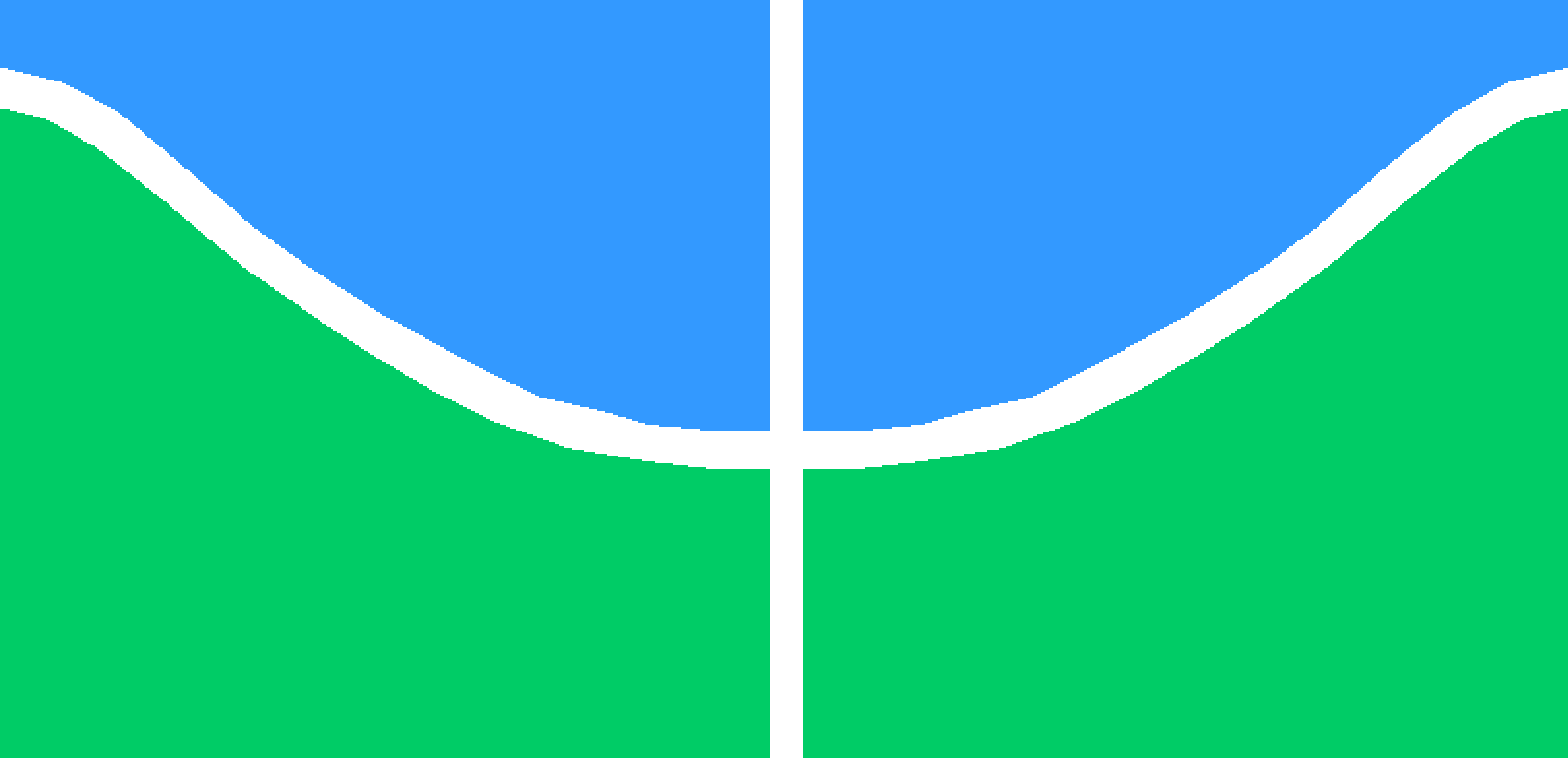


TRABALHO DE GRADUAÇÃO

Implementação de uma ResNet em FPGA, como método de identificação de objetos

Antônio Carlos Oliveira Ferreira

Moisés Silva de Sousa

Brasília, 17 de Maio de 2021

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

Implementação de uma ResNet em FPGA, como método de identificação de objetos

Antônio Carlos Oliveira Ferreira

Moisés Silva de Sousa

Relatório submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiro de Redes de Comunicação

Banca Examinadora

Prof. Alexandre Solon Nery, ENE/UnB ______________________
Orientador

Prof. Daniel Guerreiro e Silva, ENE/UnB ______________________
Examinador Externo

Prof. William Ferreira Giozza, ENE/UnB ______________________
Examinador Externo

Um especialista é um homem que cometeu todos os erros que podem ser cometidos em um campo muito restrito.

Traduzido, Niels Bohr

Agradecimentos

Agradeço a Deus, por todas as oportunidades concedidas e apoio nos momentos difíceis e pela intercessão de Nossa Senhora do Rosário, que não me deixou cair por sequer 1 segundo. Aos meus pais, Francisco Eudes Ferreira Manço e Jucilene Oliveira Ferreira, que sempre se dedicaram para me dar uma boa educação, fazendo o possível e o impossível para isso, sem os quais seria impossível eu estar aqui hoje. Ao nosso orientador, o professor Alexandre Nery, que nunca nos deixou sem apoio. A todos os meus amigos que me ajudaram nos momentos mais difíceis, me dando forçar para persistir, não só na realização desse trabalho como do curso como um todo. Agradeço ainda ao meu parceiro nesse trabalho, Moisés, não só por ter aceitado fazê-lo comigo, como também por todo esforço e dedicação dele nesse período.

Antônio Carlos Oliveira Ferreira

Agradeço, primeiramente, a Deus, por me possibilitar mais um passo em minha carreira profissional. À minha família pelo apoio ao longo de todos os anos de estudo, especialmente aos meus pais, Geovane Devesa de Sousa e Marta Salvadora Aguiar Silva, por todo apoio e esforço que realizaram durante a minha vida acadêmica. Aos meus amigos que se fizeram presentes nesta caminhada e à minha comunidade Neocatecumenal que sempre rezou por mim. Também agradeço ao Antônio, por termos trabalhados não apenas como dupla de trabalho, mas como amigos e ao meu orientador, o professor Alexandre Solon Nery que se fez presente durante toda essa jornada de construção do trabalho.

Moisés Silva de Sousa

RESUMO

O aumento na quantidade de dados e dispositivos embarcados com capacidade de processamento têm resultado na criação de muitos sistemas de tomada de decisão. O uso de aprendizado de máquina é cada vez mais presente em nosso meio, principalmente no que tange a agilidade em se computar dados e inferir resultados de maneira automática e eficiente. Essa automação ganha cada vez mais espaço à medida que os recursos computacionais evoluem e a necessidade de otimização de recursos cresce. É importante ressaltar que o intuito ainda não é substituir por completo o trabalho humano por computadores, mas viabilizar melhores estratégias para que a obtenção de dados e resultados aconteça de forma mais fluída, assertiva e com economia de tempo. O aprendizado de máquina pode proporcionar esses fatores aos seus usuários.

Tendo esse pressuposto, este trabalho apresenta a descrição de uma ResNet (*Residual Network*), completamente do início, em linguagem C para implementação em sistemas FPGA (*Field-Programmable Gate Array*) e afins, usando para isto tecnologia de síntese de alto nível HLS (*High-Level Synthesis*). Para isso, e para a comprovação dos resultados dessa rede, há a necessidade da comparação dos resultados com modelos já prontos e disponíveis. Para essa finalidade, a ferramenta Tensorflow foi escolhida para aferir essa paridade entre elas. Resultados de consumo de área de circuito e desempenho são apresentados.

ABSTRACT

The increase in the amount of data and embedded devices with processing capacity has resulted in the creation of many decision-making systems. The use of machine learning is increasingly present in our environment, especially with regard to the agility in computing data and inferring results automatically and efficiently. This automation gains more and more space as computing resources evolve and the need for resource optimization grows. It is important to emphasize that the intention is not yet to completely replace human work with computers, but to enable better strategies so that obtaining data and results happens in a more fluid, assertive and time-saving manner. Machine learning can provide these factors to its users.

With this assumption, this work presents the description of a ResNet (*Residual Network*), from the ground up, in C language for implementation in FPGA systems (*Field-Programmable Gate Array*) and the like, using HLS high level synthesis technology (*High-Level Synthesis*). For this, and to prove the results of this network, there is a need to compare the results with ready-made and available models. For this purpose, Tensorflow was chosen to measure this parity between them. Results of circuit-area consumption and performance are presented.

SUMÁRIO

LISTA DE FIGURAS .... **v**

LISTA DE TABELAS .... **vi**

1 Introdução **1**
- 1.1 Definição do Problema 1
- 1.2 Objetivos 2
- 1.3 Estrutura do Trabalho 3

2 Fundamentação Teórica **4**
- 2.1 Redes Neurais Convolucionais 4
- 2.2 Redes Residuais 7
- 2.3 Linguagem *Python* 11
- 2.4 Biblioteca *TensorFlow* 12
- 2.5 *Field-Programmable Gate Arrays* – FPGAs 12
- 2.6 *High-Level Synthesis* – HLS 15

3 Implementação **17**
- 3.1 Arquitetura Proposta da CNN 17
- 3.2 Implementação *Tensorflow* 18
- 3.3 Implementação em linguagem C 20
- 3.4 Implementação em HLS 23

4 Análise e Resultados **26**
- 4.1 Resultados da convolução 27
- 4.2 Implementação em FPGA 28

5 Conclusão **30**
- 5.1 Trabalhos Futuros 30

Bibliografia **31**

Anexos **34**

I Códigos fonte **35**

I.1 *Python* 35
I.2 C 37
I.3 FPGA 41

LISTA DE FIGURAS

2.1 Estrutura básica de uma CNN. Fonte: (CAMACHO, 2018) 5
2.2 Esquemático de uma convolução em duas dimensões. Fonte: (GANESH, 2019). 6
2.3 Esquemático do funcionamento da técnica de *shortcut*. 7
2.4 Representação gráfica da *ReLU*. Fonte: (BECKER, 2018). 8
2.5 Demonstração da regra *Zero Padding*. 8
2.6 Demonstração de um conjunto de *kernels*. 9
2.7 Demonstração de uso das regras de *pooling*, com base na Figura 2.5. À esquerda, *max pooling*; à direita, *average pooling*. 10
2.8 Representação gráfica da *sigmoid function*. Fonte: (SHARMA, 2017). 11
2.9 Representação funcional do LUT. Fonte: (XILINX, 2019). 14
2.10 Representação funcional da DSP. Fonte: (XILINX, 2019). 15
2.11 Diagrama de funcionamento da conversão do C para utilização do HLS. Fonte: (XILINX, 2018) 16

4.1 Imagem utilizada para a primeira parte da construção da rede neural. 26
4.2 Imagem de cachorro utilizada para testes de classificação de imagens nas duas redes. 28
4.3 Imagem de gato utilizada para testes de classificação de imagens nas duas redes. 28

LISTA DE TABELAS

3.1 Descrição da arquitetura *ResNet-18* implementada (NAPOLETANO; PICCOLI; SCHETTINI, 2018). 18
3.2 Argumentos utilizados para o recebimento das imagens na rede. 19
3.3 Argumentos utilizados para a regra de *Zero Padding*. 19
3.4 Argumentos utilizados para a regra de *Max Pooling*. 20
3.5 Argumentos utilizados para a regra de convolução. 20

4.1 Valores encontrados na saída da *ResNet* para classificação de cão e gato. 28
4.2 Valores de latência encontrados na simulação em HLS. 29
4.3 utilização dos recursos na simulação em HLS. 29

LISTA DE ABREVIATURAS

Acrônimos

API	*Application Programming Interface*
ASIC	*Application-Specific Integrated Circuit*
CNN	*Convolutional Neural Network*
CPU	*Central Process Unit*
DSP	
FPGA	*Field Programmable Gate Array*
GCC	*GNU Compiler Collection*
GPU	*Graphics Processing Unit*
HLS	*High-Level Synthesis*
JPEG	*Joint Photographic Experts Group*
LUT	*Look-up table*
PNG	*Portable Network Graphics*
PNM	*Portable Any Map*
R-CNN	*Region-based Convolutional Neural Network*
ReLU	*rectified linear activation*
ResNet	*Residual Neural Network*
RGB	*Red Blue and Green*
RTL	*Register Transfer Level*
TI	Tecnologia da Informação
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*

Capítulo 1

Introdução

1.1 Definição do Problema

A quantidade de informações geradas cotidianamente em nossa sociedade, bem como a necessidade de absorção de boa parte delas, promovem o crescimento de técnicas de Aprendizado de Máquina (do inglês *Machine Learning*) na área da Tecnologia da Informação. Isso se deve não somente pela agilidade e capacidade de processamento que os computadores possuem atualmente, mas pela necessidade cada vez maior de minerar os dados recebidos, o que humanamente é praticamente impossível sem a ocorrência de erros.

Isso não significa que os computadores estão isentos de erros, mas em um impasse como este, fica clara que a escolha por métodos computacionais é a preferida, dada a velocidade em se obter resultados, ainda que possa existir uma irrisória possibilidade de falhas. Com isso, o uso de Redes Neurais Convolucionais (*Convolutional Neural Networks – CNNs*) tem se tornado bastante abrangente. Mesmo que exista um consumo bem maior de memória e de recursos computacionais (SZE et al., 2017)), a adoção dessas técnicas de processamento de informações ganha força à medida que a velocidade para obtê-las se equipara ou até mesmo supera em relação a agentes humanos (HE et al., 2016).

O consumo de memória é um dos maiores desafios na construção dessas redes. Basicamente, uma rede neural é composta por neurônios. Tal como ocorre no ser humano, estes neurônios serão os responsáveis pelo acúmulo do aprendizado e pela classificação de elementos. Esses componentes estão organizados em camadas, e estas são construídas a partir de um conjunto de operações de multiplicação e de soma entre o material que é recebido na entrada com os parâmetros estimados pelo programador (SANTOS KOROL, 2019).

Portanto, o crescimento no número de camadas implica em um aumento na quantidade de dados a serem salvos e, consequentemente, processados. Surge, então, um outro fator de atenção na construção de redes neurais convolucionais, que é a capacidade de processamento total que o equipamento possui para receber esse quantitativo de informações. Sob essa perspectiva, abre-se a possibilidade de implementação em pelo menos três arquiteturas de *hardware* diferentes: GP-GPU (General Purpose Graphics Processing Unit), ASIC (Application-Specific Integrated Circuit)

e FPGA (Field-Programmable Gate Array) (SANTOS KOROL, 2019).

Com seu alto grau de paralelismo em nível de *Threads*, a implementação em GPU é escolhida principalmente quando se deseja um máximo desempenho na fase de treinamento, já que este processo não é executado regularmente. Sua capacidade de realizar operações em ponto flutuante de precisão dupla (64-bits) também é um outro fator que permite a utilização para treinamento de redes neurais, já que o processo de retropropagação requer tal funcionalidade.

