizados conforme as equações 3.26 e 3.27, onde a taxa de aprendizado η é um hiperparâmetro que deve ser ajustado durante a fase de treinamento.

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{3.26}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l} \tag{3.27}$$

Um problema que ocorre ao se usar Gradiente Descendente em bases de dados muito grandes é que a função custo C é calculada como uma média das funções custo C_x para exemplares individuais de treinamento x. Logo, seria preciso calcular todos os gradientes ∇C_x para então se obter ∇C, o que na prática pode retardar em muito o procedimento de treinamento.

Esse problema é solucionado utilizando-se somente alguns exemplares de ∇C_x para se estimar o gradiente real ∇C e então atualizar os pesos e polarizações. À essa abordagem dá-se o nome de Gradiente Descendente Estocástico. Para uma batelada de m exemplares de treinamento escolhidos aleatoriamente, espera-se que a média dos seus gradientes individuais ∇C_x se aproximem do real ∇C quanto maior for o número de amostras m.

O procedimento para o treinamento utilizando Gradiente Descendente Estocástico é feito primeiramente embaralhando todos os exemplares de treinamento para que eles sejam selecionados aleatoriamente. Em seguida, divide-se todo o conjunto de treinamento em bateladas de tamanho m. Então, estima-se a função custo real calculando-se a média das funções custo para os exemplares $X_1, X_2, ..., X_m$ da batelada em questão e atualiza-se a rede conforme as equações 3.28 e 3.29.

$$w_k \to w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{3.28}$$

$$b_l \to b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l} \tag{3.29}$$

O número de exemplares de treinamento por batelada é um hiperparâmetro da rede e deve ser avaliado durante o processo de treinamento.

4 REDES NEURAIS EM FPGA

A primeira tentativa bem-sucedida de implementar uma rede neural artificial em um FPGA aconteceu no início dos anos 90 (a Xilinx apresentou o primeiro FPGA comercial do mundo, o XC2064, em 1985) (ZHU; SUTTON, 2003). Desde então, muitas abordagens e taxonomias diferentes foram experimentadas e publicadas.

Apesar dos avanços das tecnologias de fabricação e do aumento da densidade dos dispositivos FPGA, ainda hoje a grande quantidade de multiplicadores limita o tamanho da rede que pode ser implementada em um único FPGA. Além disso, o número de interconexões pode se tornar bastante grande com o aumento do número de neurônios. Mais precisamente, a complexidade da matriz de interconexões cresce exponencialmente com o número de neurônios, tornando o roteamento de interconexões outro fator limitante em redes muito grandes (NOORY; GROZA, 2003).

Mesmo com esses desafios, dispositivos FPGA são excelentes para implementar redes neurais artificiais pois combinam três características bastante desejáveis: paralelismo, modularidade e adaptação dinâmica. Dado que elas são redes paralelas e distribuídas de unidades simples de processamento não-linear interconectadas em um arranjo em camadas (ZHU; SUTTON, 2003), pode-se explorar ambos o processamento concorrente do FPGA, não oferecido por um Processador de Sinais Digitais (DSP), e reconfiguração rápida e de baixo custo, não oferecido por um Circuito Integrado de Aplicação Específica (ASIC) (YOUSSEF; MOHAMMED; NASAR, 2014) (HIMAVATHI; ANITHA; MUTHURAMALINGAM, 2007) (NOORY; GROZA, 2003).

4.1 TIPOS DE TREINAMENTO

O treinamento refere-se ao procedimento em que as regras de aprendizagem são utilizadas para ajustar os pesos. Em se tratando de aplicações em FPGA, o treinamento pode ser:

Offline - o treinamento é dito *offline* quando acontece em um computador e somente após o seu fim os pesos e polarizações são transferidos para o FPGA. É a abordagem mais comum de ser encontrada na literatura pois o *hardware* necessário para realizar somente o passe direto é consideravelmente menor e mais simples.

Online - o treinamento é dito *online* quando acontece dentro do FPGA, ou seja, o próprio circuito é responsável pelo processo de atualização dos pesos e polarizações. O treinamento *online* requer uma quantidade expressiva de *hardware* adicional.

4.2 SISTEMA DE REPRESENTAÇÃO DE DADOS

O sistema de representação de dados é uma decisão de projeto que tem impacto direto tanto sobre a precisão das saídas quanto no tamanho do circuito final. Há a princípio dois sistemas de representação de dados mais utilizados: ponto fixo e ponto flutuante.

A representação em ponto flutuante oferece uma faixa maior de precisão dinâmica, sendo portanto adequado para qualquer tipo de aplicação (CAVUSLU; KARAKUZU; SAHIN, 2006). Por outro lado, a representação em ponto fixo é muito mais eficiente: uma representação compacta de pesos e dados em ponto fixo pode reduzir significativamente o *footprint* de memória e recursos computacionais utilizados (QIU et al., 2016).

Sabe-se que o efeito da precisão dos números na fase de treinamento é grande, por isso é fundamental que durante essa fase a precisão dos números seja a mais alta possível (CAVUSLU; KARAKUZU; SAHIN, 2006). Quando o treinamento é feito *online*, ou seja, no próprio FPGA, pode ser desejável que a implementação seja em ponto flutuante.

Contudo, quando o treinamento é feito *offline* em um computador, já em ponto flutuante, e no final tem-se conhecimento da faixa de valores que os pesos e polarizações podem assumir, pode-se optar por uma implementação em ponto fixo. Evidentemente, ao realizar essas conversões de ponto flutuante para inteiro, os valores são degradados devido aos truncamentos. Ainda, truncamentos realizados em cascata dentro da rede também podem vir a comprometer sua saída: daí a importância de escolher corretamente a quantidade de bits para a representação das partes inteira e fracionária.

O grande desafio para o projetista é que não é evidente quais valores escolher. Uma possível abordagem é definir o erro máximo que se quer nas saídas e reduzir os parâmetros até que se chegue a esse valor. Alternativamente, pode-se simular o passe direto da descrição e comparar o número de classificações corretas com o resultado em *software*. Ambas abordagens permitem, via tentativa e erro, reduzir o *hardware* até que se perceba uma degradação na performance.

4.3 FUNÇÃO DE ATIVAÇÃO

Na seção 3.2 foram apresentadas as funções de ativações mais encontradas na literatura. Um olhar rápido sobre os gráficos da Figura 2 é o suficiente para perceber que a complexidade do projeto da implementação de função de ativação em *hardware* é altamente dependente da função que se escolhe.

Por exemplo, para a função degrau, basta um comparador para o *bit* de sinal de entrada para determinar a saída. Para a função linear, basta um multiplicador para o coeficiente angular e opcionalmente um somador para o coeficiente linear caso este seja diferente de zero. Via de regra, qualquer função composta por retas (Degrau, Linear,

Sigmóide *Hard*, ReLu e *Leaked* ReLU) pode ser implementada de forma bastante simples. Aproximar uma função de ativação naturalmente curvilínea por meio de retas, técnica conhecida como aproximação linear por partes, já foi utilizada em trabalhos como (OMONDI; RAJAPAKSE, 2006), (ZHU; SUTTON, 2003) e (CHEN; PLESSIS, 2002).

Para funções com contornos mais suaves, a exemplo da Sigmóide, Tanh, Exponencial e suas variações, a implementação de blocos para realizar os cálculos se torna bastante complexa e custosa.

Himavathi, Anitha e Muthuramalingam (2007) realizaram a implementação de um neurônio de três entradas para as funções de ativação Sigmóide e Tanh buscando comparar as diferenças entre ter blocos para calcular a função desejada e ter os resultados já calculados armazenados em tabelas ou *Look-Up-Tables* (LUTs). Esse experimento mostrou que é possível se obter uma economia superior a 70% tanto em recursos de *hardware* quanto em tempo de execução ao se implementar a função de ativação com LUTs. A título de exemplo, os resultados do artigo foram replicados na Tabela 1.

Função de Ativação		Tipo de Implementação		Economia
		LUT	Blocos calculando	%
Sigmóide	Slices	281	953	70,50
	Tempo (us)	5	24,2	79,33
Tanh	Slices	282	1096	74,27
	Tempo (us)	5	29,8	83,22

Tabela 1 – Comparação de implementações de funções de ativação utilizando LUTs e blocos para realizar os cálculos.

5 ARQUITETURA PROPOSTA

A arquitetura proposta neste trabalho prevê o treinamento *offline* da rede, isto é, os pesos e polarizações de uma rede já treinada devem ser carregados no módulo. O *script* desenvolvido é responsável por ler esses valores de um arquivo e com isso gerar a rede correspondente. O sistema de representação de dados é o ponto-fixo, com partes inteira e fracionária parametrizáveis. Além disso, são parametrizáveis: a resolução ou quantidade de *bits* da entrada da rede, a resolução da LUT, ou seja, a quantidade de bits com que o conteúdo da LUT é representado, o número de camadas e o número de neurônios por camada. A função de ativação também é facilmente modificável em *script*, desde que sua imagem esteja compreendida no intervalo [-1,1], como será visto na subseção 5.2.1. Caso deseje-se trabalhar com funções de ativações cujas imagens extrapolem esse intervalo, deve-se fazer algumas alterações na arquitetura proposta.

5.1 FUNÇÃO DE ATIVAÇÃO

Foi visto na Seção 4.3 que a complexidade da implementação de uma função de ativação em *hardware* varia bastante de função para função. Funções compostas por retas são simples de se implementar, pois para cada trecho só são precisos um multiplicador por uma constante para o coeficiente angular e um somador opcional para a constante do coeficiente linear caso ele seja diferente de zero. Para funções curvilíneas, Himavathi, Anitha e Muthuramalingam (2007) mostraram por meio das funções Sigmóide e Tanh que, tanto em economia de recursos quanto em temporização, a melhor abordagem é a implementação por LUTs.

Visando ter uma arquitetura que pudesse aceitar mais de um tipo de função de ativação, optou-se para este trabalho a implementação por LUT. Apesar de funcionar com qualquer tipo de função, a abordagem apresentada é especialmente voltada para funções cuja imagem seja limitada a uma faixa de valores, como a Sigmóide, Tanh e *Softsign*. A ocorrência de saturações positiva e negativa permitem determinar valores que extrapolam os limites da LUT. Sendo assim, o procedimento geral para se obter uma LUT de tamanho mínimo para uma dada resolução, publicado em Himavathi, Anitha e Muthuramalingam (2007) e ilustrado conforme a Figura 11, é apresentado para o exemplo da função Sigmóide:

1. Seja n o número de bits e $y = f(z) = 1/(1 + e^{-z})$ a função de ativação Sigmóide.
2. Determinar a faixa de valores de entrada z para a qual a faixa da saída esteja entre 2^{-n} e $(1 - 2^{-n})$.
3. Sejam z_{min} e z_{max} os limites mínimo e máximo da entrada. Determine-os igualando-os

à função de ativação: $2^{-n} = 1/(1 + e^{-z_{max}})$ e $(1 - 2^{-n}) = 1/(1 + e^{-z_{min}})$.

$$z_{min} = -ln(2^n - 1) \; e \; z_{max} = +ln(2^n - 1) \tag{5.1}$$

4. Determinar a variação na entrada Δz que produz a variação na saída Δy no ponto de maior inclinação. No caso da Sigmóide, a maior inclinação ocorre em $z = 0$. O valor de Δz para a variação de 2^{-n} é:

$$\Delta z = ln\left(\frac{0,5 + 2^{-n}}{0,5 - 2^{-n}}\right) \tag{5.2}$$

5. O valor mínimo de entradas da LUT é dado por:

$$(LUT)_{min} = \frac{z_{max} - z_{min}}{\Delta z} \tag{5.3}$$

6. Determinar o número mínimo de bits m que pode endereçar $(LUT)_{min}$.

$$LUT = \frac{z_{max} - z_{min}}{2^m} \tag{5.4}$$

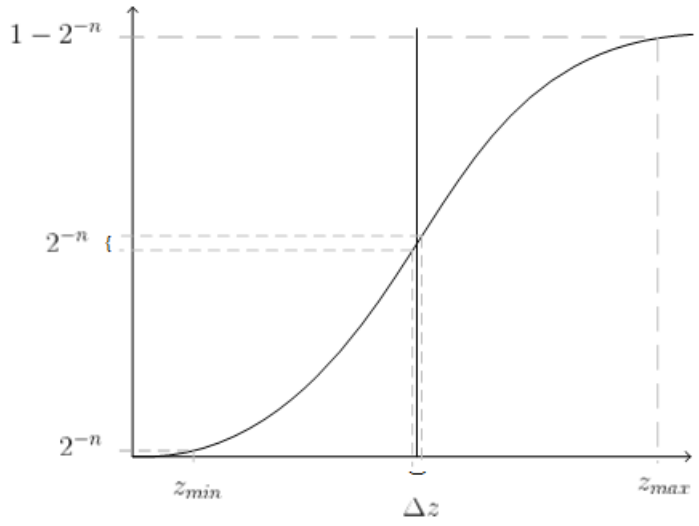


Figura 11 – Método para implementação da LUT.

Para a função Sigmóide com $n = 8$ bits de resolução, z_{min}, z_{max} e Δz são calculados a partir de 5.1 e 5.2 (vide Tabela 2), obtendo-se $-5,54$, $+5,54$ e $0,015625$ respectivamente. De 5.3 tem-se que $(LUT)_{min} = 709,28$. Portanto, são necessários $m = 10$ *bits* de endereçamento, resultando em uma LUT de 1024 entradas. Assim, considerando $\Delta z = 0,015625$, a LUT pode acomodar valores de z entre a faixa de -8 a 8 (pois $1024 * 0.015625 = 16$). e_{max} representa o erro máximo de representação na LUT, ou seja, $y_{real} - y_{binário}$, o qual nessa abordagem deve ser sempre igual ou menor que 2^{-n}.

