Embedded Artificial Neural Networks
Optimized for Low-cost and Low-Size-Memory Devices

Caio José Borba Vilar Guimarães

Supervisor: Prof. Dr. Marcelo Augusto Costa Fernandes

Master Dissertation presented to the Graduate Program in Electrical and Computer Engineering of UFRN (concentration area: Computer Engineering) as part of the requirements for obtaining the title of Master of Science.

PPgEEC Order Number: M613
Natal, RN, Outubro de 2020
Guimarães, Caio José Borba Vilar.
Embedded Artificial Neural Networks Optimized for Low-cost and Low-Size-Memory Devices / Caio José Borba Vilar Guimarães - 2020
66 f.: il.

Orientador: Prof. Dr. Marcelo Augusto Costa Fernandes

1. MLP - Dissertação. 2. AI - Dissertação. 3. 8-bit - Dissertação. 4. Microcontrollers - Dissertação. 5. Artificial Neural Networks - Dissertação. 6. Embedded Systems - Dissertação. I. Fernandes, Marcelo Augusto Costa. II. Título.
To everyone that supported me through this time in my life.
Acknowledgements

To my supervisor, professor Dr. Marcelo Augusto Costa Fernandes, I am thankful for all the lessons, support, teachings and mostly the vast experience passed down to me.

To all my colleagues, for all the constructive criticism and suggestions.

To my family, all the support into going through this challenging step in my life.

To the Coordination for the Improvement of Higher Education Personnel (CAPES) due to the financial support.
Abstract

Artificial Neural Networks (ANNs) are bio-inspired systems with a high level of parallelization and almost infinite applications. However, due to the associated high computational power requirements, most application demands powerful processing characteristics and consequently, high-costs and not-so-small form-factors. This work presents an implementation of a Multilayer Perceptron (MLP) for 8-bit microcontrollers in two different scenarios, embedded training, and inference. Analysis of training convergence, inference time duration, and program code occupation into the internal memories and a technique to optimize this implementation to fit bigger MLP architectures. The aim of this work is to provide an overview of the feasibility of ANNs on this low-cost, low-size-memory devices, known as microcontrollers. This work shows a successful implementation of an MLP on a microcontroller with a linear behavior between the increase in hyperparameter values and the time-to-inference and code size. Also, an optimization to include more synaptic weights is presented for this same implementation, showing that even so the same behavior persists, validating further both implementations of the same solution proposal.

Keywords: MLP, AI, 8-bits, microcontroller, artificial neural networks, embedded systems.
### Resumo

Redes Neurais Artificiais (RNAs) são sistemas bioinspirados com um alto nível de paralelização e beira a infinidade de aplicações. Entretanto, o esforço computacional associado a essa tecnologia requer que hardwares usados em suas aplicações sejam de alta capacidade de processamento, o que implica em altos custos e formatos de encapsulamento que ocupam muito espaço. Este trabalho apresenta uma implementação de uma Rede Neural Artificial (RNA) do tipo Perceptron de Multiplas Camadas (MLP) para microcontroladores μCs de 8-bits em dois cenários diferentes com treinamento e inferência embarcados. São apresentadas análises de convergência de treinamento, tempo para inferência e ocupação de código-fonte nas memórias internas do microcontrolador. Uma técnica de otimização de armazenamento de pesos sinápticos na memória de programa é apresentada, com o intuito de aumentar a capacidade de aplicações de RNAs com arquiteturas maiores. O objetivo deste trabalho é apresentar uma implementação de RNA e a viabilidade deste tipo de aplicação em dispositivos de baixo custo e baixa capacidade de armazenamento em memória, conhecidos como microcontroladores. Este trabalho apresenta duas aplicações de RNA-MLP de uma operação XOR e um sistema que previne colisões para um robô virtual, analisados em três mapas virtuais. Além disso, é implementada uma classificação de dígitos numéricos do dataset MNIST, com inferência em 1,6 segundos e 50 neurônios em um μC com 256kB de memória de programa e 8kB de memória de trabalho.

**Palavras-chave:** MLP, IA, 8-bits, microcontroladores, redes neurais artificiais, sistemas embarcados.
Contents

Table of Contents i
List of Figures iii
List of Tables v
List of Symbols vii

1 Introduction 1
   1.1 Embedded Systems .............................................. 1
   1.2 Artificial Neural Networks in Machine Learning ............. 2
   1.3 Related Works ............................................... 2
   1.4 Objectives ................................................ 3
   1.5 Submitted and Published Works ................................ 4
   1.6 Dissertation Outline ....................................... 4

2 Neural Networks for 8-bit Microcontrollers 5
   2.1 Introduction ............................................... 5
   2.2 System Description ......................................... 7
      2.2.1 Associated Variables .................................. 7
      2.2.2 Feedforward Module - (FFM − k) .................. 9
      2.2.3 Error Module - (EM − k) ............................. 9
      2.2.4 Backpropagation Module - (BPM − k) ............. 10
      2.2.5 Basic Operations .................................... 10
   2.3 Methodology ............................................... 11
      2.3.1 Hardware in the Loop Simulation ..................... 13
      2.3.2 XOR Operation ...................................... 14
      2.3.3 Virtual Car-Like Robot ................................ 15
   2.4 Results ................................................ 17
      2.4.1 XOR Operation ...................................... 17
      2.4.2 Virtual Car-like Robot .............................. 19
   2.5 Comparison with the State-of-the-Art ....................... 20
   2.6 Conclusions ............................................. 21
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Block diagram detailing the modules of the matrix implementation for a</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>MLP trained with BP algorithm.</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Hardware In the Loop assembly scheme in block diagram.</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Hardware In the Loop assembly on the test bench.</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Hardware In the Loop measurements of the embedded XOR operation using an</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>oscilloscope.</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>a) Execution time of FFM − 1 by number of neurons in the hidden layer,</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>$H^1$. b) Execution time of FFM − 2 by the number of neurons in the hidden</td>
<td></td>
</tr>
<tr>
<td></td>
<td>layer, $H^1$.</td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>a) Execution time of BPM − 1 by the number of neurons in the hidden layer,</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>$H^1$. b) Execution time of BPM − 2 by the number of neurons in the hidden</td>
<td></td>
</tr>
<tr>
<td></td>
<td>layer, $H^1$.</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>Mean Squared Error curve fitting comparison with $H^1 = {2, 20, 38}$</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>neurons in the hidden layer.</td>
<td></td>
</tr>
<tr>
<td>2.8</td>
<td>Virtual map with the resulting trajectory of the virtual car-like robot for</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>the 1st training dataset.</td>
<td></td>
</tr>
<tr>
<td>2.9</td>
<td>Virtual map with the resulting trajectory of the virtual car-like robot for</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>the 2nd training dataset.</td>
<td></td>
</tr>
<tr>
<td>2.10</td>
<td>Virtual map with the resulting trajectory of the virtual car-like robot for</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>the 3rd training dataset.</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Standard model of an artificial neuron.</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Architecture of a Multi-Layer Perceptron.</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Arrangement of memory sections in the program memory of an AVR $\mu$C.</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Block diagram of the implementation modules with $k$ hidden layers.</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Feedforward time in the hidden layer, $t_{FFM-1}$</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Feedforward time in the output layer, $t_{FFM-2}$</td>
<td>37</td>
</tr>
<tr>
<td>3.7</td>
<td>Occupancy of program memory for $P = {10, 20, 30, 40, 50}$ and $M = 10$.</td>
<td>38</td>
</tr>
<tr>
<td>3.8</td>
<td>Occupancy of data memory for $P = {10, 20, 30, 40, 50}$ and $M = 10$.</td>
<td>39</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Truth table for the XOR operation</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Training dataset for the first and simplest case. With $H^0 = 3$, $H^1 = 5$ and $H^2 = 2$.</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Training dataset for the second case. With $H^0 = 3$, $H^1 = 5$ and $H^2 = 2$.</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Training dataset for the third and most complex case. With $H^0 = 3$, $H^1 = 10$ e $H^2 = 2$.</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>Duration of execution for each module of the implementation of the MLP-BP with $P = H^0 = 2$, $M = H^2 = 1$ and sigmoid activation function.</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>Execution times measured with HIL for the implementation proposal in the virtual car-like robot validation.</td>
<td>20</td>
</tr>
<tr>
<td>2.7</td>
<td>ANN results for the state-of-the-art works,*Time of inference, not taking the EM-k and BPM-k modules time into account.</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Measured processing times of the MLP-ANN implementation with $M = 10$ and $N = 785$.</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>Hardware occupancy results for data and program memories of the MLP-ANN implementation with $M = 10$ and $N = 785$.</td>
<td>35</td>
</tr>
</tbody>
</table>
# List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>µC</td>
<td>Microcontroller</td>
</tr>
<tr>
<td>µP</td>
<td>Microprocessor</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>ANNs</td>
<td>Artificial Neural Networks</td>
</tr>
<tr>
<td>BP</td>
<td>Back Propagation Algorithm</td>
</tr>
<tr>
<td>BPM</td>
<td>Back Propagation Module</td>
</tr>
<tr>
<td>DL</td>
<td>Deep Learning</td>
</tr>
<tr>
<td>EM</td>
<td>Error Module</td>
</tr>
<tr>
<td>FFM</td>
<td>Feed Forward Module</td>
</tr>
<tr>
<td>GPP</td>
<td>General-Purpose-Processor</td>
</tr>
<tr>
<td>HIL</td>
<td>Hardware In the Loop</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IRPM</td>
<td>Input Random Permutation Module</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>MLP</td>
<td>Multilayer Perceptron</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean Square Error</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The use of Artificial Neural Networks (ANNs) on Embedded Systems (ES) is a recurring matter for many, since the capabilities of ANNs provide intelligence to ES applications could help to reduce costs and leverage more information from data, directly at the source. In fields like the Internet of Things (IoT), remote sensing and monitoring with intelligent ES enables the deployment of low-power systems for longer periods (Sarwesh et al. 2017). Also on the remote sensing, ANN-based ES allows for better disaster prevention IoT solutions, (Meyer et al. 2019). These are only a few of many other use cases cited in this work that elucidates the need for ANN solutions using simple and even really advanced models that employ Deep Neural Networks and State-of-The-Art (SOTA) Artificial Intelligence (AI) technologies.