Já a implementação ASIC, dentre as três citadas, é a que alcança o melhor desempenho, pois trata-se da implementação de um Circuito Integrado dedicado para a execução de uma dada aplicação (KUON; ROSE, 2007). Entretanto, o seu elevado custo de desenvolvimento associado ao seu alto grau de especificação desde sua fabricação, impedem uma utilização mais abrangente. Neste ponto, as FPGAs ganham força dada a sua flexibilidade para a utilização e custos mais reduzidos em relação a GPU e ASIC, respectivamente (QASAIMEH et al., 2019). As FPGAs são uma classe de circuitos integrados com arquitetura reprogramável, o que permite a especificação de sistemas digitais e algoritmos complexos em hardware. A especificação da arquitetura é feita através de linguagens de descrição de hardware.

Tendo esse pressuposto, entender o funcionamento das técnicas de aprendizado de máquina não apenas se torna relevante, mas também essencial no contexto de adoção de tais ferramentas. Além disso, a implementação de uma rede neural em linguagem C/C++ torna possível a portabilidade da arquitetura para diferentes classes de circuitos integrados, incluindo FPGAs e ASICs.

1.2 Objetivos

O objetivo geral deste trabalho consiste na especificação e implementação de uma rede neural ResNet (*Residual Networks*) em linguagem C/C++ e, em seguida, transcrevê-la para uma FPGA usando tecnologia de síntese de alto nível (*High-Level Synthesis*). Portanto, trata-se também de um trabalho de engenharia reversa por meio do estudo da biblioteca Tensorflow, que é voltada para aprendizado de máquina. É importante ressaltar que o treinamento da rede neural em FPGA foge do escopo deste trabalho. Isso significa que a FPGA não processará o código de treinamento, pois não é essa a finalidade empregada a ela no presente trabalho. Além disso, para treinamento, sabe-se que as GPUs são atualmente a classe de dispositivos aceleradores mais veloz. Dada essa condição, utilizou-se a API *Tensorflow* para confeccionar e treinar a ResNet projetada. Os objetivos específicos são listados a seguir:

- Compreender a estrutura interna de uma rede neural;
- Identificar tipos de redes neurais convolucionais que geram menos dependência de recursos computacionais;
- Compreender o funcionamento de diferentes técnicas de aprendizado de máquina;
- Selecionar uma ferramenta de aprendizado de máquina adequada para validação de dados;

- Implementar a rede neural convolucional escolhida em linguagem C para estudos em FPGA;
- Compreender o funcionamento de uma FPGA e a especificação de circuitos;
- Entender o funcionamento de tecnologias de Síntese de Alto Nível;
- Comparar os resultados obtidos pela rede criada;

1.3 Estrutura do Trabalho

Além de uma contextualização acerca do tema proposto neste trabalho, o referido possui a seguinte estruturação de conceitos. Para a abordagem de conceitos básicos que permeiam o tema proposto, há o Capítulo 2. O Capítulo 3 apresenta a arquitetura, descreve o desenvolvimento de cada camada da rede neural convolucional, bem como as dificuldades no processo de implementação de todas elas. O Capítulo 4 contém as análises dos resultados obtidos, bem como explicita o uso do *Tensorflow*, como método de comparação de resultados de um modelo reconhecido internacionalmente. O Capítulo 5 contém a discussão final do estudo, apresentando possibilidades para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo são abordados alguns conceitos que estão inseridos no desenvolvimento deste trabalho e que são fundamentais para o melhor entendimento do mesmo.

2.1 Redes Neurais Convolucionais

Uma rede neural convolucional (*Convolutional Neural Networks – CNNs*) é formada, basicamente, por neurônios que carregam pesos acumulados em seu processo de aprendizagem (DOSHI, 2019). É um algoritmo de aprendizado de máquina que, dentre outras funções, permite a identificação de objetos, levando em consideração o seu processo de aprendizagem e as técnicas de facilitação do aprendizado.

Nesse processo, uma coleção de imagens, chamada *dataset* é introduzida na rede, que executa o processo de convolução entre a entrada de cada camada com as matrizes de pesos. Através de algoritmos próprios de redes neurais, os pesos das matrizes são ajustados a valores que diminuem o erro, tentando minimizá-lo a fim de aumentar a acurácia da rede.

Esse processo segue uma arquitetura pré-estabelecida pelo desenvolvedor de acordo com a problemática a ser tratada, e necessita do estudo de alguns algoritmos básicos como: convolução, *zero padding*, *maxpolling* e *fully connected*. Toda essa organização pode ser observada na Figura 2.1, seguida das suas devidas definições nas Seções 2.1.1 a 2.1.7.

2.1.1 Entrada da Rede

A entrada de uma rede neural precisa ser numérica, dado que serão feitos cálculos convolucionais com ela. Entretanto, isso não impede o uso de imagens em redes neurais. O que se faz é interpretar a imagem de uma maneira numérica. Ela nada mais é que uma matriz de números, com ou sem tridimensionalidade. Essa última característica depende da forma que a figura é apresentada, ou seja, se ela é colorida ou está em escala de cinza.

Entender essa característica é fundamental para o entendimento do processamento da entrada,

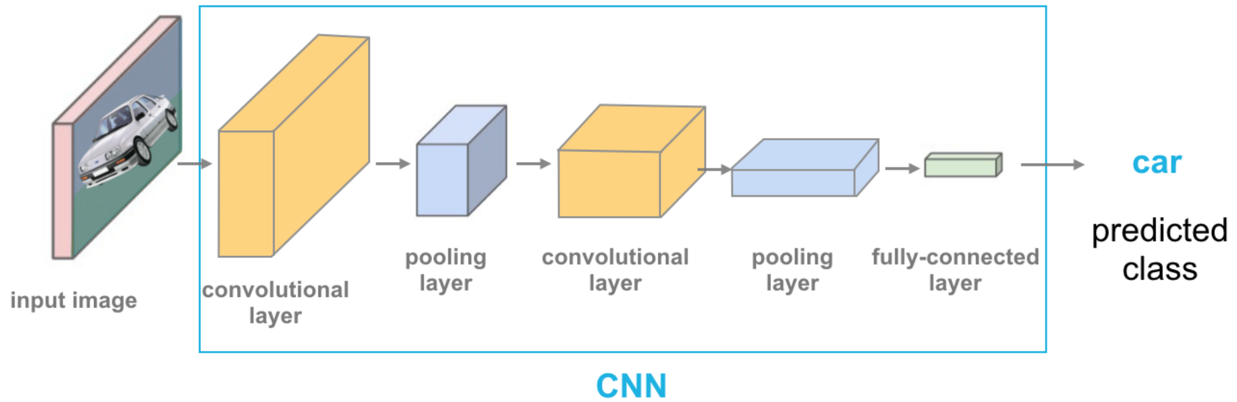


Figura 2.1: Estrutura básica de uma CNN. Fonte: (CAMACHO, 2018)

dado que o aprendizado, e consequentemente a geração de resultados passíveis de identificação, dependem de como esses valores são tratados. Por exemplo, tomemos uma imagem preto e branco, de tamanho 100x100. Neste caso, é necessária a criação de pelo menos 10.000 neurônios para a rede, uma vez que cada um deles carregarão as informações pixel a pixel. Se tivermos uma imagem colorida, de mesmas dimensões, a geração de neurônios será três vezes maior (DOSHI, 2019).

Porém, com o avanço das câmeras, imagens com essas resoluções são minoria em um contexto real. Suponha, então, que temos uma imagem de tamanho 1024x1024 colorida. Então agora precisa-se de pouco mais de 3 milhões de neurônios em uma camada de entrada. Isso geraria ainda mais pesos treináveis ao decorrer da rede, e o treinamento da rede acabaria ficando inviável, passando dos limites computacionais. Entretanto, não é necessária uma abordagem tão detalhista das imagens no contexto de CNNs para se classificar corretamente, podendo ser utilizadas imagens em escala de cinza em tamanhos menores, como 224x224, que não aparenta trazer muitos detalhes a olho nu, porém que traz informações suficientes se o *dataset* for bem montado.

2.1.2 Algoritmo de Convolução

A convolução é uma operação matemática, assim como a soma. O sinal de saída é o resultado obtido a partir de um sinal de entrada. Em visão computacional, os algoritmos de convolução são responsáveis por um grande consumo de recursos da máquina onde está implementada a rede neural, já que nestas circunstâncias elas ganham uma segunda dimensão, proveniente do tratamento das imagens, que são matrizes bidimensionais (GANESH, 2019).

Uma convolução de duas dimensões é descrita pela seguinte fórmula:

$$z_{ij} = y_{ij} + \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} x_{i+m,j+n} w_{mn}$$

Onde x_{ij} é o valor da posição ij (altura e largura) da matriz que representa a imagem de entrada. Já w_{mn} é a matriz de tamanho K que se refere ao *kernel*, y_{ij} será somado ao resultado

presente no cálculo da convolução em uma respectiva janela e z_{ij} é o valor de saída na matriz respectiva. Dessa forma, a variação dos índices i e j dependem, portanto, do tamanho original da imagem a ser processada na camada (ALMEIDA, 2015).

No contexto de redes neurais, a camada de convolução calculará o produto entre o *kernel* - matriz de pesos - e a submatriz da imagem, denominada janela de convolução. Em seguida, todos os valores resultantes serão somados (DOSHI, 2019).

Cada *kernel* gera uma matriz de resultado de duas dimensões e o tamanho da terceira dimensão do processo convolucional é dado pela quantidade de matrizes de pesos utilizada. Esse processo matemático é uma filtragem na qual o *kernel* é o filtro, e o resultado final, após passar por todos os filtros, é comumente chamado de *feature map*.

A Figura 2.2 esquematiza o funcionamento dessa convolução. Ela recebe esse nome, pois seu resultado é uma matriz bidimensional. Nota-se que cada *kernel* (em laranja) possui a mesma profundidade da entrada (marcada em azul) e executa os cálculos matemáticos com os valores da janela de convolução (marcada em rosa).

Cada janela de convolução gera um número (marcado em vermelho) e cada convolução completa entre um filtro e a entrada gera uma saída bidimensional (marcada em verde) na Figura 2.2. Com cada matriz de pesos gerando uma saída em duas dimensões, teremos N saídas, sendo N o número de *kernels*. O já citado *feature map* de uma camada de convolução será a concatenação de resultados de cada convolução, gerando uma matriz tridimensional.

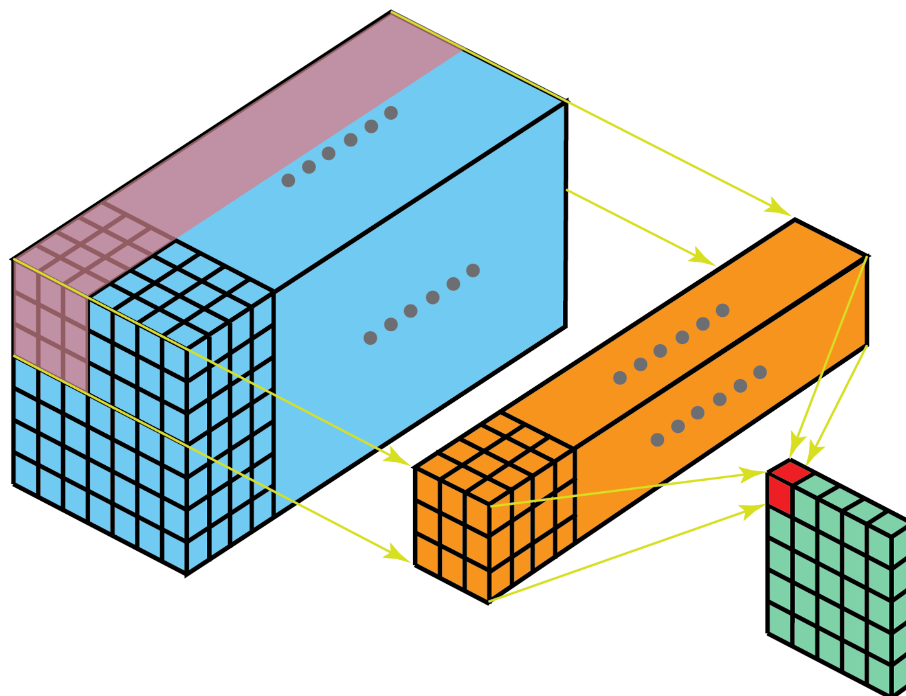


Figura 2.2: Esquemático de uma convolução em duas dimensões. Fonte: (GANESH, 2019).

2.2 Redes Residuais

Outro detalhe a ser considerado nestas convoluções, é a profundidade da rede. Quanto mais convoluções a rede faz, mais profunda fica, e essa profundidade pode acabar aumentando o erro e, portanto, diminuindo a acurácia. Isso ocorre porque a profundidade da rede significa mais operações matemáticas e mais pesos, podendo acarretar em perdas de informações. As ResNets foram concebidas estrategicamente para mitigar essas perdas, adotando a técnica de atalhos.

Esta técnica é empregada somando a entrada de um bloco de convolução à saída. O bloco de convolução corresponde a uma sequência de duas convoluções com janela de convolução 3x3. Na Figura 2.3 há uma representação desse esquema, em que a saída y recebe não somente o resultado do bloco de convolução, nomeado de F, mas sim uma soma entre x (entrada) com o seu próprio resultado (SOUGHT, 2018).

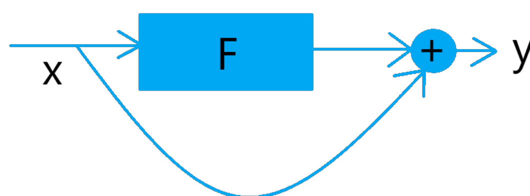


Figura 2.3: Esquemático do funcionamento da técnica de *shortcut*.

É ainda importante frisar que há, a cada dois blocos de convolução, aumento do número de filtros utilizados. Sempre que isso ocorre, há redução nas dimensões de largura e altura dos *feature maps* gerados, através de um aumento no tamanho do passo que a janela de convolução dá, de 1 para 2.

2.2.1 *ReLU*

A ReLU é uma etapa complementar a camada de convolução. Ela funciona da seguinte maneira: a função retificadora elimina quaisquer valores negativos da expressão, tornando-os zeros e mantendo os valores naturalmente positivos como estão. Assim, quando um peso negativo é preponderante na multiplicação da janela de convolução, aquele resultado tende a 0, agindo para penalizar aquele tipo de formato presente na imagem (AGARAP, 2019).

