Proposta de Implementação de Tempo Real de Redes Neurais MLP em Microcontroladores de 8-bits

Caio B. Vilar, Deângela Neves, and Marcelo A. C. Fernandes

Departamento de Computação e Automação (DCA)
Universidade Federal do Rio Grande do Norte (UFRN)
caio.b.vilar@gmail.com, deangelacgn@gmail.com,
mfernandes@dca.ufrn.br

Resumo Este artigo tem como objetivo apresentar o desenvolvimento de uma estratégia de implementação de Redes Neurais Artificiais (RNA) do tipo Perceptron de Múltiplas Camadas, *Multilayer Perceptron* (MLP), treinada com algoritmo de *Backpropagation* (BP), em microcontroladores (μC) de 8-bits. Esta estratégia busca viabilizar a utilização de RNA-MLP-BP em Sistemas Embarcados (SE) de baixo consumo e custo como μC de 8-bits. Todos os detalhes da estratégia de implementação são apresentados, bem como resultados para várias configurações da RNA utilizada. Os resultados mostram que processadores de baixo consumo e velocidade podem trabalhar com MLP-BP em várias aplicações que não precisam de um tempo de resposta curto.

Keywords: Redes Neurais, Microcontroladores, Perceptron de Múltiplas Camadas, MLP, 8-bits

1 Introdução

A plataforma de hardware chamada de microcontrolador (μC) tem sido aplicada em várias áreas como automação industrial, controle, equipamentos de medição, eletrônica de consumo e outras. É possível até afirmar que existe uma crescente demanda da utilização dessas plataformas, principalmente em setores emergentes como Internet das Coisas (*Internet of Things* - IoT), *Smart Grid*, *Machine to Machine* (M2M) e outros. Os μCs podem ser classificados como plataformas de hardware programáveis que permitem a utilização de Sistemas Embarcados (SEs) para aplicações específicas. É importante entender que os μCs são na maioria formados por um microprocessador (μP) de uso geral (*General Purpose Processor* - GPP) de 8, 16 ou 32 bits acoplado a vários periféricos de hardware internos como memória RAM (*Random Access Memory*), memória flash, contadores, geradores de sinal, protocolos de comunicação, conversores analógico digital e outros. Na maioria dos produtos, hoje disponíveis, os μC são de 8-bits com um baixo poder de processamento e armazenamento. Todavia, esses dispositivos possuem baixo consumo e custo quando comparado a outras plataformas de hardware o que o viabiliza em várias aplicações [19,8].

O uso de sistemas inteligentes com Redes Neurais Artificiais (RNA) embarcados em hardware para aplicações de tempo real tem sido alvo de investigação por vários grupos [11,13,17,16,12]. Grande parte dos estudos são motivados em face da demanda crescente da utilização de técnicas de Inteligência Artificial em IoT, M2M e outros. Um dos grandes problemas associados à complexidade computacional referente à RNA, principalmente à rede do tipo Perceptron de Múltiplas Camadas (*Multilayer Perceptron* - MLP), que possui várias operações de multiplicação e funções não lineares [11,13,17]. Além do processamento direto entre entrada e saída, a rede MLP possui também um algoritmo de treinamento associado para encontrar os ganhos ótimos da rede [7]. Caso o processo de treinamento também seja realizado em tempo real (treinamento *on line*) a complexidade será aumentará bastante, elevando o tempo de processamento e a capacidade de armazenamento necessária, do hardware utilizado [11,13,17]. Existem vários algoritmos de treinamento e neste trabalho será utilizado o algoritmo da retropropagação (*Backpropagation* - BP) [7].

Na literatura existem vários trabalhos que focam na utilização da RNA do tipo MLP-BP para aplicações em tempo real com μCs. Dentre os vários trabalhos encontrados na literatura destacam-se os apresentados em [5,15,4,14,3,20]. Nos trabalhos apresentados em [5,15,4] são propostas aplicações nos quais o MLP-BP é implementado em microcontroladores, apesar de apresentarem bons resultados nenhum estudo de ocupação e tempo de processamento é feito para vários parâmetros da rede MLP. Já nos artigos [14,3,20] são apresentados alguns resultados relativos a tempo de processamento todavia não são realizados testes de tempo em função do número de neurônios e também os tempos associados a utilização ou não do algoritmo de treinamento em tempo real.