The greatest hurdle to the development and implementation of ANNs on ES is the intrinsic high computational cost of neural network models, the simplest architectures requires thousands of multiplications and additions and also a lot of memory is required. Usually, the devices that are employed for these applications are Graphical Processing Units (GPUs), high-performance servers, Application Specific Integrated Circuits (ASICs) and custom-synthesized Field Gate Programmable Arrays (FPGAs). These devices have costs and form factors that are most of the times not adequate for ES applications and the required expertise to handle them is beyond what is usually necessary to ES solutions.

1.1 Embedded Systems

Embedded Systems are specialized computational systems that perform an specific action or specific set of actions and in a deterministic fashion. Also it can be defined as combination of computer hardware and software designed to perform a dedicated function, (Barr & Massa 2006). The definition by (Gajski D. et al. 2009) characterizes an embedded system as a set of hardware and software solutions that together must precisely produce an output, most of the times within a certain real-time constraint. Overall, the definition of embedded systems encompasses a solutions composed of hardware and software modules, that could be comprised of smaller embedded systems, that must do work deterministically within a certain time and resource constraint of size, clock, power, etc.

These definitions sets the hardware platform of Microcontrollers (μCs) as one of the
mostly used devices for embedded systems. Microcontrollers are comprised of 8-bit, 16-bit, 32-bit and even 64-bit CPUs and other peripheral devices inside a single silicon die. Program memory, work memory, registers, I/O ports, various communication protocol-specific circuits, timers and Pulse Width Modulation (PWM) generators are also easily found inside this devices. Their main issue is the innate low capacity of the internal memories and the overall slow clock that normally limits the applicability of these devices on more computationally intensive tasks.

Microcontrollers (µCs) are Systems on a Chips (SoCs) that fit in this low-cost and low-size niche, meanwhile being widely used in fields such as IoT, Smart Grid, Machine-to-Machine Communication (M2M), consumer electronics and automotive industry, among others.

1.2 Artificial Neural Networks in Machine Learning

Artificial Neural Networks (ANNs) are computational systems based on neural networks of the animal brain, (Haykin 1999). This field of Machine Learning (ML) models processing units as artificial neurons, as seen in Figure 3.1. The networks themselves are formed of various of those units connected in different topologies with different internal functions to provide a diverse range of functionalities and applications, as seen in Figure 3.2.

MLPs are one of the most versatile and widely used ANNs in ML. Also regarded as universal function approximators by the literature, (Haykin 1999), MLPs are feedforward ANNs composed, usually, of fully-connected layers, Figure 3.2 of elements named as artificial neurons, nodes, or units, such as the ones in Figure 3.1, with its synaptic weights and activation functions. The MLP is used as an end layer on almost all modern applications of AI where you need to perform a classification task, be it the MobileNet and MobileNetV2, (Howard et al. 2017, Sandler et al. 2018) or even the ShuffleNet, (Zhang et al. 2017).

1.3 Related Works

The usage of SoCs for intelligent embedded hardware platform has been a recurring research topic for many groups. The work in (Farooq et al. 2010) presents a hurdle-avoidance neural-network-based controller for a car-like robot. This is implemented on a Microchip AT89C52 µC, employing an MLP-like ANN. They show results for an architecture with up to 20 neurons in the hidden layer and the robot avoiding obstacles successfully. The ANN is trained offline using Matlab and a tangent-sigmoid activation function. Unfortunately, the authors do not provide source-code size and time-to-inference results, that could prove useful on determining the limitations of the technique they proposed.

The work on (Saad Saoud & Khellaf 2011a) proposes an implementation of an MLP on a PIC16F876A µC with embedded inferencing and training using backpropagation. This proposed ANN is applied to a nonlinear chemical process to demonstrate the feasibility and performance of the results, also comparing the µC implementation with an
1.4. OBJECTIVES

The authors on (Cotton et al. 2008) describe a ANN MLP implementation, that is trained offline using Matlab Packages, on a low-end inexpensive µC, PIC18F45J10. The ANN is applied to approximate a nonlinear control surface, with also an implementation of a fast-multiplication algorithm for fractional value using an internal 8-bit hardware multiplier. The results show a really accurate approximation of a parametric control surface and approximation of a hyperbolic tangent activation function using assembly language. The authors concluded that their implementation successfully performed regarding industrial application’s requirements, such as motor drives and harmonic distortion on power distributions, providing fast and accurate floating-point calculations on cases that using computers may not be practical. Also, this work highlighted that the only limitation to their implementation proposal is the amount of weights needed, which was limited to 256 weights, but that could be expanded.

Mancilla-David and colleagues presented a ANN-based solar irradiance sensor to maximize the efficiency of photo-voltaic power-plants on a PIC18F6627 µC (Mancilla-David et al. 2014). They presented source-code occupation for an MLP, trained using Levenberg-Marquardt Algorithm (LMA), on program memory of close to 5.9 kB and 186 bytes of data or work memory. The MLP architecture uses three input, eight neurons in the hidden layer and a single output neuron and hyperbolic-tangent as activation function. Their main goal was to show that an inexpensive 8-bit microcontroller was able to implement a powerful ANN, such as MLP, and accurately provide a precise solar irradiance tracking at such a low cost of tens of dollars.

A fact is that different authors use different techniques to show feasibility of ANN applications. However, different techniques present different results, and techniques mostly used on other platforms renders a high and prohibitive computational cost, especially for applications where time is critical. Hence, the biggest challenge for research groups is to optimize the AI algorithms with different embedding techniques to make applications on SoCs such as µCs appease to the industry.

1.4 Objectives

The presence of MLPs in AI applications inspired this work in proposing an implementation of an MLP on a microcontroller in different scenarios, showing how suitable this platform is for embedded real-time AI applications.

The first scenario it is implemented training and inference of an MLP on a µC. Analyzing how the Mean Squared Error (MSE) decreases with time, with successful training and how the increase in hyperparameters affect the overall duration of the training and inferencing steps of this application.

In the second scenario, the concern is on how to implement bigger architectures on an
intrinsically limited platform, with small internal memory and reduced clock. Also, the
time duration of an inference operation is analyzed as well as how the proposed technique
makes it possible to fit a bigger architecture on such a set of internal memories of the
chosen $\mu$C.

1.5 Submitted and Published Works

- Vilar, Caio B., Deângela Neves and Marcelo A. C. Fernandes (2018), Proposta
de implementação de tempo real de redes neurais MLP em microcontroladores
de 8-bits, In ‘Proceedings XIII Brazilian Congress on Computational Intelligence’,

- Vilar, Caio B. and Marcelo A. C. Fernandes (2019), Otimização de Redes Neurais
MLP em Microcontroladores de 8 bits Utilizando Memória de Programa, In ‘Pro-
ceedings XV Brazilian Congress on Computational Intelligence’, ABRICOM. DOI
10.21528/CBIC2019-44.

- Vilar, Caio B. and Marcelo A. C. Fernandes (2020), Real-time Neural Networks
Implementation Proposal for Microcontrollers. Electronics 2020, 9, 1597. DOI
10.3390/electronics9101597.

1.6 Dissertation Outline

- Chapter 2 presents a real-time neural networks implementation proposal for 8-bit
microcontrollers where an MLP ANN with embedded training and inference is pre-
 sented and analyzed through two application scenarios.

- Chapter 3 presents an optimization of the implementation proposed in the previous
chapter using the $\mu$C’s program memory. This shows a technique for optimizing the
storage of synaptic weights allow bigger ANN architectures to be used on real-time
applications on $\mu$Cs.

- Chapter 4 presents the conclusions of this dissertation and considerations for future
works in this area.
Chapter 2

Neural Networks for 8-bit Microcontrollers

2.1 Introduction

The microcontrollers (µCs) have been applied in many areas: industrial automation, control, instrumentation, consumer electronics, and other various areas. Nonetheless, there is an ever-growing demand for these devices, especially in emerging sectors like the Internet of Things (IoT), Smart Grid, Machine to Machine (M2M) and Edge Computing. A µC can be classified as programmable hardware platform that enables Embedded System applications in specific cases. It is important to know that µCs are mainly composed of a General-Purpose Processor (GPP) of 8, 16 or 32 bits. Then this GPP is connected to some peripherals like Random Access Memory (RAM), flash memory, counters, signal generators, communication protocol specific hardware, analog to digital and digital to analog converters and others.

An important fact is that on most products that are available today, the µCs embedded into them encapsulate an 8-bit GPP with enough computational power and memory storage, to show itself as a resourceful platform for many embedded applications. However, those same 8-bit µCs are considered low-power and low-cost platforms when compared with other platforms that are used to implement AI applications with Artificial Neural Networks (ANNs) (Ursuți et al. 2012, de Souza et al. 2014).

The use of ANNs for embedded intelligent systems with real-time constraints has been a recurrent research topic for many (Misra & Saha 2010, Ortega-Zamorano et al. 2016, de Souza & Fernandes 2014, L. da Silva & Fernandes 2016, Noronha & Fernandes 2016). A large part of the works devised from this topic is driven by the growing demand for AI techniques for IoT, M2M and Edge Computing applications.

