Proposta de implementação em Hardware de Rede Neural Profunda baseada em Stacked Sparse Autoencoder

Maria G. F. Coutinho e Marcelo A. C. Fernandes
Departamento de Engenharia da Computação e Automação (DCA)
Universidade Federal do Rio Grande do Norte (UFRN)
Natal, Brasil
gracielly@dca.ufrn.br, mfernandes@dca.ufrn.br

***Resumo*—O objetivo deste trabalho consiste em propor a implementação em hardware de uma Rede Neural Profunda (*Deep Neural Network* - DNN) baseada na técnica *Stacked Sparse Autoencoder* (SSAE). O hardware proposto foi desenvolvido em *Field Programmable Gate Array* (FPGA) utilizando ponto fixo. A técnica de matriz sistólica (*systolic array*) foi adotada em todo o circuito com a finalidade de permitir a utilização de DNNs com muitas entradas, neurônios e camadas na FPGA. Todos os detalhes da arquitetura desenvolvida são apresentados, incluindo informações referentes a taxa de ocupação dos recursos de hardware e ao tempo de processamento para uma FPGA Virtex 6 xc6vlx240t-1ff1156. Os resultados indicam que a implementação foi capaz de atingir *throughputs* elevados, além de alcançar um *speedup* significativo em comparação com um trabalho do estado da arte, o que aponta a viabilidade da aplicação da proposta apresentada neste artigo em problemas de dados massivos.**

***Palavras-chave*—Aprendizagem Profunda, Stacked Sparse Autoencoder, Hardware, FPGA, Matriz Sistólica.**

I. Introdução

Tendo em vista a crescente utilização de técnicas de Inteligência Artificial (IA) para resolução de problemas de diversas áreas, as técnicas de *Deep Learning* (DL), também conhecidas como Redes Neurais Profundas (*Deep Neural Network* - DNN), vêm ganhando grande destaque nos últimos anos. As DNNs são capazes de prover alto poder computacional, concomitantemente com a utilização de várias camadas ocultas. Entre as várias técnicas de *Deep Learning* existentes na literatura, encontram-se as baseadas em *Autoencoders* (AEs), aplicadas principalmente a problemas de predição e classificação [1]–[3].

A técnica de utilizar vários *autoencoders* encadeados, formando uma única DNN, vem se mostrando muito útil no treinamento de redes de aprendizagem profunda, como pode ser visto em [4], [5]. Os *Stacked Sparse Autoencoders* (SSAE) vêm sendo utilizados em problemas de classificação [5]. Nesta abordagem, cada camada oculta é composta por um *sparse autoencoder* treinado individualmente, de forma não supervisionada. A saída da camada oculta de cada AE é utilizada como entrada no AE seguinte, de modo que as características dos dados de entrada sejam propagadas pela rede camada a camada, possibilitando que a camada de saída seja capaz de realizar a classificação dos dados, após um treinamento supervisionado. Entretanto, as redes neurais profundas possuem uma complexidade computacional elevada em decorrência da grande quantidade de camadas ocultas adicionadas, o que dificulta ou até inviabiliza sua utilização em várias aplicações comerciais. Em contrapartida, vários trabalhos voltados para acelerar algoritmos complexos vêm sendo desenvolvidos, e entre eles a utilização de computação reconfigurável tem se mostrado uma ótima alternativa.

É possível encontrar na literatura uma grande variedade de trabalhos com implementações em hardware para algoritmos de Inteligência Artificial. Como é o caso dos trabalhos desenvolvidos em [6], [7], nos quais foi possível atingir elevados ganhos de velocidade com a utilização de computação reconfigurável em *Field Programmable Gate Array* (FPGA), em comparação com implementações em processadores gráficos (*Graphics Processing Unit* - GPU) e processadores de uso geral utilizados em computadores de alto desempenho (*High Performance Computer* - HPC).

Em [6] foi implementada uma Rede Neural Artificial (RNA) em hardware, do tipo Funções de Base Radial (RBF), treinada com o algoritmo *Least Mean Square* (LMS). A implementação foi analisada em termos de taxa de ocupação do hardware, resolução em bits e atraso de processamento. Os resultados das sínteses para dois cenários distintos e várias resoluções de ponto fixo apontam para a possibilidade de utilização da proposta em situações práticas mais complexas. Já no trabalho de [7] foi proposto uma implementação paralela de algoritmo genético, em FPGA. A arquitetura implementada em *Register Transfer Level* (RTL) compreendeu testes com diferentes parâmetros. As funções utilizadas pelo algoritmo genético foram armazenadas em *Lookup Tables* (LUTs), eliminando assim a necessidade de um circuito dedicado para realizar cada função e, consequentemente, permitiu a redução da área ocupada na FPGA e tornou a implementação mais flexível, além disso, proporcionou o aumento do *throughput*, já que construir um circuito complexo para estas funções aumentaria o caminho crítico da implementação.