Sua função é descrita pela equação abaixo e está representado na Figura (TEAM, 2018).

$$f(x) = max(0, x)$$

2.2.2 *Zero Padding*

O *Zero Padding* é uma técnica que adiciona zeros nas bordas das imagens, tanto nas superiores e inferiores, quanto nas laterais. Esse tipo de funcionalidade é importante porque permite que

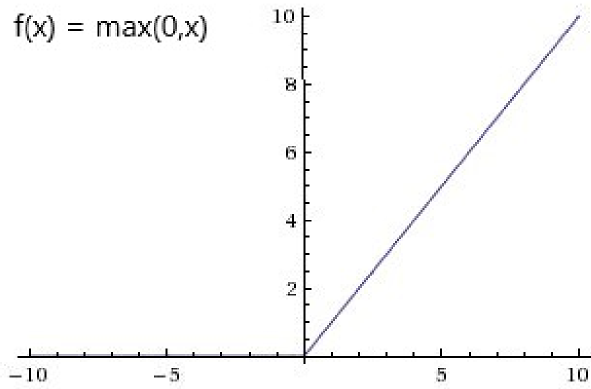


Figura 2.4: Representação gráfica da *ReLU*. Fonte: (BECKER, 2018).

todos os valores existentes na imagem sejam utilizados mais de uma vez durante as convoluções.

A espessura dessa borda dependerá do tamanho do filtro utilizado. Ou seja, quanto maior for, maior será a adição de zeros as bordas da imagem, caso o implementador deseje utilizar os elementos do *feature map* sem mudar a dimensionalidade (DOSHI, 2019).

Figura 2.5: Demonstração da regra *Zero Padding*.

Como podemos ver na Figura 2.5, a adição de duas colunas extras, a direita e a esquerda, e de duas linhas, acima e abaixo, permite que todos os elementos dos *feature maps* sejam usados três vezes nos cálculos de convolução, quando a janela, representada pelas linhas vermelhas, for uma matriz 3x3. Sem essa técnica, os elementos nas extremidades seriam utilizados em menor quantidade e a saída seria reduzida em relação à entrada.

2.2.3 *Pooling*

Trabalhar com processamento de imagem requer uma considerável quantidade de memória disponível (DANCKAERT; CATTHOOR; DE MAN, 1996). Dessa forma, adotar medidas que mitiguem esse tipo de consumo é essencial no processo de criação de redes neurais.

Para reduzir essa necessidade de poder computacional na implementação de redes neurais, existem as camadas de *pooling*. É importante frisar que, por mais que haja a redução dos dados que uma imagem pode entregar, essa camada não precariza o processo de aprendizado da máquina, já que no seu funcionamento ela extrai a informação dominante existente (DOSHI, 2019).

Dentre os utilizados, estão o *max pooling* e *average pooling*. Como os próprios nomes sugerem, essas camadas usarão ou os valores máximos ou médios de uma matriz para decidir qual o dado exato será passado para a nova matriz redimensionada. Tomemos como exemplo a Figura 2.6.

12	10	5	2
6	11	4	5
4	12	14	6
8	10	9	8

Figura 2.6: Demonstração de um conjunto de *kernels*.

Com o uso das regras de *pooling*, cada região colorida da Figura 2.6 será convertida em apenas um valor, seja ele máximo ou médio. A Figura 2.7 simula esse comportamento.

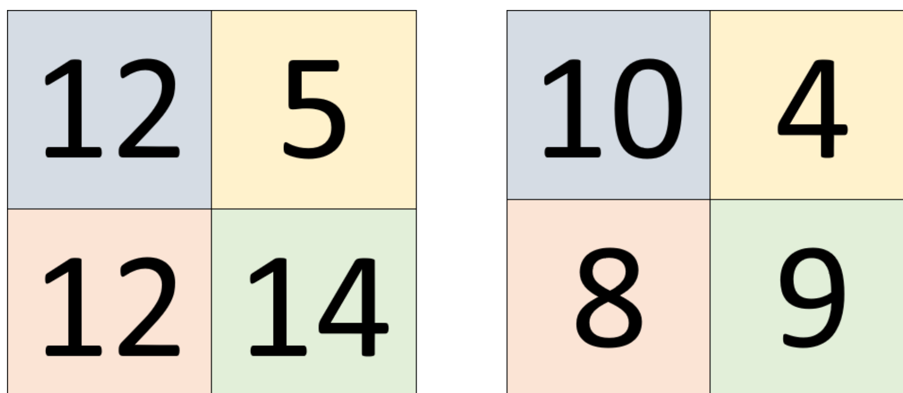


Figura 2.7: Demonstração de uso das regras de *pooling*, com base na Figura 2.5. À esquerda, *max pooling*; à direita, *average pooling.*

Além disso, a adoção de técnicas de *pooling* permite trabalhar com as informações mais relevantes que a imagem entrega, e filtra pequenos ruídos que possam existir.

2.2.4 Camada totalmente conectada

A camada *fully connected* recebe em sua entrada uma versão achatada da imagem proveniente da última convolução da rede. Dessa forma, todos os seus elementos, representados de forma matricial em 3 dimensões, são convertidos em apenas uma coluna, com um número relativamente grande de linhas. Essa é a última camada na qual há atuação de pesos (DOSHI, 2019).

Nesta camada, os dados são preparados para gerar a classificação dos objetos com base nas informações extraídas ao longo da rede. Assim como nas demais redes neurais, essa camada multiplica cada elemento da coluna citada por um peso e soma seus resultados. A saída, por sua vez, é calculada através de uma função de ativação, cuja entrada é o resultado anteriormente obtido, que imprime a probabilidade de cada classe.

Ou seja, se a rede está treinada para identificar cães e gatos, a saída dessa camada será a probabilidade que aquela imagem possua, em termos percentuais, quaisquer um destes animais. Entretanto, a depender do número de classes existentes, a função de ativação será diferente.

2.2.5 Funções de ativação

Existem muitas funções de ativação. Dentre as principais estão: *softmax* e *sigmoid function* e a já citada anteriormente ReLU. A função *softmax* transforma um vetor de K valores reais em um vetor de K elementos reais que somam 1. Os valores que estão na entrada da função só precisam ser números pertencentes ao conjuntos dos reais (WOOD, 2021).

Dado que a saída destas funções possui apenas valores entre 0 e 1, eles podem ser interpretados como a probabilidade do conteúdo da imagem original pertencer a cada classe. Sendo assim, se a entrada for muito pequena ou até mesmo negativa, a sua saída será mais próxima de zero, por exemplo (WOOD, 2021).

A função sigmoide possui o mesmo objetivo em uma CNN: transformar um número real em uma probabilidade. Entretanto, ela atua apenas em problemas com duas classes, pois possui apenas uma entrada e uma saída, gerando ao final um número entre 0 e 1. Esse valor corresponde à chance da imagem ser de uma dada classe, e o valor da probabilidade da outra classe será o complementar do resultado gerado (SHARMA, 2017).

Como ambas possuem a mesma funcionalidade, o emprego ou não de uma delas se deve ao número de classes existentes. Caso seja binário, utiliza-se a *sigmoid function* (SHARMA, 2017). Entretanto, caso existam pelo menos três classes diferentes, vale o uso da *softmax*. A Figura 2.8, representa o comportamento gráfico da função sigmoide, enquanto a expressão que representa a *softmax* é escrita abaixo.

$$\sigma(\vec{z}) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

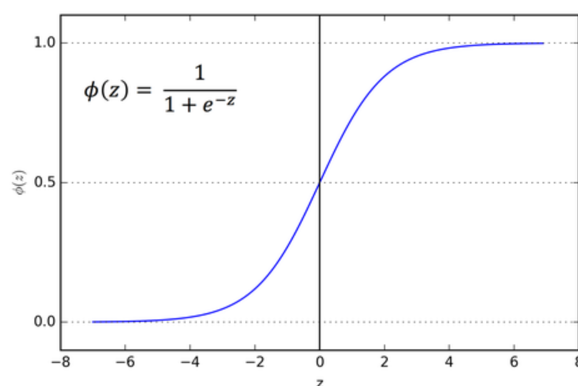


Figura 2.8: Representação gráfica da *sigmoid function*. Fonte: (SHARMA, 2017).

2.3 Linguagem *Python*

O *Python* é uma linguagem de programação *open source* [1] e de alto nível. Lançada em 1991, possui uma vasta e poderosa biblioteca padrão, além da possibilidade de instalação de módulos e *frameworks*[2] desenvolvidos por terceiros. A quantidade de *frameworks* permite maior robustez e de criação de programas com diferentes características e funcionalidades.

Um de seus aspectos mais importantes é a facilidade no processo de aprendizagem da linguagem, uma vez que possui uma comunidade ativa e extensa, com a possibilidade de listas de discussão e documentação ampla. Soma-se a isso, o fato de sua sintaxe ser enxuta e de fácil entendimento.

[1] *Open source*: projetado para ser acessado abertamente pelo público: todas as pessoas podem vê-lo, modificá-lo e distribuí-lo conforme suas necessidades (HAT, 2018).

[2] *Framework*: em desenvolvimento de software, é uma abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica.

2.3.1 Bibliotecas

O *Python* é uma ferramenta muito poderosa e conta com muitos recursos funcionais e interessantes. Porém, como dito anteriormente, essa linguagem possui um acervo extenso de bibliotecas capazes de aprimorar a experiência de quem programa. Elas são instaladas a parte e permitem aos programadores um uso sob demanda, ou seja, as instalações dessas *libraries* só serão realizadas, caso seja necessário.

Uma das bibliotecas existentes é a *matplotlib.pyplot*. Seu funcionamento se baseia em plotagens de gráficos, tal qual ocorre com o MATLAB [3], garante gráficos interativos e é muito funcional em esquemas que necessitam de demonstração visual de alguns dados. O próprio *Tensorflow*, discutido na seção 2.4 e que pode ser usado em *Python*, precisa desse processo de instalação.

Esses processos de instalação de cada biblioteca estão disponíveis nas suas respectivas documentações. Geralmente, utilizam o `pip`[4] neste processo, garantindo simplicidade e agilidade durante a instalação. Além dessa etapa, é necessário ressaltar que é obrigatória a importação do pacote no código construído.

2.4 Biblioteca *TensorFlow*

Além do aprendizado em cima dos conceitos abordados neste processo de construção de redes neurais, por exemplo, muitas vezes os cientistas da computação ou de áreas afins acabam encontrando uma quantidade considerável de ferramentas disponíveis para auxiliá-los.

Entre as ferramentas mais utilizadas para este fim (aprendizado de máquina), podemos listar: *TensorFlow*, *PyTorch*, *Keras* etc. A escolha de uma boa ferramenta é essencial neste processo.

Para este estudo, a ferramenta escolhida para o suporte na criação de redes neurais próprias foi o *TensorFlow*. A sua relevância no mercado permitiu a construção de uma comunidade extensa e bem estabelecida ao redor do mundo desde 2015.

Isso facilita, por exemplo, na resolução de possíveis problemas durante o processo de aprendizado de máquina, o que de certa maneira agiliza a criação e implementação de redes neurais capazes de obedecer ao que é proposto.

Além disso, contar com uma grande comunidade, permite também a existência de uma quantidade considerável de bibliotecas a disposição. Outro fator importante é que se trata de uma ferramenta *open source* (TENSORFLOW, 2021).

2.5 *Field-Programmable Gate Arrays* – FPGAs

Comumente associada à criação de instruções, a programação baseada em *software* se apropria de uma arquitetura geral para prover os resultados desejados. Para esse tipo de abordagem,

[3]https://la.mathworks.com/products/matlab.html
[4]Gerenciador de pacotes *Python*, usado para instalar e gerenciar pacotes de *software* escritos nessa linguagem

profissionais de TI, em geral, fazem a utilização de CPUs ou GPUs.

Quando há o emprego de FPGAs para a implementação de algum recurso, fala-se na construção de circuitos específicos, projetados exclusivamente para aquela aplicação. Apesar dessa estrutura, estes circuitos são reconfiguráveis, e podem ser atualizados conforme as necessidades apareçam.

A partir desse ponto, pode-se notar as vantagens na utilização da FPGA, em detrimento do uso de CPUs e/ou GPUs. Por conta dessa arquitetura, as perdas por latência caem, por exemplo (PLOEG, 2018).

Isso não significa que o tempo entre uma entrada e sua resposta em CPUs sejam altas. Na verdade, a média de 50 microssegundos já é suficiente para grande parte das aplicações. Entretanto, quando se requer uma precisão muito alta, como em pilotos automáticos de aviões, cada fração de segundo a menos é essencial(PLOEG, 2018).

Essa baixa latência, algo em torno de 1 microssegundo, se deve ao grau de especialização que o circuito implementado possui. Soma-se a isso, o fato de FPGAs não dependerem de barramentos ou sistemas operacionais genéricos. (PLOEG, 2018)

Outro ponto crucial a favor da FPGA, é o fator conectividade. Os periféricos que porventura serão alocados a ele são instalados diretamente nos pinos do *chip*. Esse fator contribui com a diminuição da latência que a FPGA apresenta (PLOEG, 2018).

Por fim, o consumo de energia menor, frente ao de GPUs, é outro ponto a ser levado em conta, (NURVITADHI et al., 2017) já que a eficiência enérgica ganha cada vez mais espaço.

2.5.1 Elementos da FPGA

Uma FPGA é constituída por diversas unidades, também chamadas de blocos, que tornam possível a potencialização dela. É por causa desses blocos que se torna interessante processar um código de uma rede neural em uma FPGA, haja visto que ela pode acelerar o processamento se utilizada de forma correta.

2.5.1.1 *Flip-Flops*

Os *Flip-Flops* são um dos mecanismos de potencialização da FPGA. Existem diversos tipos de *Flip-Flops*, mas o mais empregado em FPGAs é o *Flip-Flop* tipo D, que nada mais é que um registrador. Ele é constituído por duas entradas, D e o *clock* e uma saída, denominada Q. Essa saída fica com o valor presente em D no momento da subida do *clock* e o mantém até que o *clock* seja ativado novamente, fazendo com que seja possível se manter em um estado na FPGA e obter informações de processamento do passado, a fim de impactar no que ocorre após esse processamento.

2.5.1.2 *Look-up Table - LUT*

O LUT é o bloco de construção básico de uma FPGA. Ela permite a implementação de um número variado de funções booleanas. Essencialmente, ela trás consigo a tabela verdade na qual diferentes wecombinações nas entradas implementam funções diferentes para a obtenção de valores na saída. O limite que essa tabela pode fornecer baseia-se na quantidade de N entradas disponíveis no bloco, obedecendo a expressão 2^N (XILINX, 2019).

Pode-se imaginar um LUT como sendo uma coleção de memória conectadas a um conjunto de multiplexadores. Suas entradas atuam como bits seletores no *multiplexer* para selecionar o resultado em um determinado momento. A Figura 2.9 mostra a representação funcional de um LUT.

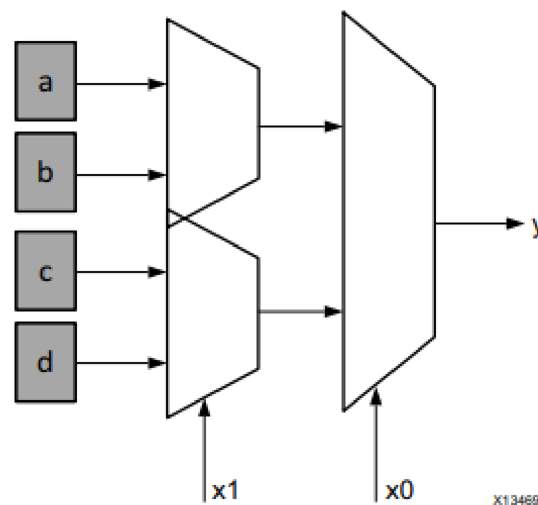


Figura 2.9: Representação funcional do LUT. Fonte: (XILINX, 2019).