A major problem with implementing ANN applications into embedded systems is the computational complexity associated with ANNs. In regards to the Multi-Layer Perceptron (MLP), described in this work, there are many inherent multiplications and calculations of nonlinear functions (Misra & Saha 2010, Ortega-Zamorano et al. 2016, de Souza & Fernandes 2014). Besides the feedforward process between the input and the synaptic weights, the MLP also has a training algorithm associated with it to find the optimum weights of the neural network. This training algorithm is very computationally expensive (Haykin 1998). If the training process is also performed in real-time, the computational
complexity is increased several times. This increase in complexity automatically raises
the processing time and requirements for memory storage from the hardware platform
used in the application (Misra & Saha 2010, Ortega-Zamorano et al. 2016, de Souza &
Fernandes 2014).

The use of MLP neural networks for real-time applications on µCs it’s not a new
effort. The work (Cotton & Wilamowski 2011) in which the authors describe a method
to linearize the nonlinear characteristics of many sensors using an MLP-NN on an 8-bit
PIC18F45J10 µC. The obtained results showed that if the network architecture is right,
even very difficult problems of linearization can be solved with just a few neurons.

In (Cotton et al. 2008), a fully-connected multi-layer ANN is implemented on a low-
end and inexpensive µC. Also, a pseudo-floating-point multiplication is devised, to make
use of the internal multiplier circuit inside the PIC18F45J10 µC used. The authors man-
aged to store 256 weights into the 1KB SRAM of the µC and deemed it being enough for
most embedded applications.

In (Farooq et al. 2010) Farooq et al implemented a hurdle-avoidance system controller
for a car-like robot using an AT89C52 µC as a system embedding platform. They imple-
mented an MLP with a back-propagation training algorithm and a single hidden layer.
The proposed system was tested in various environments containing obstacles and was
found to avoid obstacles successfully.

The paper (Saad Saoud & Khellaf 2011b) presents a neural network that is trained
with the backpropagation (BP) algorithm and validated using a low-end and inexpensive
PIC16F876A 8-bit µC. The authors chose a chemical process as a realistic example of
a nonlinear system to demonstrate the feasibility and performance of this approach, as
well as the results found using the microcontroller, against a computer implementation.
With three inputs, five hidden neurons and an output neuron on the MLP, the application
showed complete suitability for a µC-based approach. The results comparing the µC im-
plementation showed almost no difference in Mean Square Error (MSE) after 30 iterations
of the training algorithm.

The work presented in (Zhang & Wang 2010) an ANN-based PID controller is shown
using an ARM9 based µC. The authors modeled the controller to overcome the nonlin-
earity of a microbiologic fermentation process and to provide a better performing control
strategy. The results showed a more accurate control over the controlled parameters with
an ANN-PID an achieve a greater control performance and ability of the system to meet
the requirements.

In (Gural & Murmann 2019) the authors implemented a classification application for
the MNIST dataset. This implementation regards the 10 digits and full classification with
99.15% testing accuracy. Also, this is implemented with a knowingly highly resource-
hungry ANN model, the Convolutional Neural Network (CNN), using less than 2KB of
SRAM memory and also 6KB of program memory, FLASH. This work was embedded
on a Arduino Uno development kit that is comprised of a breakout board for the 8-bit
ATMega328p µC, with 32KB of program memory and 2KB of work memory, running at
16MHz.

The reference works presented above have shown different aspects of implementing
ANNs on µCs. In (Cotton et al. 2008, Farooq et al. 2010, Saad Saoud & Khellaf 2011b)
2.2. SYSTEM DESCRIPTION

you will find application proposals showing MLP-ANNs trained with the Backpropagation algorithm (MLP-BP) implemented on μCs with good results, but none of those talk about memory usage and processing-time parameters that vary according to the MLP hyperparameters. In (Cotton & Wilamowski 2011, Oyamada et al. 2008, Zhang & Wang 2010) the authors presented some results regarding processing-time, but none dependant on the number of artificial neurons or even comparing the time required to train the ANN in real-time or not.

Therefore, this work proposes an implementation of an MLP-ANN that can be trained with the BP algorithm into an ATmega2560 8-bit μC in the C language, to show that many applications with ANNs are suitable on this μC platform. We also present two implementations regarding a model that is trained on the μC in real-time and another implementation that is trained with Matlab and then ported to the same architecture to execute classification in real-time. It is important to notice that although the second model was trained using Matlab, the implementation proposal here presented is programming language and platform-independent.

In addition to the implementation proposal, parameters of processing time to each feedforward step and backward steps in the training and classification process are presented. Also, the variation of these parameters is shown due to the variation of the hyperparameters of the MLP. We also validate the classification and training results on a Hardware-in-the-Loop (HIL) strategy.

2.2 System Description

Figure 2.1 shows a block diagram detailing the modules implemented into the μC. This implementation has the work (Fernandes 2012) as a direct reference, that models the MLP into a matrix form, simplifying and modularizing all the feedforward and backwards propagations As seen in Figure 2.1 the MLP here implemented is structured as four main modules, the Input Random Permutation Module (IRPM), FeedForward Module (FFM-k), Error Module (EM) and Backpropagation Module (BPM-k), at which the variable $k$ represents the ANN layer. It is important to add that the implementation presented in this work is shown with two layers (one hidden and the output layer), but it can be easily extended for more. The modules and associated mathematical modeling will be detailed in the following subsections.

2.2.1 Associated Variables

The implementation is composed of four main variables that are passed by reference between the modules. These variables are the input signals matrix, $Y^{k-0}(n)$ defined as

$$Y^0(n) = \left[ y_1^0(n), y_2^0(n), \ldots, y_s^0(n), \ldots, y_N^0(n) \right]$$

(2.1)

where $k = 0$ means that this matrix is dealing with the input layer of the MLP, $n$ represents the iteration number of the training algorithm. $N$ is defined as the number of samples of a
training set and $y_s$ the $s$-th sample defined as

$$y_s^0(n) = [y_{s1}^0(n), y_{s2}^0(n), \ldots, y_{s}^0(n), \ldots, y_P^0(n)]^T$$  (2.2)

where $P$ is the number of available inputs of the MLP. The synaptic weights matrix from the $k$-th layer, $W^k(n)$, is defined as

$$W^k(n) = \begin{bmatrix}
w_{k10}^k(n) & w_{k11}^k(n) & \cdots & w_{k1h}^k(n) & \cdots & w_{1H^k-1}^k(n) \\
\vdots & \vdots & \ddots & & \ddots & \vdots \\
w_{j0}^k(n) & w_{j1}^k(n) & \cdots & w_{jh}^k(n) & \cdots & w_{jH^k-1}^k(n) \\
\vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\
w_{H^k0}^k(n) & w_{H^k1}^k(n) & \cdots & w_{y_{H^k}}^k(n) & \cdots & w_{H^kH^k-1}^k(n)
\end{bmatrix}$$  (2.3)

where $H^k$ is the number of neurons from the $k$-th layer and $w_{ij}^k(n)$ represents the synaptic weight associated with the $i$-th artificial neuron, from the $j$-th input signal of the $k$-th layer at the $n$-th iteration.

The output signals matrix, $Y^L(n)$ defined as

$$Y^L(n) = [y_{1}^L(n), y_{2}^L(n), \ldots, y_{s}^L(n), \ldots, y_{M}^L(n)]^T$$  (2.4)

where $L$ is the number of layers of the MLP and also defines which layer is the last one. Where $y_{s}^L(n)$ represents the output signal associated with the $s$-th sample input $y_s^0$ that is defined as

$$y_{s}^L(n) = [y_{s1}^L(n), y_{s2}^L(n), \ldots, y_{s}^L(n), \ldots, y_{M}^L(n)]^T$$  (2.5)

where $M$ is the number of output neurons.

The $D(n)$ variable represents the desired values or labels of the training set composed of the $D(n)$ and the $Y^0(n)$, that is defined as

$$D(n) = [d_1(n), d_2(n), \ldots, d_s(n), \ldots, d_N(n)]$$  (2.6)

where $d_s(n)$ is the vector of desired values referring to the $s$-th sample associated with
the $y_s^0(n)$ input signal, that is defined as
\[
d_s(n) = [d_1(n), d_2(n), \ldots, d_s(n), \ldots, d_M(n)]^T. \tag{2.7}
\]

For an MLP with two layers, $W^1(n)$ represents the synaptic weights matrix from the hidden layer ($H^0 = P$) and $W^2(n)$ is the weights matrix from the output layer ($H^2 = M$) at the $n$-th iteration. In addition to those two matrices, a few others are created to accommodate intermediary results from feedforward and backward propagation operations of the MLP.

### 2.2.2 Feedforward Module - (FFM $- k$)

This module is responsible for running the feedforward operation of an MLP, propagating the inputs through each $k$-th layer during each $n$-th iteration. As most BP implementations, this proposal can operate in online mode or batch mode (defining $N > 1$). At each $k$-th FFM-$k$ at each $k$-th layer the following equation is calculated as
\[
Y^k(n) = \varphi \left( W^k(n) Y^{k-1}(n) \right) \tag{2.8}
\]
where $Y^{k-1}(n)$ is the output signals matrix from the previous layer. The $Y^k(n)$ represents the output of the current layer and $\varphi(.)$ is the activation function of the current $k$-th layer. As previously said a few other matrices are devised as part of the BP calculation and some of them store these calculation results from Equation [2.8]

Algorithm 1 describes the pseudocode executed when the FFM-$k$ module is called. The function prodMatrix() implements the matrix product between $W^k(n)$ and $Y^{k-1}(n)$ and in the end it stores the product result in $Y^k(n)$. The actFun() function executes the activation function of the $k$-th layer to each element of $Y^k(n)$ and stores the result in itself ($Y^k(n)$) as it runs.