Alguns trabalhos envolvendo implementações de técnicas de DL em FPGA motivaram o desenvolvimento deste trabalho.

Em [8] foi proposto uma implementação de Redes Neurais Convolucionais (*Convolutional Neural Network* - CNN) em FPGA, utilizando *high level synthesis* (HLS). Os experimentos mostraram que a implementação em FPGA alcançou até 3 vezes mais eficiência energética que uma implementação em CPU, e eficiência energética equivalente a implementação na mesma CPU com 16 *threads* operando em paralelo. Além disso, com relação a implementação em SoC GPU para aplicações mobile, foi possível obter um ganho de quase 15 vezes em termos de velocidade e 16 vezes em termos de eficiência energética. Já no trabalho de [9] foi proposto um compilador RTL para acelerar algoritmos de DL, que visava alcançar desempenho semelhante às implementações em RTL, tendo em vista que implementações que utilizam *high level synthesis* não conseguem otimizar o consumo dos recursos da FPGA. Comparações com trabalhos da literatura deixam claro que o compilador RTL proposto consegue resultados bastante superiores às implementações com HLS.

Uma das primeiras implementações de DNN em FPGA utilizando *Stacked Sparse Autoencoders*, é apresentada no trabalho de [5]. A DNN foi utilizada para realizar classificação de imagens do conjunto de dados chamado de CIFAR-10. A arquitetura da rede utilizou duas camadas escondidas, contendo 2000 neurônios na primeira camada escondida e 750 na segunda, além de 3072 entradas e 10 neurônios na camada de saída (chamada de arquitetura 3072-2000-750-10). Para a implementação foi aplicado um *framework* de *high level synthesis* em OpenCL, o que possivelmente contribuiu para que os resultados para a FPGA não fossem superiores quando comparados a implementação em GPUs. Já no trabalho de [10] são mostradas as vantagens da utilização de ponto fixo em implementações de *autoencoder*, tendo em vista que, ao utilizar uma resolução de apenas 4 bits na parte inteira foi possível obter a mesma acurácia da classificação da resolução em ponto flutuante. Para os experimentos foi utilizada a arquitetura de rede 784-400-10 e uma base de dados de imagens de dígitos manuscritos, chamada de MNIST. No entanto, não foram apresentadas informações referentes a taxa de ocupação dos recursos de hardware da FPGA, e ao tempo de processamento.

Nos trabalhos de [11], [12] são apresentadas propostas de redes tradicionais utilizando *autoencoders*. Em [11] foi implementada uma arquitetura de *sparse autoencoder*, em *VerilogHDL*. Os testes foram realizados com uma rede de arquitetura 196-100-196, utilizando o conjunto de dados de imagens naturais da cidade de Kyoto no Japão. Já no trabalho de [12] foi proposta, além de arquiteturas tradicionais de *autoencoders*, uma estrutura com dois *autoencoders* encadeados, a arquitetura 4-2-1-2-4, no entanto, objetivando apenas reconstruir a entrada na saída, como nos modelos convencionais. Neste trabalho, diferentemente dos trabalhos que implementaram técnicas de *autoencoders* citados anteriormente, o circuito foi implementado em RTL, o que possibilitou alcançar eficiência superior às demais implementações de *autoencoder* comparadas no trabalho. A FPGA alvo foi uma Xilinx Virtex-6 xc6vlx240t.

Com o objetivo de garantir a eficiência da implementação em FPGA, através da otimização da utilização de seus recursos, a implementação proposta neste trabalho foi construída em RTL, diferentemente do trabalho de [5]. A implementação em RTL também foi adotada em [12], no entanto, neste trabalho foram implementadas estruturas de *autoencoders* mais semelhantes às tradicionais do que às DNNs. Assim, este artigo apresenta uma proposta de implementação em hardware de uma Rede Neural Profunda baseada na técnica *Stacked Sparse Autoencoder*. O hardware foi desenvolvido para a fase *feedfoward* adotando a técnica de matriz sistólica, o que permitiu a utilização de muitos neurônios e várias camadas. Dados com relação a taxa de ocupação em hardware e ao tempo de processamento serão apresentados para uma FPGA Virtex 6 xc6vlx240t-1ff1156.

