INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia Eletrónica e Telecomunicações e de Computadores

ISEL

Otimização de Redes Neuronais Convolucionais em FPGA com Redução do Tamanho dos Operandos

ANA GONÇALVES

(Licenciada em Engenharia Eletrónica e Telecomunicações e de Computadores)

Trabalho Final de Mestrado para obtenção do grau de Mestre

em Engenharia de Eletrónica e Telecomunicações

Orientador:

Doutor Mário Pereira Véstias

Júri:

Presidente: Professor Doutor António Couto Pinto
Vogal (Arguente): Professor Doutor José João Henriques Teixeira de Sousa
Vogal (Orientador): Professor Doutor Mário Pereira Véstias

Novembro de 2018

Agradecimentos

Em primeiro lugar quero agradecer ao meu orientador Engenheiro Mário Véstias por todo o apoio e disponibilidade durante o desenvolvimento desta dissertação.

Agradeço também ao meu namorado e familiares que me apoiaram e aconselharam durante todo o percurso académico.

Resumo

As Redes Neuronais Convolucionais (conhecidas como CNN - *Convolutional Neural Networks*) são projetadas tendo como inspiração o funcionamento do cérebro e têm obtido grandes avanços nos últimos anos estando a ser amplamente utilizadas na visão computacional, mais especificamente na classificação de imagens. Uma CNN com o propósito de classificação de imagens, após estar implementada e treinada, retorna para cada uma das classes a ser testada a probabilidade dessa imagem pertencer a essa classe.

As CNN são computacionalmente bastante exigentes e requerem uma elevada largura de banda de acesso à memória onde são guardados os pesos da rede, sendo geralmente executadas em sistemas de elevado desempenho. Contudo, a execução de CNN em sistemas embebidos próximos do sistema de recolha de dados evita a comunicação dos dados e o seu processamento em tempo real. Assim, é importante o estudo e desenvolvimento de métodos que permitam a execução de CNN em sistemas embebidos com recursos reduzidos com tempos de execução aceitáveis.

A FPGA (*Field Programmable Gate Array*) é um dispositivo cujo hardware pode ser reprogramado de acordo com as necessidades específicas de cada projeto, permitindo uma implementação bastante eficiente de sistemas embebidos. Quando utilizada na implementação de CNN, a sua arquitetura pode ser configurada de forma dedicada conforme as características da CNN a implementar.

As CNN são bastante exigentes em termos computacionais e de memória, o que dificulta a sua implementação em FPGA de baixo custo com poucos recursos computacionais e de memória. Para reduzir essas necessidades, pode reduzir-se o tamanho dos operandos através de métodos de quantificação utilizando, por exemplo, vírgula fixa dinâmica e reduzindo o número de bits para representar os dados.

O objetivo desta dissertação é investigar sobre o impacto que a redução do tamanho dos operandos da CNN tem sobre a sua precisão, sobre a ocupação de recursos da FPGA e sobre o desempenho do sistema.

Com a utilização deste método verificámos que é possível implementar redes CNN de grande dimensão em FPGA de baixa dimensão (e.g. a ZYNQ7020 considerada neste trabalho) com um desempenho que pode atingir os 600 GOPs (*Giga Operations per second*).

Palavras Chave: Rede Neuronal Convolucional, FPGA, Redução do Tamanho dos Dados, sistema embebido

Abstract

Convolutional Neural Networks (CNN) are designed with the inspiration of brain functioning and have made great strides in recent years being widely used in computer vision, more specifically in the classification of images. A CNN for the purpose of image classification, after being implemented and trained, returns to each one of the classes to be tested the probability of that image belonging to that class.

The FPGA (Field Programmable Gate Array) is a device that can be programmed according to the specific needs of the project. This feature is very useful in the implementation of CNN in FPGA, since its architecture can be programmed in a dedicated way according to the characteristics of the CNN to be implemented.

CNN have high computational and memory requirements not compatible with low cost FPGAs with scarce hardware logic and memory. To reduce these needs, the size of the operands can be reduced using quantization methods, such as dynamic fixed point and reduction of the number of bits to represent the data.

The main goal of this thesis is to research about the impact that the reduction of CNN operand size has on its precision, on the occupation of FPGA resources and on the performance of the system.

Our research and development concluded that it is possible to implement large CNN in low cost FPGAs (e.g. in a ZYNQ7020 considered in this work) with a performance that may achieve 600 GOPs (Giga Operations per second)

Keywords: Convolutional Neural Network, FPGA, Datawidth Reduction, Embedded System

Índice

Índice de Figuras

Índice de Tabelas

Índice de Equações

Lista de Acrónimos

ASIC – *Application Specific Integrated Circuits*

BRAM – *Block Random Access Memory*

CNN – *Convolution Neural Network*

CPU – *Central Processing Unit*

DMA – *Direct Memory Access*

DNN – *Deep Neural Network*

DSP – *Digital Signal Processor*

FLOP –*Floating-Point Operation*

FPGA – *Field-programmable gate array*

GPU – *Graphics Processing Unit*

IA – Inteligência Artificial

ICAN – *Input-recycling Convolutional Array of Neurons*

LUT – *Look-up Tables*

MAC – *Multiply–Accumulate*

NaN – *Not a Number*

OFM – *Output Feature Map*

PE – *Processing-Element*

RAM – *Random Access Memory*

SoC – *System-on-Chip*

Capítulo 1 - Introdução

Com o avanço da tecnologia, a área da inteligência artificial (IA) ganha cada vez mais relevância e levou a um aumento no estudo de diversos temas relacionados com implementação de aplicações de IA. Um dos temas que passou a ter grande destaque é a Rede Neuronal Profunda, (DNN – *Deep Neural Network*). A DNN é amplamente utilizada na visão computacional, pois melhora significativamente a precisão de muitas tarefas da visão computacional, como por exemplo a classificação de imagem, a localização e deteção de objetos, a segmentação de imagens e o reconhecimento de ação. A DNN também é utilizada no reconhecimento de fala, em jogos com grande complexidade, como por exemplo no jogo das damas e no jogo *Go*, na robótica e na medicina, mais especificamente na genética e na análise de imagens médicas.

A arquitetura da rede DNN é inspirada no cérebro humano. O cérebro humano é constituído por aproximadamente 86 biliões de neurónios, sendo estes neurónios responsáveis por transmitir o impulso nervoso. Na constituição do neurónio existe o dendrito (recebe o impulso nervoso) e o axónio (transmite o impulso nervoso vindo do neurónio), sendo que a conexão de um ramo do axónio e do dendrito designa-se por sinapse. A característica principal da sinapse consiste em conseguir escalar um sinal proveniente do dendrito (x_i) (ver Figura 1).

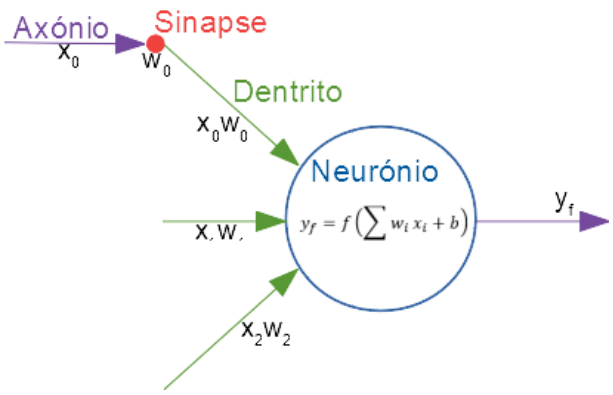


Figura 1 - Conexões do neurónio.

Esse fator de escala é conhecido como peso (w_i), e o cérebro é capaz de aprender através do ajuste dos valores desses pesos. O parâmetro *bias* (b) é um fator de ajuste que permite deslocar o resultado, ou seja, realiza um *offset*. Assim, a forma como o cérebro aprende é uma ótima inspiração para o algoritmo de aprendizagem, pois aprender consiste apenas no ajuste dos pesos em resposta a um estímulo e não na organização do cérebro [2].

Assim, inspiradas no cérebro humano, surgiram as DNN que são constituídas por uma camada de entrada, uma camada de saída e várias camadas ocultas, em que cada camada

modela um conjunto de neurónios. A Rede Neuronal Convolucional (CNN - *Convolution Neural Network*) é um tipo de DNN que tira proveito da correlação existente entre pontos próximos dos dados de entrada da rede, ou seja, tendo como exemplo uma CNN utilizada para classificação de imagem, esta tira partido da relação existente entre um conjunto de pixels próximos. O que difere na arquitetura da CNN de outro tipo de DNN é que esta possui na sua constituição camadas convolucionais nas camadas ocultas. Estas camadas realizam a operação de convolução e partilham pesos, o que diminui a quantidade de memória necessária ao seu armazenamento, tornando estas redes bastante atrativas.

As redes CNN são computacionalmente bastante exigentes e requerem uma elevada largura de banda de acesso à memória onde são guardados os pesos da rede. Por esta razão, a implementação e a execução de CNN é em geral realizada em sistemas de elevado desempenho, tornando difícil a sua aplicação em sistemas embebidos, com menor desempenho e memória. Contudo, a execução de CNN em sistemas embebidos próximos do sistema de recolha de dados tem muitas vantagens, como melhoria da privacidade dos dados, menos requisitos em termos de largura de banda de rede de comunicação e processamento em tempo real, que evita também a necessidade de guardar os dados recolhidos. Neste seguimento, o objetivo desta tese é conseguir reduzir as necessidades computacionais e de memória das CNN que permita a sua execução em sistemas embebidos com recursos reduzidos e a consequente redução do consumo de energia.

Para conseguir atingir este objetivo, foi considerada a utilização de dispositivos de hardware programável (FPGA - *Field Programmable Gate Array*) que permitem implementar uma arquitetura hardware dedicada bastante eficiente em termos computacionais e em termos de consumo energético. Tendo como base estes dispositivos, consideraram-se métodos que permitem reduzir o peso computacional e os requisitos de memória das CNN, em particular a redução do tamanho dos pesos.

Existem vários trabalhos recentes que propõem a redução do tamanho dos pesos como forma de reduzir o peso computacional e os requisitos de memória. No entanto, aplicam-se maioritariamente a plataformas com processadores de uso genérico ou com unidades de processamento gráfico (GPU - *Graphics Processing Unit*).

Neste trabalho, estabelece-se uma relação entre a redução do tamanho dos pesos, a precisão das redes e a área e o desempenho das implementações em FPGA. Na implementação das CNN, considerou-se a arquitetura LiteCNN [23] que é uma arquitetura genérica de execução de CNN em FPGA de baixo custo. Consegue-se, assim, determinar a eficiência dos métodos aplicados a implementações hardware.

1.1 Objetivos da Dissertação

Nesta dissertação, pretende-se fazer um estudo sobre o impacto que a redução do tamanho dos operandos das CNN tem sobre as implementações em FPGA.

No estudo, será considerada uma proposta de extensão da arquitetura LiteCNN para dar suporte a pesos de tamanho variável. Sobre esta rede, serão mapeadas CNN com pesos com diferentes representações em vírgula fixa dinâmica.

O estudo da precisão da rede em função da representação dos pesos é feito na plataforma Ristretto [9] que corre sobre a plataforma Caffe [21]. A implementação das redes é feita sobre a SoC (*System-on-Chip*) FPGA Zynq Z-7020. O estudo é generalizado com base em dois modelos propostos para a área da arquitetura e outro para o desempenho. Os modelos permitem estimar a área e o desempenho da arquitetura em função da rede CNN a executar e da respetiva dimensão dos dados.

O estudo permite-nos verificar o impacto que a redução do tamanho dos operandos da CNN tem sobre a ocupação dos recursos da FPGA e no desempenho do sistema.

Neste projeto foi utilizada a FPGA Zynq Z-7020 que é constituída por 53200 LUTs, 106400 *Flip-Flops*, 4.9Mb blocos BRAM (140 blocos RAM de 36Kb) e 220 DSP, que apesar de ser constituída por pouco recursos lógicos comparativamente com outras FPGA mais dispendiosas já permite implementar CNN de elevada complexidade em tempo real.

O presente documento encontra-se organizado em seis capítulos.

Capítulo 1 – No primeiro capítulo é realizado o enquadramento da tese nas Redes Neuronais Convolucionais e é descrito os objetivos desta dissertação.

Capítulo 2 – No segundo capítulo é feito uma explicação sobre as Redes Neuronais Convolucionais, incluindo os diferentes tipos de camadas que a estrutura da CNN pode conter e como é realizado o treino destas redes.

Capítulo 3 – O terceiro capítulo contêm exemplos de diversas CNN existentes e de projetos que implementaram CNN em FPGA, incluindo trabalhos com redução do tamanho dos operandos.

Capítulo 4 – O quarto capítulo descreve os diferentes algoritmos implementados nesta arquitetura para obter o melhor desempenho possível, a constituição da arquitetura implementada na FPGA e a forma como os algoritmos foram implementados nessa arquitetura. Este capítulo inclui também o modelo de área e de desempenho obtidos para esta arquitetura.

Capítulo 5 – O quinto capítulo apresenta todos os resultados obtidos para as arquiteturas com diferentes representações dos dados para diversas CNN.

Capítulo 6 – O último capítulo contém as conclusões finais deste projeto e propostas para um trabalho futuro.

Capítulo 2 - Redes Neuronais Convolucionais

A arquitetura das Redes Neuronais Convolucionais é baseada no cérebro humano, mais especificamente na forma como este processa uma entrada visual. Os neurónios responsáveis por esse processamento apenas disparam quando ocorre um determinado fenómeno no campo de visão, por exemplo, um neurónio dispara apenas quando no campo de visão está uma linha vertical e outro neurónio dispara apenas quando no campo de visão está uma linha horizontal. As imagens no cérebro são processadas em camadas de crescente complexidade, em que inicialmente são identificados atributos básicos e no final são identificados os objetos das imagens. As Redes Neuronais Convolucionais, como têm em conta essas características na sua arquitetura, são utilizadas em casos em que os dados podem ser descritos como um "mapa", em que a proximidade entre dois pontos desse dado indica como estes estão relacionados, como acontece com uma imagem. Para as CNN uma imagem é uma matriz de números, à qual são aplicadas matrizes de pesos (designadas por filtros) por diversas camadas. No final, a rede classifica a imagem indicando a probabilidade de um objeto na imagem pertencer a uma determinada classe de objetos, como no exemplo ilustrado na figura 2.

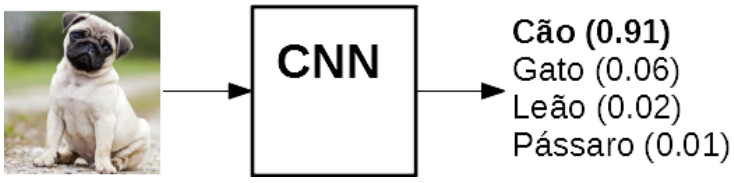


Figura 2 - Exemplo de classificação de uma imagem com CNN.

Tal como acontece no cérebro, os filtros das primeiras camadas são responsáveis por reconhecer atributos básicos e, à medida que se progride nas camadas, os filtros são capazes de detetar atributos mais complexos da imagem.

2.1 Estrutura das Redes Neuronais Convolucionais

As Redes Neuronais Convolucionais são constituídas por uma camada de entrada, uma camada de saída e várias camadas ocultas (ver figura 3).

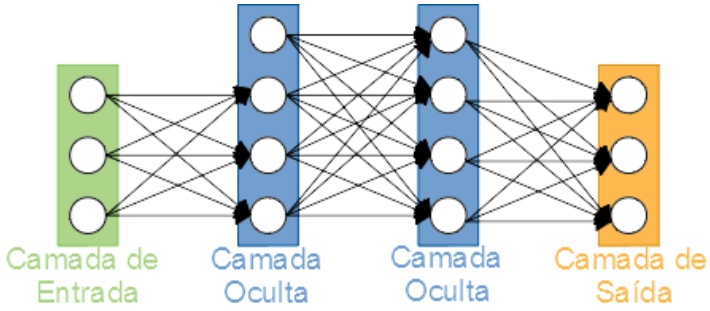


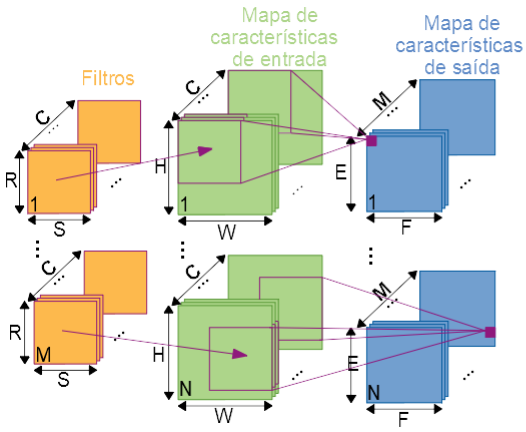
Figura 3 - Exemplo de uma estrutura de uma CNN.

A camada de entrada recebe a imagem e propaga os valores que recebeu para a primeira camada oculta. A camada de saída contém os resultados da imagem propagada pela rede, ou seja, contém as probabilidades de um objeto da imagem pertencer a cada classe que a rede classifica.

As camadas ocultas podem ser de diferentes tipos: convolucional, agrupamento (*pooling*), de não-linearidade, de perda, *softmax*, ou totalmente conectada.

A. Camada convolucional

A camada convolucional é uma das camadas principais da Rede Neuronal Convolucional. O resultado gerado por esta camada é designado por mapa de características de saída (OFM – *Output Feature Map*) ou mapa de ativação (ver figura 4).



Legenda:

C	Número de canais do filtro e do mapa de características de entrada
R/S	Dimensão do filtro (R=H e S=W nas camadas totalmente conectadas)
M	Número de filtros e número de canais do mapa de características de saída
H/W	Dimensão do mapa de características de entrada
N	Número de mapa de características de entrada e de saída (*batch*)
E/F	Dimensão do mapa de características de saída (E=1 e F=1 nas camadas totalmente conectadas)

Figura 4 - Exemplo genérico da aplicação de uma camada convolucional.

Nesta camada, a matriz de pesos (filtro) é deslizada sobre a imagem e é calculado o produto escalar do filtro com a matriz da imagem com a mesma dimensão do filtro.

Tendo em conta os parâmetros ilustrados na figura 4, a aplicação da camada convolucional é definida pela equação 1.

$$OFM[n][\mathrm{m}][\mathrm{f}][\mathrm{e}] = \sum_{k=0}^{C-1}\sum_{i=0}^{S-1}\sum_{j=0}^{R-1}(IFM[n][k][f\times P+i][e\times P+j]\times Filtro[m][k][i][j]) + Bias[m]\,,$$
$$0 \le n < N, 0 \le m < M, 0 \le f < F, 0 \le e < E, P - Passo\ (Stride)$$

Equação 1 - Aplicação da camada convolucional.

O exemplo desse cálculo para um canal é ilustrado na figura 5.

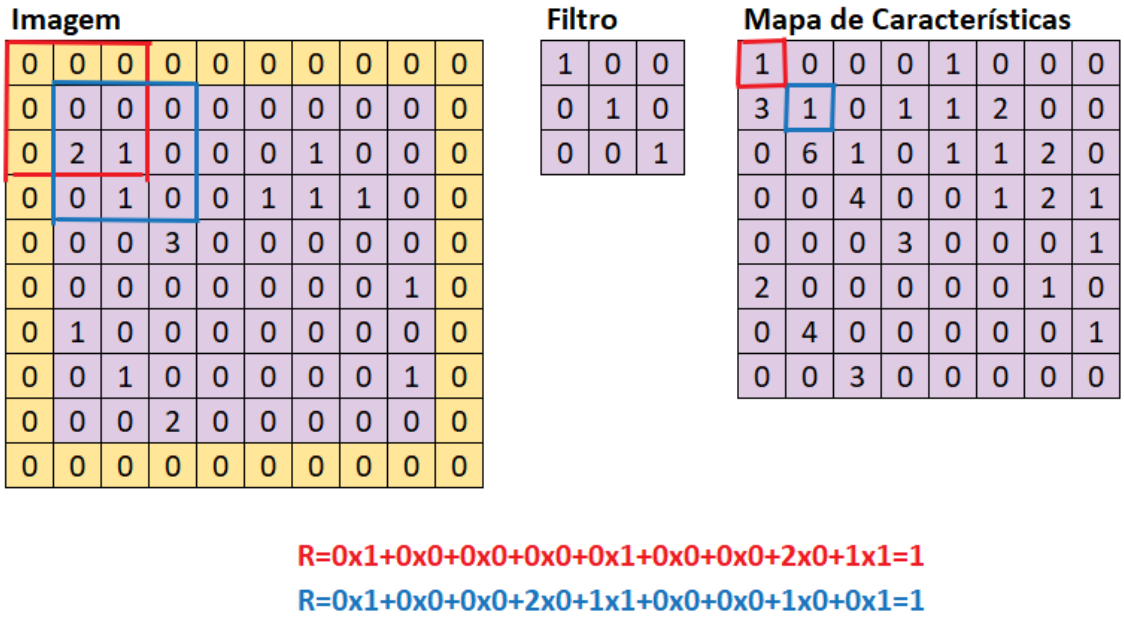


Figura 5 - Exemplo do cálculo do produto escalar entre um filtro e uma imagem.

No exemplo ilustrado na figura 5, ilustra-se como calcular o produto escalar da imagem com um filtro 3x3. Para realizar o produto escalar, é multiplicado cada peso da matriz pelo valor da imagem na posição correspondente e o seu resultado final é dado pelo somatório de todas as multiplicações (neste caso como o filtro têm uma dimensão de 3x3 corresponde a 9 multiplicações). O resultado do produto escalar é guardado no mapa de características e para este ficar com o tamanho do mapa de características de entrada foi adicionado um *padding* de valor um ao mapa de características de entrada, ou seja, à imagem foi adicionada uma linha e uma coluna preenchidas com o valor zero em todos os extremos da imagem. Este processo deve ser repetido, mas deslocando a matriz da imagem a utilizar no cálculo para a direita no valor do passo definido (*stride*), o passo corresponde ao número de pixels com que se move o filtro entre cada operação do produto escalar. No exemplo da figura 5 o passo tem o valor de um. Quando se chega ao canto direito da imagem, este processo repete-se novamente, mas utilizando a matriz da imagem no cálculo deslocada para baixo no valor do passo definido.

Os filtros são utilizados para encontrar a localização de detalhes da imagem, como por exemplo, diferentes tipos de linhas, curvas e cores. No exemplo da figura 5, o filtro permite encontrar linhas diagonais inclinadas à direita. Como resultado, verifica-se que os resultados com valores maiores no mapa de características correspondem a linhas diagonais inclinadas à direita.

B. Camada de agrupamento (*Pooling*)

A camada de agrupamento (*pooling*) reduz o tamanho do mapa de características, reduzindo a necessidade computacional para as camadas posteriores, sendo

normalmente utlizada entre camadas convolucionais sucessivas. A CNN ao possuir camadas de agrupamento torna-se menos sensível a pequenas alterações na localização de um objeto, ou seja, fica com a propriedade de invariância de translação em que a saída da camada de agrupamento permanece igual mesmo quando o objeto se move um pouco.

A camada de agrupamento pode utilizar várias funções, como a função *Average Pooling*, a função *L2-norm Pooling*, e a mais utilizada a função *Max Pooling*. A camada de agrupamento funciona como uma janela deslizante no mapa de características que aplica uma das funções mencionadas em cada deslocamento. Na função *Average Pooling*, para cada posição da janela é calculada a média de todos os valores da janela e esse é o valor guardado. Na função *L2-norm Pooling* para cada posição da janela é calculada a raiz quadrada da soma dos quadrados de todos os valores da janela e esse é o valor guardado. Na função *Max Pooling*, para cada posição da janela é retirado o valor mais alto e descartados os restantes. Geralmente, a janela desloca-se num passo com o valor igual à dimensão da janela de forma a não existir sobreposição de blocos. Um passo com valor maior que a dimensão da janela deslizante é utilizado para reduzir drasticamente o mapa de características. Um exemplo da aplicação da camada de agrupamento está ilustrado na figura 6, em que se utiliza uma janela deslizante de dimensão 2x2 e um passo de 2. É de notar que no exemplo da figura 6 a camada de agrupamento reduz o tamanho do mapa de características em 75%.

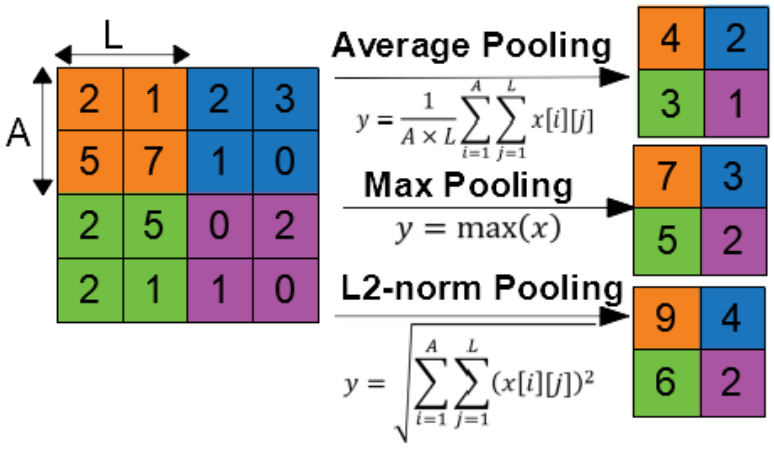


Figura 6 - Exemplo da aplicação de Average Pooling, Max Pooling e L2-norm Pooling.

C. Camada de não-linearidade

A camada de não-linearidade aplica uma função de ativação não-linear e é geralmente aplicada após as camadas de convolução ou após as camadas totalmente conectadas para introduzir não-linearidade no sistema. Existem várias funções não-lineares, como a função tangente hiperbólica ou a função sigmoide. No entanto, a

função mais utilizada é a função ReLU. Na figura 7 estão ilustradas algumas das funções não-lineares mais utilizadas.

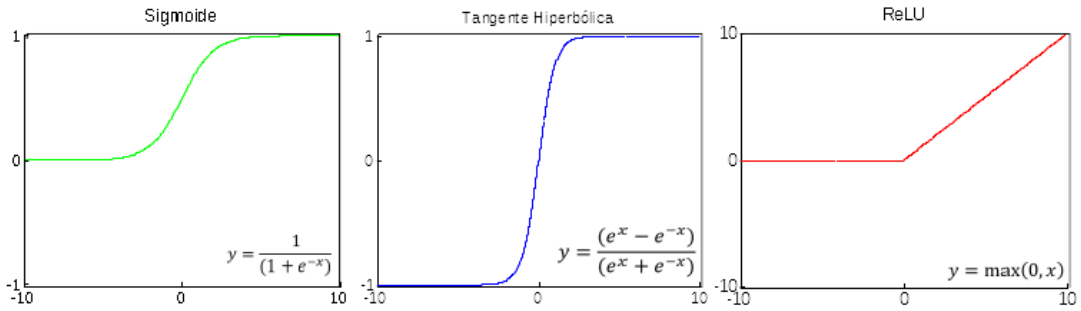


Figura 7 - Funções não-lineares.

A função ReLU é a função de ativação não-linear mais utilizada visto que treina a CNN de uma forma mais rápida sem prejudicar a sua precisão e a sua simplicidade leva a que precise de poucos recursos computacionais. A camada de não-linearidade aplica a função ReLU a cada ponto do mapa de características, resultando num mapa de características sem valores negativos, como ilustrado na figura 8.

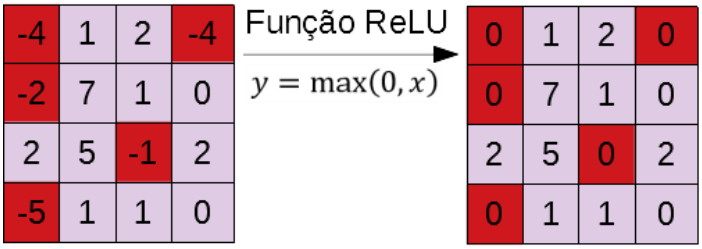


Figura 8 - Exemplo da aplicação da função ReLU.

D. Camada totalmente conectada

As camadas totalmente conectadas, ou camadas densas, são camadas em que todos os nós de entrada estão conectados a todos os nós de saída. Esta característica das camadas totalmente conectadas aumenta a necessidade de capacidade computacional e de memória, levando a que normalmente as CNN utilizem apenas duas camadas totalmente conectadas no final.

A camada totalmente conectada é constituída por três partes, a camada de entrada (saída da camada anterior à camada totalmente conectada), a camada oculta e a camada de saída. A cada conexão de um valor de entrada com um valor de saída (neurónio) é atribuído um peso diferente, sendo que o valor de saída corresponde a uma função aplicada ao somatório de todos os produtos dos valores de entrada conectados a essa saída com os seus respetivos pesos.