**Algorithm 1** Description of the algorithm implemented by the FFM $- k$ module.

1: function FFM $- k(W^k(n), Y^{k-1}(n), Y^k(n))$
2: prodMat ($W^k(n), Y^{k-1}(n), Y^k(n)$)
3: actFun ($Y^k(n)$)
4: end function

### 2.2.3 Error Module - (EM $- k$)

The error module EM-$k$ calculates the error between the desired values matrix $D(n)$ and the output signals matrix $Y^L(n)$ of the last layer, $L$. The equation that is evaluated at the EM-$k$ is defined as
\[
E(n) = D(n) - Y^L(n) \tag{2.9}
\]

Algorithm 2 describes how the EM-$k$ module functions, in which the difMatrix() function implements the element-by-element difference between $D(n)$ and $Y^L(n)$ and stores
the result into the $E(n)$ matrix.

**Algorithm 2** Pseudocode implementation of the EM module.

1: function EM($Y^L(n), D(n), E(n)$)
2:   difMatrix ($Y^L(n), D(n), E(n)$)
3: end function

### 2.2.4 Backpropagation Module - (BPM − k)

The last module is responsible for the main training part of the implementation. The BPM-$k$ calculates the new updated values of the synaptic weights matrices of the $L$ layers that better approximate the desired values, $W^k(n)$. The equation that the BPM-$k$ implements is defined as

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

(2.10)

where

$$\Delta W^{(k)}(n) = \eta \frac{\prod \phi'(\cdot)}{N}^k(n) \left[ -1 \right] T + \alpha \Delta W^{(k)}(n-1),$$

(2.11)

where $\eta$ is the learning-rate of the BP algorithm, $\alpha$ is the learning-moment rate factor and $\phi^k(n)$ is defined as

$$\phi^{k}(n) = \prod \phi'(\cdot) \left( Y^k(n) \right)^T E(n)$$

(2.12)

where

$$z^{(k)}(n) = \prod \phi'(\cdot) \left( Y^k(n) \right)^T W(n)^T g^{(k+1)}(n)$$

(2.13)

and

$$z^{(k)}(n) = \frac{z^{(k)}_{11}(n) \cdots z^{(k)}_{1N}(n)}{z^{(k)}(n)} .$$

(2.14)

The $\text{prod}()$ function in the Equation 2.13 implements an element-wise product between two matrices, implemented in Algorithm 3 and $\phi'(\cdot)$ is the derivative of the activation function.

### 2.2.5 Basic Operations

In this section, we describe how we implemented a few basic operations used in this work. First, Algorithm 3 implements the matrix product. A very important detail is that all the modules were implemented following pointer arithmetic procedures. With this in mind, the source code is very optimized for memory usage, reducing the overall duration of execution from each module. Every matrix used in this implementation are floating-point (IEEE754) (**IEEE standard for binary floating-point arithmetic 1985**) representation values.
2.3. **METHODOLOGY**

Algorithm 3 Matrix product implementation pseudocode

1: function prodMat($W^k, Y^{k-1}, Y^k, P, M, N$)
2: Initialize $(i, j, s, \text{accumulate}) \leftarrow 0$
3: for $i \leftarrow 1$ to $M$ do
4:    for $j \leftarrow 1$ to $N$ do
5:       accumulate $\leftarrow 0$
6:       for $s \leftarrow 1$ to $P$ do
7:          accumulate = accumulate + $W^k[i][s] \times Y^{k-1}[s][j]$
8:       end for
9:       $Y^k[i][j] = \text{accumulate}$
10:    end for
11: end for
12: end function

Algorithm 4 shows the pseudocode of an element-wise product between two matrices used in this work, more specifically in the BPM-\(k\) module implementation. This same pseudocode in Algorithm 4 can be slightly altered to perform subtraction or addition by editing the operation between matrices on line 5. It’s important to notice that Algorithm 4 requires that both matrices have the same dimensions.

Algorithm 5 presents to us the implementation of the calculation of the Trace of the Product between two matrices at a single operation inside a nested loop. This simple optimization saved a few \(\mu\)s of processing time for the overall training process.

Algorithm 6 shows the implemented steps for calculating the activation function of the induced-local-field of each neuron, an element-wise operation. This Algorithm 6 is implementing a sigmoid activation function.

### 2.3 Methodology

The implementation was validated using the well-known HIL simulation strategy, as explained in (de Souza et al. 2014) and illustrated in Figure 2.2. The tested variables are, execution time for the FFM-1 for the hidden layer as seen in Figure 2.1 defined as \(t_{\text{FFM-1}}\), execution time for the FFM-2, feedforward for the output layer of neurons, \(t_{\text{FFM-2}}\), error
CHAPTER 2. NEURAL NETWORKS FOR 8-BIT MICROCONTROLLERS

Algorithm 5 Pseudocode implementation of the trace of the product between two matrices.

1: function trace($E^k, E^{kT}, P, M$)
2:     Initialize ($i, j, \text{trace}$) ← 0
3:     for $i ← 1$ to $P$ do
4:         for $j ← 1$ to $M$ do
5:             trace = trace + $E^k[i][j] \times E^{kT}[i][j]$
6:         end for
7:     end for
8: end function

Algorithm 6 Pseudocode implementation of the sigmoid activation function

1: function activfun($Y^k, Y^{k+1}, P, M$)
2:     Initialise ($i, j$) ← 0
3:     for $i ← 1$ to $P$ do
4:         for $j ← 1$ to $M$ do
5:             $y^{k+1}[i][j]$ = $(1.0)/(1.0+\text{exponential}(-Y^k[i][j]))$
6:         end for
7:     end for
8: end function

module $t_{EM}$, backpropagation of the output neurons layer, $t_{BPM-1}$, and the backpropagation of the hidden neurons layer, $t_{BPM-2}$.

A parameter that was also analyzed is the memory occupation, regarding both memories of the ATmega-2560 $\mu$C used in this paper, FLASH or program memory and SRAM or work memory.

As previously mentioned the modules were implemented in the C program language for AVR $\mu$Cs using the avr-gcc version 5.4.0, inside the Atmel Studio 7 development environment, an Integrated Development Environment (IDE) made available by Microchip. After compilation and binary code generation the solution was embedded into an Atmega-2560. This $\mu$C has an 8-bit GPP integrated with 256KBytes of FLASH program memory and 8KBytes of SRAM work memory, its maximum processing speed is 1 MIPS/MHz.

The Atmega-2560 is associated to an Arduino Mega v2.0 development kit, the Arduino Mega is a development kit that provides a breakout board for all the ATmega-2560 pins and some other components required for the $\mu$C to function properly. A great feature of this development kit is the onboard Universal Serial Bus (USB) programmer that enables the developer to simply connect a USB port to a computer and test various implementations easily on the $\mu$C.

This work is further validated using two cases. First, we train the MLP-BP to behave as an XOR operation as the simplest case possible to train an MLP and evaluate its ability to learn a nonlinear relationship between two inputs. Secondly, we train the network to aid a car-like virtual robot in avoiding obstacles in a virtual map using Matlab and 3 cases of increasing ANN architecture complexity. The assembly, and analysis of these two validation cases are presented in the following subsections.
2.3. METHODOLOGY

2.3.1 Hardware in the Loop Simulation

The tests were executed with the $\mu$C running at a clock of 16MHz. The results are obtained by setting the level of a digital pin of the ATMega-2560 to HIGH, executing one of the modules and setting the same digital pin logical value to LOW and measuring the time of logic HIGH on an oscilloscope for this digital pin. This can be easily seen on Figure 2.2 and a picture of the HIL assembly is shown in Figure 2.3. In Figures 2.5, 2.6 we present curved plots of the results described into 2.5 where we performed curve-fitting on the measured points with Polynomial Regression. Figure 2.4 shows a closer look on the oscilloscope measurements, taking in regard all the modules being measured by their duration of execution for the XOR validation case described below.

The calculation of the MSE during training was also embedded into the $\mu$C implementation and they’re transmitted through Universal Serial Synchronous Asynchronous Receiver Transmitter (USART) protocol to a computer. The calculation of the MSE is defined as

$$MSE(n) = \frac{1}{2N} \text{trace} \left( E(n)^T E(n) \right)$$  \hspace{1cm} (2.15)

where $\text{trace}(\cdot)$ is the implementation seen in Algorithm 5.
2.3.2 XOR Operation

A validation of the real-time embedded training using the BPM – $k$ module was devised for a XOR operation, as described on Table 2.1 the most basic MLP validation case. This test was performed with a training and classification phases, both running in the µC, considering a configuration as seen in Figure 2.1. The test was executed on batch mode with $N = 4$, two layers of neurons ($L = 2$), two input signals ($P = H^0 = 2$), a varying amount of neurons in the hidden neurons layer ($H^1$) as seen in 2.5 and a single neuron in the output layer ($H^2 = 1$). It is important to notice that the strategy here presented can be assembled with various different configurations just modifying the $P, H^k$ and $M$ parameters, being the µCs internal memories the limiting factor for the ANN architecture size.

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.1: Truth table for the XOR operation
### 2.3. METHODOLOGY

The methodology section of the document explains the approach used in the research. It includes a table that outlines the conditions and parameters used in the experiments, as well as a detailed description of the virtual car-like robot used in the testing.

<table>
<thead>
<tr>
<th>Condition</th>
<th>FS (m)</th>
<th>RS (m)</th>
<th>LS (m)</th>
<th>LW (rad/s)</th>
<th>RW (rad/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.50</td>
<td>-0.50</td>
</tr>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
<td>-0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>3</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.50</td>
<td>-0.50</td>
</tr>
<tr>
<td>4</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
<td>-0.50</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>7</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>8</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Table 2.2: Training dataset for the first and simplest case. With $H_0 = 3$, $H_1 = 5$ and $H_2 = 2$.