2.5.1.3 *Block RAM*

As *Block RAMs* são blocos de memória capazes de armazenar grande quantidade de dados. O custo e o tamanho das FPGAs, no geral, são proporcionais à quantidade de *Block RAMs* que elas possuem. Assim, elas se tornam fator de análise crucial no desenvolvimento de aplicações em uma FPGA, especialmente no que diz respeito a CNNs, pois grande parte da dificuldade histórica do desenvolvimento de algoritmos de aprendizado de máquina é sobre processamento e armazenamento de muitos dados, tornando a quantidade desses blocos um fator limítrofe para desenvolvimento em FPGAs. Em termos programacionais, essa unidade é a responsável pelo armazenamento dos *arrays*, amplamente utilizados no contexto de redes neurais (XILINX, 2019).

É possível ler e escrever dados em uma *Block RAM*. As mais típicas em FPGAs, *dual-port Block RAMs*, permitem ainda que se façam duas operações ao mesmo tempo, desde que em endereços diferentes da *Block RAM* (XILINX, 2019).

2.5.1.4 DSP

A DSP48 é um bloco computacional disponível nas FPGAs Xilinx, cuja estrutura pode ser vista na Figura 2.10. Esse bloco é uma unidade lógica aritmética composta por 3 outros blocos computacionais. A primeira parte é um somador (também capaz de fazer subtrações), que por sua vez é conectado a um multiplicador. O último bloco consiste em mais um somador, mas que também possui a operação de acumulador, que se trata de um armazenamento dos dados calculados para posterior utilização, em conjunto com outros resultados (XILINX, 2017).

Assim como mostrado na Figura 2.10, esse módulo contém 4 entradas, denominadas A, B, C e D, tal que é possível fazer as operações $P = Bx(A+D)+C$ ou $P = P+Bx(A+D)$ (XILINX, 2017).

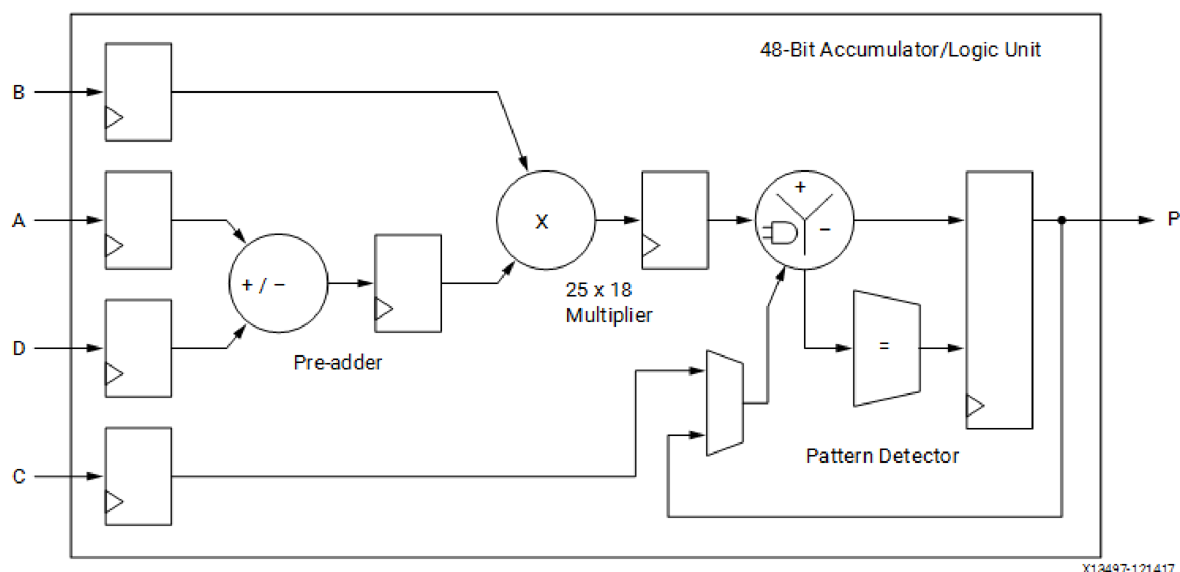
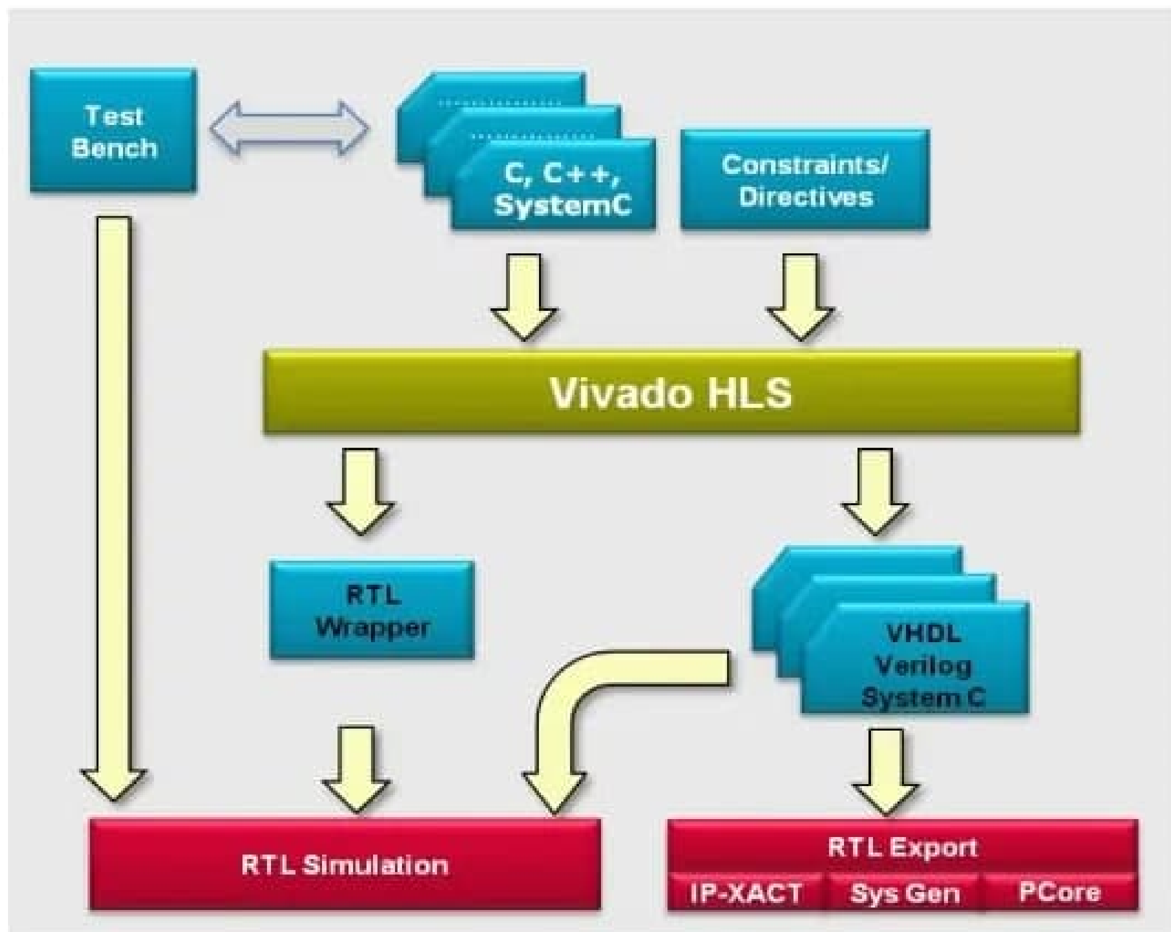


Figura 2.10: Representação funcional da DSP. Fonte: (XILINX, 2019).

2.6 *High-Level Synthesis* – HLS

A síntese de alto nível diz respeito a uma tecnologia de síntese de circuitos capaz de traduzir uma arquitetura descrita em linguagem de alto nível (ex: C/C++) para uma descrição a nível de registradores (*Register Transfer-Level* – RTL), usando para isso uma linguagem de descrição de circuitos como VHDL (*VHSIC Hardware Description Language*) ou Verilog. Ao término deste processo, a arquitetura RTL corresponde a uma implementação da arquitetura em um dispositivo FPGA (JORGE; NERY; MELO, 2019). O passo-a-passo dessa conversão que possibilita a utilização do C para programar em FPGA pode ser vista no diagrama presente na Figura 2.11



Source: Vivado Design Suite User Guide High-Level Synthesis UG902

Figura 2.11: Diagrama de funcionamento da conversão do C para utilização do HLS. Fonte: (XILINX, 2018)

Esta arquitetura é descrita pela sintetização em alto nível através de uma linguagem de descrição de *hardware* como VHDL. O contexto da introdução do HLS permeia o fato dele ser um facilitador no momento das especificações e implementações de arquiteturas RTL baseadas em linguagens de alto nível. Isso reduz significativamente o tempo de projeto de sistemas complexos (JORGE; NERY; MELO, 2019).

Por fim, o HLS contribui de maneira significativa na portabilidade do código, dado que sistemas reproduzidos em linguagem de alto nível, como o C, podem ser compilados e executados tanto em suas arquiteturas originais, como recompilados para implementação em RTL (JORGE; NERY; MELO, 2019).

Capítulo 3

Implementação

Esse capítulo descreve como a arquitetura da rede neural convolucional proposta foi construída, descrevendo as etapas de desenvolvimento, os desafios e as limitações que este projeto poderá ter. A organização do capítulo será primeiramente focada na arquitetura e posteriormente em suas entidades separadamente.

3.1 Arquitetura Proposta da CNN

A construção de uma arquitetura de uma rede neural convolucional é essencialmente decidida de acordo com o problema que se pretende resolver. Por não ser genérico, não há uma fórmula capaz de servir e alcançar resultados idênticos ou similares em diferentes contextos de implementação. Dessa forma, a escolha da arquitetura de uma CNN depende do complexidade do tema abordado e de qual tipo de resultado a rede deve gerar.

Da mesma forma, é relevante considerar a capacidade de processamento e armazenamento do dispositivo que fará o treinamento da rede. Destaca-se isso, pois, à medida que a quantidade de camadas convolucionais aumentam, também será maior a quantidade de dados gerados ao final de cada uma delas. Optou-se pela criação de cinco estruturas convolucionais, as quais cada uma possui dois blocos convolucionais, representados pelas indicações entre colchetes e que corresponde à rede neural *ResNet-18*, cuja estrutura pode ser vista na Tabela 4.1. Essa escolha foi feita para se obter uma rede convolucional efetiva na classificação de imagens, mas que não fosse excessivamente grande, para assim haver menos chances de exigir mais memória do que a FPGA seria capaz de fornecer.

Essa escolha se deve ao fato de ser a menor *ResNet* existente. Com isso, fatores de limitação de memória podem ser corrigidos com a adoção dessa rede, já que menos camadas de convolução resultam em menos geração de dados para a arquitetura, seja ela qual for, processar.

Pela Figura **??**, é possível notar também qual o tamanho da entrada a ser recebida pela rede. Por conta da saída ser um *feature map* de dimensões 112x112x64, a entrada deverá conter dimensão 224x224x1, haja visto que se trata de uma imagem em escala de cinza, e que o ajuste da janela de

convolução, indicado por *stride*, é 2, o que faz com que o resultado final tenha metade das dimensões de altura e largura da entrada. É importante salientar que a Figura **??** traz a função *softmax* como geradora das probabilidades de forma generalizada, mas que essa função é escolhida com base na quantidade de classes que se deseja classificar. Como o caso de estudo é uma classificação binária entre cães e gatos, a escolha feita foi de se implementar a função sigmoide.

Tabela 3.1: Descrição da arquitetura *ResNet-18* implementada (NAPOLETANO; PICCOLI; SCHETTINI, 2018).

Nome da camada	Tamanho da saída	ResNet-18
conv1	112x112x64	7x7, 64, *stride* 2
conv2	56x56x64	$\begin{bmatrix} 3\times3, & 64 \\ 3\times3, & 64 \end{bmatrix} x2$
conv3	28x28x128	$\begin{bmatrix} 3\times3, & 128 \\ 3\times3, & 128 \end{bmatrix} \times 2$
conv4	14x14x256	$\begin{bmatrix} 3\times3, & 256 \\ 3\times3, & 256 \end{bmatrix} \times 2$
conv5	7x7x512	$\begin{bmatrix} 3\times3, & 512 \\ 3\times3, & 512 \end{bmatrix} \times 2$
average	1x1x512	7x7 *average pool*
fully connected	1	512x1 *fully connections*
sigmoid	1	cálculo da sigmoide

3.2 Implementação *Tensorflow*

O objetivo da construção da rede em C é para que seja possível a portabilidade para sistemas embarcados em geral e FPGAs, especialmente para inferência. Isso significa que a FPGA não processará o código de treinamento, pois não é essa a finalidade empregada a ela no presente trabalho. Para treinamento, sabe-se que as GPUs são mais eficientes. Dada essa condição, utilizou-se a API *Tensorflow* para confeccionar e treinar a ResNet projetada. Além disso, é possível também comparar os resultados obtidos por ela com os do código em C, assim validando os processos matemáticos das convoluções feitas.

Por conta disso, esta seção apresentará as etapas de implementação utilizando o *Tensorflow*. O uso dessa ferramenta possibilita o treinamento e armazenamento dos pesos gerados. Em posse dessas informações, é possível gerar inferência pela rede construída em C.

3.2.1 Módulos do Tensorflow

A construção do código, utilizando as ferramentas da biblioteca *Tensorflow*, é iniciada com as importações das bibliotecas (módulos) necessárias. No caso, serão elas que farão com que o código fique mais compacto, uma vez que já carregam consigo funções genéricas.

Além disso, é necessário destacar que o *Tensorflow* possui uma quantidade muito extensa de bibliotecas próprias, sendo necessária a importação dessas. O destaque é que essas importações partem da própria ferramenta e não do *Python*. Como podemos ver pelo trecho de código no Anexo I.1.1, as importações serão utilizadas, principalmente, na manipulação das imagens recebidas pela rede, nas camadas e nos seus modelos.

3.2.2 Recebimento das imagens

Como foi visto na Fundamentação Teórica, a primeira etapa de uma construção de uma rede neural é o recebimento das imagens para serem pré-processadas e servirem como entrada da rede. Como visto pelo Anexo I.1.2, percebe-se que existem alguns argumentos necessários para a implementação efetiva do recebimento desses dados. Essas imagens são então transformadas em um tipo especial de estrutura de armazenamento, chamados *tensors*, que nada mais são do que *arrays* n-dimensionais. Os valores contidos neles são os valores dos *pixels* da imagem pré-processada.