Um exemplo de uma camada totalmente conectada está ilustrado na figura 9. No exemplo é possível verificar que todos os valores de entrada, *Xa* a *Xd*, estão conectados

a todos os valores da camada oculta, *Ya* a *Yd*, e que cada conexão de um valor de entrada com um valor da camada oculta tem atribuído um peso, *Waa* a *Wdd*. O resultado de cada neurónio da camada oculta e da camada de saída é dado pela função aplicada ao somatório de todos os produtos das entradas conectadas a esse neurónio com os seus respetivos pesos, sendo que os neurónios da camada de saída retornam a probabilidade de a imagem pertencer à classe "cão" e à classe "gato", no exemplo.

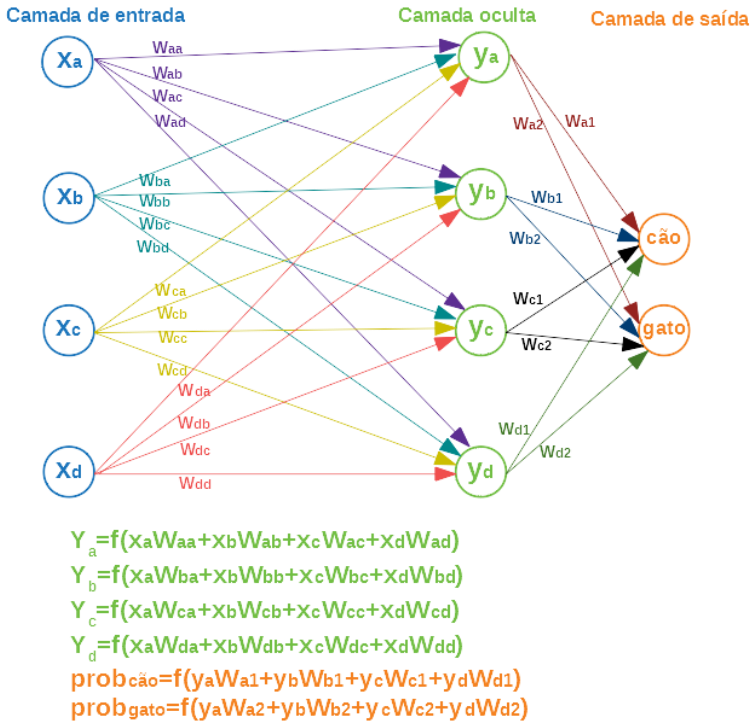


Figura 9 - Exemplo de uma camada totalmente conectada.

E. Camada de perda

A camada de perda segue-se à última camada totalmente conectada, e é responsável pelo ajustamento dos pesos e dos valores *bias* de toda a CNN. Durante o treino das CNN, realizado com lotes de imagens rotuladas, existe a propagação para a frente, que é responsável por classificar a imagem e consiste em processar pelas diferentes camadas da rede no sentido direto uma imagem colocada na entrada da rede, e a propagação para trás, na qual a camada de perda compara os resultados obtidos com os valores reais e calcula o gradiente de erro de cada parâmetro da rede que permite que este seja atualizado de forma a obter uma melhor precisão.

F. Camada *softmax*

A camada *softmax* aplica uma função que permite alterar os valores recebidos para uma gama de valores entre zero e um, e divide esses valores de forma a que o somatório dos resultados na sua saída seja um. Assim a camada *softmax* é equivalente

a uma distribuição de probabilidade, indicando a probabilidade de cada classe (cada resultado de saída desta camada) ser a correta.

2.2 Treino e Inferência das CNN

Uma CNN é um algoritmo de aprendizagem em que é necessário determinar o valor dos seus pesos: fase de aprendizagem ou treino. Depois de treinada, a rede realiza a tarefa para a qual foi treinada calculando a saída considerando os pesos determinados durante o treino. Esta fase designa-se de inferência. O objetivo do treino é ajustar os valores dos pesos e os valores bias de forma a aumentar a probabilidade da classe correta, classe a que o objeto da imagem na entrada pertence, e diminuir a probabilidade das restantes. O treino divide-se também em duas fases, propagação para a frente (*fowardpropagation*) e propagação para trás (*backpropagation*). A propagação para a frente consiste em colocar uma imagem na entrada e obter os resultados na saída e a propagação para trás consiste em atualizar os parâmetros da rede. Ao treinar a rede, a classe correta é geralmente conhecida porque as imagens utilizadas no treino são rotuladas com a classe correta. A diferença entre a probabilidade correta ideal e a probabilidade resultante na CNN designa-se por perda (*loss*), ou seja, no treino pretende-se minimizar a perda média de um grande conjunto de imagens de treino. Uma abordagem de treino normalmente utilizada designa-se por *fine-tuning* e consiste em treinar a rede com pesos previamente treinados, assim o treino é normalmente mais rápido visto que o ponto de partida do treino deixa de ser aleatório. Após a rede estar treinada, o funcionamento da CNN encontra-se na fase de inferência, em que a CNN classifica as imagens colocas na entrada utilizando os pesos determinados na fase de treino.

2.2 Complexidade das CNN

A complexidade de uma CNN além de depender do número e do tipo de camadas que constitui a sua arquitetura, depende também do número de filtros das camadas convolucionais e das suas dimensões e do tamanho da janela das camadas de agrupamento. Assim, a complexidade da rede aumenta com o aumento do número de camadas da rede, do número de filtros das camadas convolucionais, da dimensão dos filtros das camadas convolucionais e com a diminuição da janela das camadas de agrupamento. A complexidade de uma CNN é algo que deve ser sempre tido em consideração, pois quanto mais complexa for a CNN mais recursos são necessários para a implementar e mais tempo demora a ser treinada.

Atualmente a tendência consiste em aumentar a complexidade da CNN em busca da melhor precisão possível. Atentando à evolução de alguns modelos de CNN existentes essa tendência é facilmente comprovada. Uma das primeiras CNN apresentada em 1998 a LeNet-5 [2], é constituída por sete camadas, filtros de 5x5 nas camadas convolucionais,

o número de filtros das camadas convolucionais varia entre 6 e 16 e as camadas de agrupamento possuem filtros de 2x2. Em 2012 foi apresentada a CNN AlexNet [4], constituída por cinco camadas convolucionais e três camadas totalmente conectadas, as dimensões dos filtros das camadas convolucionais variam entre 5x5 e 11x11, o número de filtros das camadas convolucionais varia entre 96 e 384 e as camadas de agrupamento utilizam janelas de 3x3. Assim comparando a CNN LeNet-5 e a CNN AlexNet verifica-se que a rede mais recente, a CNN AlexNet, é mais complexa pois o número de camadas, as dimensões dos filtros das camadas convolucionais e o número de filtros das camadas convolucionais aumentaram. Passou-se de cerca de 60000 pesos para cerca de 60000000, um aumento de 1000x.

Capítulo 3 – Estado da Arte

Neste capítulo, descrevemos algumas das redes CNN mais conhecidas, trabalhos de implementação de CNN em FPGA e os métodos de melhoria da eficiência das implementações.

3.1 Redes Neuronais Convolucionais

Uma das primeiras CNN apresentadas foi a LeNet em 1989, com o propósito de classificar dígitos em imagens em tons de cinza com tamanho de 28x28 pixels [1]. Esta rede foi implementada em caixas eletrónicas para reconhecer dígitos nos cheques depositados e a sua versão mais conhecida é a LeNet-5 [2]. A LeNet-5 é uma rede constituída por sete camadas, duas camadas convolucionais, duas camadas totalmente conectadas, duas camadas de agrupamento e uma camada de não-linearidade [3] (ver figura 10).

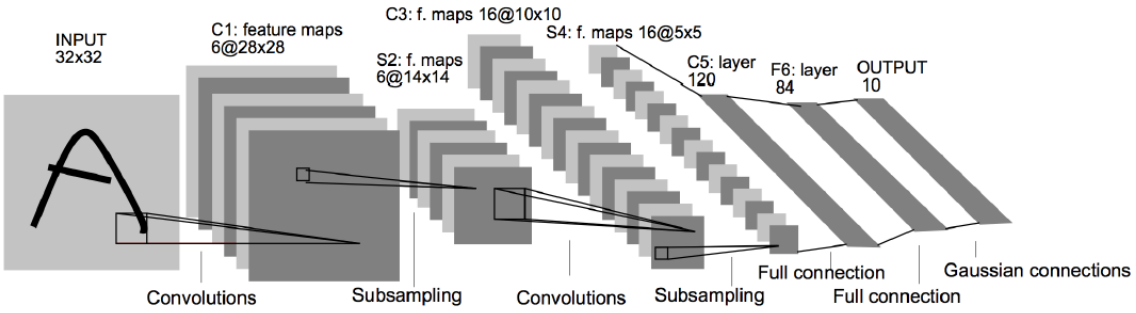


Figura 10 - Rede para classificação de dígitos: LeNet-5 [3].

No total, a LeNet-5 possui 340908 conexões e 60000 pesos. O número reduzido de pesos deve-se à partilha de pesos. As camadas convolucionais aplicam filtros de 5x5, sendo que a primeira camada convolucional possui seis filtros e a segunda camada convolucional possui dezasseis filtros. Após cada camada convolucional há uma camada de agrupamento que aplica a função *Average Pooling* com filtros de 2x2 e passo de um. O número de filtros de cada camada de agrupamento é igual ao número de filtros da camada convolucional que a antecede. No final da rede existem duas camadas totalmente conectadas e a camada de não-linearidade que aplica a função sigmoide.

Em 2012 foi apresentada na competição ILSVRC (*ImageNet Large Scale Visual Recognition Challenge*) a CNN *AlexNet*. Esta competição é dedicada à avaliação de algoritmos de classificação de imagens e deteção de objetos em larga escala. Uma das tarefas nesta competição consiste em treinar a rede com 1.2 milhões de imagens a cores, de 256x256, rotuladas em mil classes diferentes e testar a precisão da rede, após o seu treino, com um grupo de imagens não utilizadas na fase de treino [2]. A CNN *AlexNet* é uma rede profunda (com várias camadas) constituída por cinco camadas convolucionais

e três camadas totalmente conectadas e que recebe imagens de 227x227x3 [4] (ver figura 11).

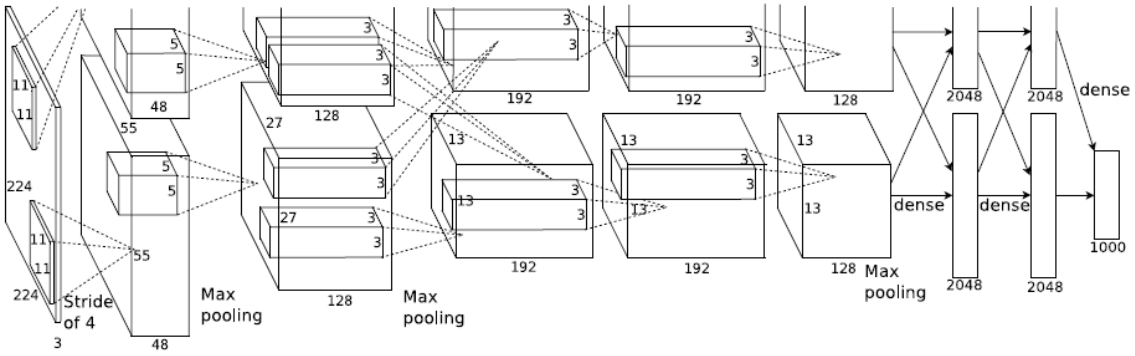


Figura 11 - Rede para classificação de imagens: AlexNet [4].

A primeira camada convolucional utiliza 96 filtros de 11x11x3 com um passo de quatro, a segunda camada convolucional utiliza 256 filtros de 5x5x48, a terceira camada convolucional aplica 384 filtros com uma dimensão de 3x3x256, a quarta camada convolucional utiliza 384 filtros de 3x3x192 e a última camada convolucional da rede utiliza 256 filtros de 3x3x192. A CNN *AlexNet* utiliza camadas de não-linearidade que aplicam a função ReLU após todas as camadas convolucionais e todas as camadas totalmente conectadas. As camadas de agrupamento aplicam a função *Max Pooling* com um filtro de 3x3 e passo de 2, e são aplicadas após a primeira, a segunda e a quinta camada convolucional. A rede *AlexNet* utiliza também duas camadas designadas por *Local Response Normalization* (LRN) antes da primeira e da segunda camada de agrupamento. Este tipo de camada aumenta o contraste entre os valores que recebe na entrada. No entanto, a camada LRN já são é muito utilizada. No total, a rede *AlexNet* utiliza 60M parâmetros para processar uma imagem de 227x227x3. A CNN *AlexNet* permitiu baixar a taxa de erro em, aproximadamente, 10% em relação aos resultados anteriores da competição ILSVRC, registando uma taxa de erro *top-5* (a classe correta encontra-se entre as cinco classes com maior probabilidade) de 15.33%.

Na competição ILSVRC de 2014 foi apresentada a CNN GoogLeNet [5], que é constituída por 22 camadas, (27 camadas contabilizando as camadas de agrupamento) (ver figura 12).

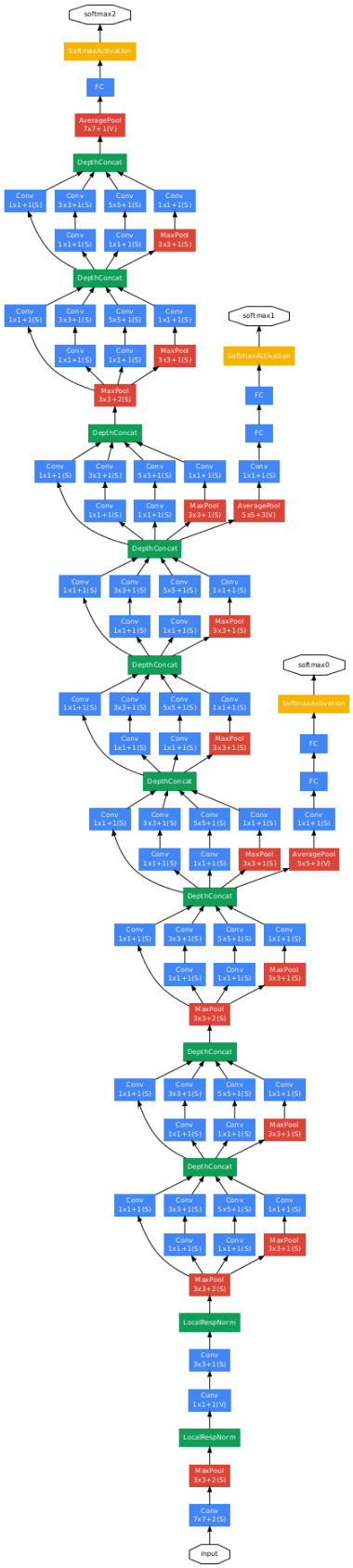


Figura 12 - Rede para classificação de imagens: GoogleNet [5].

A rede GoogLeNet introduziu o módulo *inception*, que consiste em conexões em paralelo utilizando filtros de diferentes tamanhos (1x1, 3x3 e 5x5) como ilustrado na figura 13.

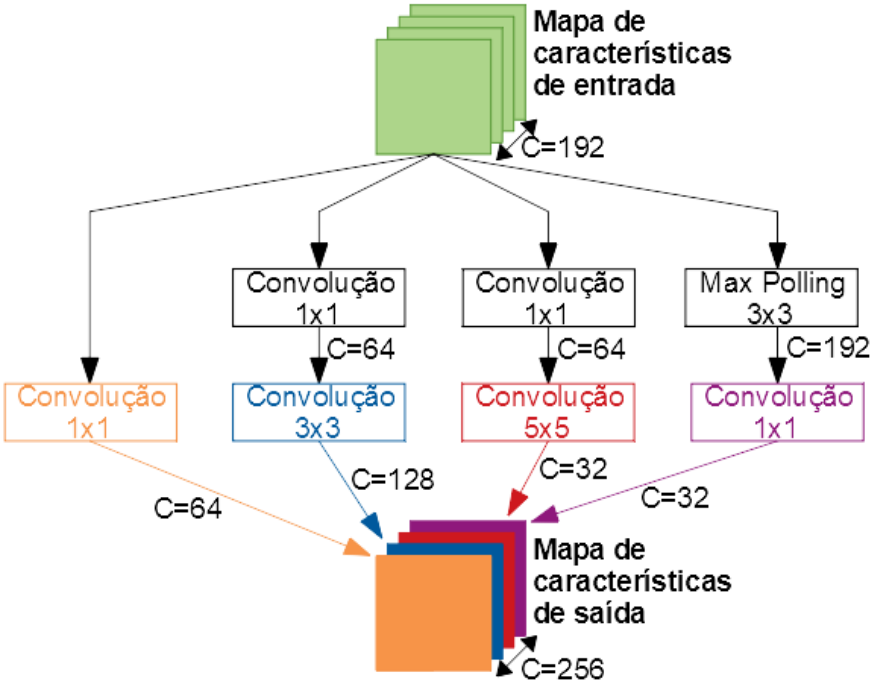


Figura 13 - Módulo inception.

Ao utilizar este módulo, o mapa de saída do módulo contém concatenados os resultados da aplicação dos diferentes filtros. A utilização dos filtros de dimensão 1x1 serve para reduzir o número de canais, como se pode verificar na figura 13, que reduz assim o número de convoluções a realizar posteriormente com os filtros de 3x3 e 5x5. A CNN GoogLeNet ganhou a competição ILSVRC de 2014 com uma taxa de erro top-5 de 6.67%.

Na competição ILSVRC de 2014 foi também apresentada a CNN VGG, esta CNN possui duas versões VGG-16, com 16 camadas (22 camadas contabilizando as camadas de agrupamento e *softmax*), e VGG-19, com 19 camadas (25 camadas contabilizando as camadas de agrupamento e *softmax*) [20]. A CNN VGG utiliza filtros 3x3 ao invés de filtros maiores, como os filtros 5x5 normalmente utilizados, o que diminui o número de pesos da rede. A VGG-16 é constituída por 13 camadas convolucionais e 3 camadas totalmente conectadas, enquanto que a VGG-19 é constituída por 16 camadas convolucionais e 3 camadas totalmente conectadas. A CNN VGG-19 obteve uma taxa de erro top-5 de 7.3%.

Um dos problemas que se verificou ao aumentar o número de camadas foi a saturação da precisão devido ao desvanecimento do gradiente de erro que impede a atualização dos parâmetros das camadas iniciais da rede. Para resolver este problema, a CNN *ResNet*

utiliza conexões residuais que permitem aumentar o número de camadas [6]. A conexão residual consiste num atalho que permite "saltar" algumas camadas convolucionais, como ilustrado na figura 14, e que permite resolver o problema do desvanecimento do gradiente de erro sem aumentar os requisitos de memória e computacionais da rede.

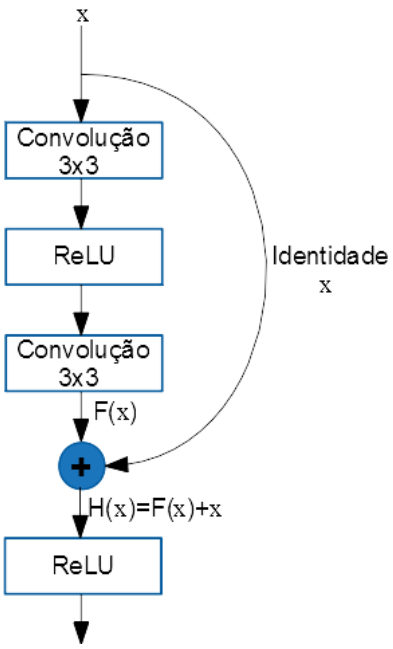


Figura 14 - Conexão residual utilizada na CNN ResNet.

Existem diversos modelos desta CNN em que varia apenas o número de camadas. Os modelos mais conhecidos são a ResNet-34, a ResNet-50, a ResNet-101 e a ResNet-152, em que o número corresponde ao número de camadas da rede. Em 2015 a CNN ResNet-152 conquistou o primeiro lugar na competição ILSVRC. A ResNet-152 atingiu uma taxa de erro top-5 de 3.57%, tendo assim uma precisão superior à da capacidade humana, que ronda os 5.1% [22].

A *SqueezeNet* é uma CNN baseada na rede AlexNet e que foi projetada para reduzir o número de parâmetros da rede sem alterar a precisão desta [7]. Assim aplicando técnicas na arquitetura da rede, como substituir os filtros de 3x3 por filtros 1x1 e diminuir o número de canais dos mapas de entrada das camadas que utilizam filtros 3x3, permite que a CNN *SqueezeNet* possua cinquenta vezes menos parâmetros que a CNN AlexNet sem alterar a sua precisão.

O modelo CIFAR-10 é uma CNN baseada também na rede *AlexNet*, mas é treinada com a base de dados CIFAR-10 ao invés da base de dados ILSVRC (imagens de mil classes diferentes utilizadas na competição ILSVRC). A base de dados CIFAR-10 consiste num conjunto de imagens a cores com dimensão 32x32 pertencentes a dez classes diferentes [2].

Atualmente a tendência é aumentar a profundidade e a complexidade da CNN de forma a obter uma maior precisão. No entanto, algumas aplicações do dia a dia onde são aplicadas as CNN, como robôs, carro autônomos e realidade aumentada, precisam de obter os resultados rapidamente e possuem recursos computacionais limitados. Desta forma, é importante o desenvolvimento de CNN orientadas para os sistemas embebidos, em que além da precisão da rede têm em atenção o tamanho e a velocidade de execução da CNN. Atualmente já existem algumas redes orientadas para sistemas embebidos como a CNN *MobileNet* e a CNN *ShuffleNet*.

A CNN *MobileNet* [10] permite delimitar os recursos (latência e tamanho) da arquitetura da rede conforme as restrições da aplicação a ser implementada. Para concretizar estas características a CNN *MobileNet* utiliza convoluções separáveis por profundidade e acrescenta dois híper-parâmetros, multiplicador de largura e multiplicador de resolução. A convolução separável em profundidade é dividida na convolução profunda, que aplica um único filtro por cada canal de entrada, e na convolução pontual, que aplica uma convolução de 1x1 para criar uma combinação linear da saída da camada de profundidade. Na figura 15 está ilustrado a diferença entre a convolução normal, a convolução profunda e a convolução pontual.

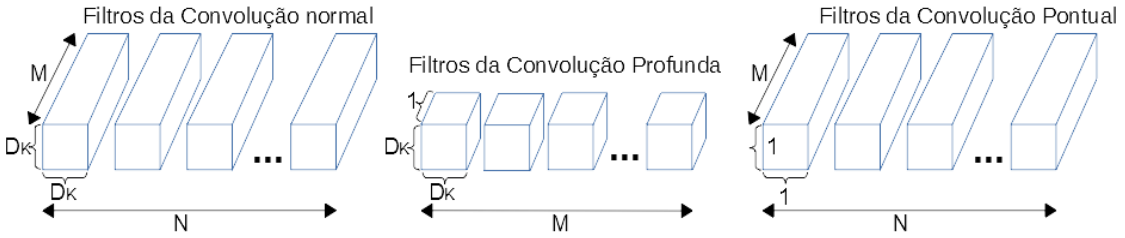


Figura 15 - Convoluções separáveis por profundidade da CNN MobileNet.

A CNN *MobileNet* utiliza convoluções separáveis em profundidade de 3x3 que permite usar perto de 9 vezes menos computações que as convoluções padrão com uma pequena redução na precisão. Para controlar a latência e o tamanho da CNN, a rede *MobileNet* introduz o parâmetro designado por multiplicador de largura, conhecido por α, que permite afinar a rede uniformemente em cada camada. Assim para uma determinada camada e multiplicador de largura, α, o número de canais de entrada, M, passa a ser dado por αM e o número de canais de saída, N, passa a ser dado por αN. Para reduzir o custo computacional da CNN a rede *MobileNet* introduziu o parâmetro multiplicador de resolução que é aplicado à imagem de entrada da rede, e subsequentemente reduz também a representação interna de cada camada. Contabilizando as convoluções profundas e as convoluções pontuais como camadas separadas, a CNN *MobileNet* é constituída por 28 camadas. Alguns resultados obtidos para a rede *MobileNet* treinada com a base de imagens ImageNet e diferentes valores para os híper-parâmetros estão ilustrados na tabela 1.

Modelo: "α MobileNet-Resolução"	Precisão	Milhões de MACs	Milhões de parâmetros
1.0 MobileNet-224	70.6%	569	4.2
0.75 MobileNet-224	68.4%	325	2.6
0.5 MobileNet-224	63.7%	149	1.3
0.25 MobileNet-224	50.6%	41	0.5
1.0 MobileNet-192	69.1%	418	4.2
1.0 MobileNet-160	67.2%	290	4.2
1.0 MobileNet-128	64.4%	186	4.2
0.5 MobileNet-160	60.2%	76	1.32

Tabela 1 - Resultados da CNN MobileNet.

Através dos resultados da *MobileNet* verifica-se que com o ajuste do parâmetro multiplicador de largura e da resolução é possível diminuir o número de MACs e de parâmetros da rede, no entanto a precisão da rede é sempre afetada. Com esta rede é possível encontrar um bom ponto de equilíbrio entre a precisão, o número de MACs e o número de parâmetros da rede, por exemplo diminuir o número de MACs em cerca de 43% e os parâmetros da rede em cerca de 38% apenas provoca uma diminuição da precisão da rede de 2.2%. Com esta rede é possível também realizar diminuições drásticas ao número de MACs e de parâmetros da rede diminuindo a precisão em apenas 20%, quando o número de MACs é reduzido em 93% e o número de parâmetros da rede é diminuído em 88%.

A *ShuffleNet* [11] é uma CNN projetada para sistemas com recursos computacionais muito limitados, entre 10 a 150 MFLOPs (*Mega Floating-Point Operation per Second*). Esta CNN utiliza convoluções de grupos e a técnica de canal *shuffle* para reduzir o custo de computação sem alterar a precisão da rede.

Uma convolução de grupo permite reduzir significativamente o custo de computação quando uma saída de um grupo apenas depende da entrada desse mesmo grupo, ou seja, quando não há interligações entre grupos. No entanto, se forem agrupadas muitas convoluções de grupo, um canal pode apenas derivar de uma pequena fração dos canais de entrada, o que pode prejudicar a rede. Para evitar esta situação, a rede *ShuffleNet* aplica a técnica de canal *shuffle* às convoluções de grupo. Assim, a convolução de grupo passa a obter dados de entrada de diferentes grupos, como se pode observar na figura 16.

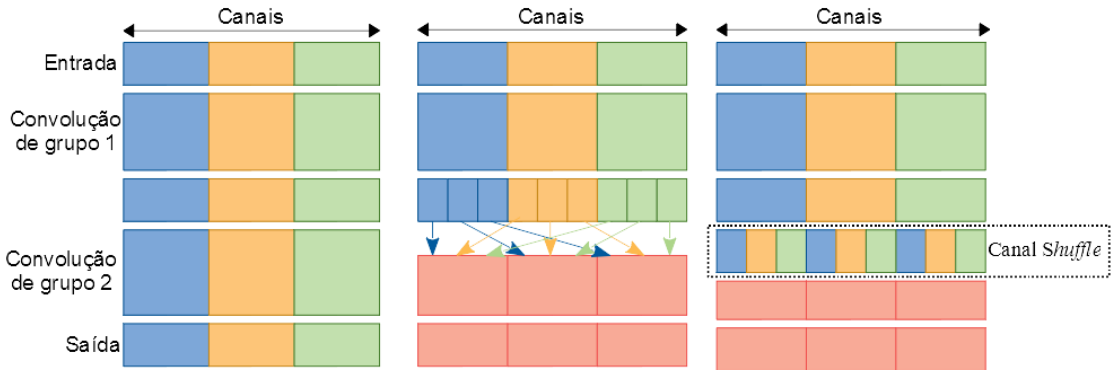


Figura 16 - Convoluções de grupo e técnica de canal shuffle utilizadas na CNN ShuffleNet.

A técnica de canal *shuffle* consiste em dividir os canais em grupo e dividir cada grupo em vários subgrupos, depois cada grupo da próxima camada recebe como entrada subgrupos de grupos diferentes. Além destas técnicas, a rede *ShuffleNet* utiliza o modelo *Bottleneck* ilustrado na figura 17, no qual as camadas convolucionais são transformadas em camadas convolucionais de grupo e é adicionado o canal *shuffle*.