II. Aprendizagem Profunda

Nos últimos anos, diversas técnicas de *Deep Learning* vêm se tornando objeto de pesquisa de muitos trabalhos acadêmicos ao redor do mundo. Tais técnicas podem ser definidas como uma modernização das redes *Multilayer Perceptron* (MLP), tendo em vista que uma das principais diferenças entre as redes MLP e as DNNs consiste na viabilidade do treinamento de redes com muitas camadas ocultas, nas redes de aprendizagem profunda, o que era um grande problema nas MLPs convencionais.

Entre as técnicas de *Deep Learning* existentes na literatura, estão as baseadas em *autoencoders*. No entanto, o termo *autoencoder* antecede o advento da *Deep Learning*. Em sua estrutura tradicional, os *autoencoders* eram principalmente aplicados a problemas de redução de dimensionalidade e aprendizagem de características. Nestas redes o treinamento ocorre para que a camada de saída forneça uma reconstrução da camada de entrada, sendo assim, ambas camada possuem o mesmo tamanho, além disso é utilizada apenas uma camada oculta, que realiza a extração de características dos dados de entrada. Sendo assim, a arquitetura do *autoencoder* é composta por três camadas: uma de entrada, uma oculta e uma de saída. As camadas de entrada e oculta formam o *encoder* da rede, e as camadas oculta e de saída, compõem o *decoder* [13].

A. Stacked Sparse Autoencoder (SSAE)

Com o surgimento da *Deep Learning*, os *autoencoders* passaram a ser utilizados de maneira encadeada, formando redes com muitas camadas ocultas, cujo treinamento é realizado previamente em cada *autoencoder*, de forma não supervisionada. Entre os tipos de *autoencoder*, estão os *sparse autoencoders*, muito aplicados a problemas de classificação. Nos *stacked sparse autoencoders*, cada camada oculta é composta pela camada oculta de um *sparse autoencoder* treinado individualmente. Cada *sparse autoencoder* recebe como entrada, a saída da camada oculta do *sparse autoencoder* que o antecede, de modo que as características dos dados de entrada sejam extraídas ao longo das camadas ocultas da rede, possibilitando que a camada de saída seja capaz de efetuar a classificação dos dados, após um treinamento supervisionado. Neste trabalho,

primeiramente realizou-se o treinamento da rede por meio desta técnica. A Figura 1 apresenta a arquitetura do SSAE proposto.

Este trabalho destinou-se a implementar a fase *feedfoward* do SSAE, na qual a equação que define a saída do i-ésimo neurônio da k-ésima camada, $z_i^k(n)$, no n-ésimo instante, pode ser expressa como

$$z_i^k(n) = \sum_{j=1}^{U^l} w_{ij}^k(n) \cdot y_j^l(n) + wb_i^k(n) \cdot b \tag{1}$$

em que $w_{ij}^k(n)$ é o peso associado a j-ésima entrada do i-ésimo neurônio da k-ésima camada no n-ésimo instante, $y_j^l(n)$ é a j-ésima entrada da l-ésima camada, na qual $l = k - 1$, no n-ésimo instante, $wb_i^k(n)$ é o peso referente ao *bias* do i-ésimo neurônio da k-ésima camada no n-ésimo instante, b é o *bias*, que possui valor de 1, e U^l é o número de entradas da l-ésima camada, sendo $U^0 = P$, $U^1 = M$ e $U^2 = N$. Nas camadas ocultas utilizou-se a função de ativação sigmoide, desse modo, a saída associada ao i-ésimo neurônio da k-ésima camada, no n-ésimo instante, $v_i^k(n)$, pode ser definida como

$$v_i^k(n) = \frac{1}{1 + e^{-z_i^k(n)}} \tag{2}$$

na qual $v_i^k(n)$ será o valor da j-ésima entrada a ser utilizada na camada seguinte no n-ésimo instante, $y_j^{l+1}(n)$, ou seja $y_j^{l+1}(n) = v_i^k(n)$, em que $j = i$.

A função de ativação *softmax* foi utilizada na camada de saída. Esta função vem sendo adotada em redes neurais de classificação, como apresentado em [5], e pode ser caracterizada como

$$s_i(n) = \frac{e^{z_i^K(n)}}{\sum_{h=1}^{H} e^{z_h^K(n)}} \tag{3}$$

na qual $s_i(n)$ consiste na i-ésima saída da última camada, K, com H neurônios, no n-ésimo instante. O valor de H é determinado pela quantidade de classes do problema, já que esta função indica a probabilidade de cada dado pertencer a uma classe específica.