Basicamente, essa parte do código prepara o ambiente, indicando em qual escala de cor estarão os dados, o tamanho da imagem e onde estão localizados. Vale ressaltar que o Anexo I.1.2 é utilizado mais de uma vez no código, pois para a classificação de dados recebidos posteriormente, também requer tal configuração. Cada argumento está explicado na Tabela 3.2

Tabela 3.2: Argumentos utilizados para o recebimento das imagens na rede.

Argumento	Definição
`directory`	Indica onde estão as imagens a serem carregadas.
`class_mode`	Indica o modo que os alvos serão produzidos.
`color_mode`	Indica a escala de cor a ser trabalhada.
`batch_size`	Indica o tamanho do conjuntos de imagens que serão processadas ao mesmo tempo no treinamento da rede.
`target_size`	Indica o tamanho das imagens a serem manipuladas.

3.2.3 Camadas de convolução

Considerando os dados expostos na Figura **??**, tem-se que nas camadas de convolução será necessária a abordagem dos conceitos de: convolução, *Zero Padding* e *Max Pooling*. Cada uma dessas etapas já estão implementadas no *Tensorflow*, bastando passar os argumentos corretos, como fora feito anteriormente com a manipulação das imagens.

Nas Tabelas 3.3, 3.4 e 3.5 são repassados os argumentos utilizados na implementação.

Tabela 3.3: Argumentos utilizados para a regra de *Zero Padding*.

Argumento	Definição
`padding`	Indica valores simétricos para altura e largura.
`data_format`	Indica a ordem das dimensões nas entradas.

Tabela 3.4: Argumentos utilizados para a regra de *Max Pooling*.

Argumento	Definição
`pool_size`	Indica o tamanho da janela sobre o qual se tira o máximo.
`strides`	Indica até que ponto a janela de *pooling* move-se para cada etapa.
`padding`	Indica se haverá preenchimento ou não, para que largura e altura tenham mesma dimensão.

Tabela 3.5: Argumentos utilizados para a regra de convolução.

Argumento	Definição
`filters`	Indica o número de filtros de saída na convolução.
`kernel_size`	Indica a altura e a largura da janela de convolução 2D.
`strides`	Indica os passos da convolução ao longo da altura e largura.
`padding`	Indica se haverá preenchimento ou não, para que largura e altura tenham mesma dimensão.
`use_bias`	Indica se a camada usa um vetor de polarização.

Conforme indicam os Anexos I.1.3.1 a I.1.3.5, os valores aplicados em cada camada de convolução são retirados da Figura **??**. A facilidade que o *Tensorflow* permite no momento da implementação é notória, uma vez que o profissional de TI deve-se preocupar apenas em substituir os valores pensados para janela, *padding* e *stride* nos locais apropriados. Isso permite com que o desenvolvedor não precise se preocupar excessivamente com a construção da estrutura, mas sim com a escolha de uma boa arquitetura e correto tratamento dos dados, uma vez que dados tratados e escolhidos sem cuidado formam um *dataset* ruim que não consegue representar com efetividade a informação que se deseja processar.

3.2.4 Passos finais

Ao término das camadas de convolução, é necessário implementar as partes finais da rede neural. Conforme indica a Figura **??**, as etapas finais deste processo incluem a adição da regra de *average pooling*, *fully connected*, que é chamada na referida API pelo nome *Dense*. Ambas estão apontadas no Anexo I.1.3.6. Além disso, há também o salvamento dos pesos em arquivo de texto, de modo que cada camada de convolução existente gera um documento próprio.

3.3 Implementação em linguagem C

Nesta seção, serão apresentadas as etapas de implementação, utilizando a linguagem C. A adoção dessa linguagem facilita a implementação em FPGA, abordagem a ser feita posteriormente.

3.3.1 Importações e definições prévias

Diferentemente da sintaxe encontrada no *Python*, a linguagem C permite a inclusão de bibliotecas externas mediante *includes* e não importações. Essas inclusões permitem a utilização de funções matemáticas (que serão adotadas nas convoluções), a leitura de arquivos (necessário para a abertura de imagens ou de matrizes de pesos), e a utilização de variáveis lógicas.

Como parte da rede neural já possui valores pré-definidos, também foram colocadas as variáveis globais. Elas serão usadas para estabelecer, dentre outros itens: o número de filtros de cada camada, as dimensões das entradas e as matrizes de pesos existentes e que são mostrados na Figura **??**. Essa configuração inicial pode ser observada no Anexo I.2.2.

3.3.2 Recebimento das imagens e matrizes de pesos

Diferentemente do que ocorre no *Python*, a linguagem C não possui um método tão didático e simples para tratar imagens[1]. Esse processo é feito mediante abertura de arquivo, na qual serão extraídos os valores de cada *pixel* da imagem em *bits*. Por assumir uma escala de cinza, esta unidade só retornará os valores 0 ou 1.

Além desse fator, é interessante ponderar algumas informações extras que estão associadas às imagens. Quando estas estão em formato PNM, é preciso desconsiderar as três primeiras linhas do arquivo. Isso se deve ao fato de haver o cabeçalho da imagem nestas linhas, não sendo importantes no processo de treinamento da rede.

Com o método *fopen* é possível abrir os arquivos com os pesos gerados na simulação com *Tensorflow*. Apesar de não existir uma manipulação em relação ao cabeçalho do arquivo, neste ponto é necessário separar cada valor do documento, de forma que a rede neural o interprete como um peso independente. Por estarem separados por * (asterisco), a cada aparição desse caractere deve ser entendida pela rede como o separador entre um peso e outro. Por isso, utilizou-se a função *strtok*, que utiliza *tokens* para divisão de *strings*.

Tanto a abertura de arquivo, quanto essa manipulação de caracteres pode ser observada no Anexo I.2.2

3.3.3 Camadas

Assim como na implementação no *Tensorflow*, as camadas de convolução criadas nesta etapa também devem levar em consideração não apenas as convoluções em si, mas também as operações de *Zero Padding* e *Max Pooling*.

Essas operações permitem, respectivamente, que os *pixels* localizados nas extremidades sejam levados em conta no momento da convolução, na mesma quantidade que os demais e que haja uma redução nos dados existentes por conta das técnicas de *pooling*, uma vez que, feita a convolução, a geração de informação cresce.

[1] Com a utilização do *Tensorflow*, a abertura de imagens era feita em apenas uma linha de código.

3.3.3.1 *Zero Padding*

Essa operação ocorre sempre que uma convolução está prestes a ser realizada. Sua estrutura, independentemente da camada que precede, é essencialmente a mesma. Conforme é observado no Anexo I.2.3.1, a função inicia-se com a declaração de algumas variáveis que serão utilizadas para manipular a inserção dos números nos devidos locais.

Os *loopings* existentes também são essenciais pois permitem um controle a respeito da devida posição que o ponteiro está, dessa maneira, evitando a inserção de informações em locais inapropriados. No caso, existem dois deles, um atrelado a linha da imagem e outra a coluna.

No caso, como a primeira borda de zeros possui espessura igual a 3, haverá a adição de seis *pixels* iguais a 0 tanto em largura quanto em coluna. Dessa forma, a imagem que possuía resolução de 224x224, sai em formato 230x230.

Para as demais funções de *Zero Padding*, as dimensões das imagens serão diferentes, pois a cada nova convolução feita, uma operação de *Max Pooling* é realizada, diminuindo a dimensão da figura. Para estes casos, ao invés de colocar 230 como limite superior do *looping*, será inserida a dimensão da imagem naquele momento (definida previamente e disponível no Anexo I.2.2), acrescido de 2, uma vez que nas demais funções, a espessura dessa borda é igual a 1.

3.3.3.2 Convolução

Assim como na operação de *Zero Padding*, as camadas de convolução na rede possuem estruturas bem parecidas. Elas, necessariamente, precisam receber a quantidade de filtros estimados para aquela camada, a dimensão da entrada recebida e o tamanho da janela deslizante que passará por toda a imagem e responsável pelos cálculos de convolução.

Pequenas diferenças ocorrem, devido a arquitetura proposta. Para a primeira camada, é exigida apenas um cálculo de convolução. Para as demais, além dessas operações, deve haver o cuidado de redimensionar a entrada para o devido padrão. Isso acontece porque a saída de cada camada possui dimensão e números diferentes de filtros da entrada das próximas.

Além disso, para as camadas subsequentes, não há apenas uma operação de convolução, mas sim quatro, conforme ilustra a Figura **??**. Apesar da quantidade extra de operações, a forma com que é feita é substancialmente a mesma. Essa montagem pode ser observada no Anexo I.2.3.2.

3.3.3.3 *Max Pooling*

Por fim, é necessário inserir a operação de *Max Pooling*. Essa implementação é observada no Anexo I.2.3.3 e sua construção também depende das dimensões da entrada e do número de filtros.

O tamanho da janela implementada é 3x3, o que indica que, a cada 9 valores recebidos, será escolhido o maior dentre eles. A saída dessa função será então a entrada da próxima operação de *Zero Padding* que será realizada pela próxima camada. Entretanto, como o afunilamento vem sendo realizado já nas convoluções, a adoção dessa técnica será feita apenas entre a primeira e a

segunda camada.

Isso não ocorre com o *Zero Padding* nem com as convoluções, que possuem representantes em cada uma das cinco camadas existentes na rede.

3.3.4 Passos finais

Finalizando a implementação da rede neural, as funções de *average pooling* e *fully connected* são apresentadas nesta seção.

A *average layer* recebe a saída da última camada de convolução da rede (quinta), ou seja, possui 512 filtros cada um com dimensão 7x7, conforme é mostrado na Figura **??**. Sua função é aferir a média dos valores obtidos para serem recebidos pela *fully connected*. Sua implementação está no Anexo I.2.4.

Já a *fully connected* está disponível no Anexo I.2.5. Note que ao final do achatamento da entrada, há a função de ativação abordada na fundamentação teórica deste documento. Como o *dataset* utilizado no experimento possui apenas duas classes diferentes: gatos e cachorros, a função de ativação a ser usada é *sigmoid*, ilustrada na Figura 2.8.

3.4 Implementação em HLS

Para a testagem na FPGA escolhida, o primeiro passo foi passar o código para HLS. Entretanto, já era esperada a possibilidade da FPGA não conseguir rodar com êxito a rede, dado que redes neurais exigem bastante poder computacional e memória. Assim se fez, e não foi possível implementar diretamente na placa designada para tal. Isso não significa que não é possível realizar a implementação, mas ela só será possível em uma FPGA com mais *Block RAMs*.

Para dar prosseguimento ao trabalho e, ainda assim, obter resultados palpáveis acerca do processamento da ResNet confeccionada em uma FPGA, utilizou-se o *software* Vivado HLS para realizar a síntese da descrição para RTL, possibilitando uma pré-análise dos requisitos em FPGA. Como dito anteriormente, a FPGA é capaz de processar as operações matemáticas paralelamente, conseguindo processar diversos dados ao mesmo tempo, chegando até mesmo a começar a processar uma janela de convolução sem ter terminado a janela anterior (projeto com pipeline). O código em HLS do primeiro bloco de convolução da ResNet é descrito na listagem 3.1 (e que também consta no Anexo I.3.2).

```
1  #include "conv1_layer.hpp"
2
3  static int image[(DIM_0 + 6) * (DIM_0 + 6)];
4  static float weights_conv1[N_FILTERS_64 * 7 * 7];
5  static float output1[(DIM_1 + 2) * (DIM_1 + 2) * N_FILTERS_64];
6  static float window[7 * 7];
7
8  void conv1layer_accel(volatile int *im, volatile float *wconv1,
9                        volatile float *output, int mode)
10 {
11 #pragma HLS INTERFACE m_axi depth = 90000 port = im offset = slave
12 #pragma HLS INTERFACE m_axi depth = 3136 port = wconv1 offset = slave
13 #pragma HLS INTERFACE m_axi depth = 831744 port = output offset = slave
14 #pragma HLS INTERFACE s_axilite port = im bundle = Ctrl
15 #pragma HLS INTERFACE s_axilite port = wconv1 bundle = Ctrl
```

```
16 #pragma HLS INTERFACE s_axilite port = output bundle = Ctrl
17 #pragma HLS INTERFACE s_axilite port = mode bundle = Ctrl
18 #pragma HLS INTERFACE s_axilite port = return bundle = Ctrl
19 #pragma HLS ARRAY_PARTITION variable = window complete dim = 1
20
21     float conv;
22
23     if (mode == INIT_MEM)
24     {
25         for (int k = 0; k < (DIM_0 + 6) * (DIM_0 + 6); k++)
26 #pragma HLS PIPELINE
27             image[k] = im[k];
28
29         for (int k = 0; k < N_FILTERS_64 * 7 * 7; k++)
30 #pragma HLS PIPELINE
31             weights_conv1[k] = wconv1[k];
32     }
33     else if (mode == READ_RESULT)
34     {
35         for (int k = 0; k < (DIM_1 + 2) * (DIM_1 + 2) *
36                                 N_FILTERS_64;
37              k++)
38 #pragma HLS PIPELINE
39             output[k] = output1[k];
40     }
41     else
42     {
43         for (int filter = 0; filter < N_FILTERS_64; filter++)
44         {
45             for (int i = 0; i < DIM_0; i = i + 2)
46             {
47                 for (int j = 0; j < DIM_0; j = j + 2)
48                 {
49
50 #pragma HLS PIPELINE
51
52                     window[0] = image[(i + 0) * (DIM_0 + 6) + j + 0];
53                     window[1] = image[(i + 0) * (DIM_0 + 6) + j + 1];
54                     window[2] = image[(i + 0) * (DIM_0 + 6) + j + 2];
55                     window[3] = image[(i + 0) * (DIM_0 + 6) + j + 3];
56                     window[4] = image[(i + 0) * (DIM_0 + 6) + j + 4];
57                     window[5] = image[(i + 0) * (DIM_0 + 6) + j + 5];
58                     window[6] = image[(i + 0) * (DIM_0 + 6) + j + 6];
59
60                     window[7] = image[(i + 1) * (DIM_0 + 6) + j + 0];
61                     window[8] = image[(i + 1) * (DIM_0 + 6) + j + 1];
62                     window[9] = image[(i + 1) * (DIM_0 + 6) + j + 2];
63                     window[10] = image[(i + 1) * (DIM_0 + 6) + j + 3];
64                     window[11] = image[(i + 1) * (DIM_0 + 6) + j + 4];
65                     window[12] = image[(i + 1) * (DIM_0 + 6) + j + 5];
66                     window[13] = image[(i + 1) * (DIM_0 + 6) + j + 6];
67
68                     window[14] = image[(i + 2) * (DIM_0 + 6) + j + 0];
69                     window[15] = image[(i + 2) * (DIM_0 + 6) + j + 1];
70                     window[16] = image[(i + 2) * (DIM_0 + 6) + j + 2];
71                     window[17] = image[(i + 2) * (DIM_0 + 6) + j + 3];
72                     window[18] = image[(i + 2) * (DIM_0 + 6) + j + 4];
73                     window[19] = image[(i + 2) * (DIM_0 + 6) + j + 5];
74                     window[20] = image[(i + 2) * (DIM_0 + 6) + j + 6];
75
76                     window[21] = image[(i + 3) * (DIM_0 + 6) + j + 0];
77                     window[22] = image[(i + 3) * (DIM_0 + 6) + j + 1];
78                     window[23] = image[(i + 3) * (DIM_0 + 6) + j + 2];
79                     window[24] = image[(i + 3) * (DIM_0 + 6) + j + 3];
80                     window[25] = image[(i + 3) * (DIM_0 + 6) + j + 4];
81                     window[26] = image[(i + 3) * (DIM_0 + 6) + j + 5];
82                     window[27] = image[(i + 3) * (DIM_0 + 6) + j + 6];
83
84                     window[28] = image[(i + 4) * (DIM_0 + 6) + j + 0];
85                     window[29] = image[(i + 4) * (DIM_0 + 6) + j + 1];
86                     window[30] = image[(i + 4) * (DIM_0 + 6) + j + 2];
87                     window[31] = image[(i + 4) * (DIM_0 + 6) + j + 3];
88                     window[32] = image[(i + 4) * (DIM_0 + 6) + j + 4];
89                     window[33] = image[(i + 4) * (DIM_0 + 6) + j + 5];
90                     window[34] = image[(i + 4) * (DIM_0 + 6) + j + 6];
91
92                     window[35] = image[(i + 5) * (DIM_0 + 6) + j + 0];
93                     window[36] = image[(i + 5) * (DIM_0 + 6) + j + 1];
94                     window[37] = image[(i + 5) * (DIM_0 + 6) + j + 2];
95                     window[38] = image[(i + 5) * (DIM_0 + 6) + j + 3];
96                     window[39] = image[(i + 5) * (DIM_0 + 6) + j + 4];
```

```
97                          window[40] = image[(i + 5) * (DIM_0 + 6) + j + 5];
98                          window[41] = image[(i + 5) * (DIM_0 + 6) + j + 6];
99
100                         window[42] = image[(i + 6) * (DIM_0 + 6) + j + 0];
101                         window[43] = image[(i + 6) * (DIM_0 + 6) + j + 1];
102                         window[44] = image[(i + 6) * (DIM_0 + 6) + j + 2];
103                         window[45] = image[(i + 6) * (DIM_0 + 6) + j + 3];
104                         window[46] = image[(i + 6) * (DIM_0 + 6) + j + 4];
105                         window[47] = image[(i + 6) * (DIM_0 + 6) + j + 5];
106                         window[48] = image[(i + 6) * (DIM_0 + 6) + j + 6];
107
108                         conv = 0.0f;
109
110                         for (int k = 0; k < 7 * 7; k++)
111                         {
112
113                             conv +=
114                                 (window[k] * weights_conv1[k + 7 * 7 * filter]);
115                         }
116
117                         if (conv < 0.0f)
118                             conv = 0.0f;
119
120                         output1[filter * (DIM_1 + 2) * (DIM_1 + 2) + ((i / 2 + 1) * (
                                DIM_1 + 2) + (j / 2 + 1))] = conv;
121                     }
122                 }
123             }
124         }
125 }
```

