PROPOSTA DE IMPLEMENTAÇÃO DE REDE NEURAL ARTIFICIAL EM HARDWARE PARA APRENDIZAGEM PROFUNDA

MARIA GRACIELLY F. COUTINHO*, MARCELO A. C. FERNANDES*

*Departamento de Engenharia da Computação e Automação - DCA
Centro de Tecnologia - CT
Universidade Federal do Rio Grande do Norte - UFRN
Natal, Brasil

Emails: gracielly@dca.ufrn.br, mfernandes@dca.ufrn.br

Abstract— The objective of this work is the hardware implementation of an artificial neural network (RNA) for the use of algorithms with deep learning. The hardware was developed on FPGA (field programmable gate array) and supports RNAs trained with the Sparse Autoencoder technique. In order to allow RNAs with many inputs and layers on the FPGA, the systolic array technique was used in all developed hardware. The details of the architecture designed were evidenced, as well as the occupation data on hardware and the processing time. The results show that the proposed implementation achieves high throughputs allowing the use of Deep Learning techniques in massive data problems.

Keywords— Deep learning, Autoencoder, Hardware, FPGA, Systolic array.

Resumo— Este trabalho tem como objetivo a implementação em hardware de uma rede neural artificial (RNA) para utilização de algoritmos com aprendizagem profunda (*Deep Learning*). O hardware proposto foi desenvolvido em FPGA (*field programmable gate array*) e suporta RNAs treinadas com a técnica *Sparse Autoencoder*. Para permitir RNAs com muitas entradas e camadas na FPGA, foi utilizada a técnica de matriz sistólica (*systolic array*) em todo hardware desenvolvido. Os detalhes da arquitetura desenvolvida são evidenciados, bem como, os dados de ocupação em hardware e o tempo de processamento. Resultados mostram que a implementação proposta consegue atingir *throughputs* elevados, permitindo a utilização de técnicas de *Deep Learning* em problemas de dados massivos.

Palavras-chave— Aprendizagem Profunda, FPGA, Autoencoder, Matriz Sistólica.

1 Introdução

Em meio aos algoritmos de Inteligência Artificial (IA), aplicáveis a diversos problemas, as Redes Neurais Artificiais (RNAs) têm ganhando destaque nos últimos anos, sobretudo, as RNAs com técnicas de aprendizagem profunda, chamadas pelo termo em inglês de *Deep Learning* (DL) (LeCun et al., 2015; Schmidhuber, 2015). Entre as várias técnicas de DL existentes, as RNAs com aprendizagem baseada em Autoencoders têm sido bastante utilizadas em várias aplicações relacionadas a problemas de predição e classificação (Baldi, 2012; Deng et al., 2014; Schmidhuber, 2015). Todavia, redes neurais baseadas em aprendizagem profunda possuem uma complexidade computacional elevada devido ao alto número de camadas escondidas, dificultando sua utilização em várias aplicações comerciais.

Por outro lado, novas alternativas voltadas para acelerar algoritmos complexos vêm sendo estudadas e entre elas as baseadas em computação reconfigurável vêm apresentando resultados bastante significativos. Os trabalhos desenvolvidos em Torquato and Fernandes (2016), Noronha and Fernandes (2016), da Silva et al. (2016) e de Souza and Fernandes (2014) mostram que a utilização de computação reconfigurável em FPGA (*field programmable gate array*) pode trazer ganhos de velocidade bastante significativos quando comparadas a implementações em processadores de uso geral utilizados em computadores de alto desempenho (*high performance computer* - HPC) e processadores gráficos (*Graphics Processing Unit* - GPU).

O trabalho apresentado em Maria et al. (2016) mostra uma das primeiras implementações, em FPGA, de uma RNA com DL utilizando *autoencoders*. Neste trabalho foi utilizada uma RNA com duas camadas escondidas, utilizando 3072 entradas, 2000 neurônios na primeira camada escondida, 750 na segunda e 10 neurônios na camada de saída (chamada de arquitetura 3072-2000-750-10) para classificar imagens do conjunto de dados chamado de CIFAR-10. Neste trabalho, a implementação foi realizada através de um *framework* de *high level synthesis* (HLS) em OpenCL e os resultados não foram significativos para a FPGA quando comparados a implementação em GPUs. Já o trabalho apresentado por Jiang et al. (2016) mostrou as vantagens da utilização de ponto fixo em implementações de *autoencoder*, onde foi possível obter a mesma acurácia da classificação da resolução em ponto flutuante, com apenas 4 bits na parte inteira. Para os experimentos foi utilizada a base de dados chamada de MNIST, com uma rede de arquitetura 784-400-10. Todavia, dados de ocupação dos recursos de hardware da FPGA e tempo de processamento não foram levantados neste artigo.