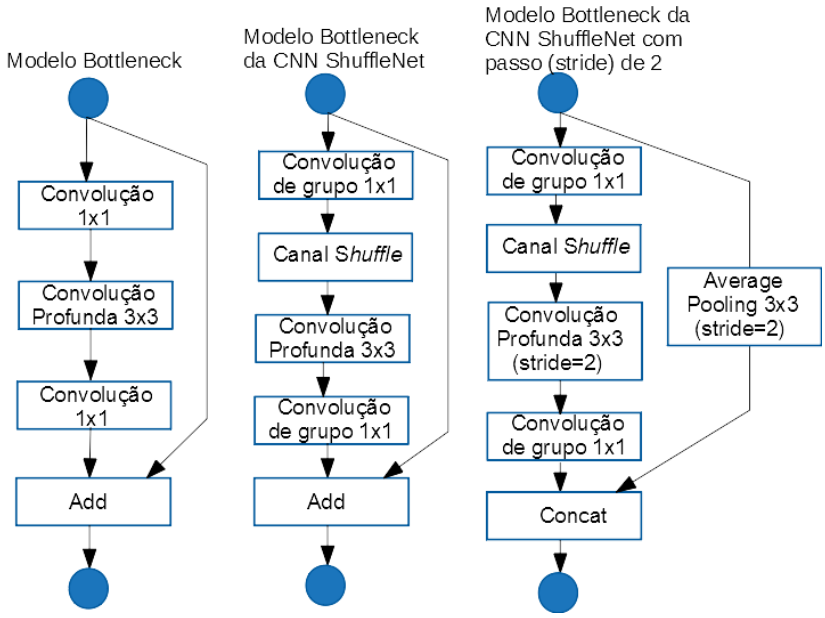


Figura 17 - Modelos Bottleneck utilizado na CNN ShuffleNet (não estão ilustradas as camadas ReLU).

A *ShuffleNet* obteve uma menor taxa de erro (cerca de 7.8%) que a *MobileNet* utilizando um sistema com recursos computacionais de 40MFLOPs. Na tabela 2 estão ilustrados alguns resultados obtidos para a rede *ShuffleNet*, em que *g* representa o número de grupos e 2x, 1.5x, 1x, 0.5x e 0.25x representam um fator de escala do número de canais que permite ajustar a complexidade da rede para um valor pretendido.

Modelo	Complexidade (MACs)	Taxa de erro				
		g=1	g=2	g=3	g=4	g=8
ShuffleNet 2x	524 MFLOPs	-	-	26,30%	-	-
ShuffleNet 1.5x	292 MFLOPs	-	-	28.5%	-	-
ShuffleNet 1x	140 MFLOPs	33.6%	32.7%	32.6%	32.8%	32.4%
ShuffleNet 0.5x	38 MFLOPs	45.1%	44.4%	43.2%	41.6%	42.3%
ShuffleNet 0.25x	13 MFLOPs	57.1%	56.8%	55.0%	54.2%	52.7%

Tabela 2 - Resultados da CNN ShuffleNet.

Com os resultados obtidos verifica-se que a rede ao ter convoluções de grupo obtém melhores resultados que convoluções sem grupo (g=1). Verifica-se também que as redes menos complexas tendem a beneficiar mais com o aumento do número de grupos, por exemplo para a rede *SheffleNet* 1x passar de uma convolução sem grupos para uma convolução com oito grupos provoca uma diminuição da taxa de erro de 1.2%, enquanto que a rede *SheffleNet* 0.25x ao passar de uma convolução sem grupos para uma convolução com oito grupos diminui a sua taxa de erro em 4.4%.

3.2 Plataformas de Treino e de Inferência de Redes CNN

O treino e a inferência das CNN são tipicamente realizados em processadores de propósito genérico (CPU - *Central Processing Unit*) ou em unidades de processamento gráfico (GPU - *Graphics Processing Unit*). O treino das redes é um processo bastante moroso, podendo demorar vários dias. Por esta razão, têm-se usado maioritariamente plataformas de computação de elevado desempenho baseadas em GPU para o treino das redes com desempenhos que atingem os 3.23 TFLOPs.

Atualmente, a dimensão da arquitetura das CNN tem vindo a aumentar para atingir melhores resultados de precisão, o que aumenta ainda mais as necessidades de memória e computacionais na implementação de uma CNN. As CNN mais recentes obtiveram maus desempenhos quando implementadas em CPU, o que aumentou a migração das implementações de CNN para GPU, FPGA e ASIC [8].

Apesar de serem computacionalmente bastante exigentes, as CNN podem ser massivamente paralelizadas. A operação mais comum das CNN nas camadas convolucionais e nas camadas totalmente conectadas é a de multiplicação e acumulação (MAC - *multiply-and-accumulate*). Os dispositivos que permitam a execução paralela de MAC permitem executar as CNN com muito melhor desempenho. As implementações de CNN em GPU permitem obter elevados níveis de desempenho comparativamente a implementações em CPU. No entanto, o consumo de energia neste tipo de implementações é bastante elevado. As implementações de CNN em ASIC aumentam

consideravelmente a relação desempenho/consumo de energia, no entanto ainda não são muito utilizadas devido ao seu elevado custo de projeto e fabricação [8] que pode facilmente deixar de ser útil devido ao dinamismo atual dos algoritmos associados às CNN. Assim, a implementação de CNN em FPGA tem ganho cada vez mais popularidade pois é possível obter bons desempenhos e níveis de consumo de energia mais baixos do que as implementações de CNN em GPU[9].

A implementação de CNN em FPGA possui inúmeras vantagens como bom desempenho, alta eficiência energética e a capacidade de reconfiguração, que permite adaptar a arquitetura à rede CNN que se pretende implementar. Apesar do número de blocos lógicos e da capacidade de memória das FPGA estar a aumentar, a execução de CNN em FPGA ainda está de certa forma limitada, comparativamente com os GPU, uma vez que não é possível atingir o mesmo nível de paralelismo e de armazenamento dos atuais GPU. Além disso, a complexidade de implementação de uma CNN em FPGA é superior à de uma CNN em GPU, visto que é necessário projetar a arquitetura hardware antes de executar o algoritmo.

Atualmente, já existem algumas implementações de CNN em FPGA que utilizam vírgula flutuante de 32 bits para representar os dados (representação de dados normalmente utilizada em GPU) que possuem bons desempenhos [13] [14] [15] [16].

No trabalho de Zhang et al. [13] verificam-se os requisitos de memória e a taxa de transferência de uma CNN através de técnicas como *loop tiling* e transformação, e posteriormente utilizam o modelo *roofline* para encontrar a solução de implementação da CNN com melhor desempenho e menor necessidade de recursos da FPGA. O *loop tiling* consiste em selecionar uma pequena parte dos dados da CNN para introduzir na FPGA (parte da arquitetura designada por *on-chip*). Uma má utilização do *loop tiling* provoca uma diminuição da reutilização dos dados e do paralelismo. A transformação consiste em organizar os elementos de processamento (PE - *processing-elements*), que consistem na unidade básica de computação para a convolução, os *buffers* de dados e as interligações entre eles para o melhor desempenho possível. Devido à limitação de recursos da FPGA, todos os dados são armazenados numa memória externa (parte da arquitetura designada por *off-chip*) que transfere dados para os *buffers* de dados da FPGA, que serão depois introduzidos nos PE. O modelo *roofline* permite relacionar o desempenho do sistema à taxa de transferência da memória externa e ao pico máximo de desempenho do hardware. A implementação descrita neste trabalho obteve um desempenho de 61.62 GFLOPs (*Giga Floating-Point Operation per Second*) utilizando a Xilinx FPGA Virtex7-485t a uma frequência de 100MHz.

Na implementação de Qiao et al. [14] é implementada uma CNN em FPGA utilizando uma arquitetura hardware/software que consiste numa FPGA que comunica com um

processador genérico, e numa memória externa que é partilhada pela FPGA e pelo processador *host*. Nesta arquitetura, a FPGA está encarregue das camadas convolucionais e das camadas totalmente conectadas da CNN e o processador está encarregue da restante implementação da CNN. A arquitetura executada na FPGA contém vários blocos, em que cada bloco é constituído por um módulo *prefetcher stream*, um módulo mapeador de fluxo e um módulo multiplicador de matriz (ver figura 18)

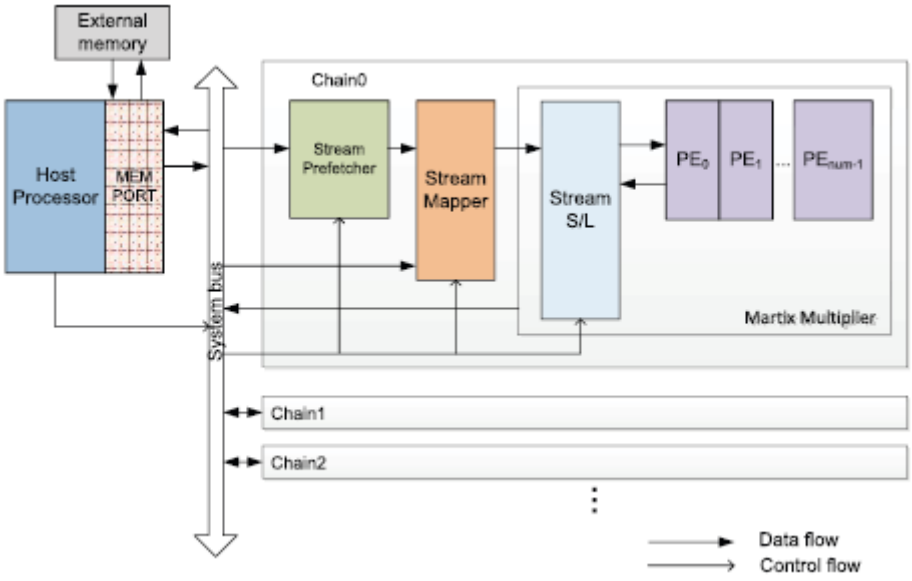


Figura 18 - Arquitetura da implementação de Qiao et al. [14].

O módulo multiplicador de matriz consiste numa unidade de S/L (*Store/Load*) e centenas de PE (ver figura 18). A unidade S/L carrega os dados para os PE e de seguida armazena os resultados. O módulo mapeador de fluxo mapeia os dados a serem utilizados nas convoluções para a unidade S/L do módulo multiplicador de matriz e o módulo *prefetcher stream* garante um acesso eficiente à memória externa. Este projeto converte as convoluções em multiplicação de matrizes para tornar a arquitetura mais flexível. Para evitar muitos acessos à memória nas camadas totalmente conectadas, são acumulados vários resultados das camadas anteriores numa matriz de forma a realizar uma única multiplicação de matrizes (assim os pesos utilizados na camada totalmente conectada são reutilizados várias vezes). A implementação deste trabalho obteve um desempenho de 77.8 GFLOPs utilizando uma FPGA Zynq-7000 XC7Z045.

Na implementação de CNN em FPGA de Ovtcharov et al. [15] é projetada uma arquitetura para obter o máximo desempenho na propagação direta nas camadas convolucionais (ver figura 19).

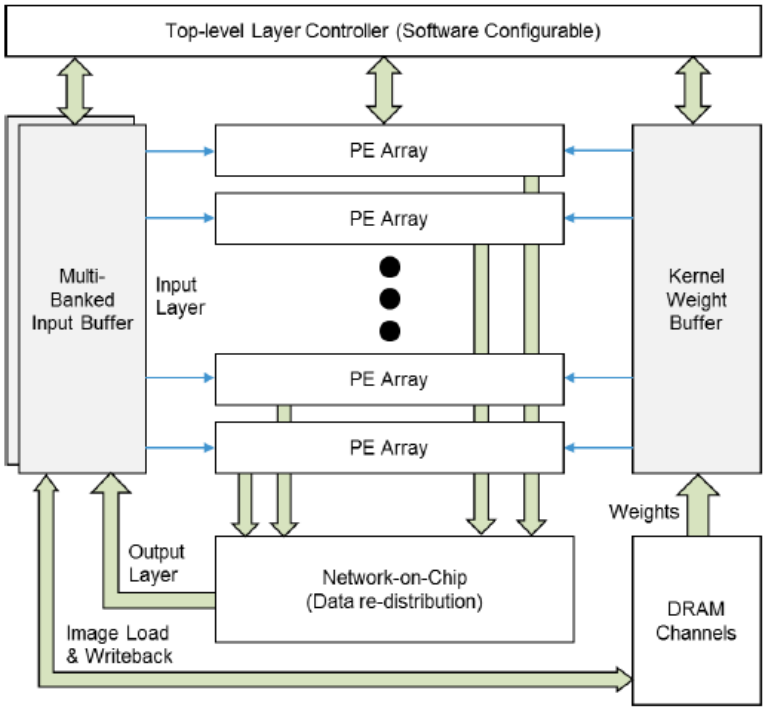


Figura 19 - Arquitetura da implementação de Ovtcharov et al. [15].

A arquitetura desta implementação é constituída por um mecanismo configurável por software, que permite a configuração de múltiplas camadas sem ser necessário a recompilação de hardware, por um esquema de buffer de dados e por uma rede de redistribuição de dados na FPGA (parte *on-chip* da arquitetura), que permitem minimizar o tráfego de dados para a memória externa (parte *off-chip* da arquitetura), e por um conjunto de PE distribuído espacialmente, que pode ser escalonado facilmente para conter milhares de PE. O projeto utilizou uma FPGA Stratix V D5 para implementar o sistema e obteve um desempenho de cerca 180 GFLOPs.

Na implementação de CNN em FPGA de Atul Rahman et al. [16] foi introduzida uma nova arquitetura designada por *Input-recycling Convolutional Array of Neurons*, ICAN (ver figura 20).

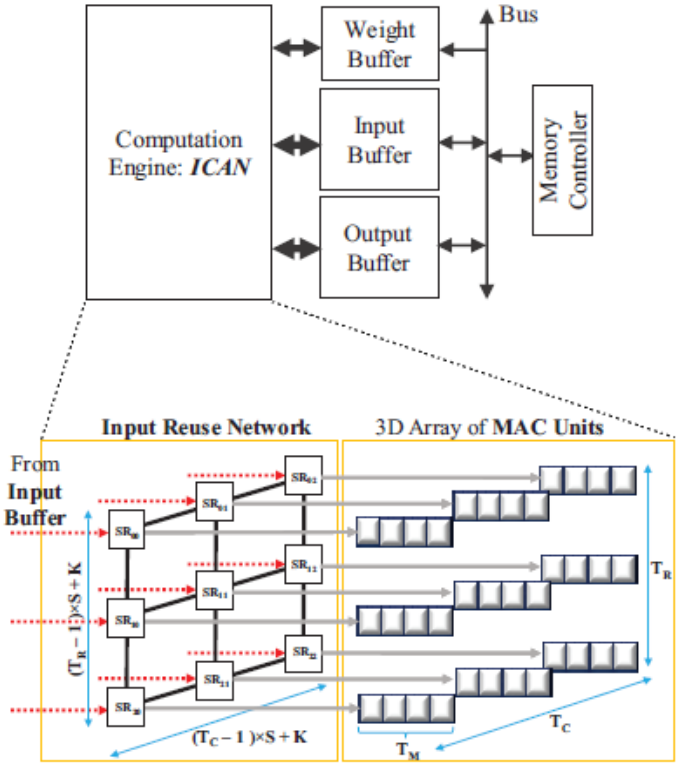


Figura 20 - Arquitetura implementada em Atul Rahman et al. [16].

Esta arquitetura é constituída por uma DRAM (*Dynamic Random-Access Memory*) externa, que guarda todos os dados da CNN, três *buffers* de dados (um para os dados de entrada, um para os pesos e outro para os dados de saída) e um módulo de computação ICAN, que é constituído por diversos registos que estão ligados a unidades MAC e que gradualmente deslocam os valores dos registos para outros registos (funcionam como registos de deslocamento), o que permite reutilizar os dados recebidos. Esta arquitetura foi desenvolvida para maximizar a taxa de computação, tendo como um dos seus critérios fundamentais a maximização da utilização dos DSP da FPGA para a implementação da CNN. Esta implementação de CNN em FPGA obteve um desempenho de 80.78 GFLOPs utilizando a FPGA Virtex-7 VX485T e uma frequência de 100MHz.

Na tabela 3 encontram-se os resultados das implementações de CNN em FPGA descritas anteriormente.

Modelo de FPGA	Virtex7 485t [13]	Zynq-7000 XC7Z045 [14]	Stratix V D5 [15]	Virtex-7 VX485T [16]
Frequência [MHz]	100	150	-	100
Representação dos dados	Vírgula Flutuante de 32bits	Vírgula Flutuante de 32bits	Vírgula Flutuante de 32bits	Vírgula Flutuante de 32bits
Desempenho [GFLOP/s]	61.62	78.8	182	80.78
Potência [W]	18.61	14.4	25	18.61
Eficiência [GFLOP/s/W]	3.31	5.47	7.28	4.34
Flip-Flop	205704 (33.87%)	218296*2 (49.93%)	-	-
LUT	186251 (61.3%)	183190*2 (83.8%)	-	-
BRAM	1024 (50%)	402*2 (73.8)	-	-
DSP	2240 (80%)	836*2 (92.8%)	-	2240 (80%)

Tabela 3 - Resultados de implementações de CNN em FPGA com representação de vírgula flutuante

Através dos resultados representados na tabela 3, verifica-se que é possível obter bons desempenhos de CNN implementadas em FPGA com um baixo consumo energético, comparativamente a implementações de CNN em GPU que para atingir valores de desempenho elevados, cerca de 3 TFLOPs, atinge um consumo energético de 250W [9]. É possível concluir também que a forma como a CNN é implementada na FPGA influencia o seu desempenho, pois comparando a implementação de Zhang et al. [13] e a implementação de Atul Rahman et al. [16] que utilizam o mesmo modelo de FPGA, a mesma representação dos dados e utilizam o mesmo número de DSP possuem um desempenho com cerca de 19 GFLOP/s de diferença.

Em geral, as arquiteturas têm procurado obter o melhor desempenho possível. A melhoria de desempenho deve-se em parte a algumas melhorias das arquiteturas e por outro devido à utilização de FPGAs de maior dimensão.

Os trabalhos descritos não procuram aplicar quaisquer otimizações em termos de algoritmo, implementando de forma direta as CNN com dados representados em vírgula flutuante.

Na secção seguinte, iremos descrever alguns métodos que têm sido utilizados para reduzir o peso computacional e as necessidades de memória. Desta forma, permitem

aumentar o desempenho de pico das plataformas e, consequentemente, a eficiência desempenho/consumo/área, mantendo praticamente a mesma precisão das redes.

3.3 Métodos de Melhoria da Eficiência das Implementações

Existem dois grandes grupos de métodos de melhoria da eficiência:

- Redução do tamanho dos operandos: este método inclui técnicas como alterar a representação dos dados de vírgula flutuante para vírgula fixa, reduzir o número de bits para representar os dados, realizar quantificação não-linear e realizar compartilhamento de pesos;
- Redução do número de operações: este método inclui técnicas como compressão, *pruning* e arquitetura de redes compactadas. A compressão permite reduzir o custo de energia e de área da rede, principalmente quando esta possui camadas ReLU que colocam muitos valores a zero. Além da compressão, a arquitetura da rede pode ser modificada de forma a não ler os pesos e não realizar os MAC de ativações com valor zero. Para facilitar o treino da rede, as CNN possuem normalmente parâmetros em excesso, assim a técnica de *pruning* consiste em remover os pesos redundantes (colocar a zero) da rede. A técnica de arquitetura de redes compactadas consiste em substituir os filtros de grande dimensão da rede por vários filtros de menor dimensão, para que o número total de pesos seja reduzido.

No trabalho desenvolvido nesta tese, centramo-nos apenas no primeiro grupo, ou seja, na redução do tamanho dos operandos. A redução do tamanho dos operandos inclui técnicas como quantificação linear e quantificação não-linear. Na CNN pode ser utilizado o mesmo método de quantificação para todos os constituintes da rede (dados de entrada, filtros, camadas) ou podem ser utilizados diferentes tipos de quantificação para os constituintes da CNN.

A. Quantificação linear

A quantificação linear inclui técnicas como mudar de uma representação de vírgula flutuante para vírgula fixa e reduzir o número de bits para representar os dados.

Uma das abordagens para reduzir o tamanho dos dados e a complexidade das operações consiste em converter os valores dos operandos e das operações da rede de vírgula flutuante para vírgula fixa. A representação da vírgula flutuante é constituída por três campos: o bit de sinal, a mantissa (ou fração), que contém o valor propriamente dito, e o expoente que representa um fator de escala. A representação da vírgula fixa dinâmica é constituída pelo bit de sinal, pela mantissa e pelo campo fracionário que representa um fator de escala que determina a localização da vírgula. O campo fracionário utilizado na representação da vírgula fixa dinâmica pode ser alterado de forma a pertencer a uma

determinada gama, o que é vantajoso para as CNN, pois a gama de valores adequada pode variar conforme as camadas e os filtros.

Utilizando a vírgula fixa dinâmica é possível reduzir o número de bits dos operandos sem ser necessário ajustar os valores dos pesos da rede. Desta forma, ao se optar por utilizar vírgula fixa dinâmica com uma largura de bit reduzida (redução do número de bits para representar os dados), estamos a reduzir a área de armazenamento necessária e a reduzir os requisitos de largura de banda de acesso à memória, com a consequente redução do consumo de energia (o acesso à memória externa é, em geral, o que consome mais energia). É de notar que optar por uma vírgula fixa dinâmica em vez de uma vírgula flutuante sem reduzir a largura de bit não reduz a área de armazenamento necessária nem o consumo de energia [2].

B. Quantificação não-linear

As distribuições dos pesos e dos valores de entrada nas CNN não são uniformes, desta forma a utilização de uma quantificação não-linear é vantajosa. Existem dois tipos de quantificações não-lineares normalmente utilizados, a quantificação de domínio logarítmico e a quantificação de aprendizagem (compartilhamento de pesos).

Na quantificação de domínio logarítmico, os níveis de quantificação são distribuídos de acordo com a distribuição logarítmica, assim os pesos e os valores de entrada são distribuídos de forma mais eficiente e justa, obtendo-se um erro de quantificação menor [2].

O compartilhamento de pesos consiste em que vários pesos compartilhem um único valor, para realizar esse compartilhamento pode-se usar a função *hash* ou o algoritmo *k-means*. A função *hash* agrupa os pesos e atribui um código para cada grupo de pesos. A função *k-means* verifica o valor dos pesos e agrupa-os conforme a proximidade dos seus valores. Ao compartilhar os pesos, é necessário guardar os valores dos pesos tal como o seu índice (código identificador respetivo gerado por uma das funções). Para obter o peso passa a ser necessário realizar duas leituras, primeiro o índice do peso e depois o peso compartilhado correspondente a esse índice. Desta forma, apenas é útil utilizar o compartilhamento de pesos se a largura de bit dos índices for menor que a largura de bit dos pesos, pois só assim reduz o custo de leitura e de armazenamento.

A técnica de redução do tamanho dos operandos já foi implementada em algumas redes. O método DOREFA-NET [12] implementa técnicas de redução do tamanho dos operandos no modelo da CNN AlexNet. Neste projeto a arquitetura da rede foi alterada para representar os pesos, as ativações e os gradientes da rede com diferentes número de bits (exceto na primeira e na última camada) e posteriormente a rede alterada foi treinada. Na tabela 4 estão representados alguns resultados obtidos através do método DOREFA-NET.

Número de bits para representar os pesos	Número de bits para representar as ativações	Número de bits para representar os gradientes	Precisão
32	32	32	55.9%
8	8	8	53.0%
1	4	32	50.3%
1	2	32	47.7%
1	1	32	40.1%
1	1	8	39.5%

Tabela 4 - Resultados do método DOREFA-NET para reduzir o tamanho dos operandos.

Como se pode verificar pela tabela 4, reduzir o número de bits para representar os dados da CNN para 8 bits apenas reduz a precisão em apenas 2.9%. É possível verificar também que reduzir o número de bits para representar as ativações e os pesos para 1 bit e reduzir o número de bits para representar os gradientes para 8 bits, faz reduzir bastante a precisão da rede, redução de 16.4%, comparativamente com a arquitetura da rede que utiliza 32 bits para representar os seus parâmetros.

Atualmente já existem algumas implementações de CNN em FPGA em que é aplicada a representação dos dados com vírgula fixa e com redução do seu tamanho [17] [18] [19].

Na implementação da CNN em FPGA proposta em [17], os dados são representados com vírgula fixa de 48 bits. A arquitetura é semelhante às utilizadas nas implementações de CNN em FPGA com vírgula flutuante de 32 bits, em que o processador executa a aplicação principal e descarrega para a FPGA os dados e as características da CNN. A FPGA tem acesso a uma memória externa dividida em três blocos para armazenar as imagens de entrada, os pesos e os dados intermédios. A FPGA contém vários elementos computacionais básicos que realizam a camada convolucional seguida por uma camada de não-linearidade, uma camada de agrupamento e por mais uma camada de não-linearidade. Cada elemento é responsável por gerar um mapa de características de saída. Nos casos em que é necessário processar mais imagens de entrada do que esse elemento é capaz, é calculado um mapa de características de saída intermédio posteriormente guardado na memória externa. Ao serem utilizados múltiplos elementos computacionais básicos é introduzido paralelismo na implementação da CNN, uma vez que uma imagem de entrada pode ser usada simultaneamente para calcular vários mapas de recursos de saída. A implementação da CNN em FPGA com vírgula fixa de 48 bits utilizando a FPGA Virtex 5 SX240T a uma frequência de 120MHz obteve um desempenho de 16 GOP/s

Qiu et al. [18] utiliza uma representação dos dados de vírgula fixa a 16 bits para os dados da CNN. A arquitetura proposta também está dividida entre um processador genérico (PS - *Processing System*) e lógica reconfigurável (PL - *Programmable Logic*) onde é implementada a arquitetura hardware dedicada. A parte PS da arquitetura consiste

num processador que ajuda na configuração da CNN e implementa a camada *softmax* da *CNN*. Os parâmetros, os resultados e as instruções envolvidas na implementação da CNN são guardados em memória externa. A parte PL consiste numa zona de hardware reconfigurável que é constituída pelo módulo computacional (conjunto de PEs), os *buffers*, módulos de acesso direto à memória (DMA - *Direct Memory Access*) e o controlador. A arquitetura contém dois *buffers* principais, o *buffer* de entrada, que armazena os pesos, os valores de *bias* e as imagens de entrada, e o buf*fer* de saída, que armazena os resultados intermédios e os resultados finais. A arquitetura do PE é divida em cinco partes: o módulo convolucional, que é responsável pelas convoluções dos dados de entrada com um filtro de tamanho três (3x3), o módulo *Adder-Tree*, que é responsável por somar todos os resultados das convoluções realizadas e pode somar o resultado intermédio caso necessário, o módulo de não linearidade, que realiza a camada de não linearidade, o módulo de *Max-Pooling* e o módulo de deslocamento de dados, que permite realizar a quantificação dinâmica. A arquitetura desta CNN em FPGA foi projetada para realizar o máximo paralelismo e reutilização de dados, e obteve um desempenho de 136.97GOP/s, utilizando a FPGA Zynq XC7Z045 à frequência de 150MHz.

Guo et al. [19] realizaram dois estudos, um com representação de dados com vírgula fixa de 16 bits e outro com representação de dados com vírgula fixa de 8 bits. Esta implementação utiliza uma quantificação dinâmica, em que é calculada a melhor posição da vírgula para o número de bits definidos para representar os dados. Este cálculo é realizado de forma a minimizar o erro entre o resultado da última camada da CNN utilizando vírgula flutuante de 32 bits para representar os dados e o resultado da última camada da CNN quando os seus dados são representados por vírgula fixa com menos bits. A posição da vírgula é calculada para cada uma das camadas para minimizar o nível computacional necessário para a implementação da quantificação dinâmica. A arquitetura da rede pode ser dividida em quatro partes: a matriz de PE, o *buffer* de mapas, a memória externa e o controlador. A matriz de PE é responsável pela convolução e utiliza dois tipos de paralelismo: o inter-PE e o intra-PE. O paralelismo inter-PE consiste em que o mesmo de dado de entrada seja utilizado para calcular diferentes resultados, ou seja, PEs diferentes compartilham os mesmos canais de entrada e usam filtros diferentes para calcular canais de saída diferentes em paralelo. O paralelismo intra-PE é definido na própria arquitetura do PE que permite a convolução de vários elementos em simultâneo. Esta implementação atingiu um desempenho de 187.8 GOP/s para uma representação de dados com vírgula fixa de 16 bits utilizando a FPGA Zynq XC7Z045 e uma frequência de 150MHz. A implementação da CNN em FPGA com representação de dados com vírgula fixa de 8 bits foi realizada com o objetivo de obter uma menor potência em detrimento de um melhor desempenho. Como tal, esta implementação obteve um

desempenho de penas 19.2 GOP/s numa FPGA Zynq XC7Z020 com uma frequência de operação de 100MHz.

Este último trabalho, apesar do baixo desempenho, é importante pois é uma das primeiras implementações de CNN grandes em FPGA de baixa densidade (ZYNQ XC7Z020). Este tipo de implementações vão permitir implementar CNN de grandes dimensões em sistemas embebidos com desempenhos razoáveis e com baixo consumo.