III. Descrição do projeto

Tendo como base a estrutura da rede detalhada na Figura 1, a arquitetura geral da implementação proposta neste trabalho é apresentada na Figura 2. As variáveis e constantes do projeto estão em ponto fixo e para cada j-ésima entrada, $y_j^0(n)$, utiliza-se 1 bit na parte inteira e 12 bits na parte fracionária, tendo em vista que as entradas estão normalizadas entre 0 e 1. Os valores dos pesos sinápticos estão guardados em memórias ROM (*Read-only Memory*) dentro de cada neurônio de cada camada da rede. Para os pesos dos neurônios das camadas ocultas, um e dois, bem como para os pesos do $bias$ de todas as camadas, são utilizados 5 bits na parte inteira (utilizando um para sinal) e 12 bits na parte fracionária. E para os pesos dos neurônios da camada de saída, são utilizados 7 bits na parte inteira (utilizando um para sinal) e 12 bits na parte fracionária.

Para a implementação da proposta foi utilizada a técnica de matriz sistólica (*systolic array*) [14], que funciona como uma abordagem intermediária entre a metodologia completamente paralela e a completamente serial. Esta técnica permite que os dados sejam recebidos de forma serial e os elementos de processamento (*Processing Elements* - PEs) executem suas operações de forma paralela [14].

A rede implementada neste trabalho utilizou duas camadas ocultas, consistindo na arquitetura 784-100-50-10. No entanto, para acrescentar mais camadas ocultas à rede é necessário apenas replicar o bloco intermediário presente na Figura 2, que representa a segunda camada oculta da rede, e realizar pequenas adaptações em cada camada para a quantidade de neurônios desejada.

A. Camadas do SSAE

A arquitetura da k-ésima camada oculta, chamada de *autoencoder* k na Figura 2, é apresentada na Figura 3 e ela implementa a técnica de matriz sistólica. Cada neurônio da k-ésima camada é representado por um i-ésimo PE_i^k e a quantidade de PEs é definida pelo valor de V^k, em que $V^1 = M$, $V^2 = N$ e $V^3 = H$. Dessa forma, cada i-ésimo PE_i^k implementa a Equação 1. Os valores computados em cada i-ésimo PE_i^k da k-ésima camada oculta passam por uma função de ativação (ver Equação 2), chamada aqui de FA^k, e são utilizados como entrada da camada seguinte. Já os valores computados nos PEs da camada de saída passam pela função de ativação definida na Equação 3, para gerar a saída da rede. O sinal sel corresponde ao seletor do multiplexador, Mux^k, utilizado para selecionar as saídas de cada i-ésimo PE_i^k para a entrada da função de ativação da k-ésima camada, FA^k.

Através da matriz sistólica, os valores das entradas fluem entre os PEs de modo que cada PE inicie suas operações no instante seguinte ao início das operações no PE que o antecede, fazendo com que estes módulos passem a operar de forma paralela. Vale ressaltar que a partir da segunda camada oculta, ou seja, quando $k > 1$, o $bias$, b, também é inserido como entrada da k-ésima camada, conforme apresentado na Figura 2. Apenas na primeira camada oculta este valor é inserido juntamente com as entradas da rede. Já os valores dos pesos sinápticos de cada i-ésimo PE_i^k encontram-se guardados em LUTs, através de memórias ROM. Isto é possível em decorrência do treinamento da rede ser feito previamente e não existir a necessidade de alterar os pesos, uma vez que a rede já tiver sido treinada para um problema específico.

1) Unidades de Processamento (PEs): A arquitetura dos PEs da primeira camada oculta difere da arquitetura dos PEs das demais camadas da rede. A arquitetura de cada i-ésimo PE_i^1, é detalhada na Figura 4 e é formada por um multiplicador, um somador, quatro registradores, R, e uma LUT para guardar os pesos do i-ésimo PE_i^1, chamada de W_i^1. Já a arquitetura de cada i-ésimo PE_i^k, para $k > 1$, é detalhada na Figura 5 e é formada por dois multiplicadores, dois somadores, seis registradores, R, uma LUT para guardar os pesos do i-ésimo PE_i^k, chamada de W_i^k, e uma constante para o peso do $bias$, wb_i^k. Cada i-ésimo PE_i^k gera uma saída após Z amostras, onde para a primeira camada oculta, $Z = P$ e para as demais, o valor da variável Z deverá ser maior ou igual