Propostas de implementação de redes tradicionais com *autoencoder* são apresentados em Jin and Kim (2014) e Suzuki et al. (2018). Em Jin

and Kim (2014) implementou-se uma arquitetura de *sparse autoencoder*, em *VerilogHDL*, utilizando para o pré-treinamento, o método de otimização L-BFGS, algoritmo popular para estimação de parâmetros em *machine learning*. Para a obtenção dos resultados foi utilizado o conjunto de dados de imagens naturais da cidade de Kyoto no Japão e a arquitetura 196-100-196. Finalmente, o trabalho apresentado em Suzuki et al. (2018) propôs a implementação de *autoencoders* com sinápses compartilhadas, em hardware reconfigurável. Tal arquitetura permite utilizar menor quantidade de recursos da FPGA. O trabalho também propõe uma estrutura com dois *autoencoders* encadeados, formando a arquitetura 4-2-1-2-4, entretanto, apenas com a finalidade de proporcionar a reconstrução da entrada na saída, como nos modelos convencionais. O circuito foi construído em *Register Transfer Level* (RTL), o que garantiu maior eficiência em comparação com as demais implementações de *autoencoder* em FPGA investigadas. A FPGA utilizada foi a Xilinx Virtex-6 xc6vlx240t.

Diferentemente do trabalho apresentado em Maria et al. (2016), a implementação proposta aqui foi desenvolvida em RTL, o que permite um maior controle e paralelização da implementação, tornando a implementação em FPGA bastante vantajosa, como também é apresentado em Suzuki et al. (2018). Por outro lado, os trabalhos apresentados em Jin and Kim (2014) e Suzuki et al. (2018) não implementaram uma RNA com várias camadas, e sim estruturas clássicas de *autoencoder*. Assim, diferentemente dos trabalhos apresentados na literatura, este artigo apresenta o desenvolvimento de uma RNA em hardware para técnicas de aprendizagem profunda baseada em *sparse autoencoder*. A implementação utilizou uma técnica de matriz sistólica com computação reconfigurável em RTL. Dados referentes a taxa de ocupação dos recursos de hardware e ao tempo de processamento serão detalhados e apresentados para uma FPGA Virtex 6 xc6vlx240t-1ff1156.

Nesta primeira seção foi apresentada uma introdução acerca do trabalho, colocando em evidência alguns trabalhos relacionados e do estado da arte. Na segunda seção será apresentada uma fundamentação teórica à respeito das técnicas de *autoencoder* para aprendizagem profunda. Na terceira seção, toda a arquitetura do projeto será descrita, bem como seus módulos e o tempo de processamento da implementação. Na quarta seção, serão apresentados os resultados obtidos pelo trabalho, que envolvem validação do hardware, taxa de ocupação do hardware e tempo de processamento da implementação, além disso, será apresentada uma comparação com um trabalho do estado da arte. Na última seção serão apresentadas as considerações finais referentes à proposta aqui apresentada.

2 Autoencoder

As técnicas de *Deep Learning* vêm ganhando grande destaque no âmbito da pesquisa mundial, esta estratégia consiste em uma derivação das redes *Multilayer Perceptron* (MLP), sendo possível, na abordagem de aprendizagem profunda, a utilização de diversas camadas ocultas, o que não era viável nas redes MLP convencionais. Os *autoencoders* constituem uma das classes das redes de aprendizagem profunda no qual sua existência antecede o termo *Deep Learning*. Primordialmente, esta técnica era principalmente utilizada para problemas envolvendo redução de dimensionalidade ou aprendizagem de características.

De modo geral, as redes neurais chamadas de *autoencoder* são treinadas para fornecer como saída, uma cópia dos dados da entrada. Sua arquitetura é composta por três camadas, sendo uma de entrada, uma oculta e outra de saída. Para que se obtenha uma reconstrução da entrada na saída, a camada de saída deve possuir o mesmo tamanho que a camada de entrada. As duas primeiras camadas, entrada e oculta, representam o *encoder* da rede, enquanto as camadas oculta e de saída, formam o *decoder* (Goodfellow et al., 2016).