Na tabela 5 são apresentados os resultados das implementações de CNN em FPGA referidas anteriormente.

Modelo de FPGA	FPGA Virtex 5 SX240T [17]	Zynq XC7Z045 [18]	Zynq XC7Z045 [19]	Zynq XC7Z020 [19]
Rede CNN [GOP]	0.52	30.76	-	-
Frequência [MHz]	120	150	150	100
Representação dos dados	Vírgula Fixa de 48 bits	Vírgula Fixa de 16 bits	Vírgula Fixa de 16 bits	Vírgula Fixa de 8 bits
Desempenho [GOP/s]	16	136.97	187.8	19.2
Potência [W]	14	9.63	9.63	2
Eficiência [GOP/s/W]	1.14	14.22	19.5	9.6
Flip-Flop	-	127653 (29.2%)	127653 (29%)	24184 (23%)
LUT	-	182616 (83.5%)	182616 (84%)	27215 (51%)
BRAM	-	486 (86.7%)	486 (89%)	68 (49%)
DSP	-	780 (89.2%)	780 (90%)	198 (90%)

Tabela 5 - Resultados de CNN em FPGA com diferentes representações de dados.

Através dos resultados da tabela 5 verifica-se que representar os dados com vírgula fixa, mas com um maior número de bits não obtém uma boa eficiência comparativamente com outras implementações de CNN em FPGA, isto porque alterar a vírgula fixa sem reduzir o número de bits não melhora o desempenho. Comparando a implementação de Qiu et al. [18] com a implementação de Guo et al. [19] verifica-se que a forma como a arquitetura da CNN é projetada influencia o seu desempenho. No entanto, ambas as implementações possuem um desempenho elevado. Com estes resultados é possível concluir que as redes quando projetadas para obter o melhor desempenho são mais eficientes quando a representação dos dados é alterada de vírgula flutuante de 32 bits para vírgula fixa de 16 bits.

As implementações de CNN em FPGA com redução do tamanho dos operandos que existem atualmente apenas se focam em alterar a representação de vírgula flutuante para vírgula fixa e em reduzir o tamanho dos dados da CNN para 16 bits e 8 bits, não utilizando outro número de bits para representar os dados. Assim os estudos existentes de CNN em FPGA com redução do tamanho dos operandos são bastante limitados, pois os poucos que existem não implementam a mesma CNN, utilizam modelos de FPGA diferentes, e por vezes o foco da implementação é obter uma potência reduzida ao invés de um bom desempenho, o que para efeitos comparativos não é útil. Assim, neste trabalho, pretende-

se verificar o impacto da redução do número de bits para representar os dados, além de 16 e 8 bits, nas características de classificação da rede e nas implementações em FPGA. São estudados e testados diferentes tamanhos para representar as ativações e os pesos e para representar os parâmetros de diferentes camadas, de forma a verificar a influência da redução do tamanho de um parâmetro da rede, em separado, nas características da CNN e da implementação hardware. Além da utilização de técnicas descritas nas implementações anteriores, como por exemplo utilizar o máximo de DSP da FPGA, pretende-se também implementar algoritmos que permitam obter um maior desempenho, como por exemplo reduzir o número de multiplicações. A implementação das arquiteturas em hardware considera a arquitetura LiteCNN. Com o objetivo de estabelecer relações entre o tamanho dos operandos, a precisão da rede e o desempenho e área da arquitetura, desenvolveu-se um modelo da LiteCNN que permite estimar facilmente estas métricas para diferentes representações de dados.

Capítulo 4 – Arquitetura LiteCNN: Otimização com Redução do Tamanho dos Dados

Neste capítulo, descrevemos a arquitetura LiteCNN que visa a implementação de CNN de grande dimensão em FPGA de baixo custo com aplicação em sistemas embebidos. A versão atual da arquitetura suporta implementações com dados representados a 8 bits em vírgula fixa dinâmica.

No trabalho desta tese, procurou-se otimizar a LiteCNN com a utilização de outros formatos de representação dos dados, que não apenas o formato a 8 bits referido. Assim, neste capítulo, começamos com a descrição do ambiente e das metodologias utilizadas para estudar o impacto da redução do tamanho dos operandos na precisão das CNN.

De seguida, descrevemos a arquitetura LiteCNN, versão atual a 8 bits. De seguida, adaptamos a arquitetura para dar suporte a dados representados com outro número de bits (e.g., 4x4, 8x4, 8x2, etc.). No âmbito deste estudo, com o objetivo de estabelecer relações entre a precisão da rede e o desempenho e a área do hardware, são propostos dois modelos da arquitetura: um de desempenho e outro de área.

4.1 Otimização da CNN com Redução do Tamanho dos Operandos

O foco principal deste trabalho é a redução do tamanho dos operandos da CNN. Esta é uma técnica que permite implementar uma CNN reduzindo as necessidades computacional e de armazenamento. Contudo, a redução do tamanho dos operandos influencia a precisão da CNN. Assim, ao implementar uma CNN com redução do tamanho dos operandos é necessário ter em atenção a relação entre a precisão da rede e os requisitos a nível computacional e de memória da rede, e encontrar a melhor solução para a implementação da CNN tendo em conta as características da rede e do hardware onde esta será implementada. Atualmente, existem diversas ferramentas que permitem implementar e testar CNN, como o Caffe [21], o TensorFlow [24] e o Theano [25]. Neste trabalho, utilizámos a plataforma Caffe, em parte devido à possibilidade de explorar mais facilmente a precisão dos operandos com a integração da ferramenta Ristretto [9], que descrevemos mais à frente.

O Caffe utiliza um ficheiro para descrever a arquitetura da rede (formato *prototxt*) e após o seu treino gera um ficheiro (formato *caffemodel*) com os parâmetros da rede (pesos e valores *bias*) já treinados. A arquitetura da rede contem diversos parâmetros para a caracterizar que permitem dar suporte a um vasto conjunto de redes CNN. Os principais parâmetros são:

- *name*: nome da camada;

- *type*: tipo da camada;
- *bottom*: nome da camada anterior;
- *top*: nome da camada atual;
- *num_output*: número de filtros;
- *kernel_size*: dimensões do filtro, caso seja um filtro com largura e comprimento diferentes utilizam-se os parâmetros *kernel_h* e *kernel_w*;
- *weight_filte*r: modo de inicialização dos filtros;
- *bias_term*: modo de inicialização dos valores *Bias*;
- *pad*: valor de *padding* (podem utilizar-se valores diferentes para horizontal e para vertical através do uso dos parâmetros *pad_h* e *pad_w*);
- *stride*: valor do passo (podem utilizar-se valores diferentes para a horizontal e para a vertical através do uso dos parâmetros *stride_h* e *stride_w*);
- *pool*: função a utilizar nas camadas de agrupamento.

O treino da rede também pode ser caracterizado de forma específica, como por exemplo no número total de iterações do treino e no número de iterações com que é gerado um modelo com os parâmetros treinados da rede (ficheiro *caffemodel*).

O Caffe contém uma extensão designada por Ristretto que permite testar e treinar a rede com diferentes números de bits para representar os dados, de forma a encontrar um bom equilíbrio entre a taxa de compressão e a precisão da rede. O Ristretto permite implementar três formas de quantificação: vírgula fixa dinâmica, *minifloat* e potências de dois. Na quantificação com vírgula fixa dinâmica, o Ristretto divide cada camada em três grupos: as entradas da camada, os pesos e as saídas da camada; para que seja possível encontrar a melhor posição da vírgula para cada um dos grupos em separado, visto que os pesos normalmente possuem valores inferiores às ativações (entradas e saídas das camadas). O Ristretto quando utilizado para a vírgula fixa dinâmica escolhe o número de bits suficientes para a parte inteira do maior valor para evitar a saturação, e cada valor é dado pela equação 2, em que *B* representa o número de bits, *s* representa o bit de sinal, *fl* representa a parte fracionária do valor e *xi* representa a mantissa do valor.

$$n = (-1)^{s} \times 2^{-fl} \times \sum_{i=0}^{B-2} \left(2^{i} \times x_i\right)$$

Equação 2 - Representação do valor no Ristretto com vírgula fixa dinâmica.

Na figura 21 estão ilustrados alguns exemplos de representação de dados com vírgula fixa dinâmica utilizada no Ristretto.

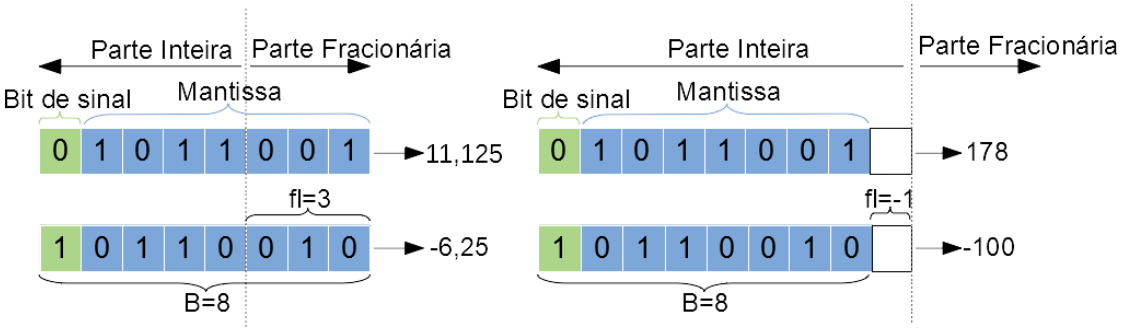


Figura 21 - Exemplo de representação dos dados com vírgula fixa dinâmica no Ristretto.

Assim, quando o Ristretto aplica a quantificação de vírgula fixa dinâmica, a arquitetura da rede passa a ser caracterizada por mais seis parâmetros: os parâmetros *bw_layer_in*, *bw_layer_out* e *bw_params*, que indicam o número de bits para representar os valores de entrada da camada, os valores de saída da camada e os pesos, respetivamente; e os parâmetros *fl_layer_in*, *fl_layer_out* e *fl_params*, que indicam quantos bits são utilizados para representar a parte fracionária dos valores de entrada da camada, dos valores de saída da camada e dos pesos, respetivamente.

Outro tipo de quantificação utilizada pelo Ristretto designa-se por *Minifloat* e consiste em reduzir o número de bits da representação de dados com vírgula flutuante. O Ristretto quando utilizado com *Minifloat* utiliza uma representação de vírgula flutuante em que o valor representado é dado pela equação 3, e a representação é constituída por três campos, o bit de sinal representado por S, a mantissa representada pela letra M, que contém o valor, e o campo expoente representado por E, que consiste num valor de escala que determina a posição da vírgula.

$$n = (-1)^S \times 2^E \times M$$

Equação 3 - Representação dos dados no Ristretto com Minifloat.

O Ristretto diminui o número de bits do *Minifloat* para 16 bits, 8 bits ou ainda menos bits, e determina o número de bits para o expoente de forma a não ocorrer saturação. O Ristretto não suporta números desnormalizados como o NaN (*Not a Number*) e o INF (infinito), desta forma o INF é substituído por números saturados e o NaN é substituído por zero. Na figura 22 está ilustrado um exemplo de representação de dados utilizando a quantificação *Minifloat* do Ristretto.

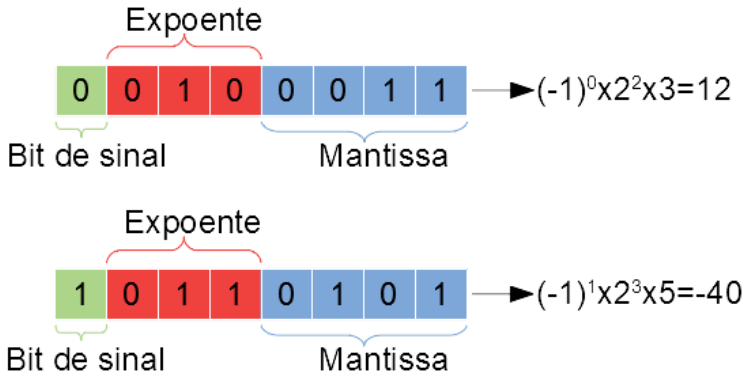


Figura 22 - Exemplo de representação dos dados com vírgula flutuante no Ristretto.

Ao ser a aplicada a quantificação *Minifloat* pelo Ristretto a arquitetura da rede passa a ser caracterizada por mais dois parâmetros, o parâmetro *mant_bits*, que indica o número de bits para representar a mantissa, e o parâmetro *exp_bits*, que indica o número de bits para representar o expoente.

Quando o Ristretto utiliza uma quantificação com parâmetros de potência de dois, os dados são representados de forma semelhante à quantificação *Minifloat* do Ristretto. Porém, não são atribuídos bits para a mantissa. Assim, o valor dos dados representados é dado pela equação 4, em que *S* representa o bit de sinal e *E* representa o valor do expoente. O valor do expoente é um inteiro e, geralmente, é negativo para os parâmetros da rede.

$$n = (-1)^S \times 2^E$$

Equação 4 - Representação dos dados no Ristretto com quantificação de parâmetros de potência de dois.

A vantagem da quantificação parâmetros de potência de dois do Ristretto é que permite que as multiplicações nas camadas convolucionais e nas camadas totalmente conectadas sejam substituídas por deslocamentos de bits (*shifts*).

O Ristretto utiliza dois ficheiros sobre a rede que se pretende reduzir o número de bits para representar os dados, um ficheiro é do formato *prototxt* que contém a características da arquitetura da CNN e outro ficheiro é do formato *caffemodel* que contém os pesos da rede CNN previamente treinada com uma representação de dados de vírgula flutuante de 32 bits. O Ristretto permite definir a quantificação que se pretende aplicar à rede e a margem de erro. Esta margem de erro é o valor que influencia o resultado final do Ristretto, pois é o valor que a precisão da rede pode diminuir ao reduzir o número de bits para representar os dados da CNN. No final, o Ristretto gera um ficheiro do formato *prototxt* que contém a arquitetura da rede alterada para representar os dados com menos bits.

O número de bits para representar os dados da CNN resultante do Ristretto apenas pode ser influenciado pelo utilizador através do parâmetro da margem de erro, o que é

limitativo quando se pretende estudar a influência de um número de bits específico para representar os dados da rede. Para ser possível treinar uma rede com um determinado número de bits para representar os dados e que não tenha sido possível extrair através do Ristretto, propôs-se e desenvolveu-se neste trabalho um fluxo de treino que permite obter pesos com um determinado tamanho (ver figura 23).

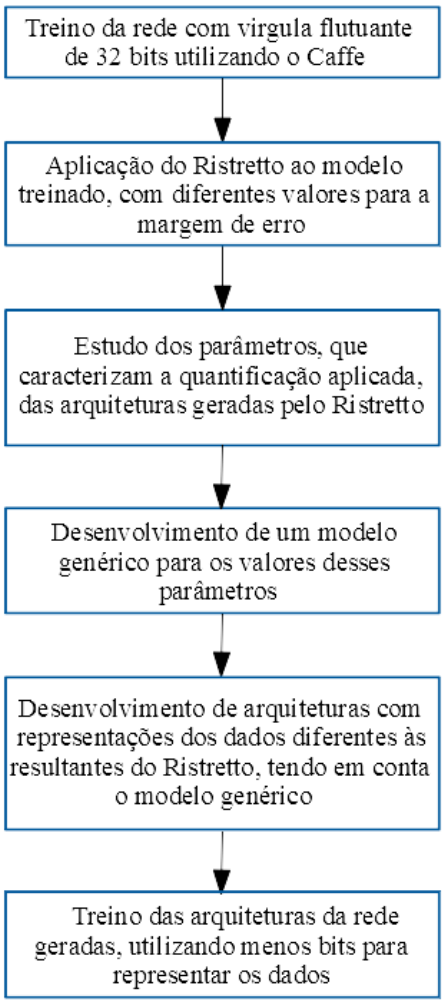


Figura 23 - Etapas a realizar neste estudo, na aplicação do Ristretto.

Como se pode verificar pela figura 23, a primeira etapa consiste em treinar a rede com uma representação de vírgula flutuante de 32 bits através da arquitetura da rede descrita no ficheiro *prototxt*. O passo de treino gera um ficheiro com o formato *caffemodel*, que contem os valores dos pesos treinados. De seguida, através do ficheiro que contêm a arquitetura da rede e do ficheiro que contém os pesos treinados é aplicado o Ristretto com a configuração da quantificação e da margem de erro pretendidos. Este passo é repetido alterando apenas o valor da margem de erro, de forma a gerar vários ficheiros com arquiteturas que utilizam diferentes números de bits para representar os dados da rede. Após a aplicação do Ristretto é feita uma extração de todos os valores dos parâmetros que caracterizam a quantificação definida. Com esses valores é verificado como estes variam consoante o número de bits que é utilizado para representar os dados da rede e é gerado

um modelo genérico que permite determinar os valores para todos os parâmetros que caracterizam a quantificação definida conforme o número de bits pretendidos para representar os dados da rede. Através dos dados do modelo genérico são gerados ficheiros *prototxt* com arquiteturas da rede utilizando o número de bits pretendido e os valores dos parâmetros que caracterizam a quantificação definida. Por fim essas arquiteturas são treinadas e além dos ficheiros com os pesos treinados associados a cada arquitetura é determinado a precisão da rede para cada uma das arquiteturas.

Neste trabalho apenas foram consideradas implementações com formatos em vírgula fixa dinâmica, uma vez que os resultados indicaram que os formatos *miniFloat* e de potências de 2 não traziam vantagens em termos de precisão da rede.

4.2 Arquitetura LiteCNN: versão 8 bits

A arquitetura LiteCNN tem uma estrutura configurável que executa uma camada de cada vez. O núcleo principal da arquitetura calcula convoluções 3D entre as ativações e os pesos explorando paralelismo de saída (calcula vários mapas de saída em paralelo), paralelismo do mapa de saída (calcula várias ativações de um mapa de saída em paralelo) e paralelismo do *kernel* (com a paralelização do cálculo das convoluções).

Em termos estruturais, a arquitetura contém um núcleo de cálculo com vários PE, um buffer de memória, responsável por armazenar a imagem inicial e os resultados intermédios, uma memória externa, e módulos responsáveis por enviar e receber ativações e pesos entre o buffer de memória e o núcleo de cálculo (ver figura 24).

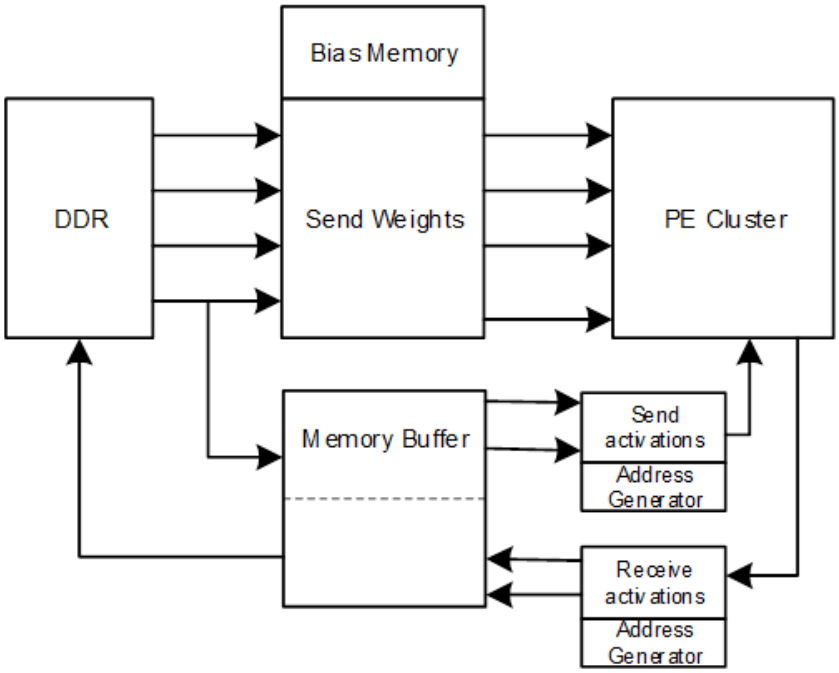


Figura 24 - Arquitetura LiteCNN.

A execução de uma CNN na arquitetura processa-se do seguinte modo:

1. Configura-se a arquitetura para uma camada específica (convolucional ou totalmente ligada). Adicionalmente, configura-se a existência de *pooling*;
2. Envia-se a imagem a processar para o buffer de memória e os pesos dos filtros da memória externa para os PE (enviados pelo módulo *sendWeights*) juntamente com o *Bias* (armazenado no *Bias memory*) e os produtos redundantes do algoritmo da redução do número de multiplicações descrito em 4.1.1 que envolvem apenas os pesos (como é o caso de W_0W_1, W_2W_3, W_4W_5 e W_6W_7). Esta soma é armazenada numa memória local do sistema. Cada PE recebe um *kernel* e é responsável por calcular o mapa de saída associado a esse *kernel*;
3. De seguida, a imagem ou as ativações são enviadas em *broadcast* do *buffer* para todos os PE (*sendActivations*). Enquanto é feita a sua leitura são calculados e armazenados os valores redundantes da aplicação do algoritmo da redução do número de multiplicações que envolvem apenas os valores do mapa de características (como é o caso de P_0P_1, P_2P_3, P_4P_5 e P_6P_7 no exemplo da figura 21);
4. Após o cálculo dos PE entre o filtro e a parte do mapa de características que o PE recebeu, os resultados são enviados de volta para o módulo recetor de ativações (*receiveActivations*) que subtrai o *Bias* e os valores redundantes das multiplicações e guarda o resultado no buffer de memória;
5. Quando a convolução entre os filtros enviados para os PE e a imagem estiver concluída são carregados novos filtros nos PE e processo repete-se até serem executados todos os filtros.

A versão atual da arquitetura organiza os PE em *clusters* (ver figura 25), em que cada cluster tem um porto dedicado de acesso à memória externa para receber os pesos.

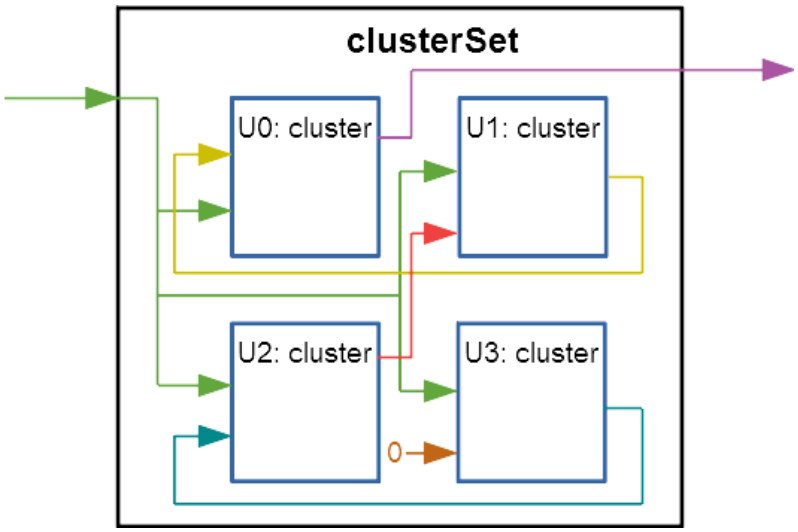


Figura 25 – Esquema genérico do módulo "clusterSet".

Cada módulo *cluster* é constituído por vários PE. O número de PE por *cluster* depende da disponibilidade de recursos da FPGA e do número de bits utilizados para representar os dados da CNN (ver exemplo da arquitetura para representação dos dados com 8 bits na figura 26).

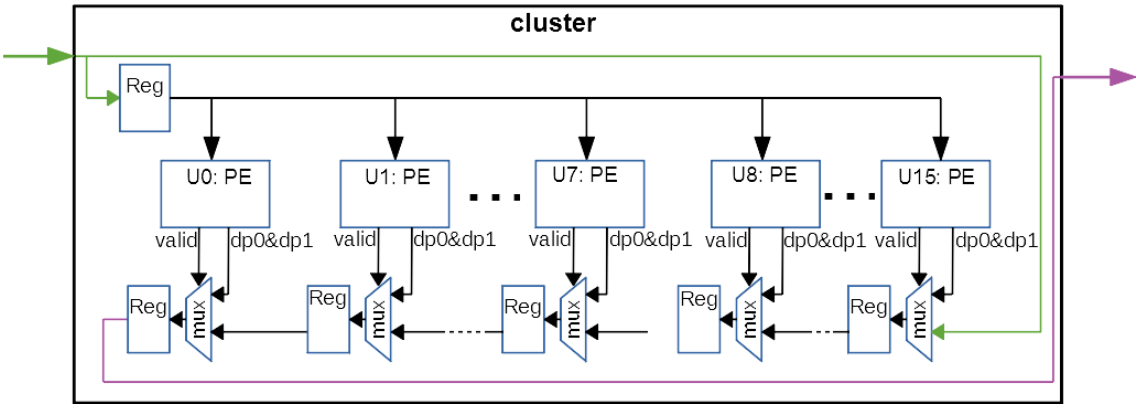


Figura 26 - Esquema genérico do módulo "cluster".

Na figura 26 podemos ver os vários elementos de processamento e a rede que permite comunicar os resultados de volta para o buffer de memória.

O elemento de processamento, PE, é constituído por um módulo designado por *Cell*, uma célula de cálculo de produtos internos, e por memória local onde são armazenados os pesos dos filtros (ver figura 27).

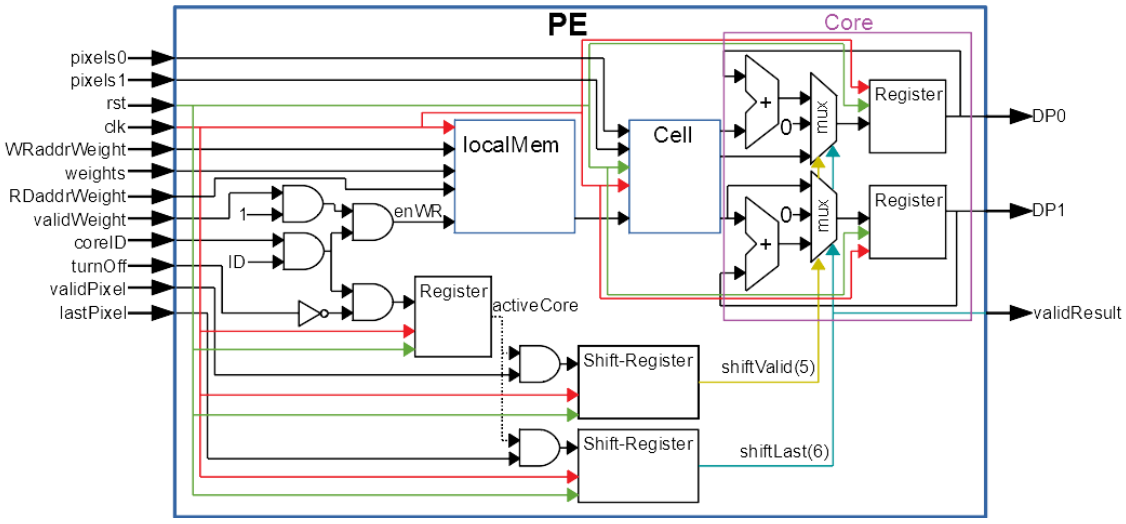


Figura 27 - Esquema genérico do PE.

Cada PE recebe e armazena na memória local os pesos de um *kernel* diferente. Cada PE pode aplicar os filtros a mais do que um bloco de ativações de entrada, produzindo assim diferentes ativações do mapa de saída. O número de ativações de saída calculadas em paralelo é configurável. No exemplo da figura, este valor é de dois, pois recebe dois blocos de ativações em paralelo (*pixels0* e *pixels1*). Por fim, o PE lê múltiplos pesos e ativações em paralelo com um único acesso à memória (a versão atual lê palavras de 64 bits a que corresponde 8 filtros de 8 bits em paralelo) permitindo explorar o paralelismo no cálculo do produto interno.

O cálculo dos produtos internos, ou seja, das multiplicações-acumulações do PE utiliza uma técnica de redução do número de multiplicações através da reorganização do produto interno. Considerando, por exemplo, uma convolução entre duas matrizes, é possível através desta técnica reduzir o número de multiplicações para metade, como está ilustrado na figura 28.

W0	W1	W2	W3
W4	W5	W6	W7

*

P0	P1	P2	P3
P4	P5	P6	P7

↓ Resultado da convolução:

W0P0+W1P1+W2P2+W3P3+W4P4+W5P5+W6P6+W7P7 → 8 multiplicações

↓ Aplicando a técnica de reduzir o número de multiplicações na convolução:

(W0+P1)(W1+P0)+(W2+P3)(W3+P2)+(W4+P5)(W5+P4)+(W6+P7)(W7+P6) → 4 multiplicações

↔W0W1+W0P0+P1W1+P1P0+W2W3+W2P2+P3W3+P3P2+W4W5+W4P4+P5W5+P5P4+W6W7+W6P6+P7W7+P7P6

Figura 28 - Técnica de redução do número de multiplicações na convolução.