Listing 3.1: Código em HLS da primeira camada de convolução ResNet

O código da implementação em HLS apresenta peculiaridades em relação ao da implementação em C. Primeiramente, é preciso especificar como a arquitetura RTL que será gerada irá interagir com outros componentes de hardware. Isto é feito através da especificação de interface, como é apresentado nas linhas 11 a 19. Tais especificações indicam para o compilador HLS que os dados de entrada e saída do projeto, ou seja, da primeira camada de convolução da ResNet, seguirá o protocolo de comunicação AXI (*Advanced eXtensible Interface*), que faz parte da família de protocolos AMBA (*Advanced Microcontroller Bus Architecture*). Isto possibilita que o circuito possa se comunicar, por exemplo, com um microprocessador central que também entenda o protocolo AXI.

A linha 19, especifica que o vetor *window* pode ser particionado em vários registradores, visto que seu tamanho é pequeno. Isto pode contribuir para melhorar o desempenho da arquitetura. Mais adiante, as linhas 23 e 33 descrevem como os dados serão transferidos para a camada de convolução 1 já em FPGA. Existem 3 modos de operação: 1. inicialização da memória com a imagem de entrada e os pesos pré-treinados, 2. transferência dos resultados e 3. execução da convolução em si.

Por fim, todas as diretivas *#pragma HLS PIPELINE* indicam que o compilador HLS deve tentar gerar a arquitetura considerando a execução em estágios. Isto é feito automaticamente pelo compilador e, em caso de sucesso, contribui drasticamente para melhorar o desempenho do design, como será apresentado no capítulo seguinte.

Capítulo 4

Análise e Resultados

As experimentações da rede neural construída foram dadas em dois momentos diferentes. A primeira delas consistia em testar camada de convolução existente, comparando os resultados obtidos na rede em *Python* e em C. Isso era feito colocando todos os pesos da rede como 1, e comparando as saídas, como dito. Essa etapa do desenvolvimento não dependia de uma correta classificação, haja visto que o mais importante era calcular corretamente, dada qualquer imagem, com esses pesos que nem se quer foram treinados. Para esses testes, foi utilizada a Figura 4.1.

Figura 4.1: Imagem utilizada para a primeira parte da construção da rede neural.

O formato da Figura 4.1 é PNM, pois a intenção da programação em C é passar para uma FPGA em seguida, assim como dito previamente nesse trabalho. Para abertura de imagens em formatos codificados, como por exemplo JPEG, seria necessário o carregamento ou desenvolvimento de uma biblioteca em C que fizesse a decodificação. No formato PNM há apenas um pequeno cabeçalho e, em seguida, os próprios bits das imagens, o que facilita o processamento na FPGA.

Como o *Tensorflow* não é capaz de processar imagens em formato PNM, ele deve receber arquivos JPEG. A imagem escolhida possuía, portanto, uma versão JPEG utilizada no *Python* e outra PNM para o C. Apesar dessas diferenças de formatos, isso não implicou em mudança nos resultados, pois os valores em *bits* de cada *pixel* das imagens não são modificados, mesmo que

as extensões sejam distintas. Após validação inicial dos processos matemáticos da convolução, o *dataset* de cães e gatos passou a ser usado, visando alcançar a probabilidade correta na classificação.

Já a segunda etapa, com a confirmação que as camadas de convolução estavam de acordo em ambos os ambientes, consistia no treinamento da rede para a elaboração de uma matriz de pesos condizente com o intuito da rede, que é de classificar as imagens passadas em cães ou gatos. Para esse procedimento de treinamento e, consequentemente de classificação, foi utilizado um *dataset* de cães e gatos fornecido pela *Kaggle* [1].

Ele possui tamanho de 217 MB, e sua estrutura interna divide aquelas imagens que servirão de treino e de teste. Em termos quantitativos, há oito mil imagens para treinamento e outras duas mil para fins de teste. Salienta-se que, como são duas classes diferentes, cada uma delas ficou com um valor igual de figuras. Isso ocorre para não criar vícios na rede e consequentemente melhorar a classificação que a rede dará.

4.1 Resultados da convolução

Para validar as operações matemáticas que constituem a ResNet, faz-se necessária mais uma etapa: que o código implementado classifique corretamente uma imagem entre cachorro ou gato com os pesos treinados para tal. Para que isso fosse possível, é essencial a existência de pesos devidamente treinados. A API *Tensorflow* foi utilizada para esse propósito, uma vez que possibilitou a geração dessas informações. Ao final deste processo, os pesos foram salvos em arquivos no formato TXT, e carregados no programa construído em C.

O objetivo deste trabalho não é obter uma boa classificação para imagens de cães e de gatos via *ResNet*, mas sim executar a rede em uma FPGA, para ver como uma rede neural desse tipo se comporta. É um estudo geral de inferência em uma FPGA. Sendo assim, o mais importante é a correta execução da etapas que constituem o processo da rede neural como um todo. Para gerar uma representação fidedigna dela, verificou-se através de uma imagem de cachorro e outra de gato se os resultados após cada convolução eram minimamente próximos. As imagens usadas para esse teste de classificação são mostradas nas Figuras 4.2 e 4.3.

Infelizmente, o carregamento dos pesos em C não era feito com tantas casas decimais quanto o *Python* era capaz de gerar, fazendo com que houvesse um leve arredondamento. Isso causou uma leve diferença entre os resultados comparativos entre as redes. Entretanto, como a diferença entre ambos não é grande e a probabilidade da rede neural convolucional classificar as Figuras 4.2 e 4.3 é muito similar a probabilidade encontrada pelo *Tensorflow*, pode-se inferir que a rede construída em C consegue utilizar os pesos para fazer a classificação com efetividade.

[1] Disponível em: https://www.kaggle.com/chetankv/dogs-cats-images

Figura 4.2: Imagem de cachorro utilizada para testes de classificação de imagens nas duas redes.

Figura 4.3: Imagem de gato utilizada para testes de classificação de imagens nas duas redes.

Tabela 4.1: Valores encontrados na saída da *ResNet* para classificação de cão e gato.

Resultados obtidos		
Linguagem	Cão	Gato
Python (TensorFlow)	1,0	0,0
C	0,999666	0,011865

4.2 Implementação em FPGA

Como dito na Sessão 3.4, não foi possível utilizar a FPGA de fato. Logo, para fins de pesquisa, parte-se para a síntese da descrição para RTL, o que possibilita uma pré-análise dos requisitos em FPGA e consequentemente um entendimento sobre os processos que ocorrerão quando a rede for fielmente implementada na placa. É importante frisar que a implementação aqui dita não se trata de uma simulação e nem de uma emulação, mas de uma implementação literal de como é em uma FPGA. Ambos os resultados para comparação foram obtidos através desse procedimento, dividida em duas etapas: uma com *pipeline* e outra sem da primeira etapa de convolução, denominada "conv1"na Tabela 4.1. Os mais importantes aspectos acerca da utilização de tal paralelismo para inferência são o tempo de processamento e o aproveitamento dos recursos disponibilizados pela

FPGA. A diferença de latência foi evidente, como mostrado na Tabela 4.2. Sem esse recurso, a latência absoluta atingiu o valor de 6,535s. Esse resultado é cerca de 32,51 vezes maior que o valor máximo atingido pela FPGA rodando as operações matemáticas de forma paralela.

Tabela 4.2: Valores de latência encontrados na simulação em HLS.

Latência absoluta		
	Mínima	Máxima
Sem *pipeline*	0.561ms	6.535s
Com *pipeline*	0.561ms	0.201s

O paralelismo é possível porque a FPGA ainda possui recursos extras ao processar as informações em série. Esse excedente, portanto, é utilizado no paralelismo, que fica evidente ao se analisar os dados apresentados na Tabela 4.3. Nela, estão mostradas as quantidades de unidades de processamento de fontes de recursos da FPGA que servem como unidade de lógica aritmética (DSP48E), os *flip-flops* (FF) e unidades de transformação de *arrays* que facilitam seus processamentos (LUT). Apenas a LUT não usa o dobro desses recursos ao ser testado o paralelismo, e ainda assim fica próximo disso.

Tabela 4.3: utilização dos recursos na simulação em HLS.

	DSP48E	FF	LUT
Sem *pipeline*	6	4114	5555
Com *pipeline*	14	9348	9887

Outro importante recurso da FPGA é o armazenamento das *Block RAMs*. Em ambas as situações, verificou-se que o número de blocos de memória dedicados aos processos foi o mesmo: 2186 unidades, estabelecendo uma quantidade mínima de *Block RAMs*. Isso faz sentido, pois o impacto esperado pela FPGA é de tempo de processamento, e não em reduzir o armazenamento. Em posse dessa informação, ficou claro que a FPGA disponível para a construção da ResNet não comportaria a rede, dada a limitação desse recurso.

Capítulo 5

Conclusão

A partir dos resultados apresentados, conclui-se que o modelo de construção de redes neurais convolucionais em C é possível, além da confecção desse tipo de tecnologia em FPGA. Isso garante, futuramente, um aprimoramento no que tange a inferência realizada por ela.

Uma das melhorias em relação à implementação em FPGA é a possibilidade do uso de paralelismo nos processos que estão sendo carregados. Apesar do maior esforço computacional, percebe-se que a latência é quase 33 vezes menor do que em um modelo sem *pipeline*. Esses recursos são próprios da FPGA, e quando utilizados de forma ótima fazem com que de fato haja uma melhora crucial no processamento do código.

Além disso, o presente trabalho se mostra uma evidente fonte de aprendizado, uma vez que possibilitou a adoção dos conceitos que estão por detrás da construção de redes neurais convolucionais, e de maneira manual.

Também reforça a importância do uso de APIs, como o *Tensorflow*, no que tange a aceleração no processo de construção dessas redes. Isso fica evidente ao se comparar o tamanho dos códigos obtidos em cada um dos ambientes. Isso aumenta ainda mais a relevância do trabalho, que trouxe para o ambiente da FPGA uma forma prática de se utilizar uma ResNet para inferência.

5.1 Trabalhos Futuros

Infelizmente não foi possível fazer a testagem e análise de dados provenientes de uma FPGA de fato. O primeiro passo a ser tomado é encontrar uma FPGA que possibilite essa implementação, e assim analisar as métricas dadas por ela. Com ela, será possível implementar toda a ResNet, e assim fazer todos os testes necessários, não apenas uma camada, embora já seja possível começar a ver os efeitos da FPGA com essa camada.

Referências

AGARAP, Abien Fred M. Deep Learning using Rectified Linear Units (ReLU). **Cornell University**, p. 1–7, 2019. Disponível em: <https://arxiv.org/pdf/1803.08375.pdf>.

ALMEIDA, Carlos Caetano de. Arquitetura do módulo de convolução para visão computacional baseada em FPGA. **Biblioteca Brasileira Digital de Teses e Dissertações**, p. 23–24, 2015. Disponível em: <http://repositorio.unicamp.br/jspui/handle/REPOSIP/265780>.

BECKER, Dans. **Rectified Linear Units (ReLU) in Deep Learning**. 2018. Disponível em: <https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning>. Acesso em: 7 mai. 2021.

CAMACHO, Cezanne. **Convolutional Neural Networks**. 2018. Disponível em: <https://cezannec.github.io/Convolutional_Neural_Networks/>. Acesso em: 13 mai. 2021.

DANCKAERT, Koen; CATTHOOR, Francky; DE MAN, Hugo. System-level memory management for weakly parallel image processing. In: ________. **Euro-Par'96 Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. p. 217–225. ISBN 978-3-540-70636-6.

DOSHI, Sanket. **Convolutional Neural Network: Learn And Apply**. 2019. Disponível em: <https://medium.com/@sdoshi579/convolutional-neural-network-learn-and-apply-3dac9acfe2b6>. Acesso em: 30 abr. 2021.