Assim, este trabalho tem como objetivo propor uma implementação da rede MLP treinada com o algoritmo BP em μC de 8 bits para proporcionar seu uso em várias aplicações. Além da proposta de implementação, foram analisados parâmetros como o tempo de processamento e a capacidade de armazenamento para vários parâmetros como o número de neurônios na camada escondida e a utilização ou não do algoritmo de treinamento em tempo real. Juntamente com os resultados de tempo de processamento e capacidade de armazenamento foram realizados testes de validação do MLP-BP em técnica de HIL (*Hardware-in-the-Loop*) [18]. Os resultados mostraram que é possível utilizar a rede MLP treinada com o BP em μCs de 8-bits em várias aplicações apresentadas na literatura, principalmente em aplicações de tempo real.

2 Proposta de Implementação

A Figura 2 detalha, em diagrama de blocos, os módulos que compõem a implementação do MLP-BP para μCs proposto neste artigo. A implementação tem como referência a proposta apresentada em [6] no qual, o MLP-BP é codificado de forma matricial simplificando e modularizando as operações de propagação direta (*feedforward*) e a operação de treinamento associada ao *backpropagation*.Como apresentado na Figura 2 o MLP-BP, aqui implementado, é formado por quatro

módulo principais chamados de *Input Random Permutation Module* (IRPM), *Feedforward Module* (FFM $-$ k), *Error Module* (EM) e *Backpropagation Module* (BPM $-$ k), nos quais a variável k representa a camada ($k = 0$ para a camada de entrada). A implementação apresentada aqui utiliza duas camadas, mas pode ser facilmente estendida para mais camadas e os módulos serão detalhados nas próximas subseções.

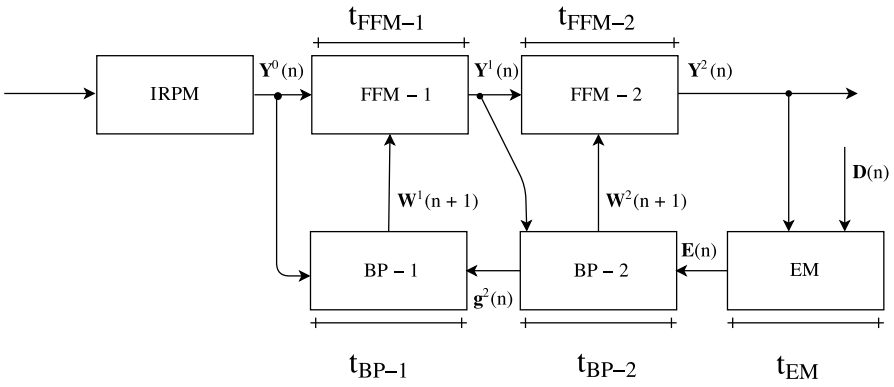


Figura 1. Diagrama de blocos detalhando os módulos da implementação do MLP-BP

2.1 Variáveis associadas

A implementação é formada basicamente por quatro variáveis principais que são passadas como referência entre os módulos. São elas a matriz de entrada, $\mathbf{Y}^0(n)$, expressa como

$$\mathbf{Y}^0(n) = \left[\mathbf{y}_1^0(n), \mathbf{y}_2^0(n), \ldots, \mathbf{y}_s^0(n), \ldots, \mathbf{y}_{\mathbb{N}}^0(n)\right] \tag{1}$$

na qual n representa o número da iteração, $\mathbb{N}$ representa o número de amostras de um dado conjunto de treinamento e $\mathbf{y}_s$ a s-ésima amostra expressa como

$$\mathbf{y}_s^0(n) = \left[y_{s1}^0(n), y_{s2}^0(n), \ldots, y_{sP}^0(n)\right]^T \tag{2}$$

onde, P é o número de entradas da MLP. A matriz de pesos da k-ésima camada, $\mathbf{W}^k(n)$, caracterizada como

$$\mathbf{W}^k(n) = \begin{bmatrix} w_{10}^k(n) & w_{11}^k(n) & \cdots & w_{10}^k(n) & \cdots & w_{1H^{k-1}}^k(n) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{j0}^k(n) & w_{j1}^k(n) & \cdots & w_{jh}^k(n) & \cdots & w_{jH^{k-1}}^k(n) \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{H^k0}^k(n) & w_{H^k1}^k(n) & \cdots & w_{H^kh}^{(y)}(n) & \cdots & w_{H^kH^{k-1}}^k(n) \end{bmatrix}, \tag{3}$$