Como se pode verificar através da figura 28, a segunda equação de cálculo do produto interno só necessita de metade das multiplicações da equação original. No entanto, além das parcelas das multiplicações originais (realçadas a verde na figura 28) obtêm-se parcelas suplementares que têm de ser determinadas e subtraídas. Apesar de estas parcelas

implicarem multiplicações, os produtos entre os pesos (W_xW_y) podem ser pré-calculados para cada um dos filtros e os produtos entre as ativações (P_xP_y) podem ser calculados apenas uma vez para todos os filtros. . Considerando *k* filtros, a razão entre o número de multiplicações da segunda equação com o da primeira é dada por 1/2 + 1/(2k). Quantos mais filtros forem processados em paralelo, mais a razão tende para o valor ideal de 1/2 (metade das multiplicações).

As multiplicações podem ser realizadas com LUTs ou com DSPs. Um DSP permite um processamento rápido de dados com baixo consumo energético e é bastante versátil pois permite a realização de diversas operações. A arquitetura do DSP está ilustrada na figura 29 e é constituída por um multiplicador, um pré-somador de 30 bits e um acumulador de saída de 48 bits.

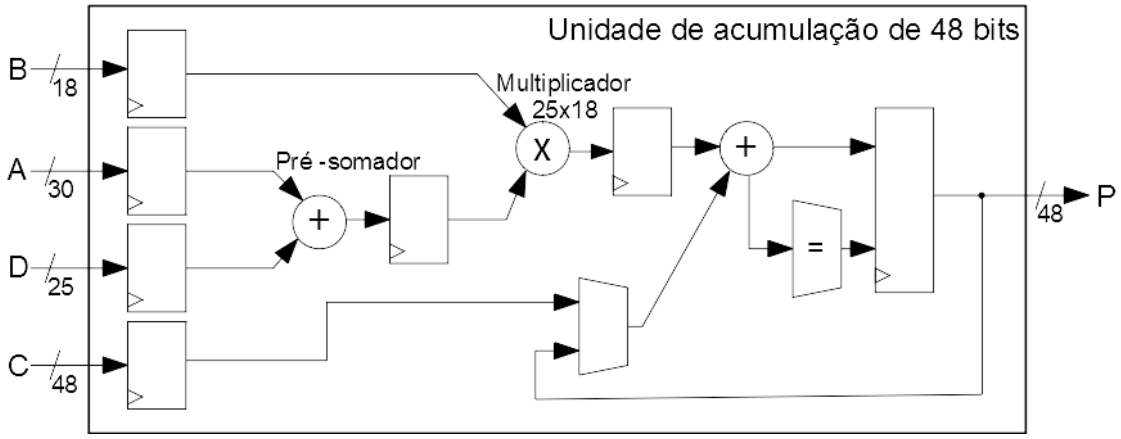


Figura 29 - Arquitetura do DSP para uma unidade de acumulação de 48 bits.

Como se pode verificar pela figura 29, o DSP realiza a operação (A+D)*B+C, esta operação permite utilizar o pré-somador para realizar uma das somas do método descrito, sendo a outra realizada com LUTs.

Utilizando esta técnica, cada PE implementa 8 multiplicações em paralelo para realizar o cálculo de dois produtos internos em paralelo com 8 parcelas cada um (ver figura 30).

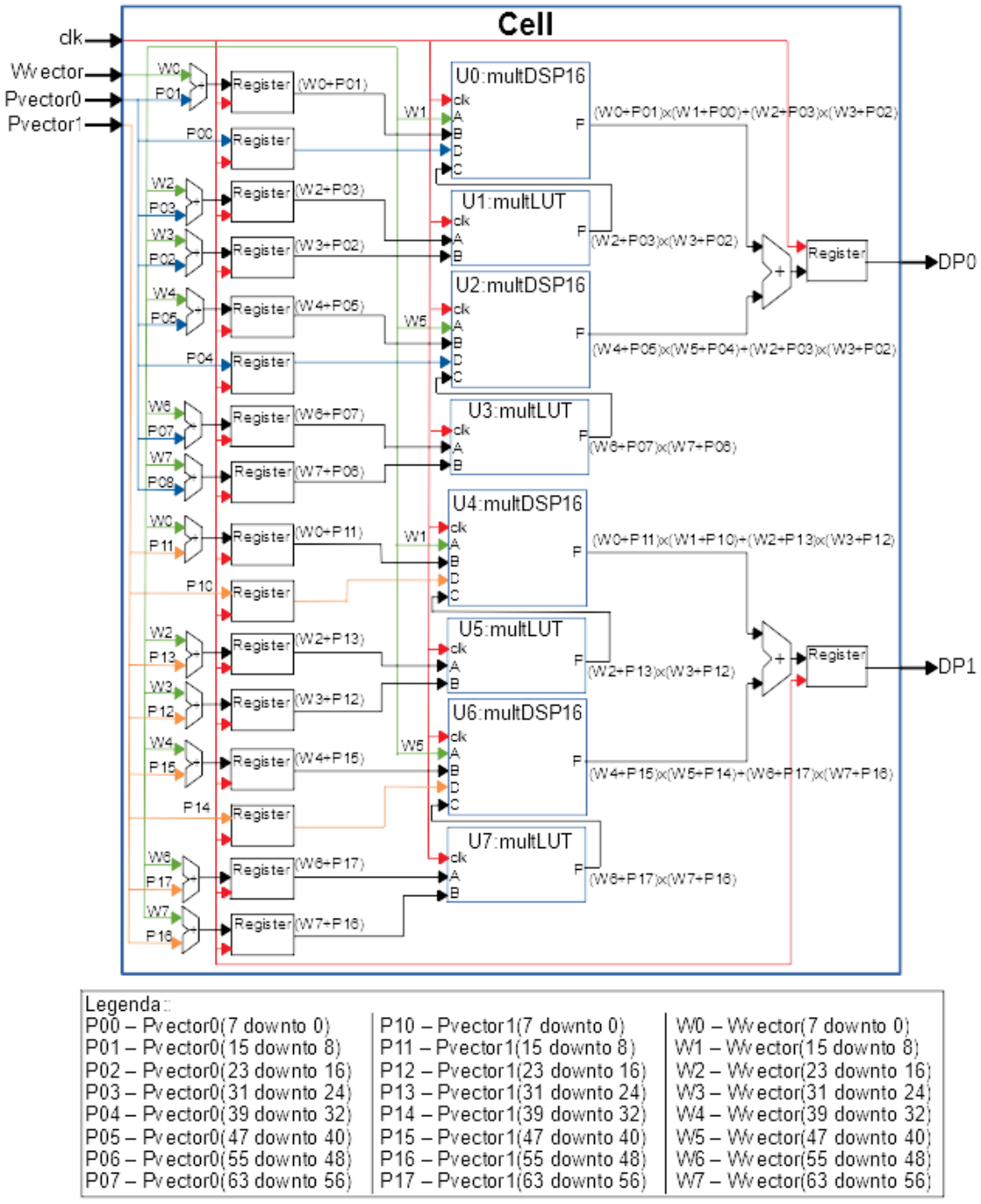


Figura 30 - Arquitetura do módulo de cálculo dos produtos internos.

Para maximizar a utilização da FPGA, A LiteCNN implementa multiplicações com DSP e com LUT.

4.3 Arquitetura LiteCNN com Tamanho de Dados Configurável

A arquitetura LiteCNN pode ser facilmente implementada com dados representados com tamanhos diferentes de 8 bits, nos casos em que as ativações e os pesos de todas as camadas têm os mesmos tamanhos (e.g., 16x16, 7x7, 5x5, 8x2, 8x4). Nestes casos, os núcleos de cálculo de produtos internos dos PE são configurados para realizar operações

de acordo com o tamanho dos dados. O número de produtos realizados em paralelo varia com estes tamanhos, considerando sempre o acesso à memória a 64 bits. Por exemplo, com dados a 5 bits, teríamos 12 unidades em paralelo.

Nos casos em que as ativações ou os pesos têm tamanhos diferentes em camadas diferentes, optou-se por armazenar os dados com o tamanho original, mas estender o tamanho dos dados de menor dimensão para o tamanho dos de maior dimensão e utilizar as mesmas unidades de cálculo (ver figura 28).

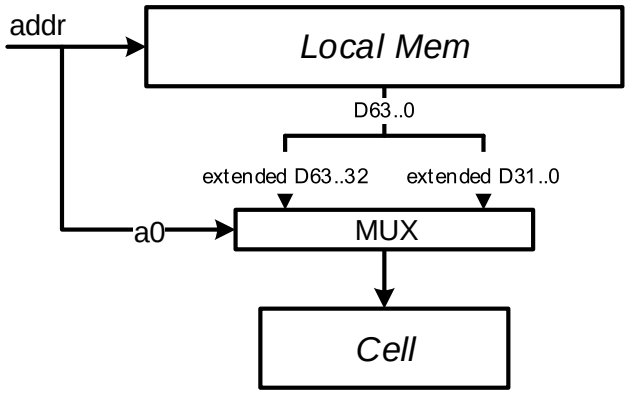


Figura 28 - Alteração do PE para suportar pesos de tamanhos diferentes.

Por exemplo, se as camadas de convolução usarem a representação 8x4 e as totalmente conectadas usarem 8x2, então as unidades de cálculo são implementadas para suportar operações de 8x4. Neste exemplo, os dados a dois bits são estendidos a quatro bits antes de entrarem no cálculo das convoluções. Esta solução não traz vantagem em termos computacionais, mas tem vantagem em termos de acesso à memória quando se utilizam dados de menor dimensão pois o acesso é feito à dimensão original. Como veremos nos resultados, a redução da dimensão dos dados verifica-se tipicamente nas camadas totalmente conectadas, que são as camadas com maior número de pesos a serem carregados de memória.

Uma outra alteração que se considerou na implementação dos núcleos de cálculo está relacionada com a aplicação do método de redução de multiplicações do produto interno. Nos casos em que as ativações têm o mesmo tamanho dos pesos, o método pode ser aplicado do mesmo modo para diferentes tamanhos de dados. Nos casos, em que são diferentes, não compensa aplicar a técnica, pois as multiplicações têm a dimensão dos dados de maior dimensão. Por exemplo, com multiplicações de 8x2, ficaríamos com multiplicações de 9x9. Para estes casos, optou-se por aplicar a técnica proposta em [26]. A técnica foi generalizada para outros tamanhos de dados seguindo o mesmo método descrito no artigo.

Considerando estas alterações, obtivemos os resultados da tabela 6, referentes às células de cálculo dos produtos internos.

Camada Convolucional (AtivaçãoxPeso)		**4x4**	**5x5**	**6x6**	**8x8**	**16x16**	**8x2**	**8x4**	**8x4**	**8x8**
Camada Totalmente Conectada (AtivaçãoxPeso)		**4x4**	**5x5**	**6x6**	**8x8**	**16x16**	**8x2**	**8x2**	**8x4**	**8x4**
Módulo PE0	**LUTs**	668	628	614	574	643	268	500	460	670
	DSP	4	4	4	4	7	4	4	4	4
	BRAM	1	1	1	1	2	1	1	1	1
Módulo PE1	**LUTs**	720	678	684	696	982	288	580	540	792
	DSP	3	3	3	3	6	3	3	3	3
	BRAM	1	1	1	1	2	1	1	1	1
Módulo PE2	**LUTs**	-	-	-	-	-	308	620	580	-
	DSP	-	-	-	-	-	2	2	2	-
	BRAM	-	-	-	-	-	1	1	1	-

Tabela 6 – Ocupação de recursos das células de cálculo dos produtos internos com diferentes configurações (ZYNQ7020).

4.4 Modelo de Área da LiteCNN

Para estimar a área da arquitetura em função da rede CNN foi desenvolvido um modelo de área genérico. Para este modelo é contabilizado o número de LUTs, de DSP e de BRAM para cada módulo da arquitetura LiteCNN com diferentes representações para as ativações e para os pesos das camadas convolucionais e camadas totalmente conectadas. O modelo de área genérico baseia-se nos valores obtidos após implementação das células (ver tabela 6) e dos restantes blocos da arquitetura LiteCNN (ver tabela 7).

Camada Convolucional (AtivaçãoxPeso)		4x4	5x5	6x6	8x8	16x16	8x2	8x4	8x4	8x8
Camada Totalmente Conectada (AtivaçãoxPeso)		4x4	5x5	6x6	8x8	16x16	8x2	8x2	8x4	8x4
Memory Buffer	**LUTs**	774	774	774	775	882	730	730	730	730
	BRAM	70	70	70	70	72	70	70	70	70
Send activations	**LUTs**	860	912	1036	1133	2784	419	419	419	419
sendWeights	**LUTs**	4*44	4*45	4*45	4*46	4*44	4*44	4*44	4*44	4*44
biasStore	**LUTs**	504	516	534	588	677	456	468	468	588
receivePixels	**LUTs**	487	496	503	555	603	458	458	458	458
	BRAM	2	2	2	2	2	1	1	1	2
Controlo	**LUTs**	52	52	52	55	28	28	28	28	28
Cluster	**Core0**	28	28	28	28	28	0	0	0	28
	Core1	36	36	36	36	4	0	64	0	36
	Core2	0	0	0	0	0	96	0	64	0
Total	**LUTs**	47477	44922	44895	44418	27082	31835	39399	39399	49671
	DSP	220	220	220	220	220	192	192	128	220
	BRAM	136	136	136	136	138	119	103	103	136
Número de Cores		64	64	64	64	32	96	64	64	64
Número de operações por Core (MACs)		32	24	20	16	8	16	16	16	16

Tabela 7 - Modelo de área.

As BRAM referidas no modelo genérico correspondem a blocos de RAM de 36kbits, exceto nas configurações em que se utiliza 8 bits para as ativações e 2 bits para os pesos, 8 bits para as ativações e 4 bits para os pesos e em que se utiliza 8 bits para as ativações e 4 bits para os pesos das camadas convolucionais e 8 bits para as ativações e 2 bits para os pesos das camadas totalmente conectadas. Nestes casos, utilizam-se as BRAM 18kbits. Assim, para esses casos, o número de BRAM total dos módulos é dividido por dois, para ser equivalente a um bloco RAM de 36kbits.Para determinar a área da arquitetura, basta multiplicar o número de PE pela sua área e somar as áreas dos restantes blocos da arquitetura. Através do modelo representado na tabela 7 verifica-se que o número de núcleos de computação que constituem o módulo *Cluster* é o que influencia a área utilizada pelas arquiteturas, sendo que os valores de LUTs, DSP e BRAM dos restantes módulos é praticamente constante para as diferentes arquiteturas apresentadas. O número

de PE utilizados na arquitetura pode ser alterado desde que o número total de LUTs, DSP e BRAM não exceda os recursos da FPGA onde é implementada esta arquitetura.

Comparando as implementações do modelo de área em que a representação das ativações e dos pesos é igual, incluído para diferentes tipos de camada, verifica-se que o número de LUTs aumenta ligeiramente com a diminuição do número de bits para representar os dados, no entanto, esse aumento é compensado com um aumento do número de operações MAC por PE, o que aumenta o desempenho da rede e por sua vez o seu paralelismo.

4. Modelo de Desempenho da LiteCNN

Para estimar o desempenho da arquitetura na execução de uma determinada CNN, é necessário saber o número de bytes que é necessário transferir entre a memória externa e a LiteCNN para cada uma das camadas, o tempo de transferência desses dados e o número de ciclos necessários para realizar todos os cálculos de cada camada. Através destes dados é possível estimar o atraso total, que corresponde ao tempo de execução de uma inferência da CNN. Este valor depende das características da rede, nomeadamente, do número e do tipo de camadas, do tamanho e da quantidade de filtros e do tamanho da imagem de entrada.

O número de bytes transferidos por camada refere-se ao número total de bytes de todos os filtros utilizados pela camada e depende do número de *Kernels*, *nKernels*, do tamanho dos *kernels* ou das convoluções 3D, *convSize*, e do número de bits utilizados para representar os pesos, *nBits*. Este valor pode ser calculado de acordo com a equação 5.

$$Número\ de\ bytes = nKernels \times convSize \times \frac{nBits}{8}$$

Equação 5 - Número de bytes transferidos por camada.

O tempo de transferência dos pesos depende do número de bytes transferidos e da largura de banda de acesso à memória externa, *BW*, onde estão armazenados os filtros. O tempo de transferência é dado pela equação 6.

$$Tempo\ de\ transferência = \frac{Número\ de\ bytes}{BW}$$

Equação 6 - Tempo de transferência.

O número de ciclos das camadas convolucionais corresponde ao número de ciclos necessários para realizar todas as convoluções da camada convolucional. O número de ciclos necessários para realizar todas as convoluções da camada convolucional depende do número de cores (que depende da configuração da LiteCNN), do número de MACs realizados por core de cada arquitetura (também configurável), do número de Kernels, do tamanho da convolução e do número de convoluções em cada camada convolucional.

Assim o número de ciclos para a cada camada convolucional é dado pela equação 7, em que *nCores* corresponde ao número de cores, *nMAC* corresponde ao número de MACs realizados por Core e *nConv* corresponde ao número de convoluções realizadas por cada camada.

$$Ciclos\ Conv = \frac{nKernels}{nCores} \times \frac{nConv \times ConvSize}{nMACs}$$

Equação 7 - Número de ciclos necessários para executar uma camada convolucional.

O cálculo do número de ciclos necessários para executar a camada totalmente conectada é semelhante ao das camadas convolucionais. No entanto, a equação que permite calcular o número de ciclos necessários para realizar o cálculo da convolução 1D das camadas totalmente conectadas tem de ser ajustado. Enquanto que no cálculo das convoluções 3D das camadas convolucionais os PEs suportam o cálculo paralelo de dois produtos internos independentes, no caso das camadas totalmente conectadas apenas é possível calcular um produto de cada vez pois os kernels não são reutilizados. Assim, a expressão utilizada para as camadas convolucionais é multiplicada por um fator, *nParallel*, igual ao número de produtos internos realizados em paralelo pela arquitetura (na versão atual da LiteCNN este valor é igual a 2) .Assim, o número de ciclos de execução da camada totalmente conectada é dado pela equação 8, na qual são utilizadas as abreviaturas já referidas anteriormente.

$$Ciclos\ FC = \frac{nKernels}{nCores} \times \frac{ConvSize}{nMACs} \times nParallel$$

Equação 8 - Número de ciclos necessários para executar uma camada totalmente conectada.

O tempo de atraso da camada convolucional corresponde ao tempo total necessário para processar a camada convolucional, ou seja, inclui o tempo de transferência dos dados utilizados nessa camada e o tempo de processamento dessa camada convolucional. O tempo de processamento da camada convolucional depende do número de ciclos necessário para executar a camada convolucional e da frequência utilizada pela arquitetura LiteCNN. O tempo de atraso da camada convolucional é dado pela equação 9.

$$Atraso\ Conv = Tempo\ transferência + \frac{Ciclos\ Conv}{Frequência}$$

Equação 9 - Tempo de atraso das camadas convolucionais.

O tempo de atraso da camada totalmente conectada é dado pelo maior tempo entre o tempo de transferência dos pesos para a convolução e o tempo de processamento (tempo que demora a realização da convolução da camada totalmente conectada. O tempo de processamento da camada totalmente conectada depende do número de ciclos necessário

para executar a camada totalmente conectada e da frequência utilizada pela arquitetura LiteCNN. O tempo de atraso da camada totalmente conectada é dado pela equação 10.

$$Atraso\ FC = \begin{cases} Tempo\ de\ transferência & ,\ se\ tempo\ transferência \geq tempo\ processamento \\ \dfrac{Ciclos\ FC}{Frequência} & ,se\ tempo\ transferência < tempo\ processamento \end{cases}$$

Equação 10 - Tempo de atraso das camadas totalmente conectadas.

O tempo de atraso total corresponde ao tempo de execução de uma inferência da CNN e depende do tempo de transferência da imagem inicial, do tempo de atraso de todas as camadas convolucionais, do tempo de atraso de todas camadas totalmente conectadas e do tempo de transferência dos resultados obtidos pela arquitetura LiteCNN. O cálculo do tempo de atraso total é realizado através da equação 11.

$$Atraso\ Total = \frac{Tamanho\ da\ Imagem}{BW} + Atraso\ Conv + Atraso\ FC + \frac{Tamanho\ do\ Resultado}{BW}$$

Equação 11 - Tempo de atraso total.

Para exemplificar a aplicação do modelo, consideremos a rede AlexNet, cujas características estão representadas na tabela 8.

	Imagem	Conv1	Conv2	Conv3	Conv4	Conv5	FC1	FC2	FC3	Total
Número de *Kernels*:	0	96	256	384	384	256	4096	4096	1000	-
Tamanho dos *Kernels* (K)	0	11	5	3	3	3	6	1	1	-
Tamanho da Convolução	-	363	1200	1152	1728	1728	9216	4096	4096	-
***Padding* - P**	-	0	2	1	1	1	0	0	0	-
Passo (*Stride*) - S	0	4	1	1	1	1	1	1	1	-
Largura do mapa de características de entrada -WI	-	227	27	13	13	13	6	1	1	-
Comprimento do mapa de características de entrada - LI	-	227	27	13	13	13	6	1	1	-
Profundidade do mapa de características de entrada	-	3	96	256	384	384	256	4096	4096	-
Largura do mapa de características de saída - WO	227	55	27	13	13	13	1	1	1	-
Comprimento do mapa de características de saída - LO	227	55	27	13	13	13	1	1	1	-
Profundidade do mapa de características de saída	3	96	256	384	384	256	4096	4096	1000	-
Número total de convoluções:	-	3025	729	169	169	169	1	1	1	4264
Número total de MACs	-	105415200	223948800	74760192	112140288	74760192	37748736	16777216	4096000	649646624

Tabela 8 - Características da rede AlexNet utilizadas no modelo de desempenho.

As características da rede AlexNet representadas na tabela 8 permitem perceber a dimensão e complexidade da rede. Para simplificar a visualização das características da rede na tabela foram omitas as camadas de agrupamento. O número de *Kernels* (matrizes de convolução) corresponde ao número de filtros, o tamanho de *Kernels* corresponde ao comprimento/largura do filtro e o tamanho da convolução está dependente do tamanho do filtro. As características do mapa de características de entrada (largura, comprimento e profundidade) de uma camada corresponde às características do mapa de características de saída da camada anterior. A largura e o comprimento do mapa de características de saída dependem da largura e do comprimento do mapa de características de entrada, da largura e do comprimento do filtro, do valor de *padding* e do valor do passo, como na equação 12 que tem em conta as abreviaturas representadas na tabela 8.

$$LO = \frac{LI - K + (2 \times P)}{S} + 1 \; ; \; WO = \frac{WI - K + (2 \times P)}{S} + 1$$

Equação 12 - Largura e comprimento do mapa de características de saída.

A profundidade do mapa de características de saída depende do número de filtros. O número de convoluções corresponde ao número de convoluções tridimensionais

realizadas em cada camada, e pode ser calculado através da largura e do comprimento do mapa de características de saída. O número total de MAC realizados por camada depende do número de filtros, da dimensão da convolução e do número de convoluções realizadas nessa camada. Com os dados da tabela 8 verifica-se que a rede AlexNet realiza o um elevado número de MACs o que provoca uma necessidade grande a nível computacional.

O modelo de desempenho proposto, representado na tabela 9, permite estimar o desempenho da implementação da rede AlexNet utilizando diferentes números de bits para representar os pesos e as ativações na rede AlexNet, quando executada na LiteCNN, (considerou-se uma frequência de 200MHz para determinar os atrasos).

		Imagem	Conv1	Conv2	Conv3	Conv4	Conv5	FC1	FC2	FC3	Total
16x16 (Ativação x Pesos)	**Pesos [Bytes]**	309174	69696	614400	884736	1327104	884736	75497472	33554432	8192000	121333750
	Tempo de transferência [ms]	0,074	0,017	0,146	0,211	0,316	0,211	17,976	7,989	1,950	28,889
	Ciclos Conv	-	173380	368337	122961	184442	122961	-	-	-	972081
	Ciclos FC	-	-	-	-	-	-	124174	55189	13474	192837
	Atraso [ms]	-	0,883	1,988	0,825	1,238	0,825	17,976	7,989	1,950	33,972
8x8 (Ativação x Pesos)	**Pesos [Bytes]**	154587	34848	307200	442368	663552	442368	37748736	16777216	4096000	60666875
	Tempo de transferência [ms]	0,03681	0,00830	0,07314	0,10533	0,15799	0,10533	8,98779	3,99458	0,97524	14,44449
	Ciclos Conv	-	102945	218700	73008	109512	73008	-	-	-	577173
	Ciclos FC	-	-	-	-	-	-	73728	32768	8000	114496
	Atraso [ms]	-	0,523	1,167	0,470	0,706	0,470	8,988	3,995	0,975	17,442
4x4 (Ativação x Pesos)	**Pesos [Bytes]**	77294	17424	153600	221184	331776	221184	18874368	8388608	2048000	30333438
	Tempo de transferência [ms]	0,01840	0,00415	0,03657	0,05266	0,07899	0,05266	4,49390	1,99729	0,48762	7,22225
	Ciclos Conv	-	51473	109350	36504	54756	36504	-	-	-	288587
	Ciclos FC	-	-	-	-	-	-	36864	16384	4000	57248
	Atraso [ms]	-	0,262	0,583	0,235	0,353	0,235	4,494	1,997	0,488	8,721
5x5 (Ativação x Pesos)	**Pesos [Bytes]**	96617	21780	192000	276480	414720	276480	23592960	10485760	2560000	37916797
	Tempo de transferência [ms]	0,023	0,005	0,046	0,066	0,099	0,066	5,617	2,497	0,610	9,028
	Ciclos Conv	-	68630	145800	48672	73008	48672	-	-	-	384782
	Ciclos FC	-	-	-	-	-	-	49152	21846	5334	76332
	Atraso [ms]	-	0,348	0,775	0,309	0,464	0,309	5,617	2,497	0,610	11,021
6x6 (Ativação x Pesos)	**Pesos [Bytes]**	115941	26136	230400	331776	497664	331776	28311552	12582912	3072000	45500157
	Tempo de transferência [ms]	0,028	0,006	0,055	0,079	0,118	0,079	6,741	2,996	0,731	10,833
	Ciclos Conv	-	82356	174960	58407	87610	58407	-	-	-	461740
	Ciclos FC	-	-	-	-	-	-	58983	26215	6400	91598
	Atraso [ms]	-	0,418	0,930	0,371	0,557	0,371	6,741	2,996	0,731	13,226
8x4 (Ativação x Pesos)	**Pesos [Bytes]**	154587	17424	153600	221184	331776	221184	18874368	8388608	2048000	30410731
	Tempo de transferência [ms]	0,037	0,004	0,037	0,053	0,079	0,053	4,494	1,997	0,488	7,241
	Ciclos Conv	-	84467	179447	59904	89856	59904	-	-	-	473578
	Ciclos FC	-	-	-	-	-	-	60495	26887	6565	93947
	Atraso [ms]	-	0,426	0,934	0,352	0,528	0,352	4,494	1,997	0,488	9,720
Conv:8x4 FC: 8x2 (Ativação x Pesos)	**Pesos [Bytes]**	154587	17424	153600	221184	331776	221184	9437184	4194304	1024000	15755243
	Tempo de transferência [ms]	0,037	0,004	0,037	0,053	0,079	0,053	2,247	0,999	0,244	3,751
	Ciclos Conv	-	91507	194400	64896	97344	64896	-	-	-	513043
	Ciclos FC	-	-	-	-	-	-	65536	29128	7112	101776
	Atraso [ms]	-	0,462	1,009	0,377	0,566	0,377	2,247	0,999	0,244	6,428
8x2 (Ativação x Pesos)	**Pesos [Bytes]**	154587	8712	76800	110592	165888	110592	9437184	4194304	1024000	15282659
	Tempo de transferência [ms]	0,037	0,002	0,018	0,026	0,039	0,026	2,247	0,999	0,244	3,639
	Ciclos Conv	-	49168	104454	34870	52305	34870	-	-	-	275667
	Ciclos FC	-	-	-	-	-	-	35214	15651	3821	54686
	Atraso [ms]	-	0,248	0,541	0,201	0,301	0,201	2,247	0,999	0,244	5,128
Conv:8x8 FC: 8x4 (Ativação x Pesos)	**Pesos [Bytes]**	154587	34848	307200	442368	663552	442368	18874368	8388608	2048000	31355899
	Tempo de transferência [ms]	0,037	0,008	0,073	0,105	0,158	0,105	4,494	1,997	0,488	7,466
	Ciclos Conv	-	109808	233280	77876	116813	77876	-	-	-	615653
	Ciclos FC	-	-	-	-	-	-	78644	34953	8534	122131
	Atraso [ms]	-	0,557	1,240	0,495	0,742	0,495	4,494	1,997	0,488	10,655

Tabela 9 - Desempenho estimado para diferentes arquiteturas da rede AlexNet.