De maneira que se possa utilizar o valor das entradas ponderadas de um determinado neurônio diretamente como o endereço da LUT, são necessários dois passos adicionais.

n	z_{min}	z_{max}	Δz	Δz_{pot2}	$(LUT)_{min}$	(LUT)	e_{max}	m
3	-1,945910	1,945910	0,510826	0,500000	7,783641	8	0,122459	3
4	-2,708050	2,708050	0,251314	0,250000	21,664402	32	0,062177	5
5	-3,433987	3,433987	0,125163	0,125000	54,943795	64	0,031209	6
6	-4,143135	4,143135	0,062520	0,062500	132,580311	256	0,015620	8
7	-4,844187	4,844187	0,031253	0,031250	310,027974	512	0,007812	9
8	-5,541264	5,541264	0,015625	0,015625	709,281734	1024	0,003906	10
9	-6,236370	6,236370	0,007813	0,007812	1596,510615	2048	0,001953	11
10	-6,930495	6,930495	0,003906	0,003906	3548,413320	4096	0,000977	12

Tabela 2 – Características da LUT em função da resolução escolhida n.

Em primeiro lugar, ao invés de Δz utiliza-se a potência de dois inferior mais próxima Δz_{pot2}. Ao se escolher um Δz_{pot2} menor que o Δz calculado, garante-se que o novo erro máximo será menor que o e_{max} previamente calculado. Somado a isso, ao restringir o valor da entrada da LUT a valores múltiplos de uma potência de dois, assegura-se que o único erro do valor armazenado para o valor real será o erro de truncamento inerente da quantidade de bits n que se escolheu para a representação.

Em segundo lugar, considerando que o acumulador que armazena as entradas ponderadas de um neurônio pode ter valor negativo e ao mesmo tempo os endereços da LUT são positivos, faz-se necessário realizar uma reordenação dos valores. A Figura 12 ilustra esse princípio: à esquerda tem-se a função Sigmóide com a abscissa em real e logo abaixo a representação desse número real em binário com sinal. À direita tem-se a função reordenada, de forma que os valores das abcissas possam ser lidos em binário sem sinal e ainda assim representar exatamente os mesmos valores da função Sigmóide.

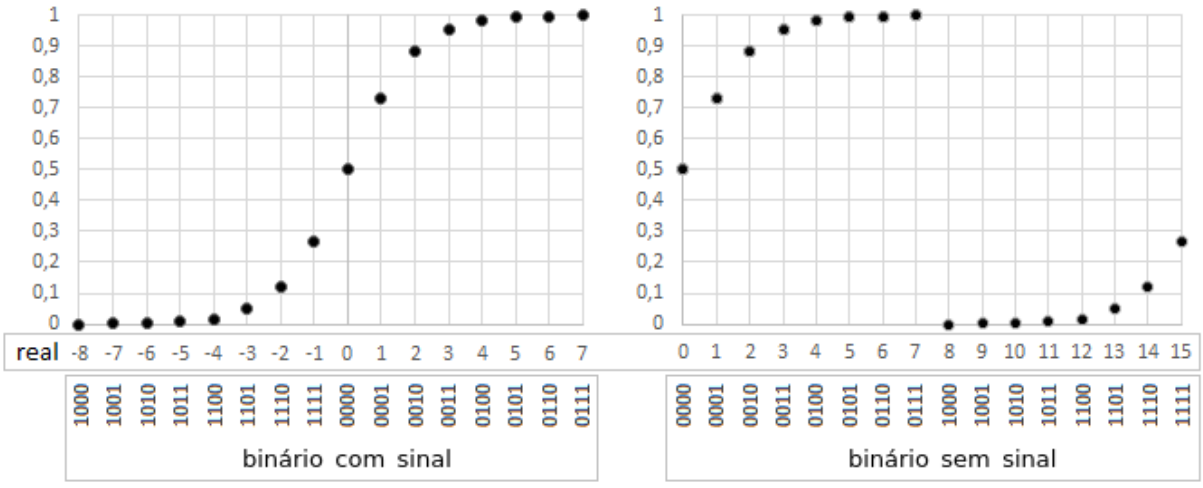


Figura 12 – Reordenação das entradas da LUT para prever abscissas negativas.

5.2 MÓDULOS

5.2.1 Multiplicador de Ponto Fixo com Sinal

O módulo do multiplicador de ponto fixo com sinal é o bloco mínimo e mais elementar do projeto. Ele é responsável por multiplicar dois números em ponto fixo com sinal com determinado número de *bits* para a parte inteira (WHOLE) e fracionária (FRACT), os quais são parâmetros do módulo, e devolver o resultado com mesmo número de *bits* das suas entradas, conforme apresentado na Figura 13.

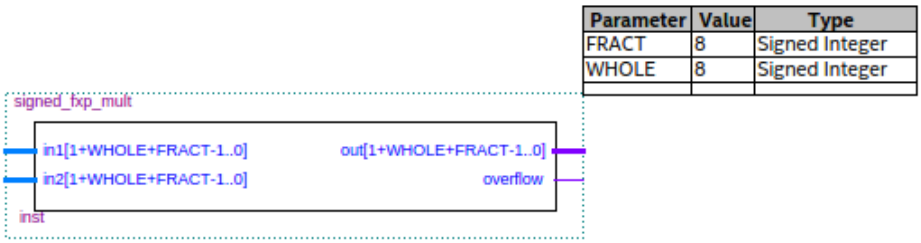


Figura 13 – Multiplicador de ponto fixo com sinal.

Em ponto fixo, o tamanho do resultado da multiplicação de dois números tem o mesmo tamanho que a soma do número de bits de suas entradas. A Figura 14 exemplifica as operações de dentro do módulo para o caso de entradas com 1 *bit* de sinal, 3 *bits* de parte inteira e 4 *bits* de parte fracionária. A saída do módulo é composta pela concatenação do *bit* mais significativo do resultado, o *bit* de sinal, com os *bits* que se deseja ao redor da vírgula - no caso desse exemplo, 3 *bits* de inteiro e 4 *bits* de fração para corresponder ao mesmo tamanho das entradas.

Para verificar se houve truncamento da parte inteira nesse processo, ou seja, se o resultado da multiplicação é inválido por não conseguir ser representado com o número de *bits* da entrada, utiliza-se o sinal out_trunc_w indicado na Figura 14. Se o resultado for negativo, ou seja, $out\_signal = 1$, diz-se que houve truncamento se houver pelo menos um *bit* em out_trunc_w que seja 0. Se o resultado $result$ for positivo, ou seja, $out\_signal = 0$, diz-se que houve truncamento se houver pelo menos um *bit* em out_trunc_w que seja 1. Em outras palavras, em complemento de dois, o valor é corrompido sempre que pelo menos um dos *bits* de inteiro desprezados for diferente do *bit* de sinal.

Pode-se pensar que seja um erro crasso projetar um multiplicador com a saída do mesmo tamanho das entradas, porém é importante lembrar que tanto as entradas da rede quanto a saída de qualquer neurônio (vide imagem da função Sigmóide) estão definidas no intervalo [0,1]. Consequentemente, uma das entradas do multiplicador será sempre menor ou igual a 1 e o valor da sua saída será sempre menor ou igual à outra entrada. Caso deseje-se utilizar uma função de ativação com imagem fora do intervalo [-1,1], deve-se alterar o multiplicador para permitir mais *bits* na saída.

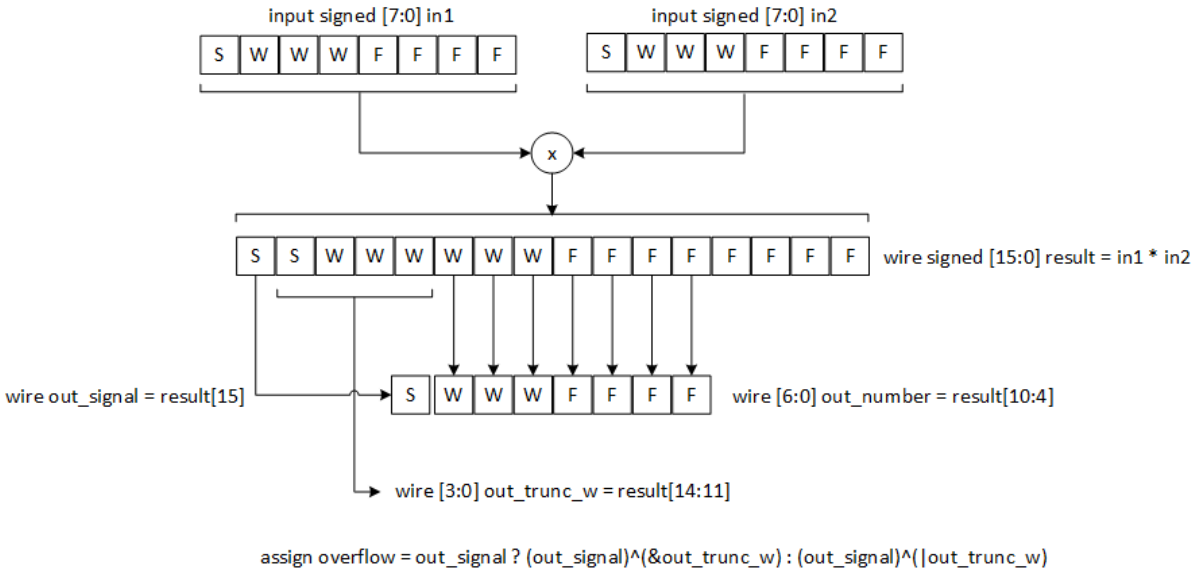


Figura 14 – Multiplicação em ponto fixo para entradas com sinal, 3 bits de parte inteira e 4 de fracionária.

5.2.2 Layer

O módulo Layer contém conceitualmente todos os neurônios de uma dada camada. Conceitualmente, pois os elementos foram arranjados um pouco diferente do que a teoria nos induz a pensar. Em rigor, o módulo contém todos os pesos e polarizações da camada armazenados em fios, módulos multiplicadores, registradores acumuladores e uma FSM para coordenar as entradas dos multiplicadores. A LUT contendo a função de ativação não faz parte do módulo, mas fica um nível acima (vide subseção 5.2.4) para que todas as camadas possam acessá-la. Dessa maneira, os cálculos do elemento neurônio não estão concentrados em um único módulo. A entrada da camada (que pode ser a entrada da rede no caso da camada de entrada ou a saída da camada anterior no caso de qualquer outra camada) são apresentadas ao módulo serialmente.

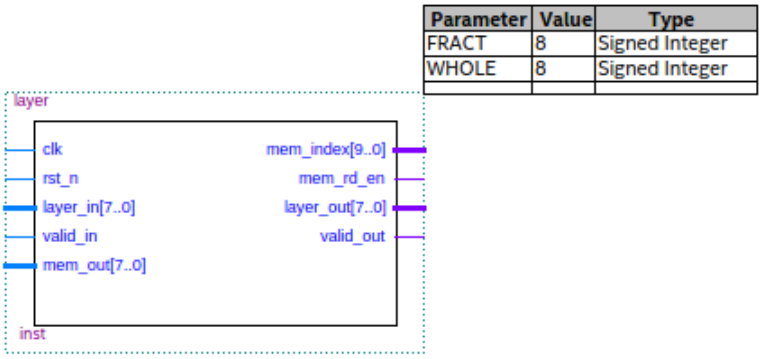


Figura 15 – Módulo Layer.

Para o projeto do cálculo do neurônio existem diversas abordagens, cabendo ao projetista escolher priorizar o tempo de execução ou a quantidade de *hardware* necessária. Para um neurônio de k entradas, pode-se projetar um módulo que execute as multiplicações em 1 ciclo de *clock* utilizando k multiplicadores ou um módulo que utilize somente 1 multiplicador mas que execute-as em k ciclos de *clock*. Naturalmente, abordagens intermediárias também são possíveis.

No caso em que a rede é voltada para o processamento digital de imagens, tem-se 1 neurônio de entrada para cada *pixel* da imagem em escala de cinza ou 3 neurônios por *pixel* no caso de uma imagem colorida no espaço de cor RGB. Como resultado, até pequenas imagens necessitam de um grande número de neurônios de entrada. Por este motivo, escolheu-se projetar cada neurônio com somente 1 multiplicador e reutilizá-lo quantas vezes fossem necessárias.

Para o cálculo das entradas ponderadas de um neurônio, descrito pela Equação 3.1, projetou-se o circuito da Figura 16. Multiplexadores controlados pelo mesmo sinal de seleção coordenam os pares entrada-peso que devem ser apresentados às entradas do multiplicador. A saída deste último entrega o resultado do cálculo à um registrador, o qual é responsável por acumular os resultados até que a última entrada seja multiplicada pelo último peso.

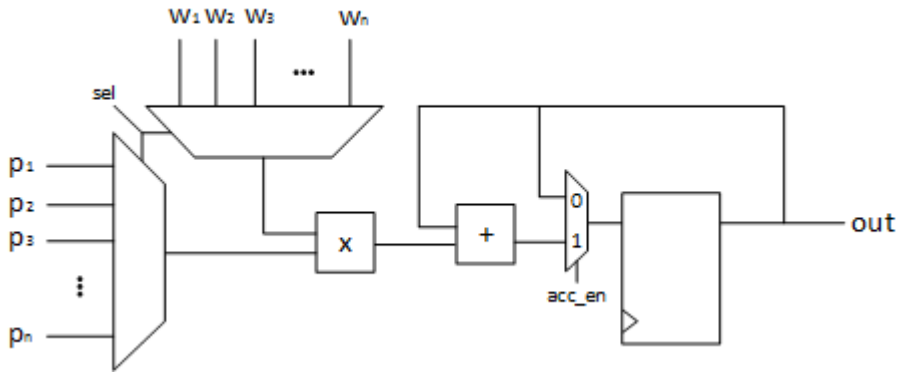


Figura 16 – Acumulador controlado por FSM.