Com o advento da *deep learning*, esta técnica passou a se mostrar muito útil no treinamento de redes com várias camadas. Nais quais cada camada consiste em um *autoencoder* treinado individualmente. Uma técnica de *autoencoder* bastante utilizada em problemas de classificação é o *sparse autoencoder*. *Sparse autoencodes* são capazes de aprender características através de aprendizagem não supervisionada. Neste trabalho, o treinamento das camadas ocultas foi efetuado previamente por meio desta abordagem. A arquitetura da rede MLP implementada é apresentada na Figura 1.

Foi implementada a fase *feedfoward* da rede, na qual a equação que define a saída do i-ésimo neurônio da k-ésima camada, $z_i^k(n)$, do n-ésimo instante, pode ser expressa como

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

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, em que $l = k - 1$, no n-ésimo instante, $wb_i^k(n)$ é o peso associado ao *bias* do i-ésimo neurônio da k-ésima camada no n-ésimo instante, b é o *bias*, que tem valor de 1, e U^l é o número de entradas da l-ésima camada, no qual $U^0 = P$, $U^1 = M$ e $U^2 = N$. A função de ativação utilizada nas camadas ocultas foi a sigmoide, desta forma a saída associada ao i-ésimo neurônio da k-ésima camada, no n-ésimo instante, $v_i^k(n)$, pode

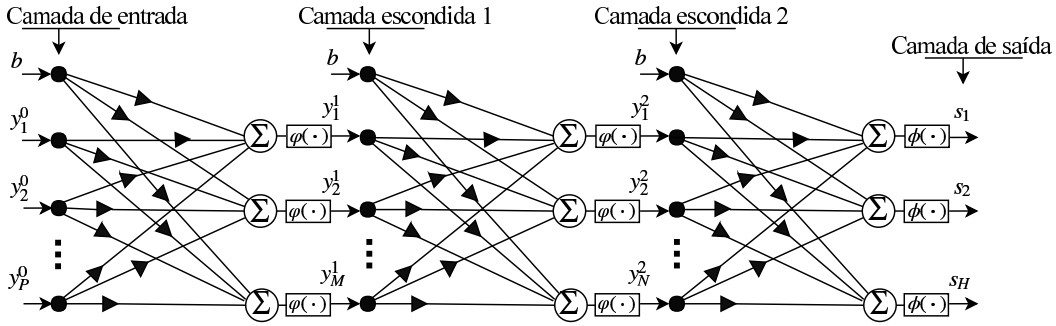

Figura 1: Arquitetura da rede MLP implementada.

ser caracterizada como

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

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

$$y_j^{l+1}(n) = v_i^k(n) \tag{3}$$

sendo $j = i$.

Na camada de saída, foi utilizada a função de ativação *softmax*, que vem sendo empregada em redes neurais de classificação como apresentado em Maria et al. (2016), e pode ser expressa como

$$s_i(n) = \frac{e^{z_i^K(n)}}{\sum_{h=1}^{H} e^{z_h^K(n)}} \tag{4}$$

onde $s_i(n)$ é a i-ésima saída da última camada, K, com H neurônios, no n-ésimo instante. Através desta função a saída da rede é forçada a representar a probabilidade dos dados pertencerem a uma determinada classe. Sendo assim, a quantidade de neurônios desta camada corresponde a quantidade de classes existentes no problema em questão.

3 Descrição do Projeto

A arquitetura geral da implementação proposta é apresentada na Figura 2 e ela tem como base a estrutura apresentada na Figura 1. 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. Já para os pesos sinápticos dos neurônios das camadas ocultas, um e dois, expressos como $w_{ij}^1(n)$ e $w_{ij}^2(n)$, bem como para os pesos do *bias* de todas as camadas, $wb_i^k(n)$, 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, $w_{ij}^3(n)$, são utilizados 7 bits na parte inteira (utilizando um para sinal) e 12 bits na parte fracionária.