#### 2.3.3 Virtual Car-Like Robot

This work also tested a MLP-BP model to control a virtual car-like robot from the *iRobot* matlab toolbox, provided by the United States Naval Academy (USNA), (USNA 2011). This toolbox provides a virtual environment and an interface to test control algorithms on a differentially steered robot on various maps with different obstacles and different combinations of distance or proximity sensors.

As previously stated, this virtual environment provides an interface to control a differentially steered virtual robot. This robot is controlled by changing the angular speed in $\text{rad/s}$ of the two wheels, Right Wheel (RW) and Left Wheel (LW). Also this work used three proximity sensors with virtual 3 meters range to provide input to the MLP. The sensors are Front-Sensor (FS), Left-Sensor (LS) and Right-Sensor (RS).

We tested the MLP model with three different datasets, increasing the hyperparameters to evaluate the behavior of the robot as we increased the network architecture’s complexity. The cases are comprised of three tables with five columns each, as seen in Table 2.2. The first case is very simple with only eight conditions provided to train the network, which means the network must be able to devise a knowledge representation on how to behave with cases not previously trained from these simple constraints.

The second case has a greater complexity with 18 conditions, as seen in Table 2.3. Also, as mentioned in Table 2.3, this case required the same amount of neurons in the hidden layer, meaning that the first architecture could still be used for a more complex case.
Table 2.3: Training dataset for the second case. With $H^0 = 3$, $H^1 = 5$ and $H^2 = 2$. 

<table>
<thead>
<tr>
<th>Condition</th>
<th>FS (m)</th>
<th>RS (m)</th>
<th>LS (m)</th>
<th>LW (rad/s)</th>
<th>RW (rad/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>2</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>3</td>
<td>2.00</td>
<td>0.00</td>
<td>1.00</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>4</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>1.00</td>
<td>2.00</td>
<td>0.20</td>
<td>-0.10</td>
</tr>
<tr>
<td>6</td>
<td>2.00</td>
<td>1.00</td>
<td>2.00</td>
<td>0.20</td>
<td>-0.10</td>
</tr>
<tr>
<td>7</td>
<td>0.00</td>
<td>2.00</td>
<td>0.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>8</td>
<td>1.00</td>
<td>2.00</td>
<td>0.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>9</td>
<td>2.00</td>
<td>2.00</td>
<td>1.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>11</td>
<td>1.00</td>
<td>0.00</td>
<td>2.00</td>
<td>0.20</td>
<td>-0.20</td>
</tr>
<tr>
<td>12</td>
<td>2.00</td>
<td>0.00</td>
<td>2.00</td>
<td>0.20</td>
<td>-0.20</td>
</tr>
<tr>
<td>13</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>14</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>15</td>
<td>2.00</td>
<td>1.00</td>
<td>1.00</td>
<td>-0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>16</td>
<td>0.00</td>
<td>2.00</td>
<td>10.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>17</td>
<td>1.00</td>
<td>2.00</td>
<td>2.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>18</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>0.50</td>
<td>0.50</td>
</tr>
</tbody>
</table>

The third and most complex case with 27 conditions to train the MLP-BP. This dataset required a bigger architecture than the previous datasets, with double the amount of neurons in the hidden layer ($H^1 = 10$)
2.4. RESULTS

<table>
<thead>
<tr>
<th>Condition</th>
<th>FS (m)</th>
<th>RS (m)</th>
<th>LS (m)</th>
<th>LW (rad/s)</th>
<th>RW (rad/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.30</td>
<td>0.30</td>
</tr>
<tr>
<td>2</td>
<td>0.50</td>
<td>0.50</td>
<td>1.00</td>
<td>0.20</td>
<td>-0.20</td>
</tr>
<tr>
<td>3</td>
<td>0.50</td>
<td>0.50</td>
<td>2.50</td>
<td>0.20</td>
<td>-0.20</td>
</tr>
<tr>
<td>4</td>
<td>0.50</td>
<td>1.00</td>
<td>0.50</td>
<td>0.30</td>
<td>0.30</td>
</tr>
<tr>
<td>5</td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
<td>0.20</td>
<td>-0.20</td>
</tr>
<tr>
<td>6</td>
<td>0.50</td>
<td>1.00</td>
<td>2.50</td>
<td>0.20</td>
<td>-0.20</td>
</tr>
<tr>
<td>7</td>
<td>0.50</td>
<td>2.50</td>
<td>0.50</td>
<td>0.30</td>
<td>0.30</td>
</tr>
<tr>
<td>8</td>
<td>0.50</td>
<td>2.50</td>
<td>1.00</td>
<td>0.30</td>
<td>0.30</td>
</tr>
<tr>
<td>9</td>
<td>0.50</td>
<td>2.50</td>
<td>2.50</td>
<td>0.30</td>
<td>0.30</td>
</tr>
<tr>
<td>10</td>
<td>1.00</td>
<td>0.50</td>
<td>0.50</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>11</td>
<td>1.00</td>
<td>0.50</td>
<td>2.50</td>
<td>0.20</td>
<td>-0.20</td>
</tr>
<tr>
<td>12</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>13</td>
<td>0.50</td>
<td>1.00</td>
<td>0.50</td>
<td>0.30</td>
<td>-0.30</td>
</tr>
<tr>
<td>14</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.30</td>
<td>0.30</td>
</tr>
<tr>
<td>15</td>
<td>1.00</td>
<td>1.00</td>
<td>2.50</td>
<td>0.20</td>
<td>-0.20</td>
</tr>
<tr>
<td>16</td>
<td>1.00</td>
<td>2.50</td>
<td>0.50</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>17</td>
<td>1.00</td>
<td>2.50</td>
<td>1.00</td>
<td>0.30</td>
<td>0.30</td>
</tr>
<tr>
<td>18</td>
<td>1.00</td>
<td>2.50</td>
<td>2.50</td>
<td>0.20</td>
<td>-0.20</td>
</tr>
<tr>
<td>19</td>
<td>2.50</td>
<td>0.50</td>
<td>0.50</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>20</td>
<td>2.50</td>
<td>0.50</td>
<td>1.00</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>21</td>
<td>2.50</td>
<td>0.50</td>
<td>2.50</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>22</td>
<td>2.50</td>
<td>1.00</td>
<td>0.50</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>23</td>
<td>2.50</td>
<td>1.00</td>
<td>1.00</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>24</td>
<td>2.50</td>
<td>1.00</td>
<td>2.50</td>
<td>0.20</td>
<td>-0.20</td>
</tr>
<tr>
<td>25</td>
<td>2.50</td>
<td>2.50</td>
<td>0.50</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>26</td>
<td>2.50</td>
<td>2.50</td>
<td>1.00</td>
<td>-0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>27</td>
<td>2.50</td>
<td>2.50</td>
<td>2.50</td>
<td>0.30</td>
<td>0.30</td>
</tr>
</tbody>
</table>

Table 2.4: Training dataset for the third and most complex case. With $H^0 = 3$, $H^1 = 10$ e $H^2 = 2$.

2.4 Results

2.4.1 XOR Operation

The binary code for the XOR operation, that is generated from this implementation as seen in Figure[2.1] resulted in 6.672 KBytes of program memory occupation (equivalent to 2.6%), a pretty compact solution if compared to the maximum program memory available for the ATMega-2560 and also comparing to the 5.904KBytes presented in (Mancilla-David et al. 2014) for and MLP implementation without the BP training, considering that these 2.6% already include the training algorithm.
The information presented at Table 2.5 show that the values obtained up to \( H^1 = 38 \) neurons in the hidden layer. The \( H^1 \) parameter is limited by the available memory in the \( \mu C \) used, ATMega-2560. However, this value is quite reasonable for most real-time applications in robotics and industrial automation. Another noticeable result is that the processing times are also reasonable. Analyzing the network without the training time results (EM, BPM-1 and BPM-2) we can see that the iteration for a batch mode of \( N = 4 \) samples takes 42.91ms for the worst case, \( H^1 = 38 \). If you analyze the duration of each iteration including the network being trained on the online mode it takes 69.88ms, again in the worst case of \( H^1 = 38 \). This shows that the implementation here presented is indeed suitable for commercial applications in fields like industrial automation, robotics, automotive industry, and others.

An important result of these tests is the behavior of these fitted curves in Figures 2.5 and 2.6. The processing time grows linearly with the number of neurons in the hidden layer (\( H^1 \)). This result is quite significant since you can estimate if a certain \( \mu C \) can be used with this implementation and also if the network architecture that was chosen will fit or not in the \( \mu C \). This result can be used as a reference by other groups that refute the usage of MLP-BP applications on \( \mu Cs \).