onde H^k representa o número de neurônios da k-ésima camada e $w_{ij}^k(n)$ é o peso associado ao i-ésimo neurônio, da j-ésima entrada da k-ésima camada no n-ésimo instante. A matriz de saída, $\mathbf{Y}(n)$, expressa como

$$\mathbf{Y}^L(n) = \left[\mathbf{y}_1^L(n), \mathbf{y}_2^L(n), \ldots, \mathbf{y}_s^L(n), \ldots, \mathbf{y}_{\mathbb{N}}^L(n)\right], \tag{4}$$

onde $\mathbf{y}_s(n)$ é a saída associada a s-ésima amostra da entrada $\mathbf{x}_s(n)$, que é expressa como

$$\mathbf{y}_s^L(n) = \left[x_{s1}^L(n), x_{s2}^L(n), \ldots, x_{sM}^L(n)\right]^T \tag{5}$$

onde M é o número de neurônios de saída. Finalmente, a variável $\mathbf{D}(n)$, que representa a matriz de valores desejados da matriz de entrada, $\mathbf{X}(n)$, que pode ser expressa como

$$\mathbf{D}(n) = [\mathbf{d}_1(n), \mathbf{d}_2(n), \ldots, \mathbf{d}_s(n), \ldots, \mathbf{d}_{\mathbb{N}}(n)] \tag{6}$$

onde $\mathbf{d}_2(n)$ é o vetor de valores desejados referente à s-ésima amostra da entrada $\mathbf{x}_s(n)$, que é expresso como

$$\mathbf{d}_s(n) = [d_{s1}(n), d_{s2}(n), \ldots, d_{sM}(n)]^T. \tag{7}$$

Para uma rede de duas camadas, $\mathbf{W}^1(n)$ representa a matriz de pesos da camada escondida ($H^0 = P$) e $\mathbf{W}^2(n)$ a matriz de pesos da camada de saída ($H^2 = M$) no n-ésimo instante.

Além destas matrizes, outras são criadas com o objetivo de propagar a informação entre os módulos e realizar todas as operações necessárias do MLP-BP. É importante destacar que a proposta aqui implementada levou em consideração vários aspectos importantes relativos à otimização do tamanho do código (redução na ocupação da memória de programa), otimização do uso de variáveis (redução da ocupação de memória RAM) e otimização das operações para reduzir o processamento do BP. Como apresentado em [10,1,2], a otimização do tamanho do código e a otimização das variáveis contribui também para a redução do processamento.

2.2 *Feedforward Module* - (FFM − k)

Este módulo tem a função de realizar a operação de propagação direta da MLP-BP em cada k-ésima camada durante cada n-ésima iteração. As iterações do algoritmo podem trabalhar em modo *batch* apenas configurando $\mathbb{N} > 1$. Em cada k-ésimo FFM $- k$ (k-ésima camada) é realizada a seguinte expressão

$$\mathbf{Y}^k(n) = \varphi\left(\mathbf{W}^k(n)\,\mathbf{Y}^{k-1}(n)\right) \tag{8}$$

na qual $\mathbf{Y}^{k-1}$ é a saída da camada anterior, no qual, $\mathbf{Y}^0$, é a entrada da MLP e $\varphi(\cdot)$ é a função de ativação da k-ésima camada. Observa-se que as saídas associadas às camadas intermediárias também devem ser armazenadas durante a execução do algoritmo, para sua utilização na etapa de treinamento. O Algoritmo 1 descreve o pseudocódigo do módulo FFM $- k$ em mais detalhes onde a função prodMatrix() deve implementar o produto matricial entre $\mathbf{W}^k(n)$ e $\mathbf{Y}^{k-1}(n)$ e armazena a resposta em $\mathbf{Y}^k(n)$ e a função actFun() deve aplicar a função de ativação elemento a elemento da $\mathbf{Y}^k(n)$ e armazenar a resposta nela mesma.

Algoritmo 1 Algoritmo do módulo FFM $-$ k

1: **função** FFM $-$ k($\mathbf{W}^k(n)$,$\mathbf{Y}^{k-1}(n)$,$\mathbf{Y}^k(n)$)
2: prodMatrix $\left(\mathbf{W}^k(n), \mathbf{Y}^{k-1}(n), \mathbf{Y}^k(n)\right)$
3: actFun $\left(\mathbf{Y}^k(n)\right)$
4: **fim função**

2.3 *Error Module* - (EM $-$ k)

O módulo EM $-$ k é responsável por calcular o erro entre o valor de referência armazenado na matriz $\mathbf{D}(n)$ e saída do neurônio da última camada L, ou seja,

$$\mathbf{E}(n) = \mathbf{D}(n) - \mathbf{Y}^L(n) \tag{9}$$

O Algoritmo 2 descreve o pseudocódigo do módulo EM cuja função difMatrix () deve implementar a diferença elemento a elemento entre duas matrizes e armazenar a resposta na matriz $\mathbf{E}(n)$.