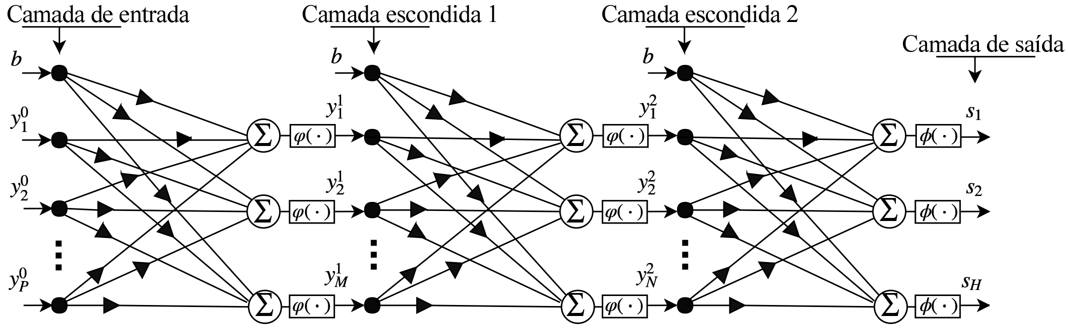

Figura 1. Arquitetura do *Stacked Sparse Autoencoder* proposto.

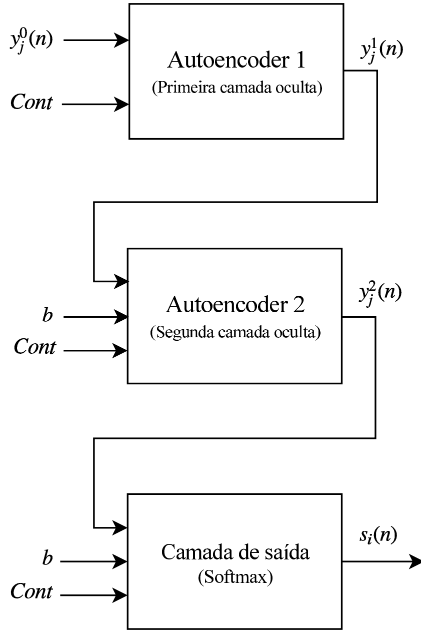

Figura 2. Arquitetura geral do projeto.

a quantidade de entradas ($Z \geq P$) e múltiplo da quantidade de neurônios da primeira camada oculta (mod(H, M) = 0). As LUTs de cada PE_i^k, W_i^k, utilizam uma memória ROM com uma profundidade de $L = Z$, armazenando palavras de 17 bits nas camadas ocultas, e 19 bits na camada de saída. Além disso, as constantes utilizadas para o peso do $bias$ de cada i-ésimo neurônio, wb_i^k, foram configuradas com 5 bits na parte inteira (utilizando um para sinal) e 12 bits na parte fracionária.

2) Funções de Ativação (FAs): A implementação de cada k-ésima função de ativação, FA^k, se deu com a utilização da técnica de *Lookup Table* (LUT), que permite aproximar funções por meio de uma tabela de L valores. Para implementar a função de ativação de cada camada oculta (Equação 2) foi utilizada uma memória ROM com profundidade de 8 bits, em que $L = 2^8$, armazenando palavras de 13 bits. Já para a implementação da função de ativação da camada de saída, a *softmax*, definida pela Equação 3, foi utilizada uma LUT para fazer a aproximação da função exponencial, e só adiante, computar a divisão da Equação 3. A LUT da camada de saída, FA^3, foi configurada com uma profundidade de $L = 2^{16}$, armazenando palavras de 57 bits.

B. Tempo de Processamento

No circuito implementado, o tempo de execução do PE, t_{PE}, corresponde ao tempo do caminho crítico do sistema.

O circuito implementado possui um atraso inicial que pode ser expresso como $d = (Q \times K + D) \times t_{PE}$, no qual Q é um número maior ou igual a quantidade de entradas da rede ($Q \geq P$) e múltiplo da quantidade de neurônios da primeira camada oculta (mod$(Q, M) = 0$), K é a quantidade de camadas, com neurônios, da rede, D é o atraso, em número de amostras, ocasionado pelas funções de ativação das camadas ocultas e de saída, e t_{PE} é o tempo do PE. Já o *throughput* (th_{ff}) da rede, em *frames* por segundo (FPS), pode ser expresso como $th_{ff} = \frac{1}{Q \times t_{PE}}$.