<table>
<thead>
<tr>
<th>( H^1 )</th>
<th>( t_{FFM-1} ) (ms)</th>
<th>( t_{FFM-2} ) (ms)</th>
<th>( t_{EM} ) (ms)</th>
<th>( t_{BPM-1} ) (ms)</th>
<th>( t_{BPM-2} ) (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.59</td>
<td>0.31</td>
<td>0.06</td>
<td>0.46</td>
<td>1.30</td>
</tr>
<tr>
<td>4</td>
<td>4.28</td>
<td>0.61</td>
<td>0.06</td>
<td>0.66</td>
<td>2.23</td>
</tr>
<tr>
<td>6</td>
<td>6.21</td>
<td>0.77</td>
<td>0.06</td>
<td>0.88</td>
<td>3.22</td>
</tr>
<tr>
<td>8</td>
<td>8.11</td>
<td>1.00</td>
<td>0.06</td>
<td>1.11</td>
<td>4.11</td>
</tr>
<tr>
<td>10</td>
<td>10.50</td>
<td>1.82</td>
<td>0.06</td>
<td>1.70</td>
<td>6.14</td>
</tr>
<tr>
<td>12</td>
<td>12.30</td>
<td>2.02</td>
<td>0.06</td>
<td>1.91</td>
<td>6.90</td>
</tr>
<tr>
<td>14</td>
<td>14.50</td>
<td>1.49</td>
<td>0.06</td>
<td>1.80</td>
<td>7.06</td>
</tr>
<tr>
<td>16</td>
<td>16.30</td>
<td>1.72</td>
<td>0.06</td>
<td>2.03</td>
<td>8.02</td>
</tr>
<tr>
<td>18</td>
<td>18.60</td>
<td>2.57</td>
<td>0.06</td>
<td>2.65</td>
<td>10.60</td>
</tr>
<tr>
<td>20</td>
<td>20.70</td>
<td>2.71</td>
<td>0.06</td>
<td>2.47</td>
<td>11.20</td>
</tr>
<tr>
<td>22</td>
<td>22.60</td>
<td>2.95</td>
<td>0.06</td>
<td>3.07</td>
<td>12.40</td>
</tr>
<tr>
<td>24</td>
<td>25.20</td>
<td>3.15</td>
<td>0.06</td>
<td>3.36</td>
<td>13.60</td>
</tr>
<tr>
<td>26</td>
<td>27.30</td>
<td>3.31</td>
<td>0.06</td>
<td>3.71</td>
<td>14.70</td>
</tr>
<tr>
<td>28</td>
<td>29.30</td>
<td>3.51</td>
<td>0.06</td>
<td>3.95</td>
<td>16.50</td>
</tr>
<tr>
<td>30</td>
<td>30.90</td>
<td>3.67</td>
<td>0.06</td>
<td>4.03</td>
<td>16.20</td>
</tr>
<tr>
<td>32</td>
<td>32.90</td>
<td>3.76</td>
<td>0.06</td>
<td>3.83</td>
<td>15.80</td>
</tr>
<tr>
<td>34</td>
<td>34.60</td>
<td>3.36</td>
<td>0.06</td>
<td>4.05</td>
<td>16.70</td>
</tr>
<tr>
<td>36</td>
<td>36.50</td>
<td>4.22</td>
<td>0.06</td>
<td>4.82</td>
<td>19.40</td>
</tr>
<tr>
<td>38</td>
<td>38.50</td>
<td>4.41</td>
<td>0.06</td>
<td>5.17</td>
<td>21.80</td>
</tr>
</tbody>
</table>

Table 2.5: Duration of execution for each module of the implementation of the MLP-BP with \( P = H^0 = 2, M = H^2 = 1 \) and sigmoid activation function.

Also we compared the MSE curve when varying the number of neurons in the hidden layer (\( H^1 \)), as shown in Figure 2.7. This shows that for 2 and 20 iterations the MSE was
really close to zero, and that with 38 neurons, 80 iterations were needed. According to Table 2.5 it took approximately 8.28 seconds to fully train the XOR case with 38 neurons.

### 2.4.2 Virtual Car-like Robot

The first validation test-case showed small MSEs, close to 1%, with the architecture described in the methodology section, three inputs and two outputs with only five neurons in the hidden layer, \( H_1 = \). Since it was trained with a maximum distance of 1.0m, the robot shows a great behavior close to obstacles, but on big empty spaces with greater than 1.0m distances between obstacles the robot drifts and spins around the same spot until the test is restarted. This showed us that this training dataset was too simple and required more complex data. The test results for this dataset are shown in Figure 2.8.

The second dataset test showed much better results, with the virtual robot being able to react quickly to farther obstacles and also correcting its trajectory faster. The robot was able to run along the map borders smoothly and keep a safe distance between parallel obstacles on the center of the map. However, the virtual robot still collided with the obstacles and sometimes it would collide and drag itself along the borders of bigger obstacles, as shown in Figure 2.9.

The dataset for the third validation case resulted with a higher MSE of 2.4%. The car-like robot was faster than the previous two tests and was able to perform a fast reaction to abrupt changes in proximity to obstacles, as shown in Figure 2.10. However, this dataset training made the robot show a behaviour that can be interpreted as overfitting, since sometimes it would react too quickly and start to spin around itself after detecting some obstacles ahead.

The overall results shows that the datasets had not enough data and this required some more tests with even more virtual sensors. We included the angle between the Front-Sensor (FS) and the virtual horizontal axis of the test map. Also we performed training tests with two and three more hidden layers \( L = 3, 4 \) and also hundreds of hidden neurons and thousands of iterations, none of which improved the results, actually preventing the Neural Network from reaching training error convergence lower than 20%.

The generated binary code from the source in C for the best dataset \( (3^{rd}) \) had 3.07KB of memory size, which amounts to 1.2% of the 256KB of program memory of the ATMega2560. This code size is quite compact compared to a similar model implementation seen in (Mancilla-David et al. 2014). The same variables and testing criteria used for the XOR operation were used in this case, with 16Mhz of clock frequency.

The timing results for the virtual car-like robot were expected, being smaller than the XOR operation that has a simpler architecture with less inputs and outputs \( (H_0 = 1 \text{ and } H_2 = 1) \) and also a wide range for \( H_1 \), Table 2.5. Also, as this implementation does not perform online real-time embedded training, only the FFM-1, FFM-2 and FFM-3 modules were used, being FFM-2 and FFM-3 for the cases with more than one hidden layer.

The timing results for the first and second datasets were the same. This is expected, since the only factor to influence the execution time is size of the synaptic weight matrices \( H_1 \) and \( H_2 \) that depends only on the amount of inputs, neurons and the amount of samples. Since this was not a batch classification, but a single sample, online classifica-
tion, the input size remained as 1 and unchanged throughout all the three tests.

The third case presented a fast timing result, but close to two times the timing results for the first and second cases. This is expected and simply reiterates what was shown in Figures 2.5 and 2.6 that the execution time grows linearly with the hyperparameters or more specifically the amount of neurons for this implementation proposal.

It is important to notice that the execution time results shown in Table 2.6 relate to what is found in the field’s literature, that more hidden layers do not necessarily improve the training MSE and can actually make it worse, as seen in (McDanel et al. 2017). In the cases where we trained the robot with more than a single hidden layer it’s ability to avoid obstacles did not improve and the best MSE was slightly higher than 20%.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>$H^1$</th>
<th>$t_{FFM-1}$ (ms)</th>
<th>$t_{FFM-2}$ (ms)</th>
<th>$t_{Total-FFM-k}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1.28</td>
<td>0.52</td>
<td>1.80</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1.28</td>
<td>0.52</td>
<td>1.80</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>2.63</td>
<td>0.88</td>
<td>3.51</td>
</tr>
</tbody>
</table>

Table 2.6: Execution times measured with HIL for the implementation proposal in the virtual car-like robot validation.

### 2.5 Comparison with the State-of-the-Art

The work McDanel et Al (McDanel et al. 2017) shows two implementations of an MLP-BP with single and dual hidden layers and their respective execution times for a classification with the results shown in Table 2.7. In (Gural & Murmann 2019), the authors were able to embed a full MNIST-10 classification model using CNNs under 2KB of SRAM being used, also the inference times were in the order of 640 ms per input sample.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Model</th>
<th>Time (ms)</th>
<th>Memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(McDanel et al. 2017)</td>
<td>MLP-1</td>
<td>17.35</td>
<td>14.73</td>
</tr>
<tr>
<td>(Gural &amp; Murmann 2019)</td>
<td>AvgPool + CNN + MaxPool + MLP</td>
<td>684</td>
<td>8.46</td>
</tr>
<tr>
<td><strong>Proposal</strong></td>
<td>MLP-2</td>
<td><strong>69.88</strong></td>
<td><strong>6.67</strong></td>
</tr>
</tbody>
</table>

Table 2.7: ANN results for the state-of-the-art works.*Time of inference, not taking the EM-k and BPM-k modules time into account.

It’s important to notice that not of the times presented in Table 2.7 take only the inferencing time. The implementation for McDanel et Al (McDanel et al. 2017) trained the model offline using MatLab. For the Gural & Murmann (Gural & Murmann 2019) paper, the training was executed offline as well on a Jupyter Notebook running Tensorflow-Keras models and optimizations before porting the synaptic weights and kernels to the µC.

It is possible to analyze this proposal, regarding how this system would behave if it had to support the same hyperparameters that some of these state-of-the-art applications
have. The work presented by McDanel et al. (McDanel et al. 2017) uses an MLP ANN with a single layer with 100 artificial neurons. It is important to notice that the proposal in their work is implemented on a Von Neumann architecture-based CPU, with 32-bits and running at 400 MHz clock speed. This same work also presents another implementation with a two-layer MLP with 200 artificial neurons. Analyzing the $t_{FFM-1}$, $t_{FFM-2}$, $t_{BPM-1}$ and $t_{BPM-2}$ fitted curves from Table 2.5 we obtain four predictive equations that define how much time is required to process this MLP implementation.

Equation 2.16 shows us again that this work’s proposal has a linear relationship between inference time and hyperparameters, specifically artificial neurons. Evaluating this Equation 2.16 with a number of 100 neurons it is observed 1.18s of feedforwarding time for the first layer.

$$t_{FFM-1}(H^1) = 11.6 \times H^1 + 20.52 \quad (2.16)$$

Taking into regard the same amount of neurons for the Equation 2.17 we get 124.49ms of feedforwarding time for the second layer. This amount to 1.30s of total inferencing time with 100 neurons.

$$t_{FFM-2}(H^2) = 1.22 \times H^2 + 2.49 \quad (2.17)$$

If the embedded training is to be also considered with this amount of neurons, the time duration for this process increases to 1.44s, by evaluating the Equation 2.18.

$$t_{BPM-1}(H^2) = 1.41 \times H^2 + 2.72 \quad (2.18)$$

Also evaluating the Equation 2.19 for the second layer, considering two hidden layers in training the time duration increases to 2.06s.

$$t_{BPM-2}(H^1) = 6.03 \times H^1 + 10.94 \quad (2.19)$$

This analysis shows that the modular implementation of a MLP here presented performs on compatible training and inferring times.