O número de *bits* do acumulador leva em consideração o somatório da magnitude dos pesos, já que tanto as entradas da rede bem como a saída de qualquer neurônio que utilize função de ativação Sigmóide terão valores menores ou iguais a 1. Caso deseje-se outra função de ativação, deve-se atentar para o número de *bits* do acumulador a fim de evitar que o mesmo não estoure.

Para eliminar a necessidade de somar a polarização ao final do cálculo, incorpora-se à lógica do *reset* do acumulador o valor correto da polarização para aquele neurônio. Os acumuladores são resetados com a ocorrência da borda de descida do sinal de *reset*, normalmente alto, ou com a borda de subida do sinal de *valid_in* da entrada da camada.

Os sinais de seleção *sel* e *acc_en* dos multiplexadores são oriundos de uma FSM (ao contrário de *acc_en*, que existe no código, *sel* foi utilizado somente na Figura 16 para fins de entendimento. Quem faz a seleção na realidade é o estado da FSM). Cada camada tem a sua própria FSM com número de estados igual ao número de entradas dos neurônios (ou ao número de neurônios da camada anterior) mais dois: *IDLE* e *DONE*.

Quando a FSM de uma camada chega ao estado *DONE*, os valores dos acumuladores são devidamente truncados para a quantidade de *bits* a ser endereçada à LUT. Caso o valor do acumulador esteja fora do domínio abrangido pela LUT, atribui-se o valor da saturação da função de ativação. No caso da função Sigmóide, atribui-se 1 à ativação que for maior que o valor máximo do endereço da LUT e 0 à ativação que for menor que o valor mínimo.

Para saber se houve *overflow* ou *underflow* investiga-se o bit de sinal e os bits da parte inteira do acumulador que não foram utilizados para o endereçamento da LUT. Não havendo *overflow* nem *underflow*, as saídas da LUT são os valores das ativações da camada e são utilizados como os valores de entrada da camada seguinte. Caso contrário, utiliza-se o valor da saturação.

Para a implementação da LUT contendo os valores da função de ativação, utilizou-se uma memória ROM *"on chip"* proveniente da biblioteca de IPs do Quartus Prime. O conteúdo dessa memória é definido por um arquivo de inicialização de memória *.mif*, gerado pelo *script* gerador junto com os demais arquivos. Quando os acumuladores terminam os seus cálculos, os pesos ponderados são apresentados serialmente à ROM, coordenados por outra FSM, obtendo-se assim os valores das ativações. Essa LUT se localiza fora do módulo Layer, de forma que todas as camadas possam acessá-la, dado que somente uma a acessará por vez.

O *valid_out* de uma camada acontece quando a FSM da camada chega ao estado DONE, servindo como *valid_in* para a camada subsequente.

5.2.3 Hardmax

Em problemas de classificação, ao apresentar um exemplar à rede, ao final do passe direto é preciso verificar qual neurônio de saída possui a maior ativação. O índice desse neurônio determina a classe atribuída ao exemplar pelo classificador. Em redes rodando em *software* é bastante comum utilizar a função *Softmax* como função de ativação da camada de saída.

A função *Softmax* aplicada às entradas ponderadas do neurônio j da última camada L, z_j^L, é dada pela equação 5.5, onde e é o número neperiano. Ela pode ser interpretada como a distribuição de probabilidade sobre k diferentes saídas. Em outras palavras, a ativação a_j^L pode ser interpretada como a probabilidade estimada de que a classe correta

é a classe j. Sendo assim, a soma de todas as ativações dos neurônios de saída deve ser igual a 1.

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{5.5}$$

Comparar as estimativas de probabilidade de cada classe pode ser interessante pois quando o número de classes possíveis é muito grande, a rede pode ficar em dúvida entre duas ou três delas e acabar se decidindo pela errada. Para não penalizar esse tipo de situação da mesma forma que uma classificação completamente errada, pode-se utilizar métricas menos restritas, como classificar como acerto se a classe correta estiver dentre as cinco maiores probabilidades.

Contudo, voltando ao contexto de implementação em *hardware*, o uso de exponenciais e divisões não são apropriadas. A função *Hardmax* é análoga à *Softmax*, porém com codificação *One-Hot*. Por conseguinte, ela atribui 1 à ativação do neurônio com maior valor de entradas ponderadas e 0 às ativações de todos os outros neurônios de saída.

A função do modulo *Hardmax* é comparar todas as suas entradas duas a duas e indicar o índice da maior (ou seja, a classe vencedora). A sua implementação se dá em duas partes: em um primeiro momento as entradas são armazenadas serialmente em registradores. Em seguida, quando as duas primeiras entradas estiverem devidamente armazenadas, os valores dos registradores são comparados - o maior deles é armazenado no registrador *max* e o respectivo índice no registrador *index*. Assim, as entradas registradas vão sendo comparadas com *max* atual até que se chegue ao final do número de entradas.

O funcionamento é coordenado por uma FSM de tantos estados quantos forem o número de entradas mais três (para IDLE, DONE e um estado extra). Como ele possui número de entradas variável, conforme o número de neurônios de saída definido pelo usuário, o *script* gerador é responsável pela criação desse arquivo.

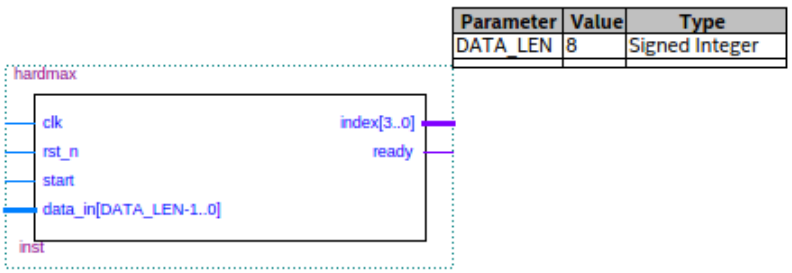


Figura 17 – Módulo Hardmax para redes com dez neurônios de saída.

Tanto as funções de ativação *Softmax* quanto *Hardmax* são funções utilizadas somente na camada de saída e o resultado de um neurônio individual depende não somente

das suas entradas ponderadas, mas das de todos os outros neurônios de saída. Por este motivo, elas não foram apresentadas na Seção 3.2.

5.2.4 Network

Como uma rede neural é um arranjo de elementos semelhantes, os neurônios, os quais podem diferir em número de entradas, função de ativação ou tipo de cálculo, mas que sempre são agrupados para formar a rede, resolveu-se seguir a abordagem *Bottom-Up* para o projeto da arquitetura.

Assim, o projeto teve início com o elemento mais básico possível: o módulo do multiplicador apresentado na subseção 5.2.1. Em seguida, projetou-se o módulo Layer para representar cada camada (ainda que ele acesse a função de ativação fora dele) e por fim, o módulo Hardmax para determinar qual neurônio obteve a maior ativação para um dado exemplar apresentado. Desse modo, módulo Network nada mais é do que um *"wrapper"* para a instanciação de tantos módulos Layer quanto forem designados, de uma ROM para representar a LUT e armazenar os valores das ativações, e por fim de um módulo Hardmax.

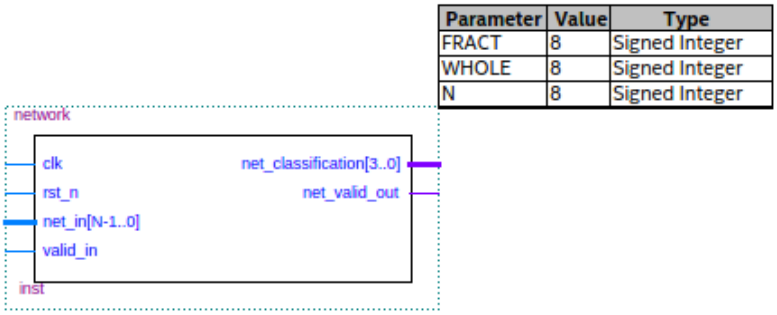


Figura 18 – Módulo Network.

Para essa arquitetura, o tempo de processamento de um passe direto em ciclos de clock é dado pela Equação 5.6.

$$clock\_cycles_{forward\_pass} = \sum neurons + 3 \cdot (\sum layers - 1) + 1 \tag{5.6}$$

A Figura 19 representa o fluxo de dados na arquitetura proposta.

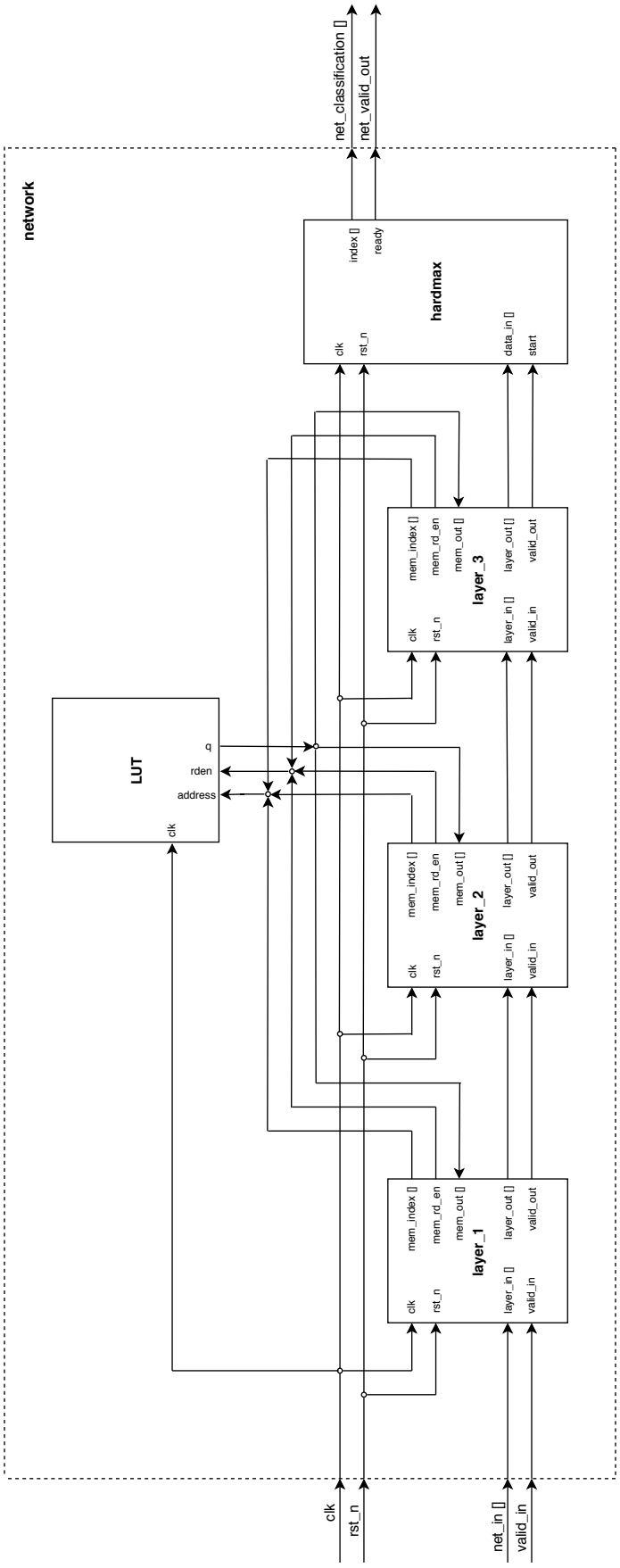


Figura 19 – Esquamático dos macroblocos da arquitetura: exemplo de rede de 3 camadas.

6 APLICAÇÃO

Como aplicação para a arquitetura proposta, é sugerido o problema de Reconhecimento Óptico de Caracteres (OCR) da base de dados MNIST. Nas próximas seções serão apresentados os passos necessários para a utilização de uma câmera como forma de fornecer dados de entrada para a rede. O conteúdo aqui apresentado é aplicável a outros trabalhos em FPGA que desejem utilizar a câmera OV7670 para alimentar uma rede neural e/ou mostrar o vídeo em um monitor VGA. Ainda, a câmera pode ser utilizada com a arquitetura deste trabalho aplicada a outras bases de dados.

6.1 BASE DE DADOS

A Base de Dados MNIST[1], ou base de dados Modificada do Instituto Nacional de Padrões e Tecnologia, é uma base bastante utilizada por pesquisadores da área de aprendizado de máquina como *benchmarking* de algoritmos classificadores. A Figura 20 apresenta alguns exemplares do MNIST.

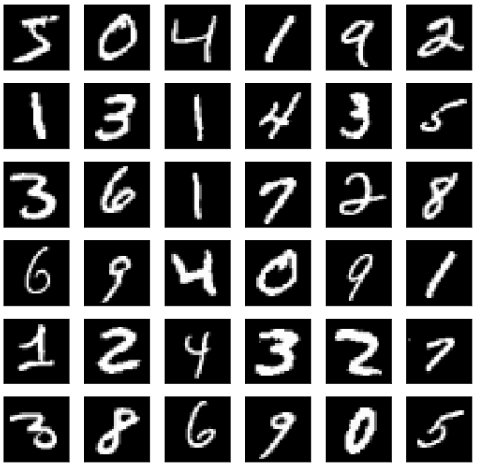


Figura 20 – Exemplos de algarismos da base de dados MNIST.