GANESH, Prakhar. **Types of Convolution Kernels : Simplified**. 2019. Disponível em: <https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37>. Acesso em: 13 mai. 2021.

HAT, Red. **O que é open source?** Disponível em: <https://www.redhat.com/pt-br/topics/open-source/what-is-open-source>. Acesso em: 22 abr. 2018.

HE, Kaiming et al. Deep Residual Learning for Image Recognition. In: PROCEEDINGS of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). [S.l.: s.n.], jun. 2016.

JORGE, Carlos; NERY, Alexandre; MELO, Alba. Uma implementação do algoritmo LCS em FPGA usando High-Level Synthesis. In: ANAIS do XX Simpósio em Sistemas Computacionais de Alto Desempenho. Campo Grande: SBC, 2019. p. 324–333. DOI: 10.5753/wscad.2019.8679. Disponível em: <https://sol.sbc.org.br/index.php/wscad/article/view/8679>.

KUON, Ian; ROSE, Jonathan. Measuring the Gap Between FPGAs and ASICs. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 26, n. 2, p. 203–215, 2007. DOI: 10.1109/TCAD.2006.884574.

NAPOLETANO, Paolo; PICCOLI, Flavio; SCHETTINI, Raimondo. Anomaly Detection in Nanofibrous Materials by CNN-Based Self-Similarity. **Sensors (Basel, Switzerland)**, v. 18, jan. 2018. DOI: 10.3390/s18010209.

NURVITADHI, Eriko et al. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In: ACM. 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. California, USA: ACM, 2017. Disponível em: <https://dl.acm.org/doi/abs/10.1145/3020078.3021740>.

PLOEG, Atze van der. **Why use an FPGA instead of a CPU or GPU?** 2018. Disponível em: <https://blog.esciencecenter.nl/why-use-an-fpga-instead-of-a-cpu-or-gpu-b234cd4f309c>. Acesso em: 18 abr. 2021.

QASAIMEH, Murad et al. Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels. **2019 IEEE International Conference on Embedded Software and Systems (ICESS)**, p. 1–8, 2019.

SANTOS KOROL, Guilherme dos. **An FPGA Implementation for Convolutional Neural Network**. 2019. Disponível em: <https://www.inf.pucrs.br/moraes/docs/tcc/tcc_korol.pdf>. Acesso em: 2 mai. 2021.

SHARMA, Sagar. **Activation Functions in Neural Networks**. 2017. Disponível em: <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>. Acesso em: 7 mai. 2021.

SOUGHT, Programmer. **Use of shortcut in deep convolutional neural network CNN**. 2018. Disponível em: <https://www.programmersought.com/article/3438854082/>. Acesso em: 7 mai. 2021.

SZE, Vivienne et al. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. **Proceedings of the IEEE**, v. 105, n. 12, p. 2295–2329, 2017. DOI: 10.1109/JPROC.2017.2761740.

TEAM, SuperDataSciente. **Convolutional Neural Networks (CNN): Step 1(b) - ReLU Layer**. 2018. Disponível em: <https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-1b-relu-layer/>. Acesso em: 3 mai. 2021.

TENSORFLOW. **Introdução ao TensorFlow**. 2021. Disponível em: <https://www.tensorflow.org/learn?hl=pt-br>. Acesso em: 18 abr. 2021.

WOOD, Thomas. **What is the Softmax Function?** 2021. Disponível em: <https://deepai.org/machine-learning-glossary-and-terms/softmax-layer>. Acesso em: 6 mai. 2021.

XILINX. **DSP48 Block**. 2017. Disponível em: <https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/uwa1504034294196.html#:~:text=The%5C%20DSP48%5C%20block%5C%20is%5C%20an,add%5C%2Fsubtract%5C%2Faccumulate%5C%20engine.>. Acesso em: 14 mai. 2021.

________. **Introduction to FPGA Design with Vivado High-Level Synthesis (UG998)**. 2019. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf>. Acesso em: 14 mai. 2021.

________. **Vivado Design Suite User Guide: High-Level Synthesis (UG902)**. 2018. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf>. Acesso em: 14 mai. 2021.

ANEXOS

ANEXO I

Códigos fonte

I.1 *Python*

Essa seção aborda a implementação da rede neural convolucional, utilizando os recursos fornecidos pelo *Tensorflow*. As subseções segmentam o código, com o intuito de facilitar o entendimento.

I.1.1 Importações

```
import tensorflow as tf
import ImageDataGenerator
import matplotlib.pyplot as plt
import ModelCheckpoint, CSVLogger
import cv2 as cv
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.preprocessing.image
from tensorflow.keras.callbacks
```

I.1.2 Recebimento das imagens

```
outgen = datagen.flow_from_directory(
        directory="/home/antonio/Imagens/gray/",
        class_mode='binary',
        color_mode='grayscale',
        batch_size=1,
        target_size=(224,224))
```

I.1.3 Camadas

Cada uma das camadas implementadas estão disponíveis nas seções abaixo:

I.1.3.1 Camada 1

```
x = layers.ZeroPadding2D((3,3))(inputs)
x = layers.Conv2D(64, (7, 7), (2, 2),
    padding='valid', use_bias=False, name='conv1')(x)
```

I.1.3.2 Camada 2

```
x = layers.ZeroPadding2D((1,1))(x)
x1 = layers.MaxPooling2D((3, 3), strides=(2,2), padding='valid')(x)
x = layers.ZeroPadding2D((1,1))(x1)
x = layers.Conv2D(64, (3, 3), padding='valid', use_bias=False,
    name='conv2_1a')(x)
x = layers.ZeroPadding2D((1,1))(x)
x2 = layers.Conv2D(64, (3, 3), padding='valid', use_bias=False,
    name='conv2_2a')(x) added = layers.add([x1,x2])
x = layers.ZeroPadding2D((1,1))(added)
x = layers.Conv2D(64, (3, 3), padding='valid', use_bias=False,
    name='conv2_1b')(x)
x = layers.ZeroPadding2D((1,1))(x)
x3 = layers.Conv2D(64, (3, 3), padding='valid',
    use_bias=False, name='conv2_2b')(x) added = layers.add([added,x3])
```

I.1.3.3 Camada 3

```
sc = layers.Conv2D(128, (1, 1), (2, 2), padding='valid', use_bias=False,
    name='shortcut3')(added)
x = layers.ZeroPadding2D((1, 1))(added)
x = layers.Conv2D(128, (3, 3), (2, 2), padding='valid', use_bias=False,
    name='conv3_1a')(x)
x = layers.ZeroPadding2D((1,1))(x)
x2 = layers.Conv2D(128, (3, 3), padding='valid', use_bias=False,
    name='conv3_2a')(x)
added = layers.add([sc,x2])
x = layers.ZeroPadding2D((1,1))(added)
x = layers.Conv2D(128, (3, 3), padding='valid', use_bias=False,
    name='conv3_1b')(x)
x = layers.ZeroPadding2D((1,1))(x)
x3 = layers.Conv2D(128, (3, 3), padding='valid', use_bias=False,
    name='conv3_2b')(x)
added = layers.add([added,x3])
x = layers.ZeroPadding2D((1,1))(added)
```

I.1.3.4 Camada 4

```
sc = layers.Conv2D(256, (1, 1), (2, 2), padding='valid',
    use_bias=False, name='shortcut4')(added)
x = layers.ZeroPadding2D((1, 1))(added)
x = layers.Conv2D(256, (3, 3), (2, 2), padding='valid', use_bias=False,
    name='conv4_1a')(x)
x = layers.ZeroPadding2D((1,1))(x)
x2 = layers.Conv2D(256, (3, 3), padding='valid', use_bias=False,
    name='conv4_2a')(x)
added = layers.add([sc,x2])

x = layers.ZeroPadding2D((1,1))(added)
x = layers.Conv2D(256, (3, 3), padding='valid', use_bias=False,
    name='conv4_1b')(x)
x = layers.ZeroPadding2D((1,1))(x)
x3 = layers.Conv2D(256, (3, 3), padding='valid', use_bias=False,
    name='conv4_2b')(x)
added = layers.add([added,x3])
x = layers.ZeroPadding2D((1,1))(added)
```

I.1.3.5 Camada 5

```
sc = layers.Conv2D(512, (1, 1), (2, 2), padding='valid',
    use_bias=False, name='shortcut5')(added)
x = layers.ZeroPadding2D((1, 1))(added)
x = layers.Conv2D(512, (3, 3), (2, 2), padding='valid', use_bias=False,
    name='conv5_1a')(x)
x = layers.ZeroPadding2D((1,1))(x)
x2 = layers.Conv2D(512, (3, 3), padding='valid', use_bias=False,
```

```
     name='conv5_2a')(x)
added = layers.add([sc,x2])
x = layers.ZeroPadding2D((1,1))(added)
x = layers.Conv2D(512, (3, 3), padding='valid', use_bias=False,
     name='conv5_1b')(x)
x = layers.ZeroPadding2D((1,1))(x)
x3 = layers.Conv2D(512, (3, 3), padding='valid', use_bias=False,
     name='conv5_2b')(x)
added = layers.add([added,x3])
x = layers.ZeroPadding2D((1,1))(added)
```

I.1.3.6 Passos finais

```
output = layers.AveragePooling2D((7,7))(added)
flat = layers.Flatten()(output)
result = layers.Dense(1, use_bias=False)(flat)
```

I.2 C

Essa seção aborda a implementação da rede neural convolucional, utilizando a linguagem C. As subseções segmentam o código, com o intuito de facilitar o entendimento.

I.2.1 Importações e definições prévias

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define N_FILTERS_64 64
#define N_FILTERS_128 128
#define N_FILTERS_256 256
#define N_FILTERS_512 512

#define DIM_0 224
#define DIM_1 112
#define DIM_2 56
#define DIM_3 28
#define DIM_4 14
#define DIM_5 7

FILE *picture, *weights;

static int image[(DIM_0+6)*(DIM_0+6)];
static float output1[(DIM_1+2)*(DIM_1+2)*N_FILTERS_64];
static float output2[(DIM_2+2)*(DIM_2+2)*N_FILTERS_64];
static float output3[(DIM_3+2)*(DIM_3+2)*N_FILTERS_128];
static float shortcut3[(DIM_3+2)*(DIM_3+2)*N_FILTERS_128];
static float output4[(DIM_4+2)*(DIM_4+2)*N_FILTERS_256];
static float shortcut4[(DIM_4+2)*(DIM_4+2)*N_FILTERS_256];
static float output5[(DIM_5+2)*(DIM_5+2)*N_FILTERS_512];
static float shortcut5[(DIM_5+2)*(DIM_5+2)*N_FILTERS_512];

float output2_temp[(DIM_2+2)*(DIM_2+2)*N_FILTERS_64];
float output2_temp2[(DIM_2+2)*(DIM_2+2)*N_FILTERS_64];
float output3_temp[(DIM_3+2)*(DIM_3+2)*N_FILTERS_128];
float output3_temp2[(DIM_3+2)*(DIM_3+2)*N_FILTERS_128];
float output4_temp[(DIM_4+2)*(DIM_4+2)*N_FILTERS_256];
float output4_temp2[(DIM_4+2)*(DIM_4+2)*N_FILTERS_256];
float output5_temp[(DIM_5+2)*(DIM_5+2)*N_FILTERS_512];
float output5_temp2[(DIM_5+2)*(DIM_5+2)*N_FILTERS_512];
float output_average[N_FILTERS_512];

static float weights_conv1[N_FILTERS_64*7*7];

static float weights_conv2_layer1a[N_FILTERS_64*3*3*N_FILTERS_64];
static float weights_conv2_layer2a[N_FILTERS_64*3*3*N_FILTERS_64];
```

```
static float weights_conv2_layer1b[N_FILTERS_64*3*3*N_FILTERS_64];
static float weights_conv2_layer2b[N_FILTERS_64*3*3*N_FILTERS_64];

static float weights_conv3_layer1a[N_FILTERS_128*3*3*N_FILTERS_64];
static float weights_conv3_layer2a[N_FILTERS_128*3*3*N_FILTERS_128];
static float weights_conv3_layer1b[N_FILTERS_128*3*3*N_FILTERS_128];
static float weights_conv3_layer2b[N_FILTERS_128*3*3*N_FILTERS_128];
static float weights_conv3_shortcut[N_FILTERS_128*N_FILTERS_64];

static float weights_conv4_layer1a[N_FILTERS_256*3*3*N_FILTERS_128];
static float weights_conv4_layer1b[N_FILTERS_256*3*3*N_FILTERS_256];
static float weights_conv4_layer2a[N_FILTERS_256*3*3*N_FILTERS_256];
static float weights_conv4_layer2b[N_FILTERS_256*3*3*N_FILTERS_256];
static float weights_conv4_shortcut[N_FILTERS_256*N_FILTERS_128];

static float weights_conv5_layer1a[N_FILTERS_512*3*3*N_FILTERS_256];
static float weights_conv5_layer1b[N_FILTERS_512*3*3*N_FILTERS_512];
static float weights_conv5_layer2a[N_FILTERS_512*3*3*N_FILTERS_512];
static float weights_conv5_layer2b[N_FILTERS_512*3*3*N_FILTERS_512];
static float weights_conv5_shortcut[N_FILTERS_512*N_FILTERS_256];

static float weights_fully[N_FILTERS_512];

static float pooling_conv2[(DIM_2+2)*(DIM_2+2)*N_FILTERS_64];
```

I.2.2 Recebimento das imagens e matrizes de pesos

```
FILE *weights_1 = fopen("weights-conv1.txt", "r");
    int counter = 0;
    int n = 0;
    float w;
    //char buffer[N_FILTERS_64*7*7*10];
    char *buffer = (char *)malloc(N_FILTERS_64*7*7*30*sizeof(char));

    do
    {
        fgets(buffer, N_FILTERS_64*7*7*30, weights_1);
    }while(atof(buffer) == 0);

    char* token = strtok(buffer, "*");

    while(token != NULL)
    {
        w = atof(token);
        weights_conv1[counter] = w;

        if(n < 63){
            counter = counter + 7*7;
            n++;
        }else if(n == 63){
            counter = counter - n*7*7 + 1;
            n = 0;
        }

        token = strtok(NULL, "*");
    }
    printf("1: %f - %f [%i]\n", weights_conv1[0], weights_conv1[1],
    counter);
```

I.2.3 Camadas de convolução

I.2.3.1 *Zero Padding*

```
int c;
    int n = 0;
    int line = 0;
    int count = 0;
    int counter = 3*(DIM_0+6) + 3;
    bool isPadding = 1;
    float w;

    if (picture != NULL)
```

```
10      {
11          for(int i = 0; i < 6; i++)
12          {
13              for(int j = 0; j < 230; j++)
14              {
15                  if(i < 3)
16                  {
17                      image[i*230 + j] = 0;
18                      count++;
19                  }
20                  else image[(i+224)*230 + j] = 0;
21              }
22          }
23
24          char *buffer = (char *)malloc(DIM_0*DIM_0*30*sizeof(char));
25
26          do
27          {
28              fgets(buffer, DIM_0*DIM_0*30,
29              picture);
30          }while(atof(buffer) == 0);
31
32          char* token = strtok(buffer, ".");
33
34          while(token != NULL)
35          {
36              w = atof(token);
37              image[counter] = w;
38
39              n = counter % (DIM_0 + 6);
40              if(n == 226) counter = counter + 7;
41              else counter++;
42
43              token = strtok(NULL, ".");
44          }
45
46
47          image[52208] = 0;
48      }
```

I.2.3.2 Convoluções