### 2.6 Conclusions

This work presents an implementation proposal of an MLP artificial neural network with embedded BP training for an 8-bit µC. Results and implementation details were presented for this proposal for an ATMega-2560 µCs. Also, the validation results of the embedding of this proposal were presented using a HIL simulation strategy. Finally, the results show that the execution times and memory occupation of the implementation were compatible with application requirements seen in the industry, seen that these requirements fall under hundreds of milliseconds.
Figure 2.5: a) Execution time of FFM – 1 by number of neurons in the hidden layer, $H^1$. b) Execution time of FFM – 2 by the number of neurons in the hidden layer, $H^1$. 
Figure 2.6: a) Execution time of BPM – 1 by the number of neurons in the hidden layer, $H^1$. b) Execution time of BPM – 2 by the number of neurons in the hidden layer, $H^1$. 

2.6. CONCLUSIONS
Figure 2.7: Mean Squared Error curve fitting comparison with $H^1 = \{2, 20, 38\}$ neurons in the hidden layer.

Figure 2.8: Virtual map with the resulting trajectory of the virtual car-like robot for the 1st training dataset.
2.6. CONCLUSIONS

Figure 2.9: Virtual map with the resulting trajectory of the virtual car-like robot for the $2^{nd}$ training dataset.

Figure 2.10: Virtual map with the resulting trajectory of the virtual car-like robot for the $3^{rd}$ training dataset.
Chapter 3

ANNs Optimization Using Program Memory

3.1 Introduction

The application of Artificial Neural Networks (ANNs) in Microcontrollers (µCs) is no longer considered as a novelty to the literature, the consumer market nor the industry. In consumer electronics, smartphones use ANNs to perform facial recognition and virtual assistants use voice recognition to assist in day-to-day activities. In the industry, temperature controllers for furnaces (Yu & Chen 2010) or even humidity controllers in greenhouses are based on Artificial Intelligence (AI) techniques with ANNs (Fuady et al. 2017). In addition, several other studies demonstrate the demand for Machine Learning (ML) techniques in low cost and low power consumption devices (Sarwesh et al. 2017, Qin & McCann 2017, Misra & Saha 2010).

Low cost and low energy consumption Systems on Chip (SoCs), such as microcontrollers, have been used for several different applications and areas such as industrial automation, control, monitoring/measurement, etc. It can be affirmed that there is a growing demand for the use of these devices, especially in emerging areas such as Internet-of-Things (IoT), Smart Grid and Machine-to-Machine (M2M). Although microcontrollers are devices with medium to low processing power, they have the advantage of costing a fraction of the price of more robust systems and requiring significantly less power when compared to other platforms, which makes it possible and advantageous to use them in various IoT applications.

In work (Saad Saoud & Khellaf 2011b), the authors chose a chemical process that is naturally modeled with a non-linear system and embedded an ANN-MLP on a PIC16F876A µC. This implementation was able to approximate the results obtained in a conventional computer application in a few iterations of the BP algorithm with the advantage of very low cost when compared to an equivalent implementation in a personal computer. Applications with similar approaches and conclusions can be seen in the works (Cotton et al. 2008, Mancilla-David et al. 2014, Oyamada et al. 2008). This papers opens the discussion in this work for wether µCs are suitable or not to implement AI techniques in real-time.

In (Bitye Dimithe et al. 2018), the authors used µCs ATMega-2560 in a heterogeneous
architecture with other embedded devices (Raspberry Pi, Odroid and a Jetson-TX2) to distribute the processing load of the control of an Unmanned Aerial Vehicle (UAV), control of a mobile ground robot and real-time image detection and recognition processing of a large amount of data.

In (Vilar et al. 2018), demonstrated a strategy for implementing MLP ANNs in a 8-bit µC, the ATMega-2560. This strategy aimed to enable the use of multi-layer perceptron trained with the backpropagation algorithm in low-power embedded systems. Timing results and the flexibility of the studied strategy with different ANN topologies were discussed. In addition, a linear relationship between the processing time of the training and classification steps and the number of ANN neurons has been established.

In the same application field, (Farooq et al. 2010), the authors implement an obstacle avoidance controller for a car-type robot using ANNs on a microcontrolled platform. It was demonstrated that in the AT-89C52 µC used, it was possible to implement up to 20 neurons in the hidden layer for an MLP-type neural network. The robot showed to be able to avoid collisions with obstacles in most cases and it has been argued that a greater number of sensors and a larger ANN topology may improve the results.

The authors from (Chen & Ran 2019, Dey & Mukherjee 2018, Huang et al. 2017, Sen & Shen 2019) discussed the importance of Edge Computing (also known as Fog Computing) applications. These studies revealed the need for an additional computing layer required to handle the real-time processing of massive amounts of high-frequency sensor data, mainly for IoT devices. It was also outlined how machine learning and Deep Learning (DL) can act as solutions for the treatment and management of such type of data.

In (Meyer et al. 2019) the authors were also interested in using distributed processing on microcontrollers, by using Convolutional Neural Networks (CNN) in an application for disaster monitoring. A network of µCs used CNN for the classification of seismic events captured by multiple geophones. It was noted in this work that CNNs demand high memory costs for their implementation and that the commercially available µCs are designed with small program and work memories for these type of applications. This fact makes it difficult to implement the system proposed by the authors. Ultimately, the solution given was to distribute the computational operations among several µCs, reducing the necessary memory usage for each individual device.

The work (Karvelis et al. 2018) deviated from distributed computing in the edge and implemented a climate prediction system for ships based on Linear Regression (LR) using microcontrollers. The authors stressed the low frequency of operation and limited memory as obstacles to the use of AI techniques µCs and sought techniques to enable the use of their solution on an ESP32, benefiting from its low cost and reduced size.

The work demonstrated in (McDanel et al. 2017) goes beyond the simple application of ML on µCs, performing optimizations in the model before anything. The authors used a 32MHz 32-bit Intel Quark x1000 System on Chip with 24 kilobytes of data memory to train a Deep Neural Network for image recognition using the MNIST (LeCun & Cortes 2010) and CIFAR-10 (Krizhevsky et al. 2008) datasets. Among the techniques used, the authors binarized the synaptic weights of the network in different architectures and performed certain steps of the forward propagation or convolution of the network layers in a single execution cycle. These techniques implemented the image recognition using only
3.2. MULTI-LAYER PERCEPTRON (MLP)

15 kilobytes of data memory for two datasets known in the literature for the large number of neurons needed to maintain acceptable classification accuracy. In addition, the authors configured the network layers in order to minimize the program memory occupation by performing a multiplication operation of the input image with the synaptic weights in the case of an MLP or a multiplication and sum of the convolution of a kernel with the input image in the case of a CNN.

As already discussed in the literature, µCs are devices with low processing power and data memory (RAM), on the other hand they also have low power consumption which is an important feature in several applications. Although the data memory is usually small these devices also have a program memory that can be used for data storage and, usually, have a larger storage capacity than data memory. Thus, the present work proposes an MLP ANN strategy embedded in an 8-bit µC whose synaptic weights are stored in the program memory. This strategy allows the use of larger ANN structures in embedded systems with a limited data memory size.

The implementation was developed targeting a µC ATmega-2560 microcontroller and the embedded MLP was trained to classify the digits from the MNIST Dataset (LeCun & Cortes 2010). Hardware-in-the-Loop (HIL) was used for analyzing the forward-propagation implementation of the proposed MLP. Finally, processing time and memory occupation results of the implementation were then compared with previous works and works from the literature in an effort to demonstrate how an increase in the network architecture size with different topologies for the same Dataset was possible.

3.2 Multi-layer Perceptron (MLP)

ANNs are biomimetic systems that use the high power of generalization, the ability to model a knowledge representation in order to carry out the activities and functions of brain structures (Haykin 1999). These structures are defined based on sets of basic processing units called Artificial Neurons.

The artificial neuron has a well-defined structure, as can be seen in Figure 3.1. It is formed by three basic elements, a set of synapses, a accumulator and a activation function.

In the $i$-th artificial neuron, the synapse set consists of $P$ synapse input signals $(x_j)$ connected to $P$ weights or forces $(w_{ij})$ corresponding to each input. The accumulator
is used to add the input signals \((x_j)\) multiplied by the weights \((w_{ij})\), an operation called calculation of the induced local field or activation potential, which produces \((v_i)\). In turn, the activation function has the role of limiting the amplitude of the neuron output, keeping the neuron signal within the required range. The standard model of neurons found in the literature, (Haykin 1999), also includes a bias which is a constant input required to change the output of the activation function, fine-tuning the result from the neuron given a set of inputs.

The architecture used in this work is a Multi Layer Perceptron which is basically an arrangement of multiple artificial neurons. This arrangement consists of at least three layers: an input layer, a hidden layer and an output layer. Since MLPs are fully connected, each node in one layer connects with a certain weight \(w_{ij}\) to every neuron in the following layer. In addition, neurons have also nonlinear and differentiable activation functions which are used to introduce non-linearity into the output of a neuron. A generic architecture of a MLP can be seen in Figure 3.2.

![Figure 3.2: Architecture of a Multi-Layer Perceptron.](image)

### 3.3 \(\mu\)C Memory Overview

Figure 3.3 shows the memory layout in the \(\mu\)C used. The MLP-ANN weights are stored in the program memory.