É importante destacar que o *throughput* da implementação independe da quantidade de camadas da rede. Sendo dependente apenas da quantidade de entradas e de neurônios utilizados na primeira camada oculta. A quantidade de camadas com neurônios, K, impactará apenas no delay inicial do sistema.

A partir da equação anterior, o tempo de execução do SSAE proposto, que consiste no tempo de *feedfoward* (t_{ff}) do SSAE, após o atraso inicial de d segundos, pode ser definido como $t_{ff} = Q \times t_{PE} = \frac{1}{th_{ff}}$, sendo assim, em cada t_{ff} é possível obter a saída de todos os H neurônios da última camada, ou seja, a saída da rede referente a uma determinada entrada.

IV. RESULTADOS

Com a finalidade de validar a implementação proposta neste artigo, utilizou-se para os experimentos, um banco de imagens de dígitos manuscritos, chamado de MNIST, que contém 60.000 imagens para o conjunto de treinamento e 10.000 imagens para o conjunto de testes, disponível em [15]. Cada imagem possui 28×28 pixels, totalizando 784 entradas para o SSAE. Os experimentos foram realizados com as 1.000 primeiras imagens do conjunto de testes do MNIST. O treinamento da rede foi realizado previamente na plataforma de simulação Matlab/Simulink [16] (*License number* 1080073)

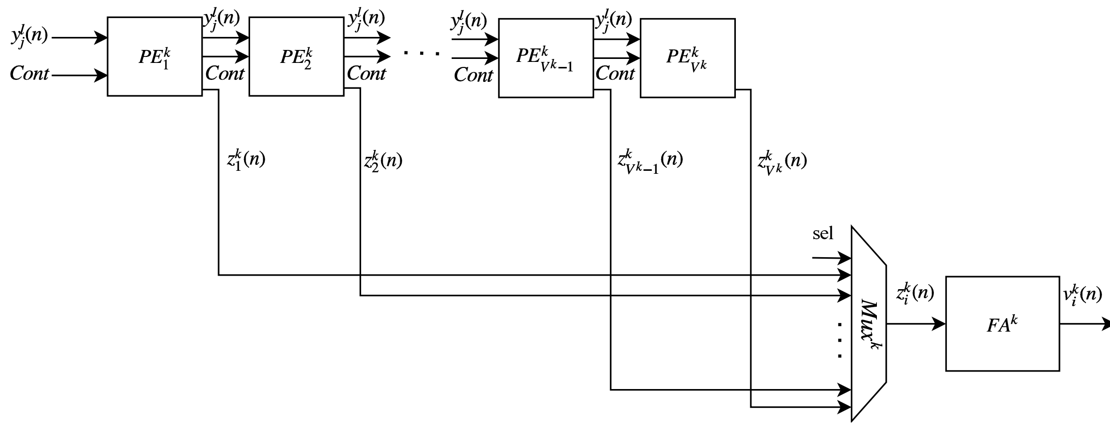

Figura 3. Arquitetura da k-ésima camada oculta do SSAE.

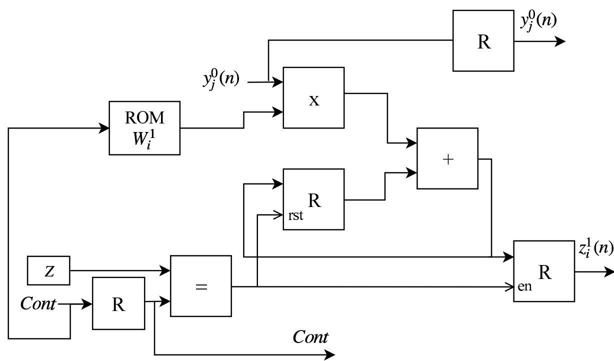

Figura 4. Arquitetura do i-ésimo PE_i^1.

e a FPGA alvo deste trabalho foi uma Virtex 6 xc6vlx240t-1ff1156.

Para a validação do hardware foi realizada uma comparação entre os resultados obtidos pela implementação na plataforma Matlab/Simulink e os resultados obtidos pela implementação em hardware. Sendo assim, foi calculado o erro quadrático médio (*Mean Square Error* - MSE) entre a saída da implementação em Matlab, $s_i^{ref}(n)$, e a saída da implementação em hardware. O cálculo do MSE, para este experimento, é definido como $MSE = \frac{1}{H \times 1.000} \sum_{i=1}^{H} \sum_{n=1}^{1000} \left(s_i^{ref}(n) - s_i(n) \right)^2$.