Ela foi criada a partir de duas outras bases contendo imagens binárias de algarismos escritos à mão: a NIST SD-1 e SD-3. A primeira foi coletada entre estudantes americanos do ensino médio, enquanto que a segunda entre funcionários do Departamento de Censo dos Estados Unidos. As imagens originais em preto e branco (dois níveis) do NIST foram normalizadas para enquadrarem-se em áreas de 20x20 pixels, mantendo a proporção original. Os níveis de cinza apresentados nas imagens resultantes foram resultado da técnica *anti-aliasing* usada pelo algoritmo de normalização. Em seguida, baseado em seus centros de massa, as imagens foram centralizadas em uma área de 28x28 *pixels*.

[1] http://yann.lecun.com/exdb/mnist/

O conjunto de treinamento MNIST é composto por 30.000 exemplares do SD-3 e 30.000 do SD-1. O conjunto de teste é composto por 5.000 exemplares do SD-3 e 5.000 do SD-1. Os 60.000 exemplares do conjunto de treinamento são oriundos de aproximadamente 250 pessoas.

Este trabalho utilizou o conjunto de teste original com 10.000 exemplares. Os 60.000 exemplares do conjunto de treinamento original foram divididos em 10.000 para validação e os 50.000 restantes para o novo conjunto de treinamento.

6.2 PROJETO DOS MÓDULOS

6.2.1 Controlador VGA

A etapa de mostrar os dados em um monitor é importante para que se possa verificar visualmente se os *pixels* advindos da câmera estão sendo corretamente subamostrados e, portanto, se a rede neural está sendo alimentada com dados coerentes.

Video Graphics Array (VGA) é um padrão de exibição introduzido pela IBM em 1987. O kit de desenvolvimento ALTERA DE2-115 conta com um circuito específico para a geração dos sinais necessários para a exibição em um monitor VGA. O referido circuito, mostrado na Figura 21, é composto por pinos específicos do FPGA EP4CE115F29C7N da família Cyclone IV, um Conversor Analógico-Digital (DAC) triplo de 10-bits ADV7123, usado para gerar os sinais analógicos de cada cor (embora neste kit somente os 8-bits mais significativos sejam usados), e um conector D-SUB de 15 pinos para a saída VGA.

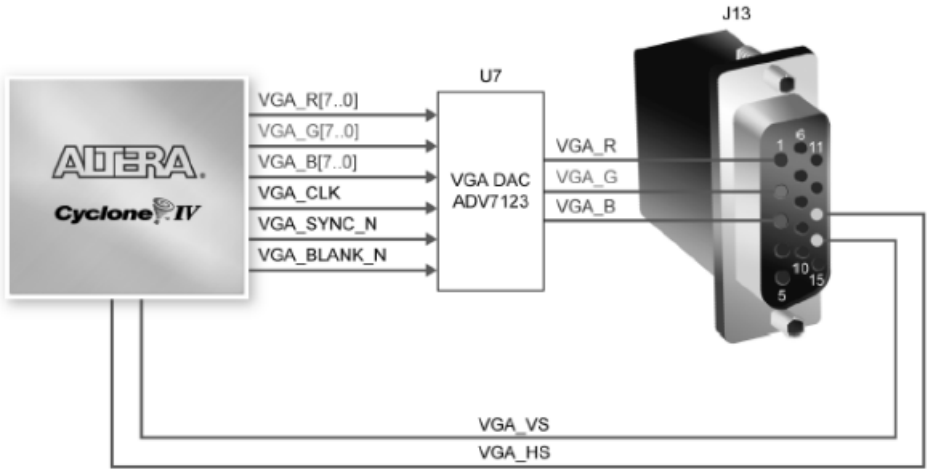


Figura 21 – Conexões entre FPGA e conector VGA.
Fonte: DE2-115 User Manual.

Do ponto de vista do FPGA, os sinais de interesse resumem-se a cinco sinais digitais: dois para sincronização, um para a horizontal e outro para a vertical, e três para definir a intensidade de cada cor em um dado *pixel* no modelo de cores RGB (vermelho, verde e azul). O princípio de funcionamento de um controlador VGA é análogo ao de monitores

de Tubo de Raios Catódicos (CRT): a tela pode ser considerada uma matriz em que os *pixels* são projetados sequencialmente em um padrão conhecido como *raster-scan*, ou seja, horizontalmente da esquerda para a direita e verticalmente de cima para baixo. A Figura 22 demonstra esse padrão.

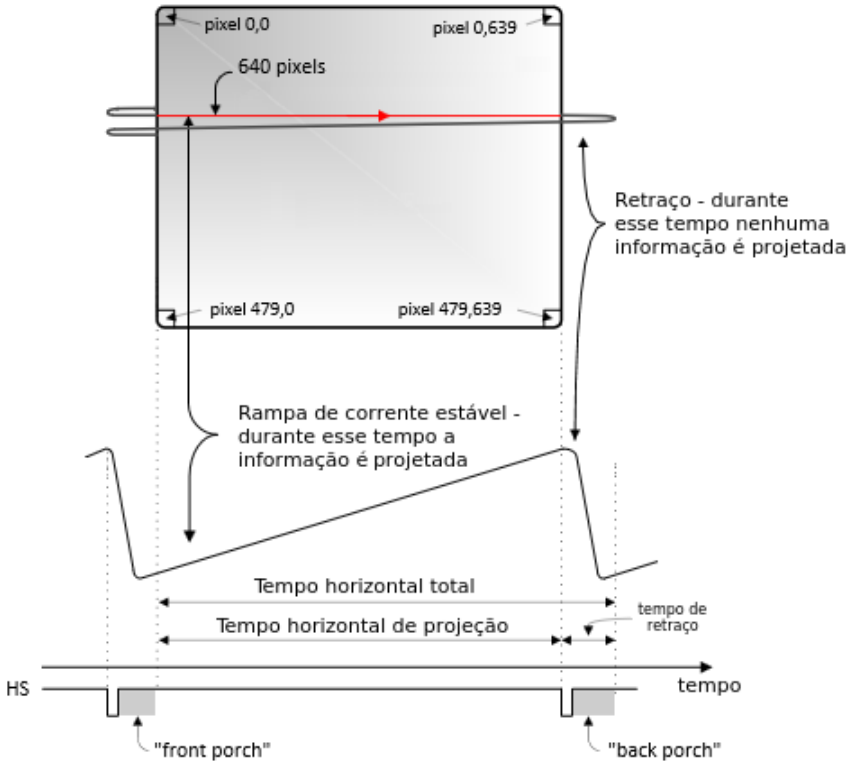


Figura 22 – VGA - Temporização horizontal.

O cursor inicia na posição (0,0) e vai projetando pixel a pixel em uma mesma linha. Quando ele chega ao término da primeira linha, posição (0,639), é preciso um tempo para que ele volte e consiga chegar ao início da linha imediatamente inferior, a posição (1,0). Esse tempo é chamado de tempo de retraço horizontal. O mesmo ocorre quando ele chega ao final da última linha, na posição (479,639), e necessita voltar para o início da primeira, momento em que ocorre o retraço vertical.

Em decorrência disso, considera-se que a tela é composta por uma área visível e uma área adicional não-visível, esta última composta por *pixels* somente para fins de temporização. Os períodos definidos como "*back porch*" e "*front porch*" horizontais são os tempos de retraço após o cursor terminar uma linha e antes começar a linha seguinte, respectivamente. O mesmo ocorre para a vertical. Assim, tem-se *pixels* adicionais não-visíveis durante os períodos de "*back porch*", "*front porch*" e sinais de sincronização, tanto para a horizontal quanto para a vertical. A duração de cada uma dessas partes é padronizada e especificada pela VESA[2] e depende da resolução e da taxa de atualização

[2] https://vesa.org/

Horizontal			Vertical		
Região	Pixels	Tempo[us]	Região	Lines	Tempo[ms]
Área visível	640	25.42	Área visível	480	l5.25
Front Porch	16	0.64	Front Porch	10	0.32
Pulso Sync	96	3.81	Pulso Sync	2	0.06
Back Porch	48	1.91	Back Porch	33	1.05
Linha completa	800	31.78	Tela completa	525	16.68

Tabela 3 – Temporização dos sinais de sincronização VGA 640x480 @60Hz.

escolhida. A configuração utilizada é para a resolução de 640x480@60Hz e a duração dos sinais é mostrada na Tabela 3.

Ao considerar todos os *pixels*, visíveis e não-visíveis, tem-se que para a resolução desejada deve-se projetar $800 \times 525 = 420.000$ *pixels* 60 vezes por segundo. Portanto, deve-se ter um sinal de *clock* de aproximadamente $420.000 \times 60 = 25,2$ MHz.

O projeto do módulo controlador baseia-se no funcionamento de dois contadores para armazenar a posição horizontal e vertical do cursor na tela. As posições são passadas para a saída juntamente com os sinais de sincronização para que o módulo que o instancie saiba quando deve ter o valor dos *pixels* disponíveis no barramento para que eles apareçam na área visível.

Ambos contadores são zerados com a borda negativa do $reset$ assim como todos os demais registradores do projeto (exceto os dos acumuladores, os quais iniciam com as suas respectivas polarizações). A posição horizontal *hpos* é incrementada a cada ciclo de *clock* até que chegue ao valor máximo da linha. Então, a posição vertical *vpos* é incrementada e *hpos* zerada para começar a contagem em uma nova linha. Quando *vpos* e *hpos* chegam aos seus valores máximos, ambos são zerados e repetem o procedimento para uma nova tela. Os contadores *vpos* e *hpos* contam tanto os pixels visíveis quanto os não visíveis.

O sinal de sincronização horizontal *hsync* permanece em nível lógico alto enquanto a posição horizontal for maior que o parâmetro H_FRONT_PORCH e menor que a soma dos parâmetros H_FRONT_PORCH e H_SYNC. Já o sinal de sincronização vertical *vsync* permanece em nível lógico alto enquanto a posição vertical for maior que o parâmetro V_FRONT_PORCH e menor que a soma dos parâmetros V_FRONT_PORCH e V_SYNC. O fluxograma dos sinais de sincronização é apresentado na Figura 23.

6.3 CÂMERA OV7670

O módulo CMOS VGA CameraChip OV7670 é, principalmente devido ao seu baixo custo, um dos dispositivos mais populares dentre projetos de eletrônica que fazem uso

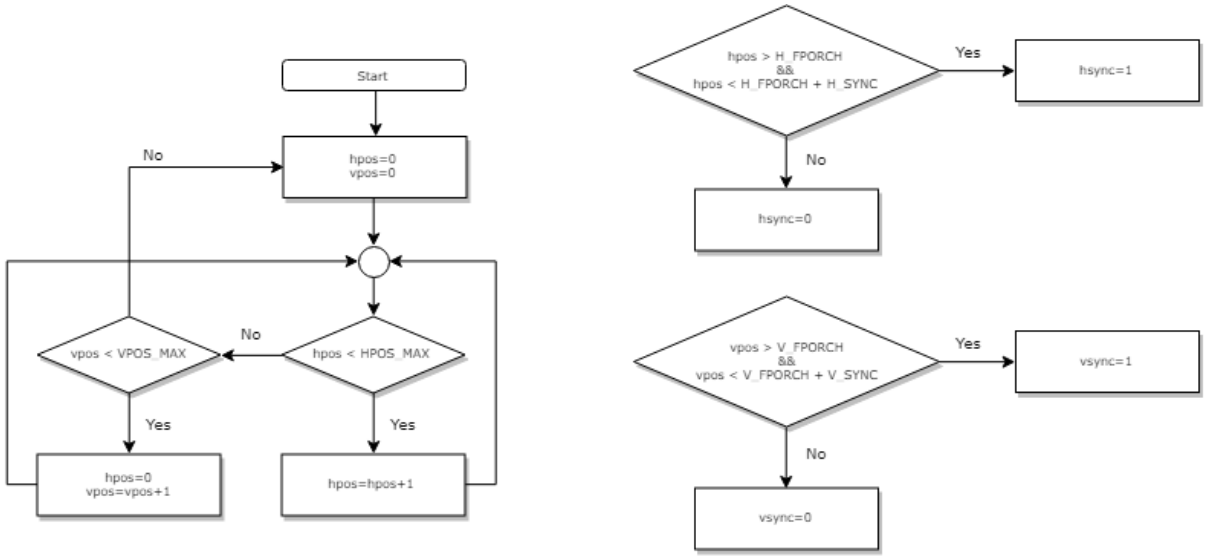
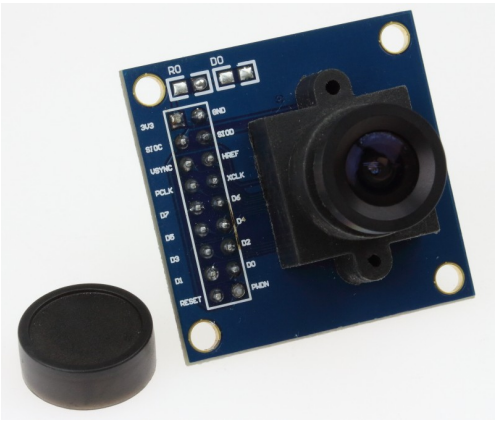


Figura 23 – Fluxograma dos sinais de sincronização *hsync* e *vsync*.



3V3	GND
SDIOC	SDIOD
VSYNC	HREF
PCLK	XCLK
D7	D6
D5	D4
D3	D2
D1	D0
RESET	PWDN

Figura 24 – Câmera OV7670.

de uma câmera. Ele é composto por uma área fotossensível, localizada na superfície da sua placa de circuito impresso e protegida do ambiente externo por uma lente rosqueada ajustável, e um circuito responsável tanto pelo processamento digital da imagem quanto pelo protocolo de comunicação. Esse módulo pode operar em até 30 quadros por segundo a uma resolução de até 640x480 *pixels*, sendo que a imagem capturada pode ser pré-processada antes de ser enviada. Embora neste trabalho essa câmera tenha sido utilizada sem nenhum tipo de configuração prévia, o pré-processamento (como controle de exposição, amplificação, balanço de branco, dentre outros) pode ser configurado via SCCB (Serial Camera Control Bus). A interface se dá por meio de uma barra de pinos no formato 9x2, conforme a Figura 24.