Todo o projeto foi implementado aplicando a técnica de matriz sistólica (Kung and Leiserson, 1978), o que possibilitou obter resultados em

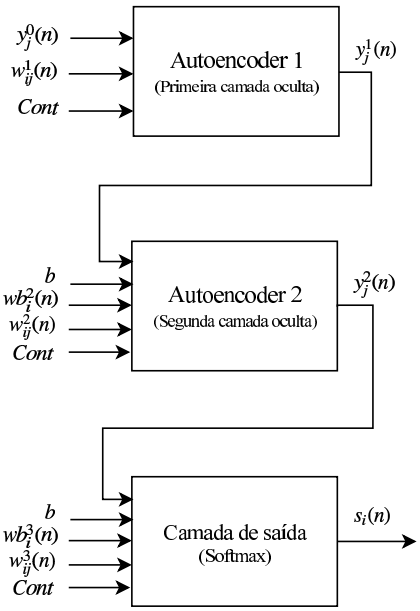

Figura 2: Arquitetura geral do projeto.

baixo tempo de processamento, mesmo utilizando muitas entradas e diversas camadas. Isso se deu em decorrência de nesta estratégia os dados fluírem entre os elementos de processamento (*Processing Elements* - PEs), permitindo que um PE inicie suas operações no instante seguinte ao início das operações no seu antecessor, de modo que estas unidades passem a operar de forma paralela. Pode-se dizer então, que a estrutura da matriz sistólica corresponde a um intermediário entre a abordagem *full parallel* e a abordagem *full serial*, já que nesta técnica os dados fluem de forma serial, e os PEs trabalham de forma paralela (Kung and Leiserson, 1978).

A arquitetura da rede implementada consistiu em 784-100-50-10. Todavia, é importante destacar que para acrescentar mais camadas à rede, se faz necessário apenas replicar a segunda camada oculta quantas vezes for necessário, e realizar pequenas adaptações para a quantidade de neurônios (PEs) desejada. Ou seja, replicar o bloco intermediário ilustrado na Figura 2.

3.1 Camadas da rede MLP proposta

A Figura 3 apresenta a arquitetura geral da k-ésima camada oculta, chamada de *autoencoder* k (ver Figura 2). Onde cada i-ésimo PE_i^k representa um neurônio e um contador em anel é utilizado para habilitar o recebimento dos pesos em cada i-ésimo PE_i^k, já que estes valores são transmitidos através de *streams* (fluxos) de pesos. A quantidade de PEs é determinada pelo valor de V^k ($V^1 = M$, $V^2 = N$ e $V^3 = H$). Basicamente cada i-ésimo PE_i^k implementa a Equação 1. Os valores resultantes das computações em cada i-ésimo PE_i^k da k-ésima camada passam por uma função de ativação, chamada aqui de FA^k (ver Equação 2), e são utilizados como entrada da camada seguinte (ver Equação 3). O sinal *sel* representa o seletor do multiplexador, Mux^k, utilizado para selecionar as saídas de cada PE_i^k para a entrada da função de ativação da k-ésima camada oculta, FA^k.

A partir da segunda camada oculta, ou seja, quando $k > 1$, o *bias*, b, e os pesos do *bias*, $wb_i^k(n)$, são inseridos como entrada da k-ésima camada, seguindo a estrutura apresentada na Figura 2. Apenas na primeira camada oculta, ou seja, quando $k = 1$, estes valores são inseridos em conjunto com as entradas da rede e com os pesos dos neurônios da camada, respectivamente.

Uma vantagem desta proposta é permitir a inserção dos pesos de cada camada de forma serial através de um único *stream* de pesos, na primeira camada oculta, e de dois *streams* de pesos, nas demais camadas da rede. Esta seria a menor quantidade de *streams* de pesos necessária para o funcionamento deste circuito. No entanto, esta arquitetura é facilmente escalonável, permitindo a paralelização dos *streams* de pesos de cada camada. A quantidade de *streams* de pesos por camada pode variar de um até a quantidade de neurônios da camada, V^1, na primeira camada oculta da rede, e de dois até $2 \times V^k$, nas demais camadas da rede, devido aos pesos do *bias*.

Vale destacar que possibilitar o recebimento dos pesos por *streams*, dispensa a necessidade de utilizar recursos de memória interna da FPGA para guardar os pesos da rede, além de viabilizar a utilização de um mesmo hardware para problemas distintos, já que os pesos treinados para um novo problema poderão ser inseridos na rede, bem como as entradas do problema em questão, sem a necessidade de reconfigurar o hardware.