Algoritmo 2 Algoritmo do módulo EM

1: **função** EM($\mathbf{Y}^L(n)$,$\mathbf{D}(n)$,$\mathbf{E}(n)$)
2: difMatrix $\left(\mathbf{Y}^L(n), \mathbf{D}(n), \mathbf{E}(n)\right)$
3: **fim função**

2.4 *Backpropagation Module* - (BPM $-$ k)

Finalmente, o módulo BPM $-$ k que tem como função calcular os novos valores das matrizes de pesos, $\mathbf{W}^k(n)$, associadas à L camadas da rede. A expressão de atualização da k ésima camada pode ser expressa como

$$\mathbf{W}^{(k)}(n+1) = \mathbf{W}^{(k)}(n) + \Delta\mathbf{W}^{(k)}(n) \tag{10}$$

onde

$$\Delta\mathbf{W}^{(k)}(n) = \frac{\eta}{\mathbb{N}}\mathbf{g}^k(n)\begin{bmatrix}-\mathbf{1}\\ \mathbf{Y}^{k-1}(n)\end{bmatrix}^T + \alpha\boldsymbol{\Delta}\mathbf{W}^{(k)}(n-1), \tag{11}$$

no qual η é fator de aprendizado, α é coeficiente de esquecimento e $\mathbf{g}^k(n)$ pode ser expresso como

$$\mathbf{g}^k(n) = \begin{cases}\text{prod}\left(\varphi'\left(\mathbf{Y}^k(n)\right), \mathbf{E}(n)\right) & \text{para } k = L\\ \mathbf{z}^k(n) & \text{para } 1 \leq k < L\end{cases} \tag{12}$$

onde

$$\mathbf{z}^k(n) = \text{prod}\left(\varphi'\left(\begin{bmatrix}-\mathbf{1}\\ \mathbf{Y}^k(n)\end{bmatrix}\right), \left[\mathbf{W}^{(y)}(n)\right]^T \mathbf{g}^{(k+1)}(n)\right) \tag{13}$$

e

$$\mathbf{z}^k(n) = \left[\frac{z_{11}^{(k)}(n)\ z_{12}^{(k)}(n) \ldots z_{1\mathbb{N}}^{(k)}(n)}{\mathbf{z}^{(k)}(n)} \right]. \tag{14}$$

A função $\text{prod}(\mathbf{A}, \mathbf{B})$ é o produto elemento a elemento das matrizes e $\varphi'(\cdot)$ é a derivada da função de ativação.

2.5 Operações Básicas da Proposta

No Algoritmo 3 é demonstrado o procedimento para calcular o produto matricial da implementação, sendo *colunas1*,*colunas2* e *linhas1*, o número de linhas e colunas da matriz 1 e o número de colunas da matriz 2, respectivamente. É importante ressaltar que todos os algoritmos foram implementados utilizando aritmética de ponteiros. Dessa forma, o consumo de memória é bastante reduzido. Todas as matrizes envolvidas são do tipo *ponto flutuante*.

Algoritmo 3 Algoritmo do módulo produto matricial prodMat()

```
 1: função prodMatrix(matriz1, matriz2, matrizresultado, linhas1, colunas1, colunas2)
 2:   linha, coluna, índice, acumula
 3:   para linha = 1 até o número de linhas da matriz-1 faça
 4:     para coluna = 1 até o número de colunas da matriz-2 faça
 5:       acumula = 0
 6:       para índice = 1 até o número de colunas da matriz-1 faça
 7:         acumula = acumula + matriz-1[linha][índice]*matriz-2[índice][coluna]
 8:       fim para
 9:       matriz-3[linha][coluna] = acumula
10:     fim para
11:   fim para
12: fim função
```

Já no Algoritmo 4 é demonstrado o modelo para a operação de produto matricial ponto a ponto. O algoritmo é utilizado para as somas, produtos e subtrações, alterando-se somente a operação entre as matrizes na linha 5. Em todos os casos, o Algoritmo 4 assume que ambas as matrizes são de mesmas dimensões.