A descrição de cada sinal pode ser visto na Tabela 4. Antes de conectar a câmera ao kit DE2-115, deve-se atentar para o fato de que as portas de entrada e saída desse

Pino	Tipo	Descrição
3V3	Alimentação	Fonte de alimentação
GND	Alimentação	Terra
SDIOC	Entrada	Clock SCCB
SDIOD	Entrada/Saída	Dados SCCB
VSYNC	Saída	Sincronização Vertical
HREF	Saída	Sincronização Horizontal
PCLK	Saída	Clock de Pixel
XCLK	Entrada	Clock de Sistema
DO-D7	Saída	Saídas de Video
RESET	Entrada	Reset (ativo em baixo)
PWDN	Entrada	Power Down (ativo em alto)

Tabela 4 – Descrição dos sinais da câmera OV7670.

kit podem ser ajustados para 3,3V, 2,5V, 1,8V ou 1,5V por meio de um *jumper* em JP6, conforme a Figura 25. Para a interface com a câmera, deve-se ajustar para 3,3V.

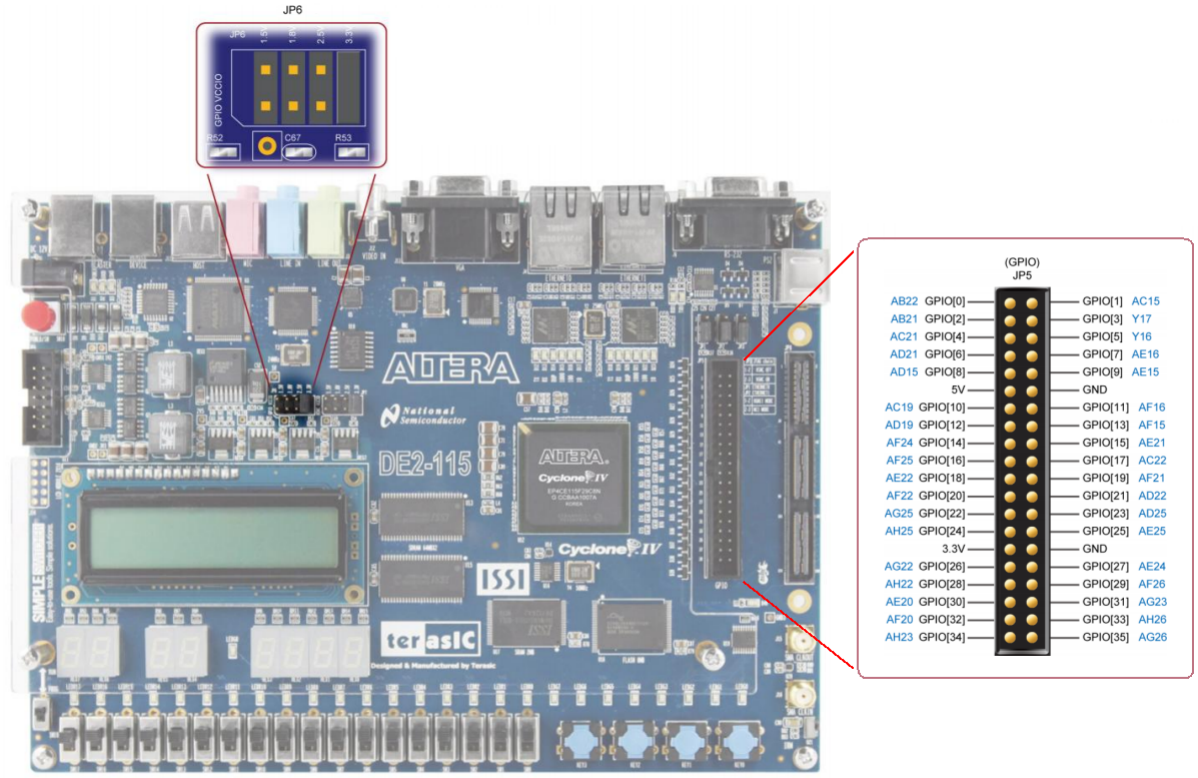


Figura 25 – Ajuste para a configuração de 3,3V nas portas de entrada e saída.

Após alimentar a câmera, basta prover um sinal de clock entre 10 e 48MHz em $XCLK$ para que apareçam dados nas saídas de interesse ($PCLK$, $VSYNC$, $HREF$ e $D0$-$D7$). O kit DE2-115 provê até três *clocks* internos de 50MHz - um deles será dividido por dois e utilizado para a entrada de $XCLK$.

De acordo com o *datasheet*, esse sinal deve ter uma frequência entre 10 e 48MHz, o que está de acordo com a frequência calculada de aproximadamente 25MHz para a resolução de 640x480@60Hz e também com o clock de 50MHz do kit, o qual pode ser facilmente dividido por dois.

A Figura 26 apresenta os sinais de sincronização horizontal, ou seja, para os *pixels* contidos em uma linha. O sinal $PCLK$ é o *clock* de saída da câmera e a referência para os demais sinais de saída. $HREF$ determina quando há dados válidos de uma linha sendo disponibilizados no barramento de dados. Assim, os *bytes* de interesse devem ser capturados sempre na borda de subida de $PCLK$ e somente quando $HREF$ estiver em nível lógico alto.

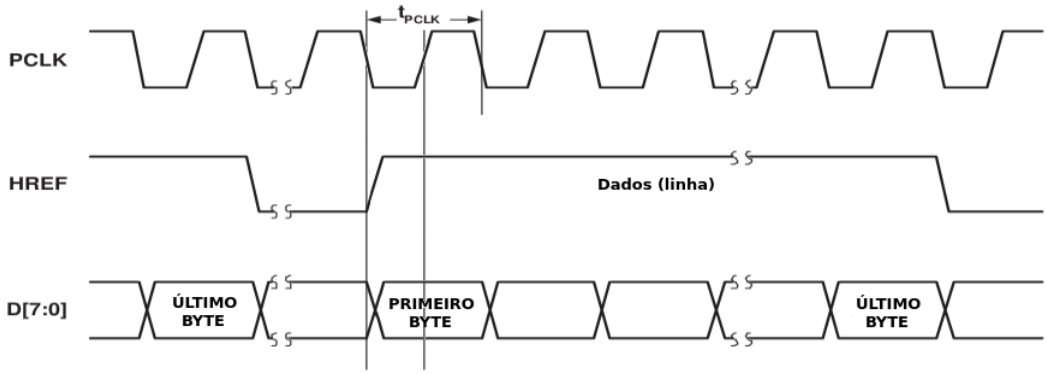


Figura 26 – Formas de onda dos sinais de sincronização da câmera (linha).

Além disso, deve-se notar também as condições dos sinais de sincronização vertical, mostrados na Figura 27. A borda de descida do sinal $VSYNC$ indica o início de uma nova tela e a borda de subida o seu fim. Portanto, os *bytes* de interesse devem ser capturados somente quando $VSYNC$ estiver em nível lógico baixo.

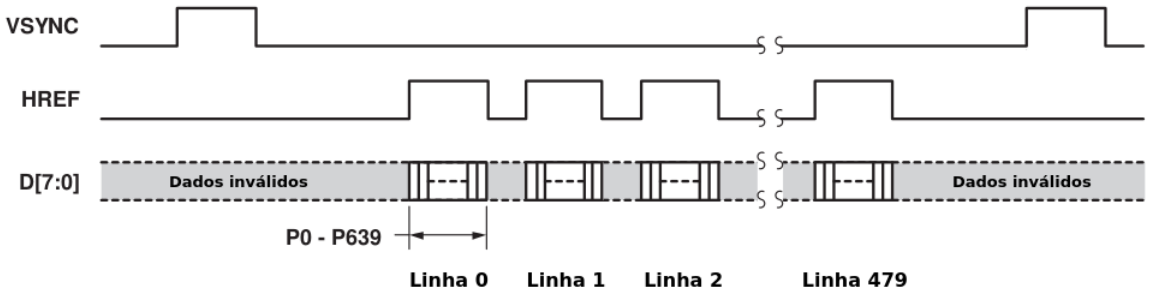


Figura 27 – Formas de onda dos sinais de sincronização da câmera (tela).

6.3.1 Espaços de Cor

Um espaço de cor é uma organização específica de cores. É possível que o espaço de cores mais conhecido seja o RGB, no qual todas as cores podem ser obtidas como uma combinação de diferentes intensidades de vermelho, verde e azul. Há, entretanto, outros

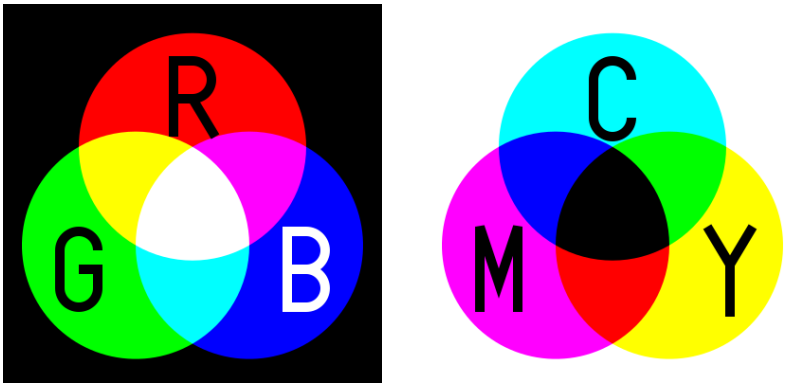


Figura 28 – Espaços de cor RGB e CMYK.

espaços talvez não tão conhecidos mas bastante utilizados no dia-a-dia como CMYK e YCbCr.

No espaço de cor CMYK, C representa a cor ciano, M magenta, Y amarelo e K preto. Esse é o espaço de cor utilizado por impressoras, no qual ao contrário do RGB, em que as diferentes intensidades das cores são adicionadas ao preto (ausência de cor) para compor a cor final, nesse espaço as cores são subtraídas do branco (combinação de todas as cores) para formar a cor final.

O espaço de cor YCbCr é o formato usado em DVDs e TV digital. Ele foi desenvolvido como parte das recomendações ITU-R BT.601 da União de Telecomunicação Internacional durante o desenvolvimento de um padrão para componentes de vídeos digitais (JACK, 2011). Nesse espaço, Y representa a luminância, ou a quantidade de luz branca na imagem, e Cb e Cr representam as crominâncias da luzes azul e vermelha, respectivamente, indicando a intensidade de cada uma dessas cores, conforme a Figura 29.

Esse espaço de cor é bastante apropriado para armazenamento e transmissão de vídeo pois o olho humano não consegue perceber tão bem a redução na resolução das crominâncias, ao contrário da luminância. Por consequência, é comum achar sistemas que adotem formatos em que as crominâncias são subamostradas visando aumentar a eficiência (reduzindo o número de bits por *pixel*) sem comprometer tanto a percepção de quem assiste. Existem vários formatos para o espaço YCbCr, tais como 4:4:4, 4:2:2, 4:1:1 e 4:2:0. A nomenclatura do formato indica quantos *bits* são utilizados para, em um *pixel*, representar a luminância Y e as crominâncias Cb e Cr, respectivamente.

A câmera OV7670 entrega por padrão *pixels* no espaço de cores YCbCr formato 4:2:2. Nesse formato, cada pixel possui o seu *byte* de luminância mas as crominâncias são repetidas para dois *pixels* adjacentes na transmissão, conforme a Figura 30. Assim, só são precisos 2 *bytes* por *pixel*. Se cada *pixel* possuísse crominâncias exclusivas, seriam precisos 3 *bytes* por *pixel*.

(a) Imagem completa (Y+Cb+Cr). (b) Luminância Y.

(c) Crominância azul Cb. (d) Crominância vermelha Cr.

Figura 29 – Decomposição de uma imagem no espaço de cor YCbCr.

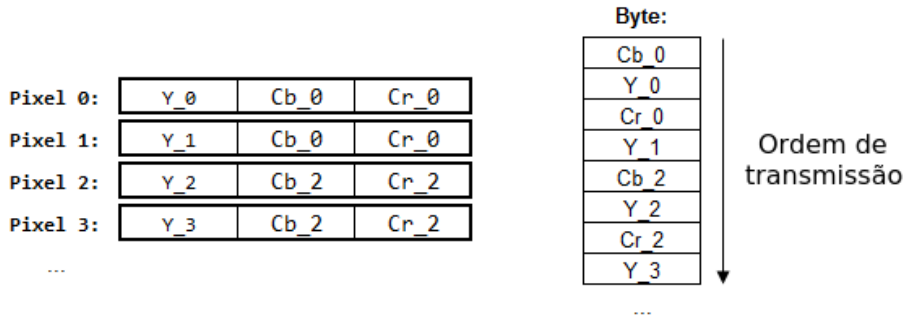


Figura 30 – Formato YCbCr422: crominâncias compartilhadas entre dois *pixels*.

Esse formato padrão é oportuno pois dessa maneira pode-se capturar o vídeo em preto e branco sem que seja necessário nenhum pré-processamento, bastando para isso sempre capturar um *byte* e deprezar o imediatamente seguinte. Caso o espaço de cor fosse RGB, por exemplo, para ter-se a imagem em preto e branco seria preciso capturar as 3 componentes e tirar a média delas.