3.1.1 Unidades de Processamento (PEs)

A arquitetura de cada i-ésimo PE_i^k da k-ésima camada é detalhada na Figura 4 e é formada por um multiplicador, um somador e quatro registradores, R. Cada i-ésimo PE_i^k gera uma saída após Z amostras, onde para a primeira camada oculta, $Z = P$ e para as demais, a variável Z deverá ser maior ou igual a quantidade de entradas ($Z \geq P$) e múltiplo da quantidade de neurônios da primeira camada oculta ($\text{mod}(H, M) = 0$). No entanto, a partir da segunda camada oculta, a multiplicação entre o *bias* e o peso do *bias* ocorre em separado da multiplicação entre as entradas e os pesos da camada, já que estes valores são recebidos à parte para $k > 1$, como mostra a Figura 2.

3.1.2 Funções de Ativação (FAs)

Para implementar cada k-ésima função de ativação, FA^k, das camadas ocultas (Equação 2) foi utilizada a técnica de *Lookup Table* (LUT) mostrada na Figura 5, que permite a aproximação da função por uma tabela de L valores. Para o desenvolvimento da LUT, utilizou-se uma memória ROM (*Read-only Memory*) com profundidade de 8 bits, onde $L = 2^8$, armazenando palavras de 13 bits. Um ponto importante associado a esta estratégia de implementação é a utilização de apenas uma única função de ativação, ou LUT, por camada oculta. Esta característica reduz de forma significativa o espaço ocupado na FPGA (ver de Souza and Fernandes (2014)).

Já para a implementação da função *softmax*, detalhada pela Equação 4, da camada de saída, foi necessário a utilização de uma LUT para fazer a aproximação da função exponencial, e só posteriormente, computar a divisão (ver Equação 4). A LUT da camada de saída, FA^3, foi configurada com uma profundidade de $L = 2^{16}$, armazenando palavras de 57 bits.

3.2 Tempo de Processamento

Um contador em anel em cada k-ésima camada, mostrado na Figura 3, é utilizado para habilitar o recebimento dos pesos em cada i-ésimo PE_i^k da k-ésima camada. Desta forma, sua operação deve ser V^k vezes mais rápida do que a operação de cada PE_i^k. Com a finalidade de manter o sincronismo da implementação, considera-se $V^k > V^{k+1}$. Com isso, o *clock* do circuito, em segundos, é determinado pelo *clock* do contador em anel da primeira camada oculta. Sendo assim, o tempo de amostragem do circuito, t_s, pode ser definido como

$$t_s = (V^1 \times clock). \tag{5}$$

A arquitetura proposta possui um atraso inicial que pode ser expresso como

$$d = (Q \times K + D) \times t_s \tag{6}$$

onde Q é um número que deve atender a dois requisitos: ser maior ou igual a quantidade de entradas da rede ($Q \geq P$) e ser múltiplo da quantidade de neurônios da primeira camada oculta ($\text{mod}(Q, M) = 0$), K corresponde a quantidade de camadas da rede, D é o atraso, em número