No Algoritmo 5 observa-se os passos necessários para se calcular o traço da diagonal principal do produto de duas matrizes em uma só operação dentro de dois loops aninhados.Logo abaixo, temos o Algoritmo 6 referente à função de ativação sigmóide utilizada.

3 Resultados

Objetivando validar a proposta de implementação, quanto ao seu funcionamento, tempo de execução, tamanho de ocupação em memória de programa e memória

Algoritmo 4 Algoritmo do produto matricial ponto a ponto

```
1: função prod(matriz1, matriz2, matrizresultado, linhas1, colunas1, colunas2)
2:    linha, coluna
3:    para linha = 1 até o número de linhas da matriz-1 faça
4:       para coluna = 1 até o número de colunas da matriz-2 faça
5:          matriz-3[linha][coluna] = matriz1[linha][coluna]*matriz2[linha][coluna]
6:       fim para
7:    fim para
8: fim função
```

Algoritmo 5 Algoritmo do traço do produto matricial

```
1: função trace(matriz1, matriz2, linhas1, colunas1)
2:    linha, coluna,traço
3:    para linha = 1 até o número de linhas da matriz1 faça
4:       para coluna = 1 até o número de colunas da matriz1 faça
5:          traço = traço + matriz1[linha][coluna]*matriz2[coluna][linha]
6:       fim para
7:    fim para
8: fim função
```

de dados, foi desenvolvido uma versão em linguagem C para microcontroladores do tipo ATmega 2560. A versão desenvolvida seguiu rigorosamente todos os algoritmos apresentados na seção anterior e foi compilada utilizando o ambiente de desenvolvimento Atmel Studio 7, ambiente este disponibilizado pela empresa Atmel fabricante do chip ATmega 2560. Após o processo de compilação e geração do código binário, o sistema foi embarcado no microcontrolador ATmega 2560 associado ao kit Arduino Mega. O ATmega 2560 é um MCU de 8 bits que trabalha à velocidade de 1 MIPS/MHz e possui 256 KBytes de memória de programa (*flash memory*) e 8 KBytes de memória de trabalho ou RAM. O Arduino Mega é um kit de desenvolvimento que agrega em um único hardware, um chip ATmega 2560 e um circuito gravador, facilitando ,assim, o processo de desenvolvimento.

O binário gerado da implementação do MLP-BP proposto ficou com 6,672 KBytes (2,6% de 256 KBytes), valor bem compacto comparado com a memória de programa máxima do ATmega 2560 e os valores de $5,904$ KBytes apresentados em [9] para a implementação do MLP sem o algoritmo do *Backpropagation*. Foram realizados testes de tempos de processamento associados aos módulos bem como testes de validação para o caso da porta XOR. O treinamento foi realizado em modo *batch* com $\mathbb{N} = 4$. Todos os testes foram realizados para uma rede com $L = 2$ camadas, duas entradas ($P = H^0 = 2$), vários neurônios na camada escondida, H^1, e um neurônio na camada de saída $M = H^2 = 1$. Mas, é importante ressaltar que o algoritmo proposto pode trabalhar com vários valores de P, H^k e M. A função de ativação utilizada foi a sigmóide.

Os testes de medição de tempo e validação da implementação foram realizados com o sistema embarcado no μC à um *clock* de 16 MHz. A Tabela 1 apresenta resultados associados ao tempo de processamento dos módulos FFM-1 ($t_{\text{FFM-1}}$),

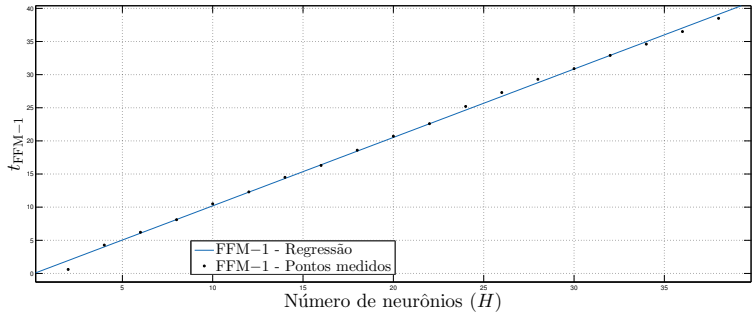
Algoritmo 6 Algoritmo da função de ativação sigmóide

```
1: função trace(matriz_in, matriz_out, linhas_in, colunas_in)
2:    linha, coluna,
3:    para linha = 1 até o número de linhas da matriz_in faça
4:       para coluna = 1 até o número de colunas da matriz_in faça
5:          matriz_out[linha][coluna]=(1.0)/(1.0+exponencial(-
   matriz_in[linha][coluna]))
6:       fim para
7:    fim para
8: fim função
```