A parte responsável pela lógica de captura dos dados da câmera (representada por interface da câmera na Figura 31) está atrelada ao gerenciamento da memória onde esse

conteúdo será armazenado. Entretanto, desconsiderando essa parte que será tratada logo a seguir na subseção 6.3.2, a captura é feita simplificadamente da seguinte forma: tem-se um bloco *always* sensível ao $PCLK$ da câmera e ao *reset* do sistema, em que sempre que $VSYNC$ e $HREF$ da câmera estiverem em nível lógico baixo e alto respectivamente, em um momento o byte será capturado e no outro será descartado. Dessa maneira garante-se a captura de somente os *bytes* de luminância.

6.3.2 Interface da Câmera

Quando se trabalha com múltiplos domínios de *clock* é preciso que haja um elemento intermediário para que se evite o problema da metaestabilidade. A melhor maneira de se tratar esse problema para vetores de dados que mudam continuamente é a sincronização por pilha FIFO assíncrona.

Nessa aplicação, em que o objetivo é filmar o algarismo e obter a classificação, o processo manual de segmentação é lento em comparação à taxa de atualização de captura da câmera (30Hz). Como resultado, há pouca mudança nos *pixels* entre quadros subsequentes. Portanto, não há prejuízo para a aplicação nos casos em que a pilha é lida em uma frequência maior do que é escrita ou vice-versa. Sem o controle de pilha vazia ou cheia, a funcionalidade pode ser simplificada por uma *dual port* RAM, conforme a Figura 31.

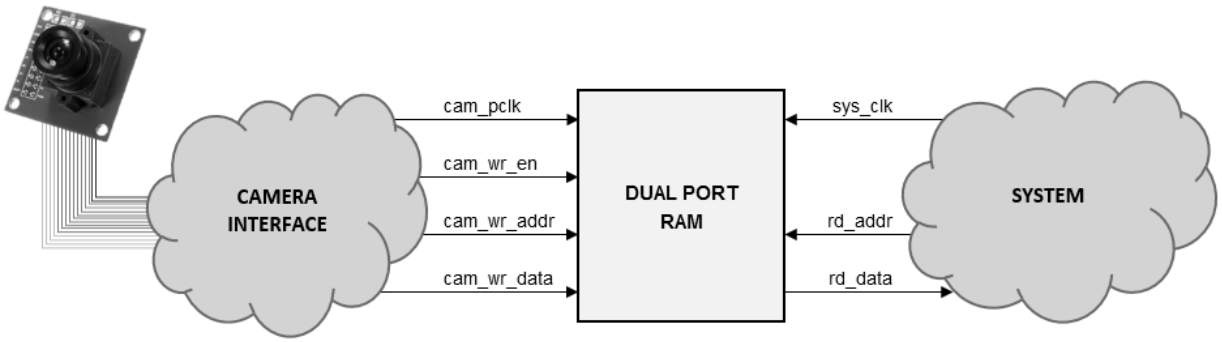


Figura 31 – *Dual Port* RAM.

Havendo a *dual port* RAM como intermediária, de um lado tem-se os dados sendo escritos conforme o *clock* e demais sinais de sincronização da câmera, e do outro, todo o sistema consumindo esses dados conforme seu próprio sinal de *clock*. No caso da utilização desse conteúdo de memória para a projeção em um monitor VGA, um cuidado adicional deve ser tomado para que o endereço de leitura só seja incrementado quando o pixel estiver em uma área visível. Adicionalmente, o endereço de leitura deve ser zerado sempre que o pixel estiver na posição visível (0,0), ou seja, no início da projeção de uma nova tela. Por fim, deve-se notar que o número de *pixels* da área visível que se quer projetar deve corresponder ao número exato de endereços da memória.

7 RESULTADOS E DISCUSSÃO

O fluxo de projeto para a utilização de uma rede com a arquitetura proposta por este trabalho se dá conforme a Figura 32.

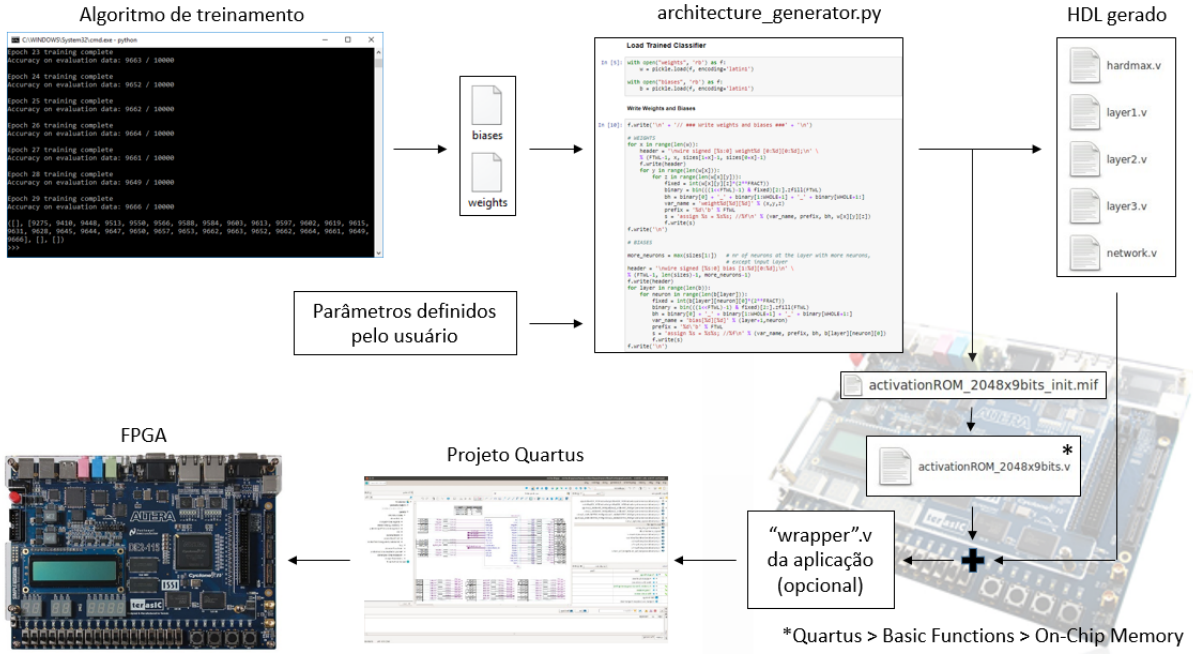


Figura 32 – Fluxo de projeto.

Em primeiro lugar, a rede deve ser treinada com o algorítmo de treinamento desejado para se obter os pesos e polarizações. Embora o *script* gerador da rede (indicado na Figura 32 por *architecture_generator.py*) tenha sido escrito para carregar os pesos e polarizações gerados pelo algorítmo de Nielsen (2015), com muito pouco esforço é possível adaptar o carregamento desses parâmetros caso o usuário faça o treinamento de outra maneira.

Em seguida, o usuário deve alimentar o *script* gerador com os parâmetros de decisão de projeto: número de camadas, número de neurônios por camada, número de bits da parte inteira e fracionária e o número de bits de resolução da LUT. Como será na Seção 7.1, a escolha desses parâmetros não é evidente e deve ser validada em *testbench* para garantir que a rede terá recursos suficientes de *hardware* para se comportar como desejado.

A saída do *script* gerador são os módulos em Verilog: *network.v*, o qual contém instâncias de *hardmax.v* e tantos *layer*.v* (sendo * o número da camada) quantos forem especificados, os quais por sua vez possuem instâncias de multiplicadores de ponto-fixo *fxp_mult.v*. Além disso, é gerado um arquivo de inicialização de memória com os

valores reordenados das ativações, conforme foi visto na Seção 5.1. A memória deve ser criada no Quartus em *Basic Functions > On-Chip-Memory* com nome no formato *activationROM_*x*bits.v*, conforme o arquivo de inicialização. Alternativamente, o usuário pode utilizar as memórias já criadas, localizadas no repositório em *src/modules/memories/*.

Os códigos fonte e todo o material desenvolvido no contexto deste trabalho estão disponíveis em um repositório do domínio *Bitbucket* [1], o qual contém todos os arquivos necessários para a reprodutibilidade dos resultados aqui apresentados e que pode ser utilizado para o desenvolvimento de trabalhos futuros.

A depender da aplicação do usuário, pode ser necessário escrever um arquivo "wrapper"para reunir tudo o que for necessário para a aplicação: uma instância da rede e módulos auxiliares, como o driver de uma câmera para alimentá-la. Por fim, cria-se um projeto Quartus para que se possa compilar o *design* e gravar no dispositivo FPGA desejado.

7.1 ESCOLHA DOS PARÂMETROS DE DECISÃO DE PROJETO

Para exemplificar a escolha dos parâmetros de decisão de projeto, quatro redes com diferentes configurações foram escolhidas para se coletar dados de simulação. Elas foram treinadas em *software* (Python) com o código disponibilizado por Nielsen (2015), autor cujo o livro foi a inspiração para este trabalho. O algorítmo de aprendizado utilizado foi o gradiente descendente estocástico e função custo entropia-cruzada por 30 épocas, batelada de 10 exemplares, taxa de aprendizado de 0,5 e parâmetro de regularização 5,0.

É importante ressaltar que não foi objetivo deste trabalho otimizar a classificação das redes procurando os melhores hiperparâmetros para cada arquitetura, mas sim garantir que a rede projetada em *hardware* se comportasse exatamente como a de *software* com a menor quantidade de recursos possível.

A Tabela 5 apresenta cada uma das redes e suas respectivas classificações corretas em *software*, ou seja, o número almejado para a classificação na simulação do *hardware*, considerando o conjunto de teste de dez mil exemplares.

De posse do número de classificações corretas para cada uma das arquiteturas, o *testbench* consistiu em apresentar cada uma das 10.000 imagens do conjunto de teste e comparar a saída da rede com a etiqueta daquela imagem. Dada uma arquitetura, como a Rede 1, a qual tem 784 neurônios de entrada, 10 ocultos e 10 de saída, ainda é possível alterar parâmetros de decisão de projeto como o número de *bits* para as partes inteira, fracionária e de representação da LUT, de forma que cada uma das colunas das Tabelas 6, 7 e 8 são redes diferentes (arquivos gerados diferentes) de uma mesma arquitetura (mesmo número de camadas e neurônios por camada).

[1] https://bitbucket.org/henriquebaqueiro/feedforward/src/master/

	Arquitetura	**Classificação em software (test_set)**
Rede 1	784-10-10	9170/10000
Rede 2	784-30-10	9669/10000
Rede 3	784-30-30-10	9692/10000
Rede 4	784-10-10-10-10	9192/10000

Tabela 5 – Arquitetura e número de classificações corretas das quatro redes utilizadas na simulação.

A Tabela 6 mostra a variação da classificação em relação ao número de *bits* da parte inteira, mantendo-se fixos os números de *bits* da parte fracionária e de representação da LUT. Esse é o parâmetro de decisão de projeto mais fácil de se obter, dada a grande influência do erro de truncamento na parte inteira no resultado de classificação: para essa aplicação que a rede foi treinada e para essa arquitetura, a rede precisa de pelo menos 4 *bits* de inteiro.

A Tabela 7 mostra que os erros de truncamento da parte fracionária influenciam a classificação de forma muito mais branda. O mínimo para essa aplicação e arquitetura são 8 *bits*, porém números próximos disso ainda produzem números de classificações que podem ser aceitáveis a depender da aplicação.

A Tabela 8 mostra o mais sutil dos parâmetros. Considerando 8 *bits* o número mínimo, uma redução de 50% acarretaria em um aumento do erro de somente 19 em 10.000 exemplares. Isso representa trocar uma LUT de 1024x8 *bits* por outra de 32x4 *bits* a um custo de somente 0,19% do número total de exemplares.

bits inteiro	2	3	**4**	5	6	7	8
bits fração	8	8	8	8	8	8	8
bits LUT	8	8	8	8	8	8	8
Classificação	1506	1571	9170	9170	9170	9170	9170

Tabela 6 – Rede 1: influência do número de *bits* de inteiro.

bits inteiro	4	4	4	4	4	4	4
bits fração	2	3	4	5	6	7	**8**
bits LUT	8	8	8	8	8	8	8
Classificação	2031	6271	8524	9019	9131	9167	9170

Tabela 7 – Rede 1: influência do número de *bits* de fração.

As tabelas 9, 10, 11 e 12 mostram as variações dos parâmetros de decisão de projeto para a Rede 2. Mais uma vez, o número mínimo de *bits* da parte inteira é fácil de determinar ao se observar a Tabela 9.

bits inteiro	8	8	8	8	8	8	8
bits fração	8	8	8	8	8	8	8
bits LUT	4	5	6	7	**8**	9	10
Classificação	9151	9164	9167	9167	9170	9170	9170

Tabela 8 – Rede 1: influência do número de *bits* da LUT.

A Tabela 10 sugere que o número mínimo de *bits* para a parte fracionária seja 11. Porém, a Tabela 11 mostra que mesmo aumentando a resolução da LUT, o número de classificações pode chegar a 9669 mas não estabiliza, indo de encontro ao que seria esperado. Em contrapartida, com 12 *bits* para a parte fracionária, o valor das classificações estabiliza em 9669 para uma resolução de LUT de 9 *bits*, como mostra a Tabela 12.

bits inteiro	2	3	**4**	5	6	7	8
bits fração	12	12	12	12	12	12	12
bits LUT	9	9	9	9	9	9	9
Classificação	4385	3831	9669	9669	9669	9669	9669

Tabela 9 – Rede 2: influência do número de *bits* de inteiro.

bits inteiro	4	4	4	4	4	4	4
bits fração	6	7	8	9	10	**11**	**12**
bits LUT	9	9	9	9	9	9	9
Classificação	9594	9642	9658	9665	9668	9669	9669

Tabela 10 – Rede 2: influência do número de *bits* de fração.

bits inteiro	4	4	4	4	4	4	4
bits fração	11	11	11	11	11	11	11
bits LUT	4	5	6	7	8	**9**	10
Classificação	9663	9667	9671	9669	9668	9669	9668

Tabela 11 – Rede 2: influência do número de *bits* da LUT para 11 *bits* de fração.