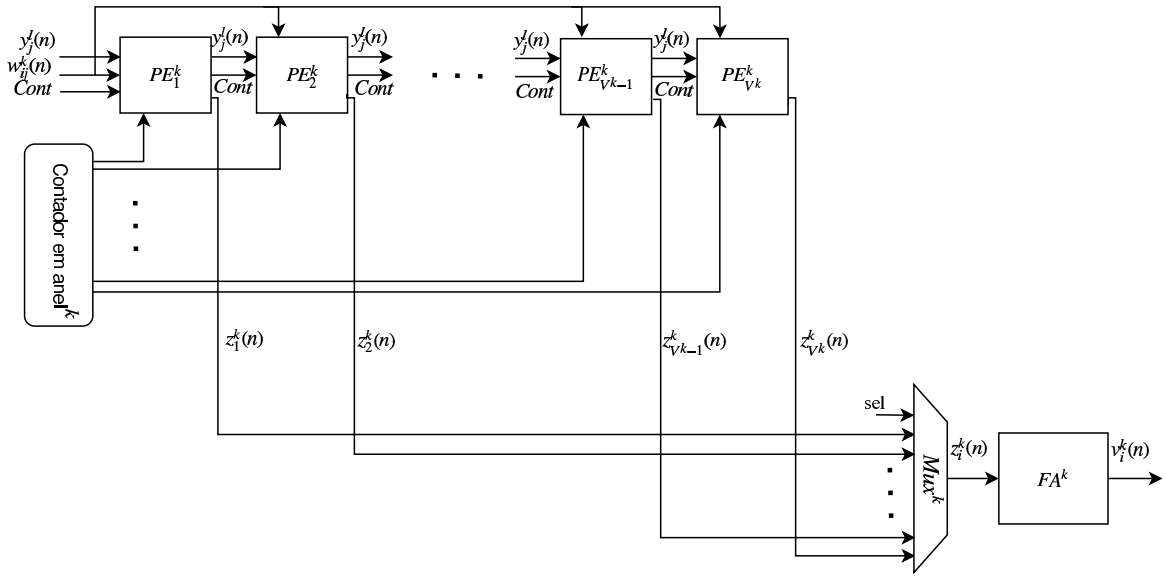

Figura 3: Arquitetura da k-ésima camada oculta da MLP.

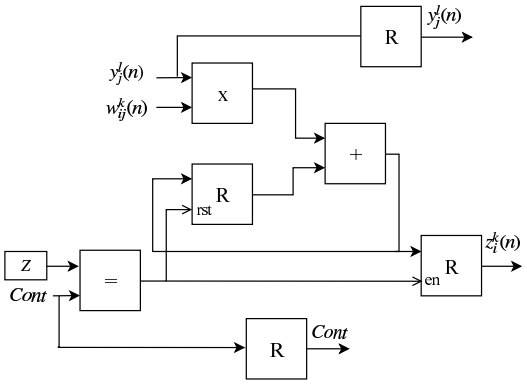

Figura 4: Arquitetura do i-ésimo PE_i^k da k-ésima camada.

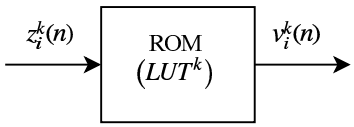

Figura 5: Arquitetura da k-ésima função de ativação, FA^k, associada as camadas ocultas.

de amostras, provocado pelas funções de ativação das camadas ocultas e de saída, e t_s é o tempo de amostragem. Já o *throughput* (th_{ff}) pode ser expresso como

$$th_{ff} = \frac{1}{Q \times t_s}. \tag{7}$$

Com base nas equações 6 e 7, o tempo de execução da RNA proposta (chamado aqui de tempo de *feedfoward* (t_{ff})), após o atraso inicial de d segundos, pode ser expresso por

$$t_{ff} = Q \times t_s = \frac{1}{th_{ff}} \tag{8}$$

ou seja, em cada t_{ff} é gerada a saída de todos os H neurônios da última camada.

4 Resultados

Para validar a implementação em hardware proposta, foram realizados testes com o banco de imagens de dígitos manuscritos, chamado de MNIST, disponível em LeCun et al. (2018). Nele estão contidas 60.000 imagens para o conjunto treinamento e 10.000 imagens para o conjunto de testes. Para os experimentos foram utilizadas as 1.000 primeiras imagens do conjunto de testes. Cada imagem possui 28×28 pixels, o equivalente a 784 entradas para a RNA. O treinamento da rede foi realizado previamente na plataforma de simulação Matlab/Simulink (The MathWorks, 2018) (*License number* 1080073) e a FPGA alvo deste trabalho foi uma Virtex 6 xc6vlx240t-1ff1156.

Uma comparação entre os resultados obtidos pela implementação na plataforma Matlab/Simulink e os resultados da implementação em hardware foi utilizada para validação. Para isto foi calculado o erro quadrático médio (*Mean Square Error* - MSE) entre a saída da implementação em hardware e a saída da implementação em Matlab, $s_i^{ref}(n)$. Para o experimento, o cálculo do MSE é expresso como

$$MSE = \frac{1}{H \times 1.000} \sum_{i=1}^{H} \sum_{n=1}^{1000} \left(s_i^{ref}(n) - s_i(n) \right)^2. \tag{9}$$

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 $1,2 \times 10^{-3}$. Este resultado é esperado dado que estão sendo utilizados apenas 12 bits na parte fracionária dos pesos as-

sociados a implementação da RNA em hardware. Todavia, apesar da pequena quantidade de bits, a implementação aqui proposta atingiu a mesma porcentagem de acerto da implementação em Matlab, ou seja, $91,4\%$ das 1.000 imagens do MNIST. Este resultado é bastante significativo pois mostra que não é necessária uma alta resolução em bits (64 bits, por exemplo) para atingir resultados significativos. Por outro lado, a utilização de uma quantidade mediana, 12 bits na parte fracionária, em ponto-fixo pode reduzir bastante a ocupação no hardware, bem como aumentar significativamente o *throughput* (Jiang et al., 2016; de Souza and Fernandes, 2014).