A tabela 9 apresenta o número de bytes ocupados pelos filtros (pesos), o tempo de transferência dos dados entre a memória externa e a LiteCNN, o número de ciclos de cálculo das camadas convolucionais (ciclos conv), o número de ciclos das camadas totalmente conectadas (ciclos FC) e o atraso total do sistema.

De acordo com o desempenho estimado apresentado na tabela 9 verifica-se que quanto menos bits a arquitetura utilizar para representar os pesos e as ativações das camadas menos bytes são necessários transferir, tal como era esperado. A arquitetura que possui um atraso menor é a arquitetura que utiliza 8 bits para representar as ativações e 2 bits para representar os pesos, e o seu atraso é 85% mais baixo que o da arquitetura com pior atraso (arquitetura que utiliza 16 bits para representar as ativações e os pesos das camadas). A arquitetura que utiliza 8 bits para representar as ativações, 4 bits para representar os pesos das camadas convolucionais e 2 bits para representar os pesos das camadas totalmente conectadas possui um atraso superior, em cerca de 20%, ao atraso da arquitetura que utiliza 8 bits para representar as ativações e 2 bits para representar os pesos, ou seja, alterar a representação dos pesos das camadas convolucionais de 4 bits para 2 bits permite reduzir a atraso em 20%.

Comparando a arquitetura que utiliza 8 bits para representar as ativações e 4 bits para representar os pesos com a arquitetura que que utiliza 8 bits para representar as ativações, 4 bits para representar os pesos das camadas convolucionais e 2 bits para representar os pesos das camadas totalmente conectadas, permite verificar que alterar a representação dos pesos das camadas totalmente conectadas de 4 bits para 2 bits reduz o atraso em, aproximadamente, 34%. Desta forma, é possível verificar também que alterar a representação dos pesos das camadas totalmente conectadas de 4 bits para 2 bits influencia mais o atraso da arquitetura LiteCNN quando executa a AlexNet do que alterar a representação dos pesos das camadas convolucionais de 4 para 2 bits.

Capítulo 5 – Resultados

As redes estudadas neste projeto foram otimizadas em termos de tamanho de dados e mapeadas na arquitetura LiteCNN para serem implementadas numa FPGA Zynq Z7020.A FPGA Zynq Z7020 é constituída por 53200 LUTs, 106400 Flip-Flops, 4.9Mb de BRAM (140 blocos RAM de 36Kb) e 220 DSP.

Para verificar o impacto que a redução do tamanho dos operandos da CNN tem sobre a sua precisão, na ocupação de recursos da FPGA e no desempenho do sistema foi realizado um estudo para diferentes redes, nomeadamente a LeNet, a Cifar10 (versão *quick* e versão *full*), a AlexNet e a SqueezeNet.

5.1 Rede LeNet

A arquitetura da rede LeNet implementada neste estudo foi ligeiramente modificada. A arquitetura da rede LeNet utilizada contém o mesmo número de camadas que a LeNet-5, no entanto a primeira camada convolucional possui vinte filtros de 5x5, a segunda camada convolucional possui cinquenta filtros de 5x5, as camadas de agrupamento utilizam a função *Max Pooling* com um passo de dois e a camada de não-linearidade aplica a função ReLU. O treino desta rede executa 10000 iterações.

Inicialmente esta rede foi treinada com vírgula flutuante de 32 bits, sendo depois aplicado o Ristretto com quantificação *Minifloat* tendo-se obtido os resultados de precisão apresentados na tabela 10.

	Vírgula flutuante de 32 bits	Camada convolucional e camada totalmente conectada	
		Minifloat de 16 bits	*Minifloat* de 8 bits
LeNet - Ristretto com margem de erro - 1%	0,991394	0,990894	-
LeNet - Ristretto com margem de erro 0%	0,991394	0,990894	-
LeNet - Ristretto com margem de erro 1%	0,991394	0,990894	0,989593
LeNet - Ristretto com margem de erro 2%	0,991394	0,990894	0,989593
LeNet - Ristretto com margem de erro 3%	0,991394	0,990894	0,989593
LeNet - Ristretto com margem de erro 10%	0,991394	0,990894	0,989593
LeNet - Ristretto com margem de erro 50%	0,991394	0,990894	0,989593

Tabela 10 - Resultados de precisão obtidos com o Ristretto com quantificação Minifloat para a rede LeNet.

Como se pode verificar pelos resultados da tabela 10 apenas foi possível obter através do Ristretto resultados para o *Minifloat* com 16 bits e com 8 bits, independentemente do

valor da margem de erro. Através dos resultados, verifica-se que passar de uma representação de vírgula flutuante de 32 bits para uma representação de vírgula flutuante de 8 bits para a rede LeNet diminui a precisão em apenas 0.18%.

O Ristretto para cada teste com uma determinada margem de erro criou um ficheiro com arquitetura da rede para a quantificação *MiniFloat*, onde inclui para cada camada convolucional e para cada camada totalmente conectada os parâmetros *mant_bits* e *exp_bits*. Esses parâmetros estão apresentados na tabela 11 para todas as margens de erro testadas no Ristretto.

	Camada Convolucional 1		Camada Convolucional 2		Camada Totalmente Conectada 1		Camada Totalmente Conectada 2	
	mant_bits	exp_bits	mant_bits	exp_bits	mant_bits	exp_bits	mant_bits	exp_bits
Margem de erro -1%	27	4	27	4	27	4	27	4
Margem de erro 0%	27	4	27	4	27	4	27	4
Margem de erro 1%	3	4	3	4	3	4	3	4
Margem de erro 2%	3	4	3	4	3	4	3	4
Margem de erro 3%	3	4	3	4	3	4	3	4
Margem de erro 10%	3	4	3	4	3	4	3	4
Margem de erro 50%	3	4	3	4	3	4	3	4
LeNet - Genérico	X	4	X	4	X	4	X	4

Tabela 11 - Resultados dos parâmetros da arquitetura LeNet aplicando o Ristretto com quantificação MiniFloat.

Através dos resultados da tabela 11 verifica-se que o número de bits para representar o expoente é sempre 4, independentemente do número de bits para representar a mantissa (o bit de sinal é omitido nestes resultados).

Para confirmar os resultados do Ristretto aplicando o *Minifloat*, resultados da tabela 10, e para testar outros números de bits para representar os dados além dos resultados do Ristretto, foram criadas e treinadas várias arquiteturas da rede LeNet com os parâmetros adicionais da quantificação *MiniFloat* (*mant_bits* e *exp_bits*) tendo em conta o modelo genérico representado na tabela 11 (ver tabela 12).

	Camada Convolucional 1		Camada Convolucional 2		Camada Totalmente Conectada 1		Camada Totalmente Conectada 2		Precisão
Tamanho MiniFloat	mant_bits	exp_bits	mant_bits	exp_bits	mant_bits	exp_bits	mant_bits	exp_bits	
32 bits	27	4	27	4	27	4	27	4	0,9894
16 bits	11	4	11	4	11	4	11	4	0,9909
16 bits	9	6	9	6	9	6	9	6	0,9917
15 bits	10	4	10	4	10	4	10	4	0,9908
14 bits	9	4	9	4	9	4	9	4	0,991
13 bits	8	4	8	4	8	4	8	4	0,9911
12 bits	7	4	7	4	7	4	7	4	0,991
11 bits	6	4	6	4	6	4	6	4	0,9907
10 bits	5	4	5	4	5	4	5	4	0,9898
9 bits	4	4	4	4	4	4	4	4	0,9907
8 bits	3	4	3	4	3	4	3	4	0,9893
7 bits	2	4	2	4	2	4	2	4	0,9856
6 bits	1	4	1	4	1	4	1	4	0,9827
6 bits	0	5	0	5	0	5	0	5	0,9549
5 bits	0	4	0	4	0	4	0	4	0,9502
5 bits	1	3	1	3	1	3	1	3	0,1135
4 bits	0	3	0	3	0	3	0	3	0,1135

Tabela 12 - Resultados da rede LeNet treinada com os parâmetros da quantificação MiniFloat para representações dos dados com diferentes números de bits.

Com os resultados da tabela 12 é possível verificar que reduzir a representação dos dados para 5 bits, 4 bits para representar o expoente e 1 bit para representar o sinal do valor, apenas faz diminuir a precisão da rede LeNet em, aproximadamente, 4.12%, quando comparada com a representação de vírgula flutuante de 32 bits inicial. Quando o número de bits para representar o expoente dos dados é 4 verifica-se que a precisão da rede é sempre inferior à precisão da rede treinada inicialmente com vírgula flutuante de 32 bits (99.1394%), mesmo quando a rede é treinada com 32 bits, ou seja, 4 bits para representar o expoente é o valor mínimo. Para confirmar esta conclusão, a rede LeNet foi também treinada com outro número de bits para representar o expoente e verificou-se que aumentar o número de bits para representar o expoente permite aumentar ligeiramente a precisão da rede e diminuir o número de bits para representar o expoente para 3 diminui a precisão da rede drasticamente, 83.67%.

O Ristretto quando aplica a quantificação *MiniFloat* é pouco flexível nos resultados, ou seja, alterando o único parâmetro que permite influenciar o número de bits para representar os dados (margem de erro) não se verificou muitas mudanças nos resultados. Por exemplo, alterar a margem de erro de 1% para 50% não alterou os resultados obtidos. Desta forma, os resultados para as restantes redes apenas têm em conta a quantificação de vírgula fixa dinâmica.

Aplicando a quantificação de vírgula fixa dinâmica do Ristretto à rede LeNet treinada inicialmente com vírgula flutuante de 32 bits obtiveram-se os resultados da precisão representados na tabela 13.

		LeNet - Ristretto utilizando quantificação de vírgula fixa dinâmica				
		Margem de erro de -1%	Margem de erro de 0%	Margem de erro de 1%	Margem de erro de 2%	Margem de erro de 3%
Vírgula flutuante de 32 bits		0,991394	0,991394	0,991394	0,991394	0,991394
Camada convolucional	VFD de 16 bits	0,9914	0,991394	0,991394	0,991394	0,991394
	VFD de 8 bits	-	0,991495	0,991495	0,991495	0,991495
	VFD de 4 bits	-	0,990194	0,990194	0,990194	0,990194
	VFD de 2 bits	-	-	0,97819	0,97819	0,97819
	VFD de 1 bits	-	-	-	0,1135	0,1135
Camada totalmente conectada	VFD de 16 bits	0,9914	0,991394	0,991394	0,991394	0,991394
	VFD de 8 bits	-	0,991394	0,991394	0,991394	0,991394
	VFD de 4 bits	-	0,990294	0,990294	0,990294	0,990294
	VFD de 2 bits	-	-	0,883801	0,883801	0,883801
Ativações das camadas	VFD de 16 bits	0,9904	0,990394	0,990394	0,990394	0,990394
	VFD de 8 bits	-	-	0,990194	0,990194	0,990194
	VFD de 4 bits	-	-	0,96659	0,96659	0,96659
	VFD de 2 bits	-	-	-	-	0
Resultados	Camada convolucional	32 bits	8 bits	4 bits	2 bits	2 bits
	Camada totalmente conectada	32 bits	8 bits	4 bits	4 bits	4 bits
	Ativações das camadas	32 bits	32 bits	8 bits	8 bits	4 bits
	Precisão	0,9904	0,990294	0,989093	0,965791	0,8974

Tabela 13 - Resultados do Ristretto com quantificação de vírgula fixa dinâmica (VFD) para a rede LeNet.

Com os resultados da tabela 13 é possível concluir que alterar a representação dos parâmetros (pesos e valores *Bias*) de vírgula flutuante de 32 bits para vírgula fixa dinâmica de 8 bits faz reduzir muito ligeiramente (inferior a 0.1%) a precisão da rede LeNet. Com estes resultados, é possível também verificar que ao alterar a representação dos dados de vírgula flutuante para vírgula fixa, reduzir o número de bits dos parâmetros (pesos e valores *Bias*) das camadas convolucionais para 2, reduzir o número de bits dos parâmetros (pesos e valores *Bias*) das camadas totalmente conectadas para 4 e reduzir o número de bits das ativações das camadas (valores de entrada e de saída das camadas) para 4, apenas provoca uma diminuição da precisão de, aproximadamente, 9.4%.

Os ficheiros da arquitetura da rede gerados pela aplicação do Ristretto utilizando a quantificação de vírgula fixa dinâmica incluem os parâmetros específicos deste tipo de quantificação para todas as camadas convolucionais e totalmente conectadas da rede. Os

valores desses parâmetros estão associados aos resultados obtidos pelo Ristretto, em que o número de bits resultantes para a camada convolucional correspondem ao número de bits utilizados para representar os pesos das camadas convolucionais (parâmetro *bw_params* das camadas convolucionais), o número de bits resultantes para a camada totalmente conectada correspondem ao número de bits que representam os pesos das camadas totalmente conectadas (parâmetro *bw_params* das camadas totalmente conectadas) e o número de bits que representam as ativações das camadas que correspondem ao número de bits para representar os valores de entrada e saída das camadas (parâmetros *bw_layer_in* e *bw_layer_out* das camadas convolucionais e das camadas totalmente conectadas).

Assim, com os valores desses parâmetros foi possível criar um modelo genérico que depois permite gerar arquiteturas com diferentes números de bits para representar os dados utilizando a mesma representação de vírgula fixa dinâmica. Os resultados dos parâmetros da quantificação de vírgula fixa dinâmica resultantes nas arquiteturas da rede LeNet geradas pelo Ristretto estão apresentados na tabela 14.

		Margem de erro - 1%	Margem de erro 0%	Margem de erro 1%	Margem de erro 2%	Margem de erro 3%	LeNet - Ristretto
Número de bits para camada convolucional: AtivaçãoxPeso		32x32	32x8	8x4	8x2	4x2	XxY
Número de bits na camada totalmente conectada: AtivaçãoxPeso		32x32	32x8	8x4	8x4	4x4	XxZ
Camada Convolucional 1	bw_layer_in	32	32	8	8	4	X
	bw_layer_out	32	32	8	8	4	X
	bw_params	32	8	4	2	2	Y
	fl_layer_in	32	32	8	8	4	X
	fl_layer_out	30	30	6	6	2	X-2
	fl_params	31	7	3	1	1	Y-1
Camada Convolucional 2	bw_layer_in	32	32	8	8	4	X
	bw_layer_out	32	32	8	8	4	X
	bw_params	32	8	4	2	2	Y
	fl_layer_in	30	30	6	6	2	X-2
	fl_layer_out	28	28	4	4	0	X-4
	fl_params	33	9	5	3	3	Y+1
Camada Totalmente Conectada 1	bw_layer_in	32	32	8	8	4	X
	bw_layer_out	32	32	8	8	4	X
	bw_params	32	8	4	4	4	Z
	fl_layer_in	28	28	4	4	0	X-4
	fl_layer_out	28	28	4	4	0	X-4
	fl_params	34	10	6	6	6	Z+2
Camada Totalmente Conectada 2	bw_layer_in	32	32	8	8	4	X
	bw_layer_out	32	32	8	8	4	X
	bw_params	32	8	4	4	4	Z
	fl_layer_in	28	28	4	4	0	X-4
	fl_layer_out	27	27	3	3	-1	X-5
	fl_params	32	8	4	4	4	Z

Tabela 14 - Resultados dos parâmetros da arquitetura LeNet aplicando o Ristretto com quantificação de vírgula fixa dinâmica.

Com o modelo genérico representado na tabela 14 foram geradas diversas arquiteturas da rede LeNet que diferem no número de bits para representar os dados da rede, mas que continuam a ser representadas com vírgula fixa dinâmica. Essas arquiteturas foram treinadas e os resultados das precisões obtidas nas diferentes arquiteturas estão apresentadas na tabela 15.

		LeNet com representação de vírgula fixa dinâmica								
Número de bits para camada convolucional: AtivaçãoxPeso		16x16	8x8	6x6	5x5	4x4	8x8	8x4	8x4	8x2
Número de bits na camada totalmente conectada: AtivaçãoxPeso		16x16	8x8	6x6	5x5	4x4	8x4	8x4	8x2	8x2
Camada Convolucional 1	bw_layer_in	16	8	6	5	4	8	8	8	8
	bw_layer_out	16	8	6	5	4	8	8	8	8
	bw_params	16	8	6	5	4	8	4	4	2
	fl_layer_in	16	8	6	5	4	8	8	8	8
	fl_layer_out	14	6	4	3	2	6	6	6	8
	fl_params	15	7	5	4	3	7	3	3	1
Camada Convolucional 2	bw_layer_in	16	8	6	5	4	8	8	8	8
	bw_layer_out	16	8	6	5	4	8	8	8	8
	bw_params	16	8	6	5	4	8	4	4	2
	fl_layer_in	14	6	4	3	2	6	6	6	6
	fl_layer_out	12	4	2	1	0	4	4	4	4
	fl_params	17	9	7	6	5	9	5	5	3
Camada Totalmente Conectada 1	bw_layer_in	16	8	6	5	4	8	8	8	8
	bw_layer_out	16	8	6	5	4	8	8	8	8
	bw_params	16	8	6	5	4	4	4	2	2
	fl_layer_in	12	4	2	1	0	4	4	4	4
	fl_layer_out	12	4	2	1	0	4	4	4	4
	fl_params	18	10	8	7	6	6	6	4	4
Camada Totalmente Conectada 2	bw_layer_in	16	8	6	5	4	8	8	8	8
	bw_layer_out	16	8	6	5	4	8	8	8	8
	bw_params	16	8	6	5	4	4	4	2	2
	fl_layer_in	12	4	2	1	0	4	4	4	4
	fl_layer_out	11	3	1	0	-1	3	3	3	3
	fl_params	16	8	6	5	4	4	4	2	2
Precisão		0,9897	0,9882	0,988	0,9832	0,9611	0.9878	0.9878	0.9777	0

Tabela 15 - Resultados da rede LeNet treinada com os parâmetros da quantificação de vírgula fixa dinâmica para representações dos dados com diferentes números de bits.

Através dos resultados representados na tabela 15 verifica-se que utilizando uma representação de vírgula fixa dinâmica de 4 bits para todos os dados da LeNet a precisão da rede diminui apenas 3.03%. Com os resultados obtidos verifica-se também que para representações dos dados com poucos bits, como por exemplo 5 bits, obtém-se melhores resultados para a representação de vírgula fixa dinâmica do que para a representação de vírgula flutuante (utilizando os parâmetros da quantificação *MiniFloat* do Ristretto). No anexo A encontram-se representados mais resultados obtidos para a rede LeNet treinada com outros números de bits para representar os dados.

Através do modelo de desempenho foi possível verificar o atraso das arquiteturas que representam as ativações e os pesos com diferentes números de bits e que utilizam um número de recursos da FPGA semelhante (45700 LUT ± 4%, 220 DSP). A relação entre o atraso e a precisão da CNN LeNet para as diferentes arquiteturas está ilustrada na figura 29.

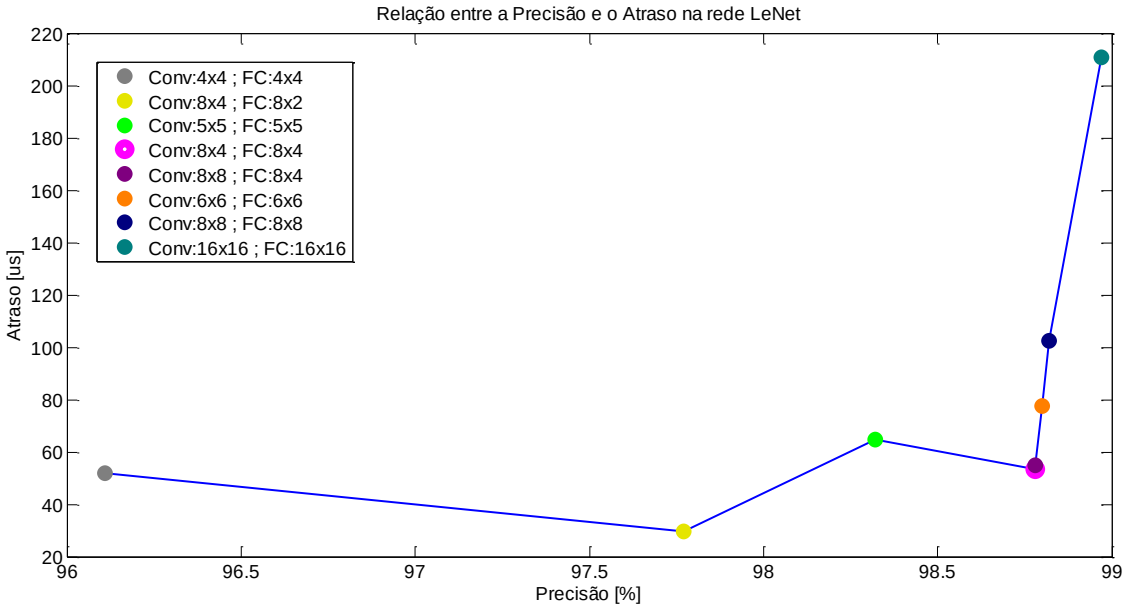


Figura 29 - Relação entre o atraso da arquitetura e a precisão da rede LeNet.

Com os resultados da figura 29 verifica-se que a arquitetura que utiliza uma representação com vírgula fixa dinâmica de 16 bits para as ativações e para os pesos das camadas convolucionais e das camadas totalmente conectadas é a que possui uma melhor precisão, no entanto, é a arquitetura que possui um atraso maior. A arquitetura que possui um atraso mais baixo é a arquitetura que utiliza vírgula fixa dinâmica e representa as ativações com 8 bits, os pesos das camadas convolucionais com 4 bits e os pesos das camadas totalmente conectadas com 2 bits. Esta arquitetura apesar de ser uma das que possui pior precisão, a redução da precisão comparativamente com a arquitetura que utiliza vírgula flutuante de 32 bits é de apenas 1.4%. Comparando a arquitetura que utiliza uma representação de vírgula fixa dinâmica de 8 bits para as ativações e 4 bits para os pesos com a arquitetura que utiliza uma representação de vírgula fixa dinâmica de 8 bits para as ativações e para os pesos das camadas convolucionais e de 4 bits para os pesos das camadas totalmente conectadas, verifica-se que alterar os pesos das camadas convolucionais de 8 bits para 4 bits não influencia a precisão e o atraso é muito semelhante para ambas as arquiteturas.

5.2 Rede Cifar10_quick

A arquitetura da versão *quick* da rede Cifar10 é constituída por três camadas convolucionais e duas camadas totalmente conectadas. Cada camada convolucional é

seguida por uma camada de não-linearidade e uma camada de agrupamento, A primeira e a segunda camada convolucional utilizam 32 filtros de 5x5, *padding* de valor 2 e passo de 1, e a última camada convolucional utiliza 64 filtros de 5x5, *padding* de valor 2 e passo de 1. A camada de não-linearidade aplica a função ReLU. A primeira camada de agrupamento aplica a função *Max Pooling* com filtros de 3x3 e passo de 2, e as duas restantes camadas de agrupamento aplicam a função *Average Pooling* com filtros de 3x3 e passo de 2. O treino desta rede é realizado com a base de imagens Cifar10 e executa 5000 iterações.

Inicialmente a rede Cifar10_quick foi treinada com uma representação de dados de vírgula flutuante de 32 bits obtendo uma precisão de 75.88%. Ao modelo treinado da rede foi aplicado o Ristretto com quantificação de vírgula fixa dinâmica e os resultados obtidos estão apresentados na tabela 16.

		Cifar10_quick - Ristretto com quantificação de vírgula fixa dinâmica					
		Margem de erro -1%	Margem de erro 0%	Margem de erro 1%	Margem de erro 2%	Margem de erro 3%	Margem de erro 10%
Vírgula flutuante de 32 bits		0,758803	0,758803	0,758803	0,758803	0,758803	0,758803
Camada convolucional	Vírgula fixa dinâmica de 16 bits	0,759002	0,759002	0,759002	0,759002	0,759002	0,759002
	Vírgula fixa dinâmica de 8 bits	-	0,757702	0,757702	0,757702	0,757702	0,757702
	Vírgula fixa dinâmica de 4 bits	-	-	0,733299	0,733299	0,733299	0,733299
	Vírgula fixa dinâmica de 2 bits	-	-	-	-	0,170901	0,170901
Camada totalmente conectada	Vírgula fixa dinâmica de 16 bits	0,758803	0,758803	0,758803	0,758803	0,758803	0,758803
	Vírgula fixa dinâmica de 8 bits	-	0,757202	0,757202	0,757202	0,757202	0,757202
	Vírgula fixa dinâmica de 4 bits	-	-	0,732001	0,732001	0,732001	0,732001
	Vírgula fixa dinâmica de 2 bits	-	-	-	-	0,482499	0,482499
Ativações das camadas	Vírgula fixa dinâmica de 16 bits	0,756402	0,756402	0,756402	0,756402	0,756402	0,756402
	Vírgula fixa dinâmica de 8 bits	-	-	0,747303	0,747303	0,747303	0,747303
	Vírgula fixa dinâmica de 4 bits	-	-	-	0,561602	0,561602	0,561602
Resultados	Camada convolucional	32 bits	16 bits	8 bits	8 bits	4 bits	4 bits
	Camada totalmente conectada	32 bits	16 bits	8 bits	8 bits	4 bits	4 bits
	Ativações das camadas	32 bits	32 bits	16 bits	8 bits	8 bits	8 bits
	Precisão	0,756803	0,756803	0,755102	0,746301	0,701198	0,701198

Tabela 16 - Resultados do Ristretto com quantificação de vírgula fixa dinâmica para a rede Cifar10_quick.

Com os resultados do Ristretto apresentados na tabela 16 verifica-se que reduzir o número de bits para as ativações das camadas (valores de entrada e saída das camadas da

rede) influência mais a descida da precisão da rede do que reduzir o número de bits para os parâmetros (pesos e valores Bias) das camadas convolucionais e das camadas totalmente conectadas. Através dos resultados da tabela 16 verifica-se também que utilizar uma representação de vírgula fixa dinâmica de 4 bits invés de uma representação de vírgula flutuante de 32 bits apenas provoca uma diminuição da precisão da rede Cifar10_quick de 5.76%.

Os parâmetros da representação de vírgula fixa dinâmica das arquiteturas da rede Cifar10_quick geradas pela aplicação do Ristretto estão apresentadas na tabela 17. Através dos resultados dos diferentes valores de margem de erros do Ristretto foi possível criar um modelo genérico que permite gerar novas arquiteturas da rede Cifar10_quick com o bom desempenho encontrado pelo Ristretto.

		Cifar10_quick - Ristretto com quantificação de vírgula fixa dinâmica						
		Margem de erro - 1%	Margem de erro 0%	Margem de erro 1%	Margem de erro 2%	Margem de erro 3%	Margem de erro 10%	Modelo Genérico
Número de bits para camada convolucional: AtivaçãoxPeso		32x32	32x16	16x8	8x8	8x4	8x4	XxY
Número de bits na camada totalmente conectada: AtivaçãoxPeso		32x32	32x16	16x8	8x8	8x4	8x4	XxZ
Camada Convolucional 1	bw_layer_in	32	32	16	8	8	8	X
	bw_layer_out	32	32	16	8	8	8	X
	bw_params	32	16	8	8	4	4	Y
	fl_layer_in	24	24	8	0	0	0	X-8
	fl_layer_out	23	23	7	-1	-1	-1	X-9
	fl_params	33	17	9	9	5	5	Y+1
Camada Convolucional 2	bw_layer_in	32	32	16	8	8	8	X
	bw_layer_out	32	32	16	8	8	8	X
	bw_params	32	16	8	8	4	4	Y
	fl_layer_in	23	23	7	-1	-1	-1	X-9
	fl_layer_out	23	23	7	-1	-1	-1	X-9
	fl_params	34	18	10	10	6	6	Y+2
Camada Convolucional 3	bw_layer_in	32	32	16	8	8	8	X
	bw_layer_out	32	32	16	8	8	8	X
	bw_params	32	16	8	8	4	4	Y
	fl_layer_in	24	24	8	0	0	0	X-8
	fl_layer_out	26	26	10	2	2	2	X-6
	fl_params	34	18	10	10	6	6	Y+2
Camada Totalmente Conectada 1	bw_layer_in	32	32	16	8	8	8	X
	bw_layer_out	32	32	16	8	8	8	X
	bw_params	32	16	8	8	4	4	Z
	fl_layer_in	26	26	10	2	2	2	X-6
	fl_layer_out	27	27		3	3	3	X-5
	fl_params	32	16	8	8	4	4	Z
Camada Totalmente Conectada 2	bw_layer_in	32	32	16	8	8	8	X
	bw_layer_out	32	32	16	8	8	8	X
	bw_params	32	16	8	8	4	4	Z
	fl_layer_in	27	27	11	3	3	3	X-5
	fl_layer_out	27	27	11	3	3	3	X-5
	fl_params	32	16	8	8	4	4	Z

Tabela 17 - Resultados dos parâmetros da arquitetura Cifar10_quick aplicando o Ristretto com quantificação de vírgula fixa dinâmica.