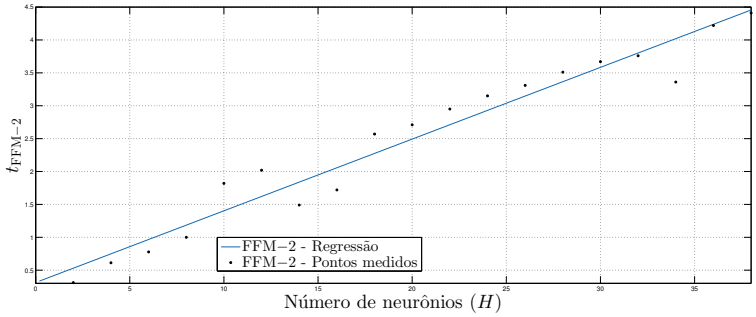
FFM-2 ($t_{\text{FFM-2}}$), EM (t_{EM}), BPM-1 ($t_{\text{BPM-1}}$) e BPM-2 ($t_{\text{BPM-2}}$) para vários valores de H^1. Os resultados foram obtidos com medições em osciloscópio com o algoritmo embarcado no μC. Já nas Figuras 2 - 3, são apresentadas curvas com os resultados descritos na Tabela 1, nas quais os pontos medidos foram ajustados por Regressão Polinomial com fator R igual a 0,9984 e 0,9496 respectivamente às tabelas. Finalmente, a Figura 4 apresenta as curvas do erro quadrático médio obtido no sistema embarcado para vários valores de H^1.

Tabela 1. Tempos de processamento medidos a partir da implementação proposta do MLP-BP com $P = H^0 = 2$, $M = H^2 = 1$ e função de avaliação sigmóide.

H^1	$t_{\text{FFM-1}}$ (ms)	$t_{\text{FFM-2}}$ (ms)	t_{EM} (ms)	$t_{\text{BPM-1}}$ (ms)	$t_{\text{BPM-2}}$ (ms)
2	0,59	0,31	0,06	0,46	1,30
4	4,28	0,61	0,06	0,66	2,23
6	6,21	0,77	0,06	0,88	3,22
8	8,11	1,00	0,06	1,11	4,11
10	10,50	1,82	0,06	1,70	6,14
12	12,30	2,02	0,06	1,91	6,90
14	14,50	1,49	0,06	1,80	7,06
16	16,30	1,72	0,06	2,03	8,02
18	18,60	2,57	0,06	2,65	10,60
20	20,70	2,71	0,06	2,47	11,20
22	22,60	2,95	0,06	3,07	12,40
24	25,20	3,15	0,06	3,36	13,60
26	27,30	3,31	0,06	3,71	14,70
28	29,30	3,51	0,06	3,95	16,50
30	30,90	3,67	0,06	4,03	16,20
32	32,90	3,76	0,06	3,83	15,80
34	34,60	3,36	0,06	4,05	16,70
36	36,50	4,22	0,06	4,82	19,40
38	38,50	4,41	0,06	5,17	21,80



(a)

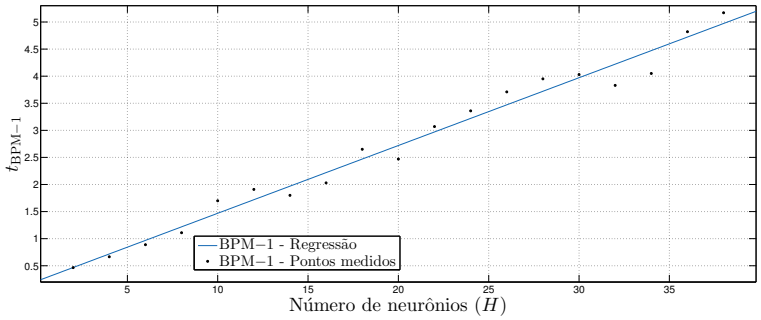


(b)