Existe a possibilidade de que para uma rede apresentar o número exato de classificações da sua contraparte em *software* tenha que se aumentar os parâmetros em *hardware* em níveis que não compensem na prática. Quando o número de *bits* não é suficiente para representar a diferença entre dois ou mais neurônios de saída, em *hardware* eles terão o mesmo valor e a classe vencedora estará empatada com outra(s). Em caso de empate, a arquitetura do módulo *Hardmax* sempre favorece a classe de índice mais alto, o que pode ou não condizer com a classificação em *software*. Com isso, existe a possibilidade da rede em *hardware* ter um número de classificações maior (caso acerte quando deveria errar) ou

bits inteiro	4	4	4	4	4	4	4
bits fração	12	12	12	12	12	12	12
bits LUT	4	5	6	7	8	**9**	10
Classificação	9661	9671	9668	9670	9670	9669	9669

Tabela 12 – Rede 2: influência do número de *bits* da LUT para 12 *bits* de fração.

menor (caso erre quando deveria acertar) que a sua contraparte em *software*. Os erros de representação no interior da rede e a propagação desses erros nas operações através das camadas podem ser interpretados da mesma maneira: pode ser que os valores truncados reduzam o número final de classificações, mas pode ser que aumentem também.

A Tabela 13 mostra a influência do número de *bits* da parte inteira no número de classificações corretas para a Rede 3. Ao contrário das redes anteriores, desta vez só foram necessários dois *bits* de inteiro. Como o múmero de *bits* para a parte fracionária e a resolução da LUT são parâmetros menos evidentes, para esta rede variou-se ambos os parâmetros fixando-se dois *bits* para a parte inteira, conforme mostra a Tabela 14. Nela, é possível notar que as classificações estabilizam em 9693 para 13 ou mais *bits* de fração e 9 ou mais *bits* de LUT, apesar de o número de classificações de referência em *software* ser 9692.

bits inteiro	1	**2**	3	4	5	6	7
bits fração	12	12	12	12	12	12	12
bits LUT	10	10	10	10	10	10	10
Classificação	1679	9692	9692	9692	9692	9692	9692

Tabela 13 – Rede 3: influência do número de *bits* de inteiro.

Neste caso, por não haver uma configuração mínima exata, cabe ao usuário final decidir qual configuração utilizar. O objetivo do usuário final de redes em *hardware* deve ser tentar alcançar números de classificações iguais ou próximos ao esperado, garantindo o funcionamento satisfatório da sua aplicação.

Analogamente, as Tabelas 15 e 16 apresentam o número de classificações para a Rede 4. Na primeira, tem-se a influência do número de *bits* de inteiro para esta rede, enquanto que na segunda tem-se a influência do número de *bits* de fração e da LUT para 4 *bits* de inteiro. Nelas, é possível observar que o número de classificações estabiliza no número almejado para a Rede 4 para 4 *bits* de inteiro, 15 de fração e 10 de LUT.

Por fim, um comentário final sobre a métrica utilizada no *testbench* deste trabalho: em projetos de módulos que executam cálculos, onde há erros de representação e eles se propagam, é comum que se defina um erro máximo na saída e que se utilize desse número para definir se o projeto atende às especificações. No caso de redes neurais, esse tipo de

		bits fração				
		10	11	12	13	14
bits LUT	6	9675	9684	9682	9684	9684
	7	9683	9687	9690	9691	9691
	8	9687	9689	9690	9690	9691
	9	9686	9690	9691	9693	9693
	10	9686	9690	**9692**	9693	9693
	11	9686	9691	9691	9693	9693
	12	9686	9691	**9692**	9693	9693

Tabela 14 – Rede 3: influência do número de *bits* de fração e da LUT para 2 *bits* de inteiro.

bits inteiro	2	3	4	5	6	7	8
bits fração	15	15	15	15	15	15	15
bits LUT	10	10	10	10	10	10	10
Classificação	1743	1154	9192	9192	9192	9192	9192

Tabela 15 – Rede 4: influência do número de *bits* de inteiro.

		bits fração					
		13	14	15	16	17	18
bits LUT	7	9194	9193	9193	9193	9192	9192
	8	9196	9195	9194	9194	9195	9195
	9	9198	9196	9195	9195	9194	9194
	10	9193	9193	**9192**	9192	9192	9192
	11	9194	9192	9192	9192	9192	9192
	12	9193	9193	9192	9192	9192	9192

Tabela 16 – Rede 4: influência do número de *bits* de fração e da LUT para 4 *bits* de inteiro.

métrica não é muito apropriada pois a depender da aplicação ou até mesmo do treinamento da rede, 10^{-2} pode ser um número pequeno ou também 10^{-5} pode ser um número grande. Tudo isso depende do quanto esse erro vai impactar no número de classificações.

	Rede 1 784-10-10	Rede 2 784-30-10	Rede 3 784-30-30-10	Rede 4 784-10-10-10-10
bits inteiro	4	4	2	4
bits fração	8	12	13	15
bits LUT	8	9	9	10

Tabela 17 – Configuração mínima para as quatro redes.

7.2 APLICAÇÃO PROPOSTA NO FPGA

A Figura 33 mostra a Rede 3 (escolhida simplesmente por ter o maior número de classificações corretas) carregada em FPGA sendo alimentada por dados da câmera OV7670, esta última localizada no canto inferior direito da imagem em um suporte de madeira. A imagem 640x480 *pixels* em preto e branco da câmera é mostrada em um monitor. Linhas vermelhas indicam o recorte no centro da tela de 28x28 *pixels*, os quais são pré-processados (inversão de cores e aplicação de *thresholds* para melhor se assemelharem à distribuição da base de dados MNIST) e então encaminhados para a RAM responsável por alimentar a rede. O conteúdo dessa RAM é mostrado no canto inferior direito do monitor (observe que o recorte no centro da tela é um 2 preto em fundo branco, enquanto que no canto inferior direito da tela temos o mesmo 2, porém branco e com fundo preto). A classificação é mostrada no display de 7-segmentos do *kit* de desenvolvimento.

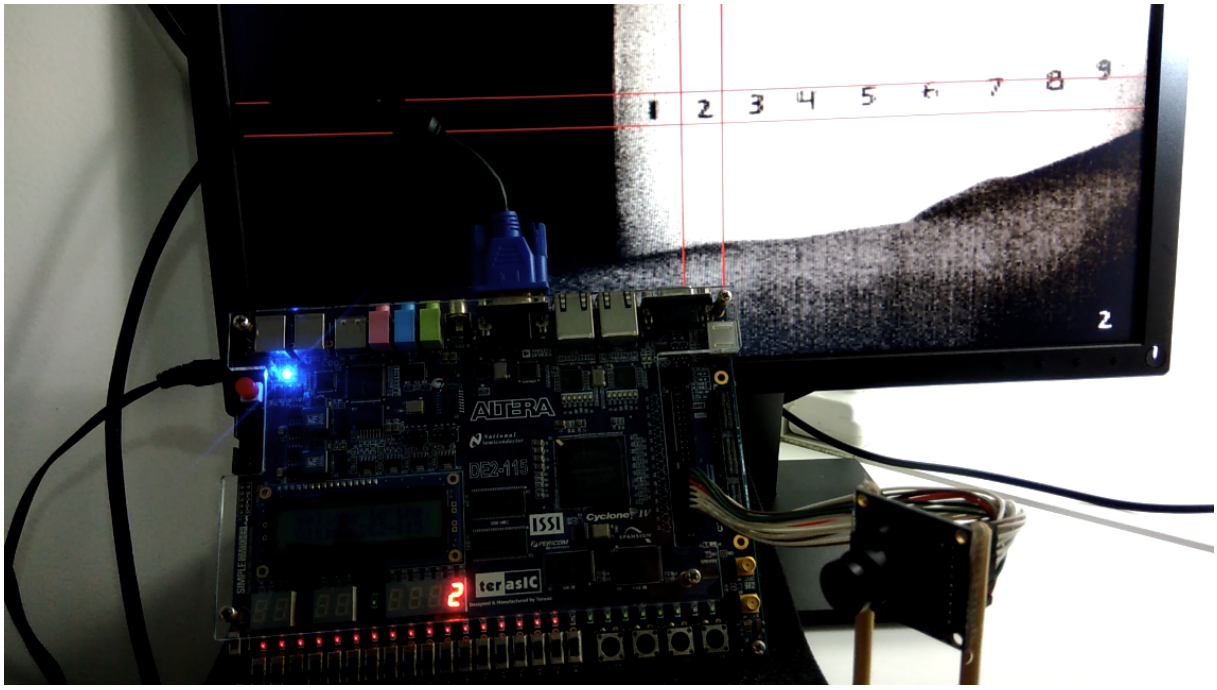


Figura 33 – Aplicação proposta no FPGA.

Os resultados de compilação podem ser vistos no Apêndice B.

8 CONCLUSÕES

Este trabalho desenvolveu uma ferramenta que gera a descrição de *hardware* em linguagem Verilog para uma rede neural do tipo *feedforward* com parâmetros variáveis (número de *bits* de entrada, número de camadas, número de neurônios por camada, número de *bits* para a parte inteira e fracionária na representação em ponto fixo e número de *bits* para a representação da *look-up table* da função de ativação).

Foi apresentada uma visão geral sobre redes neurais artificiais: a estrutura do neurônio, funções de ativação, topologias, funções custo e o algorítmo de treinamento mais conhecido, o gradiente descendente, o qual utiliza a retropropagação para achar as derivadas parciais da função custo. Foram apresentadas também as considerações que se deve ter ao implementar uma rede neural artificial em *hardware*: tipos de treinamento, sistemas de representação de dados e formas de se implementar a função de ativação. Em seguida, a arquitetura proposta foi apresentada, evidenciando cada um dos módulos constituintes e as decisões de projeto que levaram ao projeto final.

Como aplicação sugerida foi apresentado o problema do reconhecimento óptico de caracteres: uma rede neural artificial, cuja descrição de *hardware* foi gerada pelo *script* desenvolvido por esse trabalho, foi utilizada para classificar em tempo real imagens de algarismos de zero a nove, provenientes de uma câmera. Essas imagens, após um pré-processamento para se assemelharem com os exemplares do banco de dados MNIST (imagens brancas com fundo preto), foram satisfatoriamente classificadas, sendo a classificação exposta em um display de 7-segmentos do *kit* de desenvolvimento DE2-115 da antiga Altera, hoje Intel.

Por fim, o trabalho gerou a publicação do artigo intitulado "*FPGA Implementation of Artificial Neural Networks: A Detailed Approach*", o qual foi apresentado na oitava edição do congresso *Workshop on Circuits and System Design* – WCAS 2018. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

8.1 TRABALHOS FUTUROS

Trabalhos futuros podem se beneficiar da arquitetura proposta para embarcar a suas redes treinadas com o mínimo de esforço, bastando para isso ajustar o trecho do *script* do carregamento de pesos e polarizações, caso o treinamento seja feito com algum código diferente do utilizado neste trabalho. Efetivamente, essa é a principal contribuição deste trabalho. Também, as considerações de projeto de elementos comuns de redes neurais em *hardware* podem ser aplicáveis na implementação de novas topologias.

Do ponto de vista da arquitetura, dois pontos podem ser sugeridos como melhorias, apesar de eles poderem acarretar em impactos consideráveis em efeito cascata no restante do projeto. São eles:

1. Permitir que o projeto aceite mais funções de ativação: o método proposto para o projeto da LUT é especialmente apropriado para funções de ativação altamente não-lineares e que saturam nos extremos. É o caso da Sigmóide, Tanh e *Softsign*. Isso foi foi feito pensando em construir um projeto que fosse o mais genérico possível. Por outro lado, funções de ativação largamente utilizadas como a ReLU ou qualquer outra composta por retas não precisariam da complexidade de uma LUT, fazendo muito mais sentido "calculá-las em tempo de execução". O *script* gerador poderia ser modificado para decidir, a depender da função de ativação escolhida, se a implementaria com LUT ou não. Deve-se ter em mente que trabalhar com funções de ativação que não saturam impõe mais um desafio no projeto, devendo-se ter cuidado para que não ocorra *overflow* nos acumuladores.

2. Tamanho dos acumuladores calculado automaticamente: o projeto atual permite que redes sejam geradas e a partir daí pode-se verificar se houve ou não degradação na classificação. Contudo, dado que uma rede treinada tem todos os pesos e polarizações bem definidos, seria possível saber o valor máximo de todos os acumuladores individualmante, considerando todos os passes diretos do conjunto de teste. Realizar acumuladores de tamanhos diferentes no código em ponto fixo seria extremamente difícil, porém uma abordagem simplificada seria obter um tamanho máximo (é possível obter esse dado em software considerando todos os exemplares da base de dados) e adotar esse valor para todo mundo, eliminando assim um parâmetro de decisão de projeto que é o número de *bits* para a parte inteira.

REFERÊNCIAS

CAVUSLU, M.; KARAKUZU, C.; SAHIN, S. Neural network hardware implementation using fpga. *3. ISEECE*, p. 287–290, 2006. Citado na página 44.