Além da validação da implementação em hardware com o Matlab, foi também realizada a síntese para gerar o relatório de ocupação da área do hardware. A Tabela 1 apresenta os dados de síntese relacionados a ocupação de área da proposta deste artigo na FPGA. A primeira coluna apresenta a quantidade de multiplicadores utilizados. Os multiplicadores estão presentes nos PEs de cada k-ésima camada. A segunda coluna exibe a quantidade de registradores utilizados em todo o circuito, e a terceira coluna, a quantidade de células lógicas.

Tabela 1: Síntese - Taxa de ocupação

Multiplicadores	Registradores	Células Lógicas
220 (28%)	28.637 (9%)	24.147 (16%)

Os dados apresentados na Tabela 1 evidenciam a viabilidade da implementação de RNA proposta neste trabalho. Verifica-se que foram ocupados apenas 9% dos registradores e 16% das células lógicas da FPGA alvo, mostrando que ainda existe bastante espaço para adição de novas camadas e neurônios. Um dos elementos que foram mais utilizados foram os multiplicadores, em torno de 28%, dado que cada PE da primeira camada consome um, e os PEs das outras camadas consomem dois, devido ao *bias*. Apesar disso, constata-se que ainda é possível aumentar bastante o número de camadas e neurônios da rede.

A Tabela 2 apresenta informações à respeito do tempo de processamento do circuito proposto. A primeira coluna expõe o *clock* em que opera o circuito. A segunda coluna exibe o atraso inicial da arquitetura, definido pela Equação 6. Na terceira coluna é mostrado o tempo de execução do *feedfoward* da rede, expresso pela Equação 8, após o atraso inicial (Equação 6), e a última coluna apresenta o *throughput* da rede, determinado pela Equação 7, que neste trabalho consiste na quantidade de imagens classificadas por segundo.

Os dados apresentados na Tabela 2 são bastante significativos. Após a síntese obteve-se um *clock* de $10ns$. Além disso, após o atraso inicial de apenas $2,4ms$ é possível obter as saídas de todos os neurônios da última camada, que neste caso é a classificação de uma imagem, a cada $0,8ms$. O que permite alcançar um *throughput* de pelo menos 1.250 imagens classificadas por segundo. Este é o valor do *throughput* mínimo atingido por esta proposta, ou seja, seu *lower bound*, já que a implementação utilizou a menor quantidade de *streams* de pesos necessária, o que implica no aumento do tempo de amostragem do circuito, e consequentemente, no aumento do delay (d) e do tempo de *feedfoward* (t_{ff}) da implementação. Ao paralelizar estes *streams* é possível atingir resultados ainda mais expressivos. De todo modo, os resultados apresentados revelam a possibilidade da utilização desta técnica de *Deep Learning* em problemas de dados massivos.

Todavia, não faz sentido comparar a proposta aqui apresentada com os trabalhos relacionados citados anteriormente, já que nestes trabalhos não se deixa claro permitir a utilização de *streams* para as entradas dos pesos na rede. Além disso, alguns trabalhos não apresentam informações sobre o tempo de processamento e outros não implementaram arquiteturas de redes para aprendizagem profunda. Porém, se esta comparação for realizada, serão obtidos como resultados para a proposta com a quantidade mínima de *streams* por camada, os valores apresentados na Tabela 3. A primeira coluna desta tabela apresenta a referência comparada e na segunda coluna é informado o dispositivo alvo do trabalho. O trabalho relacionado também é aplicado a um problema de classificação e faz uso de uma FPGA equivalente em termos de processamento. Na terceira coluna é mostrada a arquitetura da rede implementada. Na quarta coluna é apresentado o *throughput* da referência comparada, e na última coluna é exibido o *throughput* obtido pela implementação da proposta aqui apresentada.