Com os resultados dos parâmetros da arquitetura gerada pelo Ristretto com quantificação de vírgula fixa dinâmica, apresentados na tabela 17, foi possível criar um modelo genérico desses parâmetros. Esse modelo genérico permitiu gerar outras arquiteturas da rede Cifar10_quick com representações de vírgula fixa dinâmica de

tamanhos diferentes às que foram apresentadas na tabela 17. Essas arquiteturas foram treinadas e os resultados da precisão obtidos estão representados na tabela 18.

		CIFAR10_quick com representação de vírgula fixa dinâmica									
Número de bits para camada convolucional: AtivaçãoxPeso		16x16	8x8	7x7	6x6	5x5	4x4	8x8	8x4	8x4	8x2
Número de bits na camada totalmente conectada: AtivaçãoxPeso		16x16	8x8	7x7	6x6	5x5	4x4	8x4	8x4	8x2	8x2
Camada Convolucional 1	bw_layer_in	16	8	7	6	5	4	8	8	8	8
	bw_layer_out	16	8	7	6	5	4	8	8	8	8
	bw_params	16	8	7	6	5	4	8	4	4	2
	fl_layer_in	8	0	-1	-2	-3	-4	0	0	0	0
	fl_layer_out	7	-1	-2	-3	-4	-5	-1	-1	-1	-1
	fl_params	17	9	8	7	6	5	9	5	5	3
Camada Convolucional 2	bw_layer_in	16	8	7	6	5	4	8	8	8	8
	bw_layer_out	16	8	7	6	5	4	8	8	8	8
	bw_params	16	8	7	6	5	4	8	4	4	2
	fl_layer_in	7	-1	-2	-3	-4	-5	-1	-1	-1	-1
	fl_layer_out	7	-1	-2	-3	-4	-5	-1	-1	-1	-1
	fl_params	18	10	9	8	7	6	10	6	6	4
Camada Convolucional 3	bw_layer_in	16	8	7	6	5	4	8	8	8	8
	bw_layer_out	16	8	7	6	5	4	8	8	8	8
	bw_params	16	8	7	6	5	4	8	4	4	2
	fl_layer_in	8	0	-1	-2	-3	-4	0	0	0	0
	fl_layer_out	10	2	1	0	-1	-2	2	2	2	2
	fl_params	18	10	9	8	7	6	10	6	6	4
Camada Totalmente Conectada 1	bw_layer_in	16	8	7	6	5	4	8	8	8	8
	bw_layer_out	16	8	7	6	5	4	8	8	8	8
	bw_params	16	8	7	6	5	4	4	4	2	2
	fl_layer_in	10	2	1	0	-1	-2	2	2	2	2
	fl_layer_out	11	3	2	1	0	-1	3	3	3	3
	fl_params	16	8	7	6	5	4	4	4	2	2
Camada Totalmente Conectada 2	bw_layer_in	16	8	7	6	5	4	8	8	8	8
	bw_layer_out	16	8	7	6	5	4	8	8	8	8
	bw_params	16	8	7	6	5	4	4	4	2	2
	fl_layer_in	11	3	2	1	0	-1	3	3	3	3
	fl_layer_out	11	3	2	1	0	-1	3	3	3	3
	fl_params	16	8	7	6	5	4	4	4	2	2
Precisão		0,7511	0,7422	0,7279	0,7096	0,6411	0,4302	0.7395	0.6757	0.5042	0

Tabela 18 - Resultados da rede Cifar10_quick treinada com os parâmetros da quantificação de vírgula fixa dinâmica para representações dos dados com diferentes números de bits.

Com os resultados da tabela 18 verifica-se que que é possível reduzir o número de bits para representar os dados da rede Cifar10_quick com representação de vírgula fixa

dinâmica até 5 bits com uma redução da precisão de apenas 11.77% Verifica-se também que a redução do número de bits influência a precisão de forma mais acentuada comparativamente à rede LeNet. No anexo B encontram-se representados mais resultados obtidos para a rede Cifar10_quick treinada com outros números de bits para representar os dados.

Através dos resultados da tabela 18 e modelo de desempenho das diferentes arquiteturas utilizando o mesmo número de recursos de FPGA (45700 LUT ± 4%, 220 DSP) foi possível verificar a relação entre a precisão e o atraso para a rede Cifar10_Quick. Esta relação está ilustrada na figura 30.

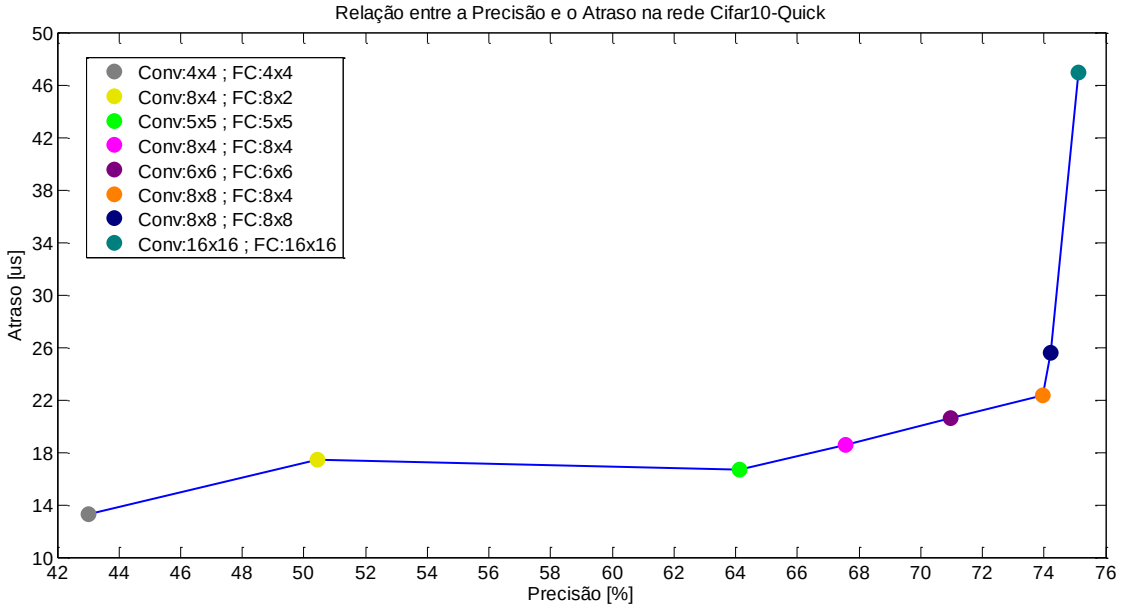


Figura 30 - Relação entre a precisão e o atraso para a rede Cifar10-Quick.

Através dos resultados ilustrados na figura 30 verifica-se que a arquitetura que utiliza vírgula fixa dinâmica de 16 bits para representar as ativações e os pesos é a arquitetura que possui melhor precisão, no entanto é a arquitetura que possui o atraso mais elevado (possui um atraso superior em cerca de 72% comparativamente à arquitetura com o atraso mais baixo). Comparando a arquitetura que utiliza vírgula fixa dinâmica de 5 bits para representar as ativações e os pesos com a arquitetura que utiliza representação de vírgula fixa dinâmica de 8 bits para as ativações, 4 bits para os pesos das camadas convolucionais e 2 bits para as camadas totalmente conectadas verifica-se que as arquiteturas têm um atraso muito semelhante, no entanto a arquitetura que utiliza 5 bits para representar os seus dados possui um precisão bastante superior, cerca de 14%. Com os resultados obtidos verifica-se também que utilizar 8 bits para representar as ativações e os pesos das camadas convolucionais e alterar a representação dos pesos das camadas totalmente conectadas de 8 bits para 4 bits faz diminuir ligeiramente a precisão, cerca de 0.3%, no entanto a

diminuição do atraso é mais elevada, cerca de 13%, o que compensa a ligeira redução de precisão.

5.3 Rede Cifar10_full

A arquitetura da versão *full* da rede Cifar10 é constituída por três camadas convolucionais e uma camada totalmente conectada. As duas primeiras camadas convolucionais utilizam 32 filtros de 5x5, *padding* de valor 2, passo de 1 e são seguidas por uma camada de não-linearidade, uma camada de agrupamento e uma camada LRN. A terceira camada convolucional utiliza 64 filtros de 5x5, *padding* de valor 2, passo de 1 e é seguida apenas por uma cada de não-linearidade e uma camada de agrupamento. A camada de não-linearidade aplica a função ReLU. A primeira camada de agrupamento aplica a função *Max Pooling* com filtros de 3x3 e passo de 2, e as duas restantes camadas de agrupamento aplicam a função *Average Pooling* com filtros de 3x3 e passo de 2. O treino desta rede é feito com a base de imagens Cifar10 e apesar de ter menos uma camada totalmente conectada do que a versão *quick* da rede Cifar10 o seu treino executa um número muito maior de iterações, 70000 iterações.

A rede Cifar10_full foi treinada com uma representação de vírgula flutuante de 32 bis, e ao modelo treinado da rede foi aplicado o Ristretto com quantificação de vírgula fixa dinâmica, estando os resultados representados na tabela 19.

		Cifar10_full - Ristretto com quantificação de vírgula fixa dinâmica					
		Margem de erro -1%	Margem de erro 0%	Margem de erro 1%	Margem de erro 2%	Margem de erro 3%	Margem de erro 10%
Vírgula flutuante de 32 bits		0,8181	0,8181	0,8181	0,8181	0,8181	0,8181
Camada convolucional	Vírgula fixa dinâmica de 16 bits	0,8181	0,8181	0,8181	0,8181	0,8181	0,8181
	Vírgula fixa dinâmica de 8 bits	-	0,8178	0,8178	0,8178	0,8178	0,8178
	Vírgula fixa dinâmica de 4 bits	-	-	0,760903	0,760903	0,760903	0,760903
	Vírgula fixa dinâmica de 2 bits	-	-	-	-	-	0,1295
Camada totalmente conectada	Vírgula fixa dinâmica de 16 bits	0,809803	0,809803	0,809803	0,809803	0,809803	0,809803
	Vírgula fixa dinâmica de 8 bits	-	-	0,809303	0,809303	0,809303	0,809303
	Vírgula fixa dinâmica de 4 bits	-	-	0,796604	0,796604	0,796604	0,796604
	Vírgula fixa dinâmica de 2 bits	-	-	-	-	0,662299	0,662299
Ativações das camadas	Vírgula fixa dinâmica de 16 bits	0,817702	0,817702	0,817702	0,817702	0,817702	0,817702
	Vírgula fixa dinâmica de 8 bits	-	-	0,81702	0,81702	0,81702	0,81702
	Vírgula fixa dinâmica de 4 bits	-	-	0,668198	0,668198	0,668198	0,668198
Resultados	Camada convolucional	32 bits	16 bits	8 bits	8 bits	8 bits	4 bits
	Camada totalmente conectada	32 bits	32 bits	8 bits	8 bits	4 bits	4 bits
	Ativações das camadas	32 bits	32 bits	8 bits	8 bits	8 bits	8 bits
	Precisão	0,809204	0,809204	0,803802	0,803802	0,791703	0,732698

Tabela 19 - Resultados do Ristretto com quantificação de vírgula fixa dinâmica para a rede Cifar10_full.

Com os resultados representados na tabela 19 verifica-se que a rede Cifar10_full treinada com vírgula flutuante de 32 bits obteve uma precisão de 81.81% e que alterar a rede para uma representação de vírgula fixa dinâmica de 8 bits diminui a precisão em apenas 1.43%. Comparativamente à CNN Cifar10_quick esta rede têm uma precisão mais alta, no entanto, verifica-se que reduzir muito o número de bits para representar os dados

com vírgula fixa dinâmica influência mais a precisão na rede Cifar10_full do que na rede Cifar10_quick.

A aplicação do Ristretto gerou arquiteturas com representação de vírgula fixa dinâmica de diferentes tamanhos para a rede Cifar10_full. Estas arquiteturas incluem parâmetros específicos dessa quantificação e os resultados obtidos para esses parâmetros estão apresentados na tabela 20.

		Cifar10_full - Ristretto com quantificação de vírgula fixa dinâmica						
		Margem de erro -1%	Margem de erro 0%	Margem de erro 1%	Margem de erro 2%	Margem de erro 3%	Margem de erro 10%	Modelo Genérico
Número de bits para camada convolucional: AtivaçãoxPeso		32x32	32x16	8x8	8x8	8x8	8x4	XxY
Número de bits na camada totalmente conectada: AtivaçãoxPeso		32x32	32x32	8x8	8x8	8x4	8x4	XxZ
Camada Convolucional 1	bw_layer_in	32	32	8	8	8	8	X
	bw_layer_out	32	32	8	8	8	8	X
	bw_params	32	16	8	8	8	4	Y
	fl_layer_in	24	24	0	0	0	0	X-8
	fl_layer_out	22	22	-2	-2	-2	-2	X-10
	fl_params	32	16	8	8	8	4	Y
Camada Convolucional 2	bw_layer_in	32	32	8	8	8	8	X
	bw_layer_out	32	32	8	8	8	8	X
	bw_params	32	16	8	8	8	4	Y
	fl_layer_in	24	24	0	0	0	0	X-8
	fl_layer_out	23	23	-1	-1	-1	-1	X-9
	fl_params	33	17	9	9	9	5	Y+1
Camada Convolucional 3	bw_layer_in	32	32	8	8	8	8	X
	bw_layer_out	32	32	8	8	8	8	X
	bw_params	32	32	8	8	8	4	Y
	fl_layer_in	24	24	0	0	0	0	X-8
	fl_layer_out	24	24	0	0	0	0	X-8
	fl_params	33	17	9	9	9	5	Y+1
Camada Totalmente Conectada 1	bw_layer_in	32	32	8	8	8	8	X
	bw_layer_out	32	32	8	8	8	8	X
	bw_params	32	32	8	8	4	4	Z
	fl_layer_in	25	25	1	1	1	1	X-7
	fl_layer_out	27	27	3	3	3	3	X-5
	fl_params	35	35	11	11	7	7	Z+3

Tabela 20 - Resultados dos parâmetros da arquitetura Cifar10_full aplicando o Ristretto com quantificação de vírgula fixa dinâmica.

Com os resultados apresentados na tabela 20 foi criado um modelo genérico, também apresentado na tabela 20, que permite gerar arquiteturas da Cifar10_full com representação de vírgula fixa dinâmica e outros tamanhos. Na tabela 21 estão ilustrados os resultados da precisão da rede Cifar10_full treinada com representação de vírgula fixa dinâmica com diferentes números de bits.

		CIFAR10_full com representação de vírgula fixa dinâmica								
Número de bits para camada convolucional: AtivaçãoxPeso		32x32	16x16	8x8	6x6	5x5	4x4	8x8	8x4	8x4
Número de bits na camada totalmente conectada: AtivaçãoxPeso		32x32	16x16	8x8	6x6	5x5	4x4	8x4	8x4	8x2
Camada Convolucional 1	bw_layer_in	32	16	8	6	5	4	8	8	8
	bw_layer_out	32	16	8	6	5	4	8	8	8
	bw_params	32	16	8	6	5	4	8	4	4
	fl_layer_in	24	8	0	-2	-3	-4	0	0	0
	fl_layer_out	22	6	-2	-4	-5	-6	-2	-2	-2
	fl_params	32	16	8	6	5	4	8	4	4
Camada Convolucional 2	bw_layer_in	32	16	8	6	5	4	8	8	8
	bw_layer_out	32	16	8	6	5	4	8	8	8
	bw_params	32	16	8	6	5	4	8	4	4
	fl_layer_in	24	8	0	-2	-3	-4	0	0	0
	fl_layer_out	23	7	-1	-3	-4	-5	-1	-1	-1
	fl_params	33	17	9	7	6	5	9	5	5
Camada Convolucional 3	bw_layer_in	32	16	8	6	5	4	8	8	8
	bw_layer_out	32	16	8	6	5	4	8	8	8
	bw_params	32	16	8	6	5	4	8	4	4
	fl_layer_in	24	8	0	-2	-3	-4	0	0	0
	fl_layer_out	24	8	0	-2	-3	-4	0	-1	-1
	fl_params	33	17	9	7	6	5	9	5	5
Camada Totalmente Conectada 1	bw_layer_in	32	16	8	6	5	4	8	8	8
	bw_layer_out	32	16	8	6	5	4	8	8	8
	bw_params	32	16	8	6	5	4	4	4	2
	fl_layer_in	25	9	1	-1	-2	-3	1	1	1
	fl_layer_out	27	11	3	1	0	-1	3	3	3
	fl_params	35	19	11	9	8	7	7	7	5
Precisão		0,8175	0,8144	0,8135	0,7926	0.7483	0,6137	0.8086	0.7592	0.7206

Tabela 21 - Resultados da rede Cifar10_full treinada com os parâmetros da quantificação de vírgula fixa dinâmica para representações dos dados com diferentes números de bits.

Com os resultados apresentados na tabela 21 verifica-se que alterar a representação dos dados da rede de vírgula flutuante de 32 bits para vírgula fixa dinâmica de 6 bits provoca uma diminuição da precisão de 2.55%.

Através da tabela 21 e do modelo de desempenho das arquiteturas foi possível verificar a influência da representação dos dados com vírgula fixa dinâmica com diferentes

números de bits na precisão da rede e no atraso da arquitetura. Na figura 31 está ilustrada essa relação para a rede Cifar10_Full.

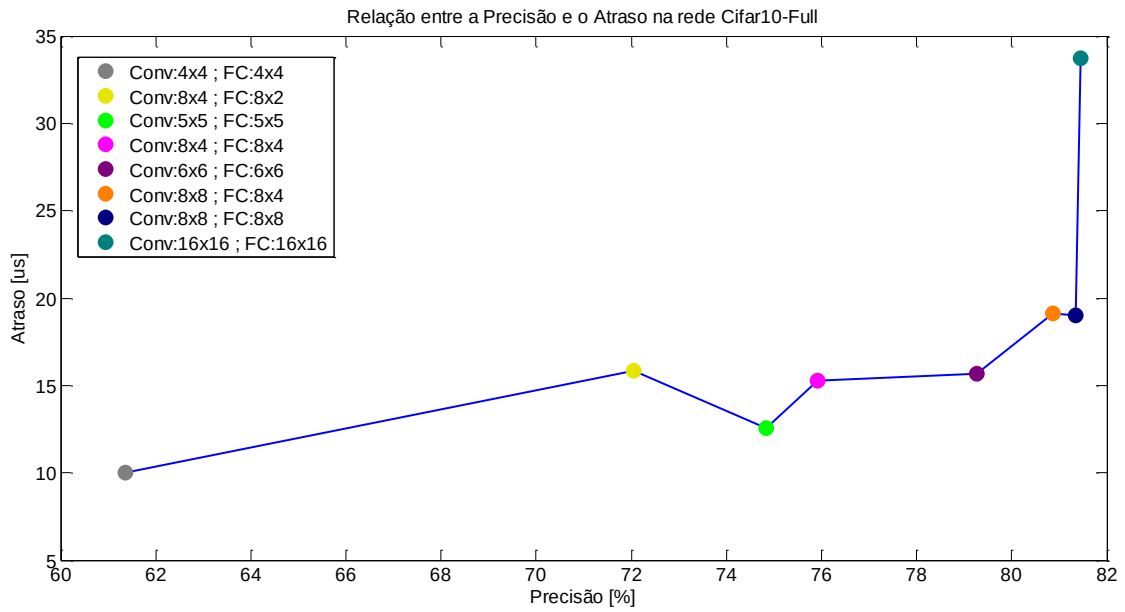


Figura 31 - Relação entre o atraso e a precisão da rede Cifar10_Full.

Com os resultados da figura 31 verifica-se que utilizar uma representação de vírgula fixa dinâmica de 8 bits ao invés de 16 bits, permite ter uma precisão semelhante com uma diminuição do atraso em cerca de 44%. A arquitetura que utiliza uma representação de vírgula fixa dinâmica de 5 bits possui um atraso inferior à maior das arquiteturas com uma redução da precisão de, aproximadamente, 7% comparativamente à arquitetura com uma representação de vírgula flutuante de 32 bits. Com os resultados obtidos verifica-se também que uma arquitetura que possui uma boa relação entre a precisão e o atraso é a arquitetura que utiliza 6 bits para representar as ativações e os pesos das camadas.

5.4 Rede AlexNet

O treino da rede AlexNet com a base de imagens ImageNet (base de imagens utilizadas na competição ILSVRC) possui requisitos computacionais muito elevados, não sendo possível realizar o treino da rede com os recursos deste projeto. Assim foi utilizado um modelo da rede AlexNet previamente treinado disponibilizado pela comunidade da ferramenta Caffe. Esse modelo foi gerado por um treino de 360000 iterações e atinge uma precisão top-1 de 57.1% e uma precisão top-5 de 80.2%. Ao modelo previamente treinado foi aplicado o Ristretto com quantificação de vírgula fixa dinâmica utilizando margens de erro diferentes, os resultados obtidos estão apresentados na tabela 22.

		AlexNet - Ristretto com quantificação de vírgula fixa dinâmica				
		Margem de erro 0%	Margem de erro 0,125%	Margem de erro 0,15%	Margem de erro 1%	Margem de erro 2%
Vírgula flutuante de 32 bits		0,56822	0,56822	0,56822	0,56822	0,56822
Camada convolucional	Vírgula fixa dinâmica de 16 bits	0,567779	0,567779	0,567779	0,567779	0,567779
	Vírgula fixa dinâmica de 8 bits	-	0,566339	0,566339	0,566339	0,566339
	Vírgula fixa dinâmica de 4 bits	-	-	-	-	0,00268
Camada totalmente conectada	Vírgula fixa dinâmica de 16 bits	0,56742	0,56742	0,56742	0,56742	0,56742
	Vírgula fixa dinâmica de 8 bits	-	0,56678	0,56678	0,56678	0,56678
	Vírgula fixa dinâmica de 4 bits	-	-	0,55183	0,55183	0,55183
	Vírgula fixa dinâmica de 2 bits	-	-	-	-	0,000999999
Ativações das camadas	Vírgula fixa dinâmica de 16 bits	0,56576	0,56576	0,56576	0,56576	0,56576
	Vírgula fixa dinâmica de 8 bits	-	-	-	0,54802	0,54826
	Vírgula fixa dinâmica de 4 bits	-	-	-	-	0,0414001
Resultados	Camada convolucional	32 bits	16 bits	16 bits	8 bits	8 bits
	Camada totalmente conectada	32 bits	16 bits	8 bits	8 bits	4 bits
	Ativações das camadas	32 bits	32 bits	32 bits	16 bits	8 bits
	Precisão	0,5658	0,56572	0,56512	0,564039	0,530769

Tabela 22 - Resultados do Ristretto com quantificação de vírgula fixa dinâmica para a rede AlexNet.

Com os resultados apresentados na tabela 22 para a rede AlexNet verifica-se que alterar de uma representação de vírgula flutuante de 32 bits para uma representação de vírgula fixa dinâmica com 8 bits para os parâmetros (pesos e valores Bias) das camadas convolucionais, 4 bits para os parâmetros (pesos e valores Bias) das camadas totalmente conectadas e 8 bits para as ativações das camadas (valores de entrada e saída das camadas) reduz a precisão em, aproximadamente, 3.81%. Com os resultados obtidos da aplicação do Ristretto com uma margem de erro de 2% verifica-se que utilizar a rede AlexNet com uma representação de vírgula fixa dinâmica com 4 bits para os parâmetros (pesos e

valores Bias) das camadas convolucionais, com 2 bits para os parâmetros (pesos e valores Bias) das camadas totalmente conectadas, ou com 4 bits para as ativações das camadas (valores de entrada e saída das camadas) não é viável, pois os resultados da precisão para esse número de bits foi praticamente zero. No anexo C encontram os resultados para os parâmetros da representação de vírgula fixa dinâmica das arquiteturas geradas pela aplicação do Ristretto à CNN AlexNet e o modelo genérico desses mesmos parâmetros, como não foi possível treinar a rede devido aos requisitos computacionais esses valores não foram utilizados.

Como não foi possível treinar a rede AlexNet com a base de imagens ImageNet, foi feito um treino da rede AlexNet com a base de imagens Cifar10. Como as imagens da Cifar10 são mais pequenas do que as imagens da *ImageNet* e apenas são rotuladas em dez classes diferentes, foi necessário realizar algumas alterações na arquitetura da rede AlexNet. Devido às dimensões reduzidas das imagens da Cifar10, comparativamente à ImageNet, as camadas mais profundas da rede AlexNet, caso não fosse realizado nenhuma alteração, recebiam mapas de recursos de entrada de dimensão 0x0. Desta forma para evitar esse problema a primeira camada convolucional foi colocada com um *padding* de valor 4, as dimensões dos filtros utilizados pelas camadas de agrupamento foram alteradas para 2x2, e como as imagens da Cifar10 pertencem apenas a 10 classes diferentes a última camada totalmente conectada passou a ter apenas 10 saídas. O treino da AlexNet modificada com a base de imagens Cifar10 obteve uma precisão de 73.88% utilizando uma representação de vírgula flutuante de 32 bits. A esse modelo treinado foi aplicado o Ristretto utilizando uma quantificação de vírgula fixa dinâmica e os resultados obtidos estão representados na tabela 23.

		AlexNet treinada com Cifar10 - Ristretto com quantificação de vírgula fixa dinâmica					
		Margem de erro -1%	Margem de erro 0%	Margem de erro 1%	Margem de erro 2%	Margem de erro 3%	Margem de erro 10%
Vírgula flutuante de 32 bits		0,7388	0,7388	0,7388	0,7388	0,7388	0,7388
Camada convolucional	Vírgula fixa dinâmica de 16 bits	0,7398	0,7398	0,7398	0,7398	0,7398	0,7398
	Vírgula fixa dinâmica de 8 bits	-	0,7386	0,7386	0,7386	0,7386	0,7386
	Vírgula fixa dinâmica de 4 bits	-	-	0,665998	0,665998	0,665998	0,665998
	Vírgula fixa dinâmica de 2 bits	-	-	-	-	-	0,1259
Camada totalmente conectada	Vírgula fixa dinâmica de 16 bits	0,735799	0,735799	0,735799	0,735799	0,735799	0,735799
	Vírgula fixa dinâmica de 8 bits	-	-	0,7356	0,7356	0,7356	0,7356
	Vírgula fixa dinâmica de 4 bits	-	-	0,733901	0,733901	0,733901	0,733901
	Vírgula fixa dinâmica de 2 bits	-	-	0,609499	0,609499	0,609499	0,609499
Ativações das camadas	Vírgula fixa dinâmica de 16 bits	0,7399	0,7399	0,7399	0,7399	0,7399	0,7399
	Vírgula fixa dinâmica de 8 bits	-	0,7374	0,7374	0,7374	0,7374	0,7374
	Vírgula fixa dinâmica de 4 bits	-	-	0,416901	0,416901	0,416901	0,416901
Resultados	Camada convolucional	32 bits	16 bits	8 bits	8 bits	8 bits	4 bits
	Camada totalmente conectada	32 bits	32 bits	4 bits	4 bits	4 bits	4 bits
	Ativações das camadas	32 bits	16 bits	8 bits	8 bits	8 bits	8 bits
	Precisão	0,7338	0,7339	0,729201	0,729201	0,729201	0,6637

Tabela 23 - Resultados do Ristretto com quantificação de vírgula fixa dinâmica para a rede AlexNet modificada e treinada com a base de imagens Cifar10.

Através dos resultados apresentados na tabela 23 verifica-se que se pode alterar a representação dos dados da rede de vírgula flutuante de 32 bits para vírgula fixa dinâmica com 8 bits para os parâmetros (pesos e valores Bias) das camadas convolucionais, 4 bits para os parâmetros (pesos e valores Bias) das camadas totalmente conectadas e 8 bits para as ativações das camadas (valores de entrada e saída das camadas) reduzindo a precisão em apenas 0.96%. É possível verificar também que utilizar uma representação de vírgula

fixa dinâmica com 2 bits para os parâmetros (pesos e valores Bias) das camadas convolucionais, com 2 bits para os parâmetros (pesos e valores Bias) das camadas totalmente conectadas ou com 4 bits para as ativações das camadas (valores de entrada e saída das camadas) não é aconselhável pois os resultados da precisão descem para praticamente zero.