CHEN, Y.; PLESSIS, W. du. Neural network implementation on a fpga. In: IEEE. *Africon Conference in Africa, 2002. IEEE AFRICON. 6th.* [S.l.], 2002. v. 1, p. 337–342. Citado na página 45.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning.* [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>. Citado 3 vezes nas páginas 25, 33 e 39.

HIMAVATHI, S.; ANITHA, D.; MUTHURAMALINGAM, A. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, IEEE, v. 18, n. 3, p. 880–888, 2007. Citado 4 vezes nas páginas 25, 43, 45 e 47.

JACK, K. *Video Demystified: A Handbook for the Digital Engineer.* Elsevier Science, 2011. (Video Demystified Series). ISBN 9780080553955. Disponível em: <https://books.google.com.br/books?id=6dgWB3-rChYC>. Citado na página 64.

MACK, C. A. Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing*, IEEE, v. 24, n. 2, p. 202–207, 2011. Citado na página 24.

MCCARTHY, J. *What is Artificial Intelligence?* 2007. Disponível em: <http://www-formal.stanford.edu/jmc/whatisai/whatisai.html>. Acesso em: 14 out 2017. Citado na página 23.

MINSKY, M.; PAPERT, S. *Perceptrons: An Introduction to Computational Geometry.* M.I.T.Pr., 1988. ISBN 9780262631112. Disponível em: <https://books.google.com.br/books?id=U-O9BHlTvJIC>. Citado na página 23.

MOORE, G. E. Cramming more components onto integrated circuits. *Electronics*, IEEE, v. 38, n. 8, 1965. Citado na página 24.

NIELSEN, M. E. *Neural Networks and Deep Learning.* [S.l.]: Determination Press, 2015. <http://neuralnetworksanddeeplearning.com/>. Citado 5 vezes nas páginas 24, 32, 40, 67 e 68.

NOORY, B.; GROZA, V. A reconfigurable approach to hardware implementation of neural networks. In: IEEE. *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on.* [S.l.], 2003. v. 3, p. 1861–1864. Citado na página 43.

OMONDI, A. R.; RAJAPAKSE, J. C. *FPGA implementations of neural networks.* [S.l.]: Springer, 2006. v. 365. Citado na página 45.

QIU, J. et al. Going deeper with embedded fpga platform for convolutional neural network. In: ACM. *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* [S.l.], 2016. p. 26–35. Citado na página 44.

ROSENBLATT, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, v. 65, n. 6, p. 306–408, 1958. Citado na página 23.

SU, J. et al. Neural network based reinforcement learning acceleration on fpga platforms. *ACM SIGARCH Computer Architecture News*, ACM, v. 44, n. 4, p. 68–73, 2017. Citado na página 24.

TURING, A. M. Computing machinery and intelligence. *Mind*, [Oxford University Press, Mind Association], v. 59, n. 236, p. 433–460, 1950. ISSN 00264423, 14602113. Disponível em: <http://www.jstor.org/stable/2251299>. Citado na página 23.

WANG, Y. et al. Deepburning: automatic generation of fpga-based learning accelerators for the neural network family. In: ACM. *Proceedings of the 53rd Annual Design Automation Conference*. [S.l.], 2016. p. 110. Citado na página 24.

YOUSSEF, A.; MOHAMMED, K.; NASAR, A. Two novel generic, reconfigurable neural network fpga architectures. In: IEEE. *Artificial Intelligence with Applications in Engineering and Technology (ICAIET), 2014 4th International Conference on*. [S.l.], 2014. p. 3–7. Citado na página 43.

ZHU, J.; SUTTON, P. Fpga implementations of neural networks–a survey of a decade of progress. In: SPRINGER. *International Conference on Field Programmable Logic and Applications*. [S.l.], 2003. p. 1062–1066. Citado 2 vezes nas páginas 43 e 45.

Apêndices

APÊNDICE A – OBSERVAÇÕES SOBRE DECISÕES DE PROJETO

Ao longo do desenvolvimento deste trabalho, o projeto da rede precisou passar por várias mudanças conforme iam surgindo novos desafios na implementação. A princípio, tinha-se pensado em uma rede sendo um módulo com tantas entradas quantos fossem seus neurônios de entrada (além do *clock*, *reset* e talvez algum *valid*). Como a ideia era fazer uma rede que pudesse ser parametrizável, surgia então o primeiro desafio: não há como fazer um modulo com número variável de entradas em Verilog. Certamente seria possível ter somente uma entrada de tamanho variável mas essa opção foi logo descartada pela complexidade que o *debug* poderia se tornar. A solução mais atraente na época foi trabalhar com *scripts* para a geração de arquivos texto - uma solução não tão elegante, mas que trouxe algumas vantagens importantes do ponto de vista de facilitar o desenvolvimento do projeto, a saber:

- A rede para a implementação que se queria adotar era treinada em Python. O processo de importação dos parâmetros treinados (pesos e polarizações) é muito mais simples quando o programa que os utiliza é na mesma linguagem. Caso o projetista optasse por não gerar a rede em Python, seria necessário de toda forma escrever um *script* Python para exportar todos os parâmetros treinados para um arquivo texto para então lê-lo do Verilog por meio das funções de sistema $\$fopen$ e $\$fscanf$ e por fim carregá-los na memória.

- O projeto propunha que se pudesse variar o número de neurônios e de camadas. Porém, como já foi comentado, a linguagem Verilog não permite que se tenha, mesmo usando a declaração *generate* e/ou parâmetros, um módulo com número realmente variável de entradas e/ou saídas. A única forma de se ter algo semelhante em Verilog seria ter somente uma porta, seja ela de entrada ou saída, encapsulando todas as entradas e/ou saídas. Por exemplo:

  ```
  input [IN_LENGTH*IN_SIZE-1:0] data_in_array,
  output [OUT_LENGTH*OUT_SIZE-1:0] data_out_array
  ```

  Dentro do módulo, a entrada encapsulada *data_in_array* teria que ser dividida nos diferentes sinais e as saídas internas desejadas deveriam ser concatenadas e então atribuídas ao sinal encapsulado de saída *data_out_array*. Apesar de ser possível fazer dessa maneira, a depuração se tornaria muito mais complicada, principalmente ao se trabalhar com ponto fixo e não números reais. No início do projeto, a arquitetura da camada era descrita por um módulo com tantas entradas quanto fossem os neurônios nela contidas. A possibilidade de se trabalhar com *strings* no Python era a única possibilidade de se ter módulos com número variável de entradas e saídas. Contudo,

durante o desenvolvimento deste trabalho, o projeto original foi sofrendo modificações e optou-se por se apresentar dados serialmente em uma única porta de entrada.

- Mais uma vez, a possibilidade de se trabalhar com *strings* no Python foi bastante útil durante a fase de desenvolvimento, pois tornou possível gerar linhas de comentários em Verilog, as quais podiam conter os valores reais de sinais em binário, a diferença entre o valor binário e o valor real esperado e etc. Também, tornou-se possível trabalhar com a edição do nome dos sinais, e não somente com índices de *arrays*.
- Python possui bibliotecas numéricas como *numpy* e de visualização como *matplotlib*, as quais foram muito úteis durante o desenvolvimento da LUT. Ademais, a biblioteca *bitstring* foi bastante útil para converter números reais em strings em binário, uma ferramenta importante para quem trabalha com ponto fixo.

Outras mudanças grandes aconteceram ao longo do desenvolvimento: na época na publicação do artigo, tinha-se a LUT puramente combinacional como uma série de *assigns* no código. Mais tarde notou-se ser mais apropriado implementar a LUT com uma ROM que pudesse ser acessada por todos os neurônios da rede, cada um por vez.

APÊNDICE B – RELATÓRIOS DE COMPILAÇÃO

A arquitetura projetada ao longo deste trabalho foi gravada no kit de desenvolvimento Altera DE2-115. A Figura 34 apresenta o resultado de compilação do módulo *Network* para a Rede 3. Ela foi a rede escolhida por ter o número mais alto de classificações corretas em *software*.

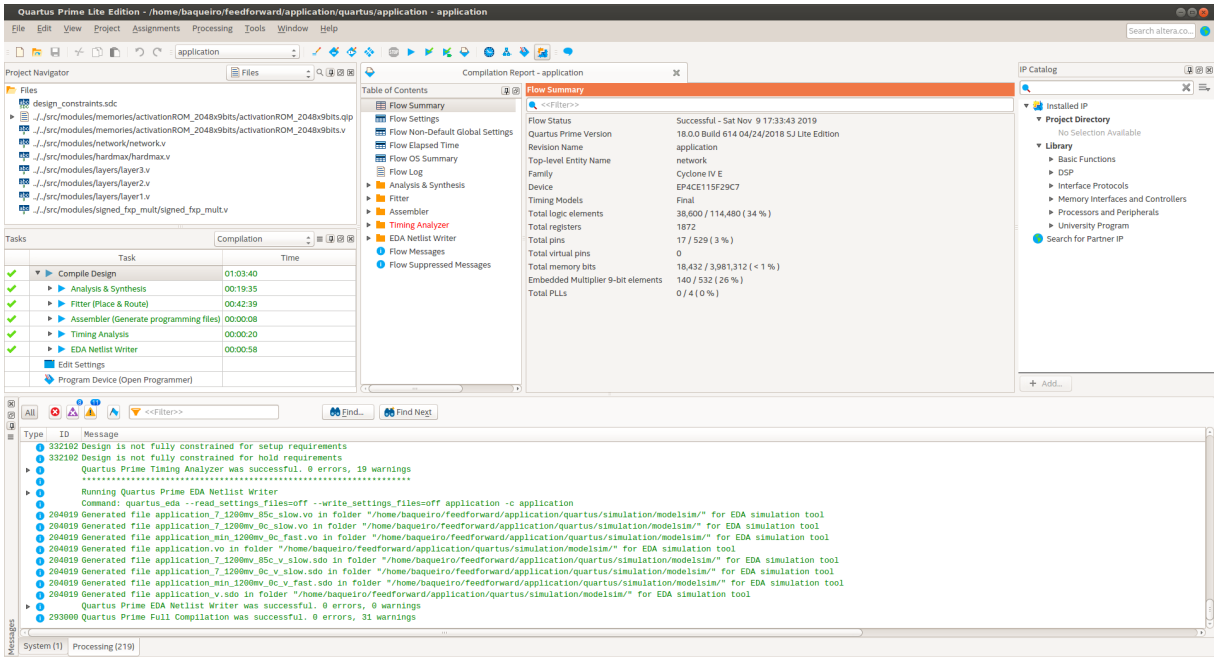


Figura 34 – Relatório de Compilação do Quartus para a Rede 3.

A Figura 35 apresenta a mesma Rede 3, porém encapsulada em um módulo com as funcionalidades necessárias para a aplicação proposta (módulos para captura de dados da câmera e de geração de sinais para o monitor, memórias - uma para o armazenamento de toda a tela VGA, outra para o armazenamento somente da região de interesse e outra para o armazenamento e envio da mesma região de interesse para a rede neural - geração de sinal de *valid_in* periódico para a rede, além de alguns trechos de código para fins de depuração).

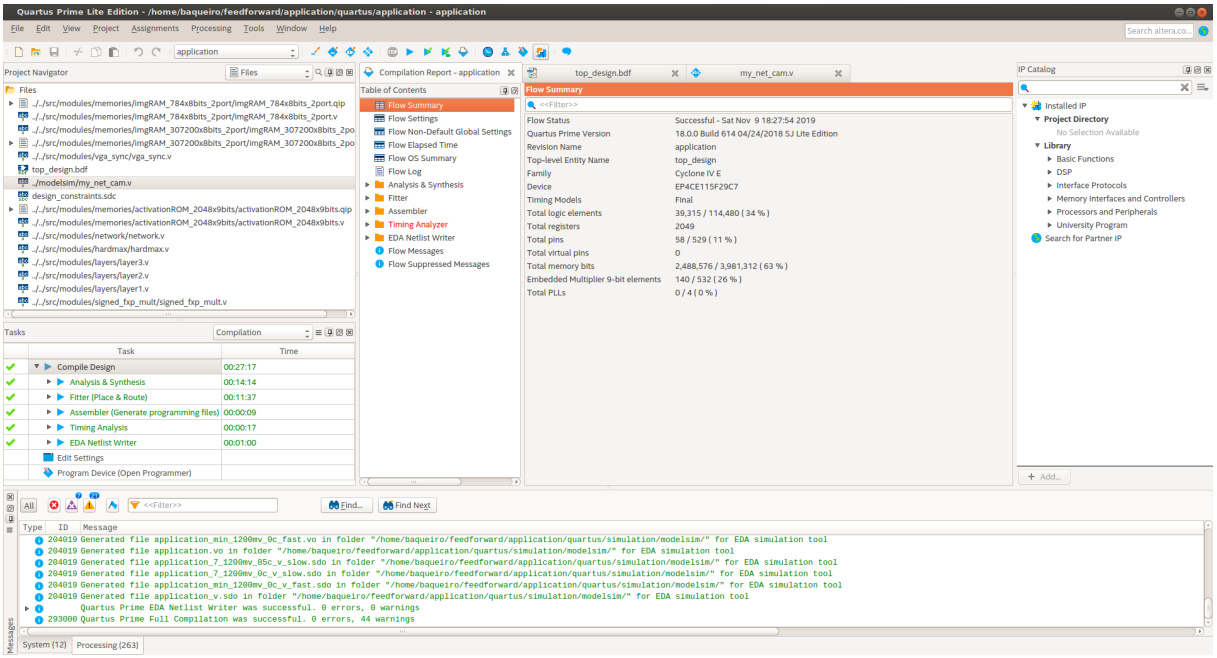


Figura 35 – Relatório de Compilação do Quartus para a aplicação proposta utilizando a Rede 3.