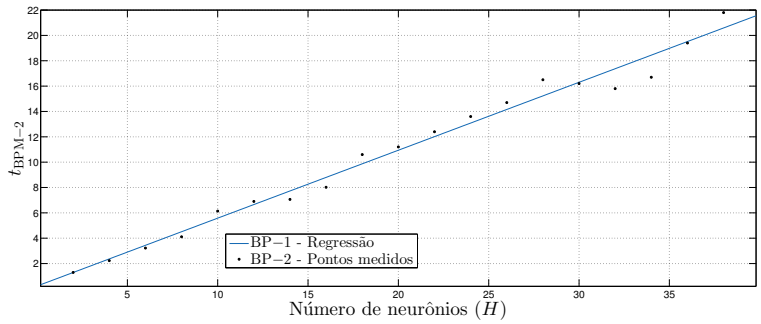
Figura 2. a) Tempo do FFM − 1 em função do número de neurônios da camada escondida, H^1. a) Tempo do FFM − 2 em função do número de neurônios da camada escondida, H^1.

As informações apresentadas na Tabela 1 mostram que os valores obtidos até $H^1 = 38$ neurônios na camada escondida, esse valor foi limitado pela capacidade de memória do μC utilizado, ATmega 2560, todavia é um valor bastante razoável para várias aplicações de tempo real voltadas para robótica, automação industrial e outras. Outro ponto importante a ser analisado são os tempos de execução da MLP-BP que também se apresentaram bastante factíveis. Analisando a utilização da rede sem treinamento, observa-se que o tempo de cada iteração em modo *batch* para $\mathbb{N} = 4$ foi em torno de $42,91\,\mathrm{ms}$ para o pior caso, $H^1 = 38$. Já se o tempo de treinamento estiver sendo contabilizado, para o caso de aplicações com treinamento *online*, o tempo de cada iteração é de $69,88\,\mathrm{ms}$ (no pior caso de $H^1 = 38$). Assim, observa-se que os resultados de tempo também se mostram bastante viáveis para várias aplicações comerciais na área de automação, robótica, automobilística e outras.

Em relação ainda ao tempo de processamento, verifica-se que seu crescimento é linear com o número de neurônios na camada escondida, H^1, como apresentado



(a)



(b)

Figura 3. a) Tempo do BPM − 1 em função do número de neurônios da camada escondida, H. a) Tempo do BPM − 2 em função do número de neurônios da camada escondida, H.

nas Figuras 2 - 3. Esse resultado é bastante significativo, dentro do ponto de vista de utilização de outros μC com maior capacidade de memória RAM, objetivando a utilização de mais neurônios. Esses resultados podem servir como referência para outros grupos que questionam a utilização do MLP-BP em suas aplicações com μCs.

Finalmente, para validar o funcionamento do algoritmo proposto embarcado no μC, foram realizados testes utilizando-se a técnica de HIL, cuja simulação é realizada no próprio hardware que está em *loop* com o computador que transmite e recebe os dados e parâmetros para análise. Os resultados dos testes em HIL foram caracterizados pelo cálculo do erro quadrático médio (*Mean Square Error* - MSE), com vários valores de H^1 para o caso da porta XOR. A Figura 4 apresenta as curvas de erro quadrático médio (*Mean Square Error* - MSE) para vários valores de H^1. O cálculo do MSE foi também embarcado no μC e sua

implementação é expressa como,

$$MSE(n) = \frac{1}{2N}\text{trace}\left(\mathbf{E}(n)^T\mathbf{E}(n)\right) \tag{15}$$

onde a função trace$(\cdot)$ calcula o traço de uma matriz.

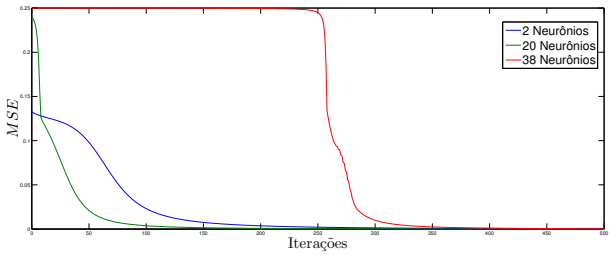


Figura 4. Erro quadrático médio com $H^1 = \{2, 20, 38\}$ neurônios na camada escondida.

4 Conclusões

Este trabalho apresenta uma proposta de implementação de um MLP-BP para μC de 8-bits. Detalhes de implementação como resultados associados às medidas de tempo de processamento foram apresentados para um μC Atmega 2560. Também foram apresentados resultados da validação do MLP-BP embarcado mediante a técnica de HIL. Os resultados mostram que existem propostas viáveis de implementação do MLP-BP para várias aplicações, cujas restrições de tempo são a partir de centenas de milissegundos, o que satisfaz as necessidades operacionais de várias aplicações na indústria de automação, robótica, entre outras.