Com isso, verificou-se que o MSE entre os resultados da implementação em Matlab, que utiliza ponto flutuante (64 bits), e da implementação em hardware, em ponto fixo, foi de apenas $1,2 \times 10^{-3}$, que é um resultado bastante aceitável, já que os pesos associados a implementação em hardware utilizam apenas 12 bits na parte fracionária. Além disso, a pequena quantidade de bits não impediu a classificação correta dos dados, já que a implementação em hardware aqui proposta obteve a mesma porcentagem de acerto da implementação em Matlab, cerca de $91,4\%$ das 1.000 imagens do conjunto de testes do MNIST. Este resultado é bastante relevante pois indica que não é necessária uma alta resolução em bits (como 64 bits, por exemplo) para alcançar resultados significativos, mostrando que ao utilizar apenas 12 bits na parte fracionária, em ponto fixo, é possível garantir a redução na ocupação de área de hardware, além de promover o aumento do *throughput* [6], [10].

Após a validação da implementação em hardware com o Matlab, foi realizada a síntese para obter o relatório de ocupação dos recursos da FPGA. Com isso verificou-se a viabilidade da implementação proposta neste artigo, tendo em vista que foram ocupados apenas 3% (11.086) dos registradores e 10% (15.412) das células lógicas da FPGA alvo, mostrando que ainda existe espaço suficiente para adição de novas camadas e neurônios. Os elementos mais utilizados foram os multiplicadores, cerca de 28% (220), visto que cada PE da primeira camada oculta consome um, e os PEs das demais camadas da rede consomem dois, devido ao *bias*. Apesar disso, verifica-se a possibilidade de aumentar significativamente o número de camadas e neurônios do SSAE.

Com relação ao tempo de processamento do circuito implementado. O tempo de execução do PE obtido foi de apenas $47ns$, que também corresponde ao valor do tempo crítico do sistema. Além disso, observou-se que é possível obter a saída da rede, que neste trabalho corresponde a classificação de uma imagem, a cada $0,03ms$, após o atraso inicial de apenas $0,1ms$. Com isso, foi possível alcançar um *throughput* de $26,5$ KFPS, ou seja, 26.500 imagens classificadas por segundo, que é um valor bastante expressivo. Estes resultados indicam a aplicabilidade da DNN apresentada neste artigo à problemas de dados massivos.

Após a obtenção dos resultados apresentados, foi efetuada uma comparação da proposta aqui apresentada com um trabalho do estado da arte, apresentado em [5]. O trabalho relacionado também é aplicado a um problema de classificação, utiliza um banco de imagens para validação da arquitetura 3072-2000-750-10 implementada, e uma FPGA Altera Stratix V D5, equivalente em termos de processamento à FPGA aqui utilizada. Neste trabalho, foi alcançado um *throughput* de 45 FPS, enquanto a implementação aqui proposta alcançaria um *throughput* de 5.319 FPS, caso a arquitetura presente em [5] fosse implementada. Ou seja, a proposta aqui apresentada