Os dados apresentados na Tabela 3 indicam que o menor valor de *throughput* obtido pela proposta aqui apresentada para a arquitetura do trabalho relacionado foi de 12,5 FPS. Apesar do valor do *throughput* obtido parecer inferior ao *throughput* de referência, este valor trata-se do limite inferior alcançado (*lower bound*), devido a utilização da quantidade mínima de *streams* de pesos por camada. De modo que, ao paralelizar os *streams* de pesos das camadas ocultas em apenas 4 vezes, já seria possível ultrapassar o valor do *throughput* atingido pelo trabalho relacionado, tendo em vista que $12,5 \times 4 = 50$ FPS, com a vantagem de permitir facilmente a alteração dos valores dos pesos sinápticos da rede.

5 Conclusões

Este trabalho teve como objetivo a implementação em hardware de uma rede neural artificial

Tabela 2: Síntese - Associação ao tempo de processamento

Clock (ns)	Delay (ms)	*Feedfoward* (ms)	*Throughput* (KFPS)
10	$\leq 2,4$	$\leq 0,8$	$\geq 1,25$

Tabela 3: Comparação da proposta com um trabalho relacionado

Referência	Dispositivo	Arquitetura da rede	*Throughput* referência	*Throughput* obtido
(Maria et al., 2016)	Altera Stratix V D5	3072-2000-750-10	45 FPS	$\geq 12,5$ FPS

(RNA) para utilização de algoritmos com aprendizagem profunda (*Deep Learning*). O hardware foi desenvolvido em RTL, utilizando ponto fixo, e implementou a técnica de matriz sistólica, permitindo a utilização de muitas entradas e camadas na FPGA. Todos os detalhes da implementação foram apresentados bem como resultados relativos a síntese para ocupação e tempo de processamento. Os resultados mostraram que a implementação proposta consegue atingir *throughputs* elevados, permitindo a utilização de técnicas de *Deep Learning* em problemas de dados massivos.

Agradecimentos

À CAPES, pelo apoio financeiro.

Referências

Baldi, P. (2012). Autoencoders, unsupervised learning, and deep architectures, *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, pp. 37–49.

da Silva, L., Torquato, M. and Fernandes, M. A. C. (2016). Proposta de arquitetura em hardware para fpga da técnica q-learning de aprendizagem por reforço, *Encontro Nacional de Inteligência Artificial e Computacional - ENIAC 2016*, Recife, PE.

de Souza, A. C. D. and Fernandes, M. A. C. (2014). Parallel fixed point implementation of a radial basis function network in an fpga, *Sensors* **14**(10): 18223–18243.

Deng, L., Yu, D. et al. (2014). Deep learning: methods and applications, *Foundations and Trends® in Signal Processing* **7**(3–4): 197–387.

Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*, MIT press.

Jiang, J., Hu, R., Wang, D., Xu, J. and Dou, Y. (2016). Performance of the fixed-point autoencoder, **23**: 77–82.

Jin, Y. and Kim, D. (2014). Unsupervised feature learning by pre-route simulation of auto-encoder behavior model, *International Journal of Computer, Electrical, Automation, Control and Information Engineering* **8**(5): 706 – 710.

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

LeCun, Y., Bengio, Y. and Hinton, G. (2015). Deep learning, *nature* **521**(7553): 436.

LeCun, Y., Cortes, C. and Burges, C. J. (2018). Yann LeCun's Home Page, http://yann.lecun.com/exdb/mnist/.

Maria, J., Amaro, J., Falcao, G. and Alexandre, L. A. (2016). Stacked autoencoders using low-power accelerated architectures for object recognition in autonomous systems, *Neural Process. Lett.* **43**(2): 445–458.

Noronha, D. and Fernandes, M. A. C. (2016). Implementação em fpga de máquina de vetores de suporte (svm) para classificação e regressão, *Encontro Nacional de Inteligência Artificial e Computacional - ENIAC 2016*, Recife, PE.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview, *Neural networks* **61**: 85–117.

Suzuki, A., Morie, T. and Tamukoh, H. (2018). A shared synapse architecture for efficient fpga implementation of autoencoders, *PLOS ONE* **13**(3): 1–22.

The MathWorks (2018). Matlab/Simlink, https://www.mathworks.com/.

Torquato, M. F. and Fernandes, M. A. C. (2016). Proposta de implementação paralela de algoritmo genético em fpga, *XXI Congresso Brasileiro de Automática, CBA 2016*.