```
1       bool first = true;
2       int filter, id = 0;
3       int counter = 0;
4       float conv;
5       float window[7*7];
6
7       for(filter = 0; filter < N_FILTERS_64; filter++)
8       {
9           for(int i = 0; i < DIM_0; i = i + 2)
10          {
11              for(int j = 0; j < DIM_0; j = j + 2)
12              {
13                  window[0] = image[(i+0)*(DIM_0+6) + j+0];
14                  window[1] = image[(i+0)*(DIM_0+6) + j+1];
15                  window[2] = image[(i+0)*(DIM_0+6) + j+2];
16                  window[3] = image[(i+0)*(DIM_0+6) + j+3];
17                  window[4] = image[(i+0)*(DIM_0+6) + j+4];
18                  window[5] = image[(i+0)*(DIM_0+6) + j+5];
19                  window[6] = image[(i+0)*(DIM_0+6) + j+6];
20
21                  window[7] = image[(i+1)*(DIM_0+6) + j+0];
22                  window[8] = image[(i+1)*(DIM_0+6) + j+1];
23                  window[9] = image[(i+1)*(DIM_0+6) + j+2];
24                  window[10] = image[(i+1)*(DIM_0+6) + j+3];
25                  window[11] = image[(i+1)*(DIM_0+6) + j+4];
26                  window[12] = image[(i+1)*(DIM_0+6) + j+5];
27                  window[13] = image[(i+1)*(DIM_0+6) + j+6];
28
29                  window[14] = image[(i+2)*(DIM_0+6) + j+0];
30                  window[15] = image[(i+2)*(DIM_0+6) + j+1];
31                  window[16] = image[(i+2)*(DIM_0+6) + j+2];
32                  window[17] = image[(i+2)*(DIM_0+6) + j+3];
33                  window[18] = image[(i+2)*(DIM_0+6) + j+4];
34                  window[19] = image[(i+2)*(DIM_0+6) + j+5];
35                  window[20] = image[(i+2)*(DIM_0+6) + j+6];
```

```
                    window[21] = image[(i+3)*(DIM_0+6) + j+0];
                    window[22] = image[(i+3)*(DIM_0+6) + j+1];
                    window[23] = image[(i+3)*(DIM_0+6) + j+2];
                    window[24] = image[(i+3)*(DIM_0+6) + j+3];
                    window[25] = image[(i+3)*(DIM_0+6) + j+4];
                    window[26] = image[(i+3)*(DIM_0+6) + j+5];
                    window[27] = image[(i+3)*(DIM_0+6) + j+6];

                    window[28] = image[(i+4)*(DIM_0+6) + j+0];
                    window[29] = image[(i+4)*(DIM_0+6) + j+1];
                    window[30] = image[(i+4)*(DIM_0+6) + j+2];
                    window[31] = image[(i+4)*(DIM_0+6) + j+3];
                    window[32] = image[(i+4)*(DIM_0+6) + j+4];
                    window[33] = image[(i+4)*(DIM_0+6) + j+5];
                    window[34] = image[(i+4)*(DIM_0+6) + j+6];

                    window[35] = image[(i+5)*(DIM_0+6) + j+0];
                    window[36] = image[(i+5)*(DIM_0+6) + j+1];
                    window[37] = image[(i+5)*(DIM_0+6) + j+2];
                    window[38] = image[(i+5)*(DIM_0+6) + j+3];
                    window[39] = image[(i+5)*(DIM_0+6) + j+4];
                    window[40] = image[(i+5)*(DIM_0+6) + j+5];
                    window[41] = image[(i+5)*(DIM_0+6) + j+6];

                    window[42] = image[(i+6)*(DIM_0+6) + j+0];
                    window[43] = image[(i+6)*(DIM_0+6) + j+1];
                    window[44] = image[(i+6)*(DIM_0+6) + j+2];
                    window[45] = image[(i+6)*(DIM_0+6) + j+3];
                    window[46] = image[(i+6)*(DIM_0+6) + j+4];
                    window[47] = image[(i+6)*(DIM_0+6) + j+5];
                    window[48] = image[(i+6)*(DIM_0+6) + j+6];

                    conv = 0.0f;

                    for(int k = 0; k < 7*7; k++)
                    {
                        conv = conv + (window[k] * weights_conv1[k +
                        7*7*filter]);
                    }

                    if(conv < 0) conv = 0;

                    output1[filter*(DIM_1+2)*(DIM_1+2) +
                    ((i/2 + 1)*(DIM_1+2) + (j/2 + 1))] = conv;
                }
            }
        }
```

I.2.3.3 *Max Pooling*

```
float window[3*3];
    float max_pool;

    for (int depth = 0; depth < N_FILTERS_64; depth++)
    {
        for (int i = 0; i < DIM_1; i = i + 2)
        {
            for (int j = 0; j < DIM_1; j = j + 2)
            {
                window[0] = output1[(depth*(DIM_1+2)*(DIM_1+2)) +
                (i+0)*(DIM_1+2) + j+0];
                window[1] = output1[(depth*(DIM_1+2)*(DIM_1+2)) +
                (i+0)*(DIM_1+2) + j+1];
                window[2] = output1[(depth*(DIM_1+2)*(DIM_1+2)) +
                (i+0)*(DIM_1+2) + j+2];

                window[3] = output1[(depth*(DIM_1+2)*(DIM_1+2)) +
                (i+1)*(DIM_1+2) + j+0];
                window[4] = output1[(depth*(DIM_1+2)*(DIM_1+2)) +
                (i+1)*(DIM_1+2) + j+1];
                window[5] = output1[(depth*(DIM_1+2)*(DIM_1+2)) +
                (i+1)*(DIM_1+2) + j+2];

                window[6] = output1[(depth*(DIM_1+2)*(DIM_1+2)) +
                (i+2)*(DIM_1+2) + j+0];
                window[7] = output1[(depth*(DIM_1+2)*(DIM_1+2)) +
```

```
                (i+2)*(DIM_1+2) + j+1];
                window[8] = output1[(depth*(DIM_1+2)*(DIM_1+2)) +
                (i+2)*(DIM_1+2) + j+2];

                max_pool = __LDBL_MIN_EXP__;

                for (int pool = 0; pool < 3*3; pool++)
                {
                    if(window[pool] > max_pool) max_pool =
                    window[pool];
                }
                pooling_conv2[(depth*(DIM_2+2)*(DIM_2+2)) +
                (i/2 + 1)*(DIM_2+2) + (j/2 + 1)] = max_pool;
            }

        }

    }
```

I.2.4 *Average Layer*

```
    float nums[DIM_5*DIM_5];
    float result;

    for(int filter = 0; filter < N_FILTERS_512; filter++)
    {
        for(int i = 0; i < DIM_5; i++)
        {
            for(int j = 0; j < DIM_5; j++)
            {
                nums[i*DIM_5 + j] = output5[(i+1)*(DIM_5+2) + (j+1) +
                filter*(DIM_5+2)*(DIM_5+2)];
            }
        }

        result = 0;

        for(int k = 0; k < DIM_5*DIM_5; k++)
        {
            result = result + nums[k];
        }

        output_average[filter] = result/(DIM_5*DIM_5);
    }
```

I.2.5 *Fully Connected*

```
    int count;
    int one = 0, zero = 0;
    float result = 0;

    for(count = 0; count < N_FILTERS_512; count++)
    {
        result = result + output_average[count]*weights_fully[count];
    }

    result = 1/(1 + exp(-1*result));

    return result;
```

I.3 FPGA

I.3.1 HPP

```
#ifndef CONV1_LAYER_HPP
#define CONV1_LAYER_HPP

```

```
#define N_FILTERS_64 64
#define N_FILTERS_128 128
#define N_FILTERS_256 256
#define N_FILTERS_512 512

#define DIM_0 224
#define DIM_1 112
#define DIM_2 56
#define DIM_3 28
#define DIM_4 14
#define DIM_5 7

#define INIT_MEM 0
#define READ_RESULT 1

void conv1layer_accel(volatile int *im, volatile float *wconv1,
volatile float *output, int mode );

#endif
```

I.3.2 CPP

```
#include "conv1_layer.hpp"

static int image[(DIM_0 + 6) * (DIM_0 + 6)];
static float weights_conv1[N_FILTERS_64 * 7 * 7];
static float output1[(DIM_1 + 2) * (DIM_1 + 2) * N_FILTERS_64];
static float window[7 * 7];

void conv1layer_accel(volatile int *im, volatile float *wconv1,
volatile float *output, int mode )
{

#pragma HLS INTERFACE m_axi depth=90000 port=im offset=slave
#pragma HLS INTERFACE m_axi depth=3136 port=wconv1 offset=slave
#pragma HLS INTERFACE m_axi depth=831744 port=output offset=slave
#pragma HLS INTERFACE s_axilite port=im bundle=Ctrl
#pragma HLS INTERFACE s_axilite port=wconv1 bundle=Ctrl
#pragma HLS INTERFACE s_axilite port=output bundle=Ctrl
#pragma HLS INTERFACE s_axilite port=mode bundle=Ctrl
#pragma HLS INTERFACE s_axilite port=return bundle=Ctrl
#pragma HLS ARRAY_PARTITION variable=window complete dim=1

  float conv;

  if (mode == INIT_MEM) {
    for(int k = 0 ; k < (DIM_0 + 6) * (DIM_0 + 6) ; k++)
      #pragma HLS PIPELINE
      image[k] = im[k];

    for(int k = 0 ; k < N_FILTERS_64 * 7 * 7 ; k++)
      #pragma HLS PIPELINE
      weights_conv1[k] = wconv1[k];

  } else if (mode == READ_RESULT) {
    for(int k = 0 ; k < (DIM_1 + 2) * (DIM_1 + 2) *
    N_FILTERS_64 ; k++)
      #pragma HLS PIPELINE
      output[k] = output1[k];

  } else {

    for (int filter = 0; filter < N_FILTERS_64; filter++) {
      for (int i = 0; i < DIM_0; i = i + 2) {
        for (int j = 0; j < DIM_0; j = j + 2) {

      #pragma HLS PIPELINE
    window[0] = image[(i + 0) * (DIM_0 + 6) + j + 0];
    window[1] = image[(i + 0) * (DIM_0 + 6) + j + 1];
    window[2] = image[(i + 0) * (DIM_0 + 6) + j + 2];
    window[3] = image[(i + 0) * (DIM_0 + 6) + j + 3];
    window[4] = image[(i + 0) * (DIM_0 + 6) + j + 4];
    window[5] = image[(i + 0) * (DIM_0 + 6) + j + 5];
    window[6] = image[(i + 0) * (DIM_0 + 6) + j + 6];

    window[7] = image[(i + 1) * (DIM_0 + 6) + j + 0];
    window[8] = image[(i + 1) * (DIM_0 + 6) + j + 1];
    window[9] = image[(i + 1) * (DIM_0 + 6) + j + 2];
    window[10] = image[(i + 1) * (DIM_0 + 6) + j + 3];
```

```
      window[11] = image[(i + 1) * (DIM_0 + 6) + j + 4];
      window[12] = image[(i + 1) * (DIM_0 + 6) + j + 5];
      window[13] = image[(i + 1) * (DIM_0 + 6) + j + 6];

      window[14] = image[(i + 2) * (DIM_0 + 6) + j + 0];
      window[15] = image[(i + 2) * (DIM_0 + 6) + j + 1];
      window[16] = image[(i + 2) * (DIM_0 + 6) + j + 2];
      window[17] = image[(i + 2) * (DIM_0 + 6) + j + 3];
      window[18] = image[(i + 2) * (DIM_0 + 6) + j + 4];
      window[19] = image[(i + 2) * (DIM_0 + 6) + j + 5];
      window[20] = image[(i + 2) * (DIM_0 + 6) + j + 6];

      window[21] = image[(i + 3) * (DIM_0 + 6) + j + 0];
      window[22] = image[(i + 3) * (DIM_0 + 6) + j + 1];
      window[23] = image[(i + 3) * (DIM_0 + 6) + j + 2];
      window[24] = image[(i + 3) * (DIM_0 + 6) + j + 3];
      window[25] = image[(i + 3) * (DIM_0 + 6) + j + 4];
      window[26] = image[(i + 3) * (DIM_0 + 6) + j + 5];
      window[27] = image[(i + 3) * (DIM_0 + 6) + j + 6];

      window[28] = image[(i + 4) * (DIM_0 + 6) + j + 0];
      window[29] = image[(i + 4) * (DIM_0 + 6) + j + 1];
      window[30] = image[(i + 4) * (DIM_0 + 6) + j + 2];
      window[31] = image[(i + 4) * (DIM_0 + 6) + j + 3];
      window[32] = image[(i + 4) * (DIM_0 + 6) + j + 4];
      window[33] = image[(i + 4) * (DIM_0 + 6) + j + 5];
      window[34] = image[(i + 4) * (DIM_0 + 6) + j + 6];

      window[35] = image[(i + 5) * (DIM_0 + 6) + j + 0];
      window[36] = image[(i + 5) * (DIM_0 + 6) + j + 1];
      window[37] = image[(i + 5) * (DIM_0 + 6) + j + 2];
      window[38] = image[(i + 5) * (DIM_0 + 6) + j + 3];
      window[39] = image[(i + 5) * (DIM_0 + 6) + j + 4];
      window[40] = image[(i + 5) * (DIM_0 + 6) + j + 5];
      window[41] = image[(i + 5) * (DIM_0 + 6) + j + 6];

      window[42] = image[(i + 6) * (DIM_0 + 6) + j + 0];
      window[43] = image[(i + 6) * (DIM_0 + 6) + j + 1];
      window[44] = image[(i + 6) * (DIM_0 + 6) + j + 2];
      window[45] = image[(i + 6) * (DIM_0 + 6) + j + 3];
      window[46] = image[(i + 6) * (DIM_0 + 6) + j + 4];
      window[47] = image[(i + 6) * (DIM_0 + 6) + j + 5];
      window[48] = image[(i + 6) * (DIM_0 + 6) + j + 6];

      conv = 0.0f;

      for (int k = 0; k < 7 * 7; k++) {

        conv +=
        (window[k]* weights_conv1[k+ 7 * 7 * filter]);
            }

      if (conv < 0.0f)
      conv = 0.0f;

      output1[filter * (DIM_1 + 2) * (DIM_1 + 2)
      + ((i / 2 + 1) * (DIM_1 + 2) + (j / 2 + 1))] = conv;
          }
        }
      }
    }
}
```