Referências

1. AVR: AVR035: Efficient C Coding for AVR - Application Note. http://www.atmel.com/images/doc1497.pdf (2003)
2. AVR: Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers - Application Note. http://www.atmel.com/images/doc8453.pdf (2011)
3. Cotton, N.J., Wilamowski, B.M.: Compensation of nonlinearities using neural networks implemented on inexpensive microcontrollers. IEEE Transactions on Industrial Electronics 58(3), 733–740 (March 2011)
4. Cotton, N.J., Wilamowski, B.M., Dundar, G.: A neural network implementation on an inexpensive eight bit microcontroller. In: 2008 International Conference on Intelligent Engineering Systems. pp. 109–114 (Feb 2008)

5. Farooq, U., Amar, M., Hasan, K.M., Akhtar, M.K., Asad, M.U., Iqbal, A.: A low cost microcontroller implementation of neural network based hurdle avoidance controller for a car-like robot. In: 2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE). vol. 1, pp. 592–597 (Feb 2010)
6. Fernandes, M.A.C.: The matrix implementation of the two-layer multilayer perceptron (mlp) neural networks. https://www.mathworks.com/matlabcentral/fileexchange/36253-the-matrix-implementation-of-the-two-layer-multilayer-perceptron–mlp–neural-networks (2012)
7. Haykin, S.: Neural Networks: A Comprehensive Foundation. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edn. (1998)
8. Johann, S.F., Moreira, M.T., Heck, L.S., Calazans, N.L., Hessel, F.P.: A processor for iot applications: An assessment of design space and trade-offs. Microprocessors and Microsystems 42, 156 – 164 (2016)
9. Mancilla-David, F., Riganti-Fulginei, F., Laudani, A., Salvini, A.: A neural network-based low-cost solar irradiance sensor. IEEE Transactions on Instrumentation and Measurement 63(3), 583–591 (March 2014)
10. Mann, R.: How to program an 8-bit microcontroller using c language. Journal of Atmel Applications 3(4), 13–16 (2015)
11. Misra, J., Saha, I.: Artificial neural networks in hardware: A survey of two decades of progress. Neurocomputing 74(1-3), 239 – 255 (2010), artificial Brains
12. Noronha, D., Fernandes, M.A.C.: Implementação em fpga de máquina de vetores de suporte (svm) para classificação e regressõ. In: XIII Encontro Nacional de Inteligência Artificial e Computacional - ENIAC 2016. Recife, PE (oct 2016)
13. Ortega-Zamorano, F., Jerez, J.M., noz, D.U.M., Luque-Baena, R.M., Franco, L.: Efficient implementation of the backpropagation algorithm in fpgas and microcontrollers. IEEE Transactions on Neural Networks and Learning Systems 27(9), 1840–1850 (Sept 2016)
14. Oyamada, M.S., Zschornack, F., Wagner, F.R.: Applying neural networks to performance estimation of embedded software. Journal of Systems Architecture 54(1-2), 224 – 240 (2008)
15. Saad Saoud, L., Khellaf, A.: A neural network based on an inexpensive eight-bit microcontroller. Neural Computing and Applications 20(3), 329–334 (2011)
16. L. da Silva, M.T., Fernandes, M.A.C.: Proposta de arquitetura em hardware para fpga da técnica q-learning de aprendizagem por reforço. In: XIII Encontro Nacional de Inteligência Artificial e Computacional - ENIAC 2016. Recife, PE (oct 2016)
17. de Souza, A.C., Fernandes, M.A.: Parallel fixed point implementation of a radial basis function network in an fpga. Sensors 14(10), 18223–18243 (2014)
18. de Souza, I., Natan, S., Teles, R., Fernandes, M.: Platform for real-time simulation of dynamic systems and hardware-in-the-loop for control algorithms. Sensors 14(10), 19176–19199 (Oct 2014)
19. Ursuţiu, D., Nascov, V., Samoilă, C., Moga, M.: Microcontroller technologies in low power applications. In: 2012 15th International Conference on Interactive Collaborative Learning (ICL). pp. 1–5 (Sept 2012)
20. Zhang, L., Wang, Z.F.: Design of embedded control system based on arm9 microcontroller. In: 2010 International Conference on Electrical and Control Engineering. pp. 3579–3582 (June 2010)