Com as arquiteturas da rede AlexNet modificada geradas pelo Ristretto para diferentes valores de margem de erro foi possível extrair os valores dos parâmetros da representação de vírgula fixa dinâmica e criar um modelo genérico para esses mesmos parâmetros. Esses resultados estão apresentados na tabela 24.

		AlexNet treinada com Cifar10 - Ristretto com quantificação de vírgula fixa dinâmica						
		Margem de erro -1%	Margem de erro 0%	Margem de erro 1%	Margem de erro 2%	Margem de erro 3%	Margem de erro 10%	Modelo Genérico
Número de bits para camada convolucional: AtivaçãoxPeso		32x32	16x16	8x8	8x8	8x8	8x4	XxY
Número de bits na camada totalmente conectada: AtivaçãoxPeso		32x32	16x32	8x4	8x4	8x4	8x4	XxZ
Camada Convolucional 1	bw_layer_in	32	16	8	8	8	8	X
	bw_layer_out	32	16	8	8	8	8	X
	bw_params	32	16	8	8	8	4	Y
	fl_layer_in	24	8	0	0	0	0	X-8
	fl_layer_out	21	5	-3	-3	-3	-3	X-11
	fl_params	33	17	9	9	9	5	Y+1
Camada Convolucional 2	bw_layer_in	32	16	8	8	8	8	X
	bw_layer_out	32	16	8	8	8	8	X
	bw_params	32	16	8	8	8	4	Y
	fl_layer_in	24	8	0	0	0	0	X-8
	fl_layer_out	23	7	-1	-1	-1	-1	X-9
	fl_params	32	16	8	8	8	4	Y
Camada Convolucional 3	bw_layer_in	32	16	8	8	8	8	X
	bw_layer_out	32	16	8	8	8	8	X
	bw_params	32	16	8	8	8	4	Y
	fl_layer_in	24	8	0	0	0	0	X-8
	fl_layer_out	24	8	0	0	0	0	X-8
	fl_params	33	17	9	9	9	5	Y+1
Camada Convolucional 4	bw_layer_in	32	16	8	8	8	8	X
	bw_layer_out	32	16	8	8	8	8	X
	bw_params	32	16	8	8	8	4	Y
	fl_layer_in	24	8	0	0	0	0	X-8
	fl_layer_out	26	10	2	2	2	2	X-6
	fl_params	34	18	10	10	10	6	Y+2
Camada Convolucional 5	bw_layer_in	32	16	8	8	8	8	X
	bw_layer_out	32	16	8	8	8	8	X
	bw_params	32	16	8	8	8	4	Y
	fl_layer_in	26	10	2	2	2	2	X-6
	fl_layer_out	26	10	2	2	2	2	X-6
	fl_params	34	18	10	10	10	6	Y+2
Camada Totalmente Conectada 1	bw_layer_in	32	16	8	8	8	8	X
	bw_layer_out	32	16	8	8	8	8	X
	bw_params	32	32	4	4	4	4	Z
	fl_layer_in	26	10	2	2	2	2	X-6
	fl_layer_out	28	12	4	4	4	4	X-4
	fl_params	35	35	7	7	7	7	Z+3
Camada Totalmente Conectada 2	bw_layer_in	32	16	8	8	8	8	X
	bw_layer_out	32	16	8	8	8	8	X
	bw_params	32	32	4	4	4	4	Z
	fl_layer_in	28	12	4	4	4	4	X-4
	fl_layer_out	29	13	5	5	5	5	X-3
	fl_params	37	37	9	9	9	9	Z+5
Camada Totalmente Conectada 3	bw_layer_in	32	16	8	8	8	8	X
	bw_layer_out	32	16	8	8	8	8	X
	bw_params	32	32	4	4	4	4	Z
	fl_layer_in	29	13	5	5	5	5	X-3
	fl_layer_out	26	11	3	3	3	3	X-5
	fl_params	34	34	6	6	6	6	Z+2

Tabela 24 - Resultados dos parâmetros da arquitetura da rede AlexNet modificada e treinada com a base de imagens Cifar10 aplicando o Ristretto com quantificação de vírgula fixa dinâmica.

Com o modelo genérico apresentado na tabela 24 foi possível gerar novas arquiteturas da rede AlexNet modificada para o treino da base de imagens Cifar10 utilizando representações para os dados de vírgula fixa dinâmica com diferentes tamanhos. Essas arquiteturas geradas foram treinadas com a base de imagens Cifar10 e os resultados obtidos estão representados na tabela 25.

		AlexNet modificada com representação de vírgula fixa dinâmica						
Número de bits para camada convolucional: AtivaçãoxPeso		16x16	8x8	5x5	4x4	8x8	8x4	8x4
Número de bits na camada totalmente conectada: AtivaçãoxPeso		16x16	8x8	5x5	4x4	8x4	8x4	8x2
Camada Convolucional 1	bw_layer_in	16	8	5	4	8	8	8
	bw_layer_out	16	8	5	4	8	8	8
	bw_params	16	8	5	4	8	4	2
	fl_layer_in	8	0	-3	-4	0	0	0
	fl_layer_out	5	-3	-6	-7	-3	-3	-3
	fl_params	17	9	6	5	9	5	3
Camada Convolucional 2	bw_layer_in	16	8	5	4	8	8	8
	bw_layer_out	16	8	5	4	8	8	8
	bw_params	16	8	5	4	8	4	2
	fl_layer_in	8	0	-3	-4	0	0	0
	fl_layer_out	7	-1	-4	-5	-1	-1	-1
	fl_params	16	8	5	4	8	4	2
Camada Convolucional 3	bw_layer_in	16	8	5	4	8	8	8
	bw_layer_out	16	8	5	4	8	8	8
	bw_params	16	8	5	4	8	4	2
	fl_layer_in	8	0	-3	-4	0	0	0
	fl_layer_out	8	0	-3	-4	0	0	0
	fl_params	17	9	6	5	9	5	3
Camada Convolucional 4	bw_layer_in	16	8	5	4	8	8	8
	bw_layer_out	16	8	5	4	8	8	8
	bw_params	16	8	5	4	8	4	2
	fl_layer_in	8	0	-3	-4	0	0	0
	fl_layer_out	10	2	-1	-2	2	2	2
	fl_params	18	10	7	6	10	6	4
Camada Convolucional 5	bw_layer_in	16	8	5	4	8	8	8
	bw_layer_out	16	8	5	4	8	8	8
	bw_params	16	8	5	4	8	4	2
	fl_layer_in	10	2	-1	-2	2	2	2
	fl_layer_out	10	2	-1	-2	2	2	2
	fl_params	18	10	7	6	10	6	4
Camada Totalmente Conectada 1	bw_layer_in	16	8	5	4	8	8	8
	bw_layer_out	16	8	5	4	8	8	8
	bw_params	16	8	5	4	4	4	2
	fl_layer_in	10	2	-1	-2	2	2	2
	fl_layer_out	12	4	1	0	4	4	4
	fl_params	19	11	8	7	7	7	5
Camada Totalmente Conectada 2	bw_layer_in	16	8	5	4	8	8	8
	bw_layer_out	16	8	5	4	8	8	8
	bw_params	16	8	5	4	4	4	2
	fl_layer_in	12	4	1	0	4	4	4
	fl_layer_out	13	5	2	1	5	5	5
	fl_params	21	13	10	9	9	9	7
Camada Totalmente Conectada 3	bw_layer_in	16	8	5	4	8	8	8
	bw_layer_out	16	8	5	4	8	8	8
	bw_params	16	8	5	4	4	4	5
	fl_layer_in	13	5	2	1	5	5	5
	fl_layer_out	11	3	0	-1	3	3	3
	fl_params	18	10	7	6	6	6	4
Precisão		0,7383	0,7421	0.693	0.339	0.7433	0.7096	0.6922

Tabela 25 - Resultados da rede AlexNet modificada e treinada com a base de imagens Cifar10, utilizando os parâmetros da quantificação de vírgula fixa dinâmica para representações dos dados com diferentes números de bits.

Com o modelo de desempenho e os resultados obtidos representados na tabela 25 foi possível verificar a relação entre o número de bits para representar os dados com a precisão da rede e o atraso das arquiteturas quando estas utilizam uma área semelhante da FPGA (45700 LUT ± 4%, 220 DSP). Essa relação para a rede AlexNet treinada com a base de imagens Cifar10 está ilustrada na figura 32.

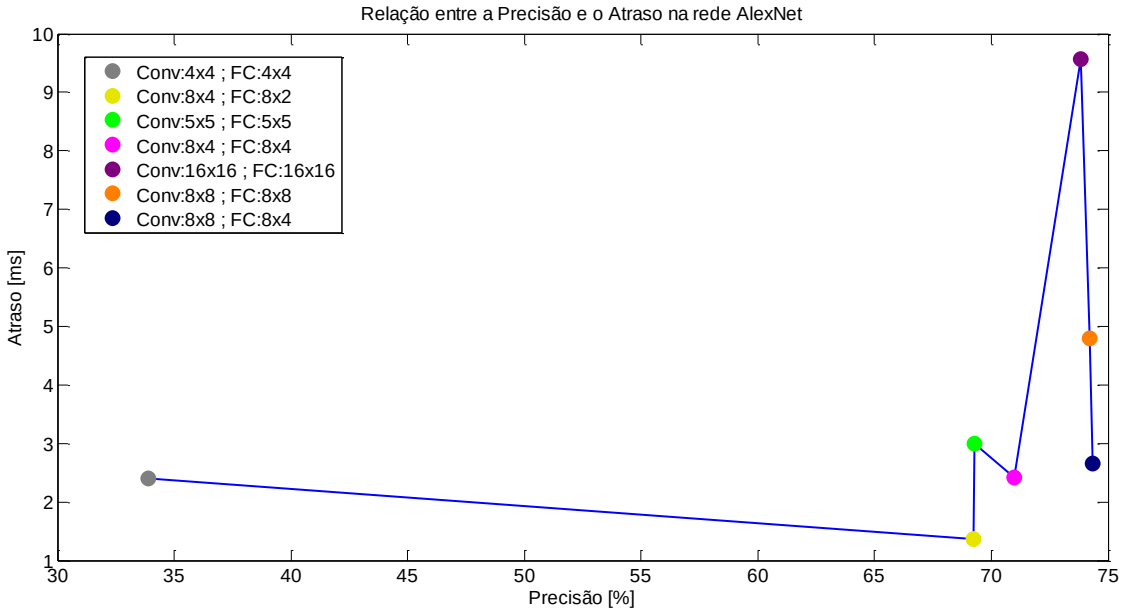


Figura 32 - Relação entre a precisão e o atraso na rede AlexNet.

Através dos resultados obtidos para a rede AlexNet treinada com a base de imagens da cifar10 verifica-se que a arquitetura que utiliza 8 bits para representar as ativações, 4 bits para representar os pesos das camadas convolucionais e 2 bits para representar os pesos das camadas totalmente conectadas é a arquitetura que possui um atraso mais baixo e a precisão é reduzida em apenas 4.65%, comparativamente com a arquitetura que utiliza uma representação de vírgula flutuante de 32 bits. A arquitetura que utiliza 8 bits para representar as ativações e os pesos das camadas convolucionais e 4 bits para representar os pesos das camadas totalmente conectadas é a arquitetura que possui melhor precisão, e possui uma boa relação entre a precisão e o atraso. Verifica-se também que alterar a representação dos pesos das camadas convolucionais de 8 bits para 4 bits não compensa neste caso, pois o atraso é semelhante e reduz a precisão.

5.5 Rede SqueezeNet

A rede SqueezeNet possui o mesmo problema que a rede AlexNet, em que o treino utilizando a base de imagens ImageNet possui requisitos computacionais muito elevados, o que impossibilita o treino da rede com os recursos deste projeto. Assim foi utilizado um modelo da rede SqueezeNet previamente treinado disponibilizado pela comunidade da ferramenta Caffe. A esse modelo foi aplicado o Ristretto com quantificação de vírgula fixa dinâmica utilizando margens de erro diferentes, os resultados obtidos estão apresentados na tabela 26.

		SqueezeNet - Ristretto com quantificação de vírgula fixa dinâmica				
		Margem de erro -1%	Margem de erro 0%	Margem de erro 1%	Margem de erro 2%	Margem de erro 3%
Vírgula flutuante de 32 bits		0,5767	0,5767	0,5767	0,5767	0,5767
Camada convolucional	Vírgula fixa dinâmica de 16 bits	0,55704	0,55704	0,55704	0,55704	0,55704
	Vírgula fixa dinâmica de 8 bits	-	-	-	0,55582	0,55582
	Vírgula fixa dinâmica de 4 bits	-	-	-	-	0,00568
Ativações das camadas	Vírgula fixa dinâmica de 16 bits	0,57542	0,57542	0,57542	0,57542	0,57542
	Vírgula fixa dinâmica de 8 bits	-	-	0.55974	0.55974	0.55974
	Vírgula fixa dinâmica de 4 bits	-	-	-	0,0059	0,0059
Resultados	Camada convolucional	32 bits	32 bits	32 bits	16 bits	8 bits
	Ativações das camadas	32 bits	32 bits	16 bits	8 bits	8 bits
	Precisão	0,55664	0,55664	0,55282	0,55282	0,53502

Tabela 26 - Resultados do Ristretto com quantificação de vírgula fixa dinâmica para a rede SqueezeNet.

Os resultados apresentados na tabela 26 não apresentam resultados para as camadas totalmente conectadas porque a arquitetura da rede SqueezeNet não é constituída por nenhuma camada totalmente conectada. Com os resultados obtidos da aplicação do Ristretto à CNN SqueezeNet pode-se verificar que representar os dados da rede com vírgula fixa dinâmica de 8 bits provoca uma diminuição da precisão da rede, comparativamente à representação de vírgula flutuante de 32 bits, de 4.17%.

Capítulo 6 – Conclusões

Com os resultados obtidos neste projeto é possível concluir que o impacto de alterar a representação dos dados de vírgula flutuante de 32 bits para uma representação dos dados de vírgula fixa dinâmica com um número menor de bits varia conforme a CNN.

Conclui-se também que para a maioria das redes estudadas neste projeto a arquitetura que utiliza 4 bits para representar as ativações e os pesos das camadas convolucionais e das camadas totalmente conectadas não é opção, apesar de ser a que possui geralmente o atraso menor, pois a precisão da rede diminui bastante inviabilizando a sua utilização na classificação de imagens. No entanto, verifica-se que utilizando 5 ou 6 bits para representar as ativações e os pesos das camadas convolucionais e das camadas totalmente conectadas se conseguem obter precisões superiores e, em alguns casos, próximas das conseguidas com representações a 8 e a 16 bits, com redução de recursos e do atraso comparado com a arquitetura típica a 8 bits.

Através dos resultados conclui-se também que a arquitetura que utiliza uma representação com vírgula fixa dinâmica de 16 bits para as ativações e para os pesos das camadas convolucionais e das camadas totalmente conectadas é, geralmente, ineficiente comparado com uma arquitetura a 8 bits pois a relação entre precisão/recursos hardware/desempenho é inferior, com precisões similares.

A arquitetura com uma representação de vírgula fixa dinâmica de 8 bits para ativações, 4 bits para os pesos nas camadas convolucionais e 2 bits para os pesos nas camadas totalmente conectadas é uma das arquiteturas que possui o atraso mais baixo, para redes com um número alto de parâmetros como é o caso da LeNet e da AlexNet treinada com a base de imagens Cifar10, sem comprometer a precisão da rede.

Por fim conclui-se que a melhor representação para os dados da CNN depende do objetivo da implementação (por exemplo, melhor precisão ou menor atraso) e das características do hardware onde a CNN será implementada.

Utilizando a arquitetura LiteCNN com otimização do tamanho de dados, é possível obter desempenhos de pico superiores a 500 GOPs (arquitetura a 6 bits) ou até 600 GOPs (arquitetura a 5 bits) com uma FPGA de baixa densidade.

6.1 Trabalho Futuro

Um trabalho futuro será estudar o impacto que as otimizações de redução do tamanho dos dados tem sobre o consumo energético. Esta métrica não foi considerada neste trabalho, mas é importante no âmbito do desenvolvimento de CNN em sistemas embebidos.

Também propomos realizar um estudo sobre o impacto da redução do número de bits na representação dos dados da rede em conjunto com técnicas de redução e compressão dos dados para diminuir os requisitos computacionais e de memória da CNN, como a compressão, a partilha de pesos e o método de *prunning*.

A arquitetura LiteCNN está pensada para permitir a execução de CNN em sistemas de baixos recursos computacionais. No entanto, acreditamos que também é uma boa solução para ser implementada em FPGA de elevado desempenho. Um outro trabalho futuro, seria implementar a LiteCNN em FPGA de elevada densidade e comparar com outras soluções no mesmo âmbito.

Referências

[1] Y. L. Cun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, W. Hubbard, “Handwritten digit recognition: applications of neural network chips and automatic learning”, IEEE Communications Magazine, vol. 27, no. 11, pp. 41–46, Nov. 1989.

[2] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey", Proceedings of the IEEE, vol. 105, no. 12, pp. 2295-2329, Dec. 2017.

[3] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-based learning applied to document recognition", Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998.

[4] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, "ImageNet classification with deep convolutional neural networks", NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing Systems, vol. 1, pp. 1097-1105, Dec. 2012.

[5] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, "Going deeper with convolutions", IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Oct. 2015.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Deep Residual Learning for Image Recognition", IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Dec. 2016.

[7] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, Kurt Keutzer, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5MB model size”, arXiv:1602.07360, Feb. 2016.

[8] Andrea Solazzo, “An automated design framework for FPGA-based hardware accelerators of Convolutional Neural Networks”, Tese de Mestrado, Politécnico de Milão, Apr. 2017.

[9] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, Soheil Ghiasi, "Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks", IEEE Transactions on Neural Networks and Learning Systems, vol. 29, no. 11, pp. 5784–5789, Mar. 2018.

[10] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, Hartwig Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", arXiv:1704.04861, Apr. 2017.

[11] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, Jian Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices", arXiv:1707.01083, Jul. 2017.

[12] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, Yuheng Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients", arXiv:1606.06160, Jun. 2016.

[13] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, Jason Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks",

FPGA '15 Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 161-170, Feb. 2015.

[14] Yuran Qiao, Junzhong Shen, Tao Xiao, Qianming Yang, Mei Wen, Chunyuan Zhang, "FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency", Concurrency and Computation: Practice and Experience, vol. 29, no. 20, May. 2016.

[15] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, Eric S. Chung, "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware", Microsoft Research, Feb. 2015.

[16] Atul Rahman, Jongeun Lee, Kiyoung Choi, "Efficient FPGA acceleration of Convolutional Neural Networks using logical-3D compute array", 2016 Design, Automation & Test in Europe Conference & Exhibition, Apr. 2016.

[17] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, "A dynamically configurable coprocessor for convolutional neural networks", ACM SIGARCH Computer Architecture News - ISCA '10, vol. 38, no. 3 pp. 247-257, Jun. 2010.

[18] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, Huazhong Yang, "Going deeper with embedded FPGA platform for convolutional neural network", FPGA '16 Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 26-35, Feb. 2016.

[19] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Song Yao, Song Han, Yu Wang, Huazhong Yang, "Angel-Eye: A Complete Design Flow for Mapping CNN onto Customized Hardware", 2016 IEEE Computer Society Annual Symposium on VLSI, Sep. 2016.

[20] Karen Simonyan, Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", arXiv:1409.1556, Sep. 2014.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, Trevor Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding", arXiv preprint arXiv:1408.5093, Jun. 2014.

[22] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge", arXiv:1409.0575, Jan. 2015.

[23] Mário Véstias, José Teixeira, Rui Duarte, Horácio Neto, "Lite-CNN: A High-Performance Architecture to Run CNNs in Low Density FPGAs", in international IEEE conference on Field Programmable Logic, FPL 2018.

[24] Martín Abadi, et al., "TensorFlow: Large-scale machine learning on heterogeneous systems", 2015.

[25] Theano Development Team, "Theano: A {Python} framework for fast computation of mathematical expressions", arXiv:1605.02688, May 2016.

[26] Mário P. Véstias, Rui Policarpo Duarte, José T. de Sousa, Horácio C. Neto, "Parallel dot-products for deep learning on FPGA", 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Sept. 2017.

Anexos

A. Resultados da rede LeNet com vírgula fixa dinâmica

		LeNet com representação de vírgula fixa dinâmica									
Número de bits para camada convolucional: AtivaçãoxPeso		32x32	15x15	14x14	13x13	12x12	11x11	10x10	9x9	7x7	3x3
Número de bits na camada totalmente conectada: AtivaçãoxPeso		32x32	15x15	14x14	13x13	12x12	11x11	10x10	9x9	7x7	3x3
Camada Convolucional 1	bw_layer_in	32	15	14	13	12	11	10	9	7	3
	bw_layer_out	32	15	14	13	12	11	10	9	7	3
	bw_params	32	15	14	13	12	11	10	9	7	3
	fl_layer_in	32	15	14	13	12	11	10	9	7	3
	fl_layer_out	30	13	12	11	10	9	8	7	5	1
	fl_params	31	14	13	12	11	10	9	8	6	2
Camada Convolucional 2	bw_layer_in	32	15	14	13	12	11	10	9	7	3
	bw_layer_out	32	15	14	13	12	11	10	9	7	3
	bw_params	32	15	14	13	12	11	10	9	7	3
	fl_layer_in	30	13	12	11	10	9	8	7	5	1
	fl_layer_out	28	11	10	9	8	7	6	5	3	-1
	fl_params	33	16	15	14	13	12	11	10	8	4
Camada Totalmente Conectada 1	bw_layer_in	32	15	14	13	12	11	10	9	7	3
	bw_layer_out	32	15	14	13	12	11	10	9	7	3
	bw_params	32	15	14	13	12	11	10	9	7	3
	fl_layer_in	28	11	10	9	8	7	6	5	3	-1
	fl_layer_out	28	11	10	9	8	7	6	5	3	-1
	fl_params	34	17	16	15	14	13	12	11	9	5
Camada Totalmente Conectada 2	bw_layer_in	32	15	14	13	12	11	10	9	7	3
	bw_layer_out	32	15	14	13	12	11	10	9	7	3
	bw_params	32	15	14	13	12	11	10	9	7	3
	fl_layer_in	28	11	10	9	8	7	6	5	3	-1
	fl_layer_out	27	10	9	8	7	6	5	4	2	-2
	fl_params	32	15	14	13	12	11	10	9	7	3
Precisão		0,9895	0,9904	0,9895	0,9895	0,9893	0,9893	0,9893	0,9898	0,9894	0,6525

Tabela 27 - Resultados restantes da rede LeNet treinada com os parâmetros da quantificação de vírgula fixa dinâmica para representações dos dados com diferentes números de bits.

B. Resultados da rede Cifar10_quick com vírgula fixa dinâmica

		CIFAR10_quick com representação de vírgula fixa dinâmica								
Número de bits para camada convolucional: AtivaçãoxPeso		32x32	15x15	14x14	13x13	12x12	11x11	10x10	9x9	3x3
Número de bits na camada totalmente conectada: AtivaçãoxPeso		32x32	15x15	14x14	13x13	12x12	11x11	10x10	9x9	3x3
Camada Convolucional 1	bw_layer_in	32	15	14	13	12	11	10	9	3
	bw_layer_out	32	15	14	13	12	11	10	9	3
	bw_params	32	15	14	13	12	11	10	9	3
	fl_layer_in	24	7	6	5	4	3	2	1	-5
	fl_layer_out	23	6	5	4	3	2	1	0	-6
	fl_params	33	16	15	14	13	12	11	10	4
Camada Convolucional 2	bw_layer_in	32	15	14	13	12	11	10	9	3
	bw_layer_out	32	15	14	13	12	11	10	9	3
	bw_params	32	15	14	13	12	11	10	9	3
	fl_layer_in	23	6	5	4	3	2	1	0	-6
	fl_layer_out	23	6	5	4	3	2	1	0	-6
	fl_params	34	17	16	15	14	13	12	11	5
Camada Convolucional 3	bw_layer_in	32	15	14	13	12	11	10	9	3
	bw_layer_out	32	15	14	13	12	11	10	9	3
	bw_params	32	15	14	13	12	11	10	9	3
	fl_layer_in	24	7	6	5	4	3	2	1	-5
	fl_layer_out	26	9	8	7	6	5	4	3	-3
	fl_params	34	17	16	15	14	13	12	11	5
Camada Totalmente Conectada 1	bw_layer_in	32	15	14	13	12	11	10	9	3
	bw_layer_out	32	15	14	13	12	11	10	9	3
	bw_params	32	15	14	13	12	11	10	9	3
	fl_layer_in	26	9	8	7	6	5	4	3	-3
	fl_layer_out	27	10	9	8	7	6	5	4	-2
	fl_params	32	15	14	13	12	11	10	9	3
Camada Totalmente Conectada 2	bw_layer_in	32	15	14	13	12	11	10	9	3
	bw_layer_out	32	15	14	13	12	11	10	9	3
	bw_params	32	15	14	13	12	11	10	9	3
	fl_layer_in	27	10	9	8	7	6	5	4	-2
	fl_layer_out	27	10	9	8	7	6	5	4	-2
	fl_params	32	15	14	13	12	11	10	9	3
Precisão		0,7524	0,738184	0,7544	0,747801	0,7531	0,7511	0,75	0,7507	0,0057

Tabela 28 - Resultados restantes da rede Cifar10_quick treinada com os parâmetros da quantificação de vírgula fixa dinâmica para representações dos dados com diferentes números de bits.

C. Resultados dos parâmetros da AlexNet com vírgula fixa dinâmica

		AlexNet - Ristretto com quantificação de vírgula fixa dinâmica					
		Margem de erro 0%	Margem de erro 0,125%	Margem de erro 0,15%	Margem de erro 1%	Margem de erro 2%	Modelo Genérico
Número de bits para camada convolucional: AtivaçãoxPeso		32x32	32x16	32x16	16x8	8x8	XxY
Número de bits na camada FC: AtivaçãoxPeso		32x32	32x16	32x8	16x8	8x4	XxZ
Camada Convolucional 1	bw_layer_in	32	32	32	16	8	X
	bw_layer_out	32	32	32	16	8	X
	bw_params	32	16	16	8	8	Y
	fl_layer_in	24	24	24	8	0	X-8
	fl_layer_out	20	20	20	4	-4	X-12
	fl_params	32	16	16	8	8	Y
Camada Convolucional 2	bw_layer_in	32	32	32	16	8	X
	bw_layer_out	32	32	32	16	8	X
	bw_params	32	16	16	8	8	Y
	fl_layer_in	24	24	24	8	0	X-8
	fl_layer_out	22	22	22	6	-2	X-10
	fl_params	32	16	16	8	8	Y
Camada Convolucional 3	bw_layer_in	32	32	32	16	8	X
	bw_layer_out	32	32	32	16	8	X
	bw_params	32	16	16	8	8	Y
	fl_layer_in	24	24	24	8	0	X-8
	fl_layer_out	23	23	23	6	-1	X-9
	fl_params	32	16	16	8	8	Y
Camada Convolucional 4	bw_layer_in	32	32	32	16	8	X
	bw_layer_out	32	32	32	16	8	X
	bw_params	32	16	16	8	8	Z
	fl_layer_in	23	23	23	6	-1	X-9
	fl_layer_out	23	23	23	7	-1	X-9
	fl_params	32	16	16	8	8	Z
Camada Convolucional 5	bw_layer_in	32	32	32	16	8	X
	bw_layer_out	32	32	32	16	8	X
	bw_params	32	16	16	8	8	Z
	fl_layer_in	23	23	23	7	-1	X-9
	fl_layer_out	23	23	23	7	-1	X-9
	fl_params	32	16	16	8	8	Z
Camada Totalmente Conectada 1	bw_layer_in	32	32	32	16	8	X
	bw_layer_out	32	32	32	16	8	X
	bw_params	32	16	8	8	4	Z
	fl_layer_in	23	23	23	7	-1	X-9
	fl_layer_out	24	24	24	8	0	X-8
	fl_params	35	19	11	11	7	Z+3
Camada Totalmente Conectada 2	bw_layer_in	32	32	32	16	8	X
	bw_layer_out	32	32	32	16	8	X
	bw_params	32	16	8	8	4	Z
	fl_layer_in	24	24	24	8	0	X-8
	fl_layer_out	25	26	26	10	2	X-6
	fl_params	34	18	10	10	6	Z+2
Camada Totalmente Conectada 3	bw_layer_in	32	32	32	16	8	X
	bw_layer_out	32	32	32	16	8	X
	bw_params	32	16	8	8	4	Z
	fl_layer_in	25	26	26	10	2	X-6
	fl_layer_out	26	26	26	10	2	X-6
	fl_params	34	18	10	10	6	Z+2

Tabela 29 - Resultados dos parâmetros da arquitetura AlexNet aplicando o Ristretto com quantificação de vírgula fixa dinâmica.