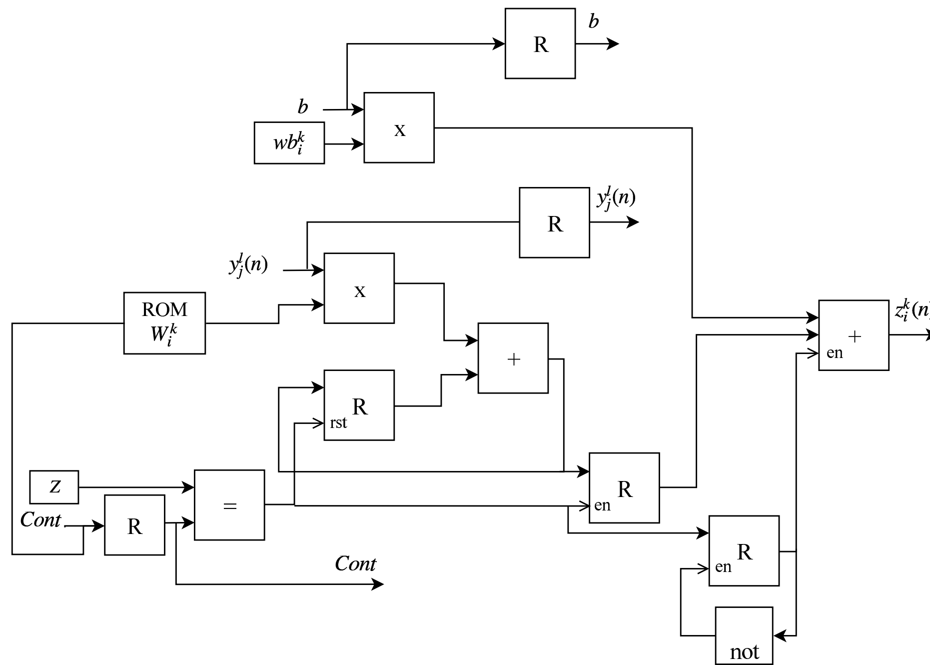

Figura 5. Arquitetura do i-ésimo PE_i^k, para $k > 1$, da k-ésima camada.

atingiu um *speedup* de 118x em comparação com o trabalho relacionado.

V. CONCLUSÕES

Este artigo apresentou uma proposta de implementação em hardware da técnica de *Deep Learning*, *Stacked Sparse Autoencoder*. Foi implementada a fase *feedfoward* da DNN, utilizando ponto fixo e design em RTL. Em toda a implementação foi aplicada a técnica de matriz sistólica, que permitiu utilizar muitos neurônios nas várias camadas da rede, além de possibilitar a obtenção dos resultados em baixo tempo de processamento. Todos os detalhes da implementação proposta foram apresentados, além dos resultados referentes à taxa de ocupação de hardware e ao tempo de processamento para uma FPGA Virtex 6 xc6vlx240t-1ff1156. Os *throughputs* elevados atingidos evidenciaram a possibilidade da utilização desta DNN em problemas de dados massivos. Além disso, uma comparação com um trabalho relacionado revelou a obtenção de um *speedup* significativo, reafirmando as contribuições da proposta aqui apresentada.

AGRADECIMENTOS

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo apoio financeiro.

REFERÊNCIAS

[1] P. Baldi, "Autoencoders, unsupervised learning, and deep architectures," in *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, 2012, pp. 37–49.

[2] L. Deng, D. Yu *et al.*, "Deep learning: methods and applications," *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014.

[3] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[4] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *J. Mach. Learn. Res.*, vol. 11, pp. 3371–3408, Dec. 2010.

[5] J. Maria, J. Amaro, G. Falcao, and L. A. Alexandre, "Stacked autoencoders using low-power accelerated architectures for object recognition in autonomous systems," *Neural Process. Lett.*, vol. 43, no. 2, pp. 445–458, Apr. 2016.

[6] A. C. D. de Souza and M. A. C. Fernandes, "Parallel fixed point implementation of a radial basis function network in an fpga," *Sensors*, vol. 14, no. 10, pp. 18 223–18 243, 2014.

[7] M. F. Torquato and M. A. Fernandes, "High-performance parallel implementation of genetic algorithm on fpga," *arXiv preprint arXiv:1806.11555*, 2018.

[8] M. Bettoni, G. Urgese, Y. Kobayashi, E. Macii, and A. Acquaviva, "A convolutional neural network fully implemented on fpga for embedded platforms," in *2017 New Generation of CAS (NGCAS)*, Sept 2017, pp. 49–52.

[9] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J. sun Seo, "Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler," *Integration*, vol. 62, pp. 14 – 23, 2018.

[10] J. Jiang, R. Hu, D. Wang, J. Xu, and Y. Dou, "Performance of the fixed-point autoencoder," vol. 23, pp. 77–82, 02 2016.

[11] Y. Jin and D. Kim, "Unsupervised feature learning by pre-route simulation of auto-encoder behavior model," *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 8, no. 5, pp. 706 – 710, 2014.

[12] A. Suzuki, T. Morie, and H. Tamukoh, "A shared synapse architecture for efficient fpga implementation of autoencoders," *PLOS ONE*, vol. 13, no. 3, pp. 1–22, 03 2018.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016.

[14] H. T. Kung and C. E. Leiserson, *Systolic arrays (for VLSI)*. Proceedings Symposium on Sparse Matrix Computations: I.S. Duff and C.G. Stewart. Eds., 1978.

[15] Y. LeCun, C. Cortes, and C. J. Burges, "Yann LeCun's Home Page," http://yann.lecun.com/exdb/mnist/, Jan 2018.

[16] The MathWorks, "Matlab/Simlink," https://www.mathworks.com/, Jan 2018.