AVR microcontrollers, the \(\mu\)Cs family to which the ATmega-2560 belongs, have a modified Harvard hardware architecture. This architecture differs from the Von Neumann architecture since both data and program memories have their own buses, allowing the execution of arithmetic operations and fetch of instructions independently. In addition, these \(\mu\)Cs are compiled using Reduced Instruction Set Computer (RISC), in which most instructions are executed in a single clock cycle (Jamil 1995).

The methodology used in (McDanel et al. 2017) is not feasible for 8-bit \(\mu\)Cs as the ATmega-2560, considering ANNs of larger topologies, (LeCun & Cortes 2010) that require hundreds of neurons. Since they used a Von Neumann-based architecture Intel
3.3. µC MEMORY OVERVIEW

Figure 3.3: Arrangement of memory sections in the program memory of an AVR µC.

Quark X1000 CPU, with 32-bits and 400MHz of clock the storage of synaptic weights in the program memory is easier. The Von Neumann architecture shares the bus between program memory and work memory, allowing for the developer to work with data stored on either one of those memories without having to rely on language or manufacturer-specific features for Harvard architecture-based µCs.

The C programming language compilation was performed using the avr-gcc compiler. For µCs, the program written in this language follows the memory layout described in Figure 3.3.

The .data section is the first one (bottom) in the program memory. This section is responsible for storing all variables initialized statically in the source code. Next, is the .bss section where the uninitialized variables are stored. Finally, the .text section stores all the machine instructions that make up the source code itself.

It is important to point out that in the Harvard architecture, as the data and program memory addresses are not shared, the compiler must copy the variables from the .data and .bss sections to the data memory in order to avoid additional search instructions during execution. This architecture feature makes it impossible to work with data larger than 8 kilobytes or data that is above the $2^{16}$ bits address.
The AVR µCs have some specific instructions in the Assembly language that allow the user to read data from flash memory at runtime, the function `pgm_read()`. This allows the maximum use of 256 kilobytes of program memory, not exceeding the memory that stores the machine instructions in the `.text` section. This type of functionality is widely used in the industry for storing strings for interactive menus, encryption keys, static audios and any type of data that will be used frequently but will not be changed during execution time.

Another feature from the architecture of this µC is the lack of hardware memory management which prevents the sections from Figure 3.3 from overwriting each other (Corporation n.d.).

### 3.4 Implementation Proposal

The block diagram from Figure 3.4 describes the modules responsible for implementing the µCs MLP-ANN here proposed. The implementation has as a direct reference to (Vilar et al. 2018), following its basic coding structure of an Fully-Connected (FC) MLP-ANN in a matrix notation. This approach simplifies and modularizes direct propagation operations (feedforward). As depicted in Figure 3.4 there are three modules: one main module (FFM – k) responsible for feedforward operations which is replicated for each network layer and two communication modules responsible for the HIL test system. The implementation proposed here uses two layers for MLP-ANN (k = 2), but can be extended to more hidden layers with different amounts of neurons.

#### 3.4.1 Variables Description

This implementation is basically formed by three main variables that are passed by reference between the modules. The first one is the input vector of the hidden layers $y^k$ which can be expressed as

$$y^k = [y_1^k, y_2^k, \ldots, y_N^k]$$

(3.1)
where \( N \) is the number of MLP inputs. Secondly, the weight matrix of the \( k \)-th layer \( W^k \) is defined as

\[
W^k = \begin{bmatrix}
w^k_{00} & \cdots & w^k_{10} & \cdots & w^k_{1H^k-1} \\
\vdots & \ddots & \vdots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
w^k_{j0} & \cdots & w^k_{jh} & \cdots & w^k_{jH^k-1} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
w^k_{H^k0} & \cdots & w^k_{H^kh} & \cdots & w^k_{H^kH^k-1}
\end{bmatrix},
\]

where \( H^k \) represents the number of neurons in the \( k \)-th layer and \( w^k_{ij} \) is the weight associated with the \( i \)-th neuron, from the \( j \)-th input of the \( k \)-th layer. Lastly, the output vector \( y^L \) is identified as

\[
y^L = [y^L_1, y^L_2, \ldots, y^L_M],
\]

where \( M \) is the number of output neurons. For a two-layer network, \( W^1 \) represents the weight matrix of the hidden layer \((H^1 = P)\), where \( P \) is the number of neurons in the hidden layer and \( W^2 \) the weight matrix of the output layer \((H^2 = M)\).

In addition to these matrices, other ones were created in order to propagate information between the modules and carry out all the necessary operations of the MLP-ANN. It is important to highlight that this proposal takes into account several important aspects related to the optimization of the code size (reduction of program memory occupation), optimization of variables usage (reduction of RAM memory occupation) and optimization of feedforward operations to reduce the computational cost and processing time. As presented in (Mann 2015, AVR 2003, AVR 2011), the optimization of the code size and the optimization of variables also contribute to a computational cost reduction.

### 3.4.2 Feedforward Module - (FFM - \( k \))

This module has the function of performing the MLP-BP direct propagation operation in each \( k \)-th layer. For each \( k \)-th FFM - \( k \) (\( k \)-th layer) the following expression is performed

\[
y^k = \varphi \left( W^k \times y^{k-1} \right)
\]

in which \( y^{k-1} \) is the output from the previous layer, \( Y^0 \) is the MLP input and \( \varphi (\cdot) \) is the activation function for the \( k \)-th layer. It is observed that the outputs associated with the intermediate layers must also be stored during the execution of the algorithm as they will need to be used in the following feedforward step of the next layer. Algorithm \( \text{Algorithm} \) shows the pseudocode of the FFM - \( k \) module in detail. The function prodMatrix() implements the product of a matrix and a vector, in this case, \( W^k \) and \( y^{k-1} \) and stores the result in \( y^k \). The function actFun() applies the activation function element-wise in \( y^k \) and overwrites \( y^k \) with the result.

Finally, the Serial Send Module (SSM) sends \( y^L \) to the Computer, a Personal Computer (PC) in the HIL setting, which will record the classification performed by the embedded ANN.
3.5 Results

The validation of the presented implementation was performed using software implemented in the C and the Atmel Studio 7 development environment with the `avr-gcc` compiler version 5.4.0. The µC used is a ATmega-2560, an 8-bit µC that works at the speed of 1 MIPS/16MHz and has 256 kilobytes of program memory (flash memory) and 8 kilobytes of data memory or RAM. The Arduino Mega is a development kit that combines in a single hardware, a 2560 ATmega chip and a recording circuit, easying the development process.

The ANN proposed is an MLP with a single hidden layer and the output layer. The ANN topology with a single hidden layer $H^k$ was chosen due to its acceptance in the literature as the most efficient architecture (McDanel et al. 2017). The dataset used was MNIST, (LeCun & Cortes 2010), which consists of 70,000 images with $28 \times 28$ pixels in grayscale. These images consist of handwritten digits from 0 to 9.

The artificial neural network was trained externally to the µC using Matlab. Matlab is a multi-paradigm numerical computing environment that allows the user to perform many simulations, calculations and sort. This tool provides an environment for simulations from simple wave functions to complex Aerospace, Artificial Intelligence, Communications systems and fluid flow simulations, a widely-used toolbox in the whole industry.

The Neural Network Pattern Recognition Tool Toolbox from Matlab was used to train the µC, with $P = 50$ neurons in the hidden layer ($H^1$) and $M = 10$ neurons in the output layer ($H^2 = H^L$). The activation function for neurons in the hidden layer was the hyperbolic tangent while Softmax was used in activation of the output layer.

The test consisted of performing the same operations from Algorithm 7 in a loop in Matlab and sending a vector of 785 elements ($28 \times 28$ grayscale pixels and the bias value) with values between 0-255 via the Universal Serial Asynchronous Receiver Transmitter (USART) communication protocol to the µC. This received vector was handled by the Serial Data Receiving Module, Serial Send Module (SRM), responsible for receiving the vector byte by byte. Then, the process described in Figure 3.4 is followed and, finally, the vector $y^L$ is sent back to the program executed in Matlab which evaluates and records the output classification of the embedded ANN.

Throughout the testing and evaluation process, the execution time of each module $FFM - k$ was evaluated using an oscilloscope. Observed times are $t_{FFM-1}$ and $t_{FFM-2}$, which are the feedforward times for each layer, respectively. The measurement of the execution time of each module was performed by raising the logical level of a µC output pin when starting the module’s execution and lowering the logical level of that same pin at the end of the module’s execution. The timing results can be seen in Table 3.1.

Time validation, as shown in Table 2.5, was performed with values of $P = \{10, 20, 30, 40, 50\}$,
3.5. RESULTS

Table 3.1: Measured processing times of the MLP-ANN implementation with $M = 10$ and $N = 785$.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$t_{FFM-1}$ (ms)</th>
<th>$t_{FFM-2}$ (ms)</th>
<th>Total (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>116.7</td>
<td>109.9</td>
<td>226.6</td>
</tr>
<tr>
<td>20</td>
<td>233.5</td>
<td>267.9</td>
<td>501.4</td>
</tr>
<tr>
<td>30</td>
<td>351.7</td>
<td>474.3</td>
<td>826.0</td>
</tr>
<tr>
<td>40</td>
<td>470.2</td>
<td>728.4</td>
<td>1198.6</td>
</tr>
<tr>
<td>50</td>
<td>587.6</td>
<td>1030.9</td>
<td>1618.5</td>
</tr>
</tbody>
</table>

Table 3.2: Hardware occupancy results for data and program memories of the MLP-ANN implementation with $M = 10$ and $N = 785$.

<table>
<thead>
<tr>
<th>$P$</th>
<th>.text (kilobytes)</th>
<th>.bss (kilobytes)</th>
<th>FLASH (kilobytes)</th>
<th>SRAM (kilobytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>37.49</td>
<td>0.094</td>
<td>38.28</td>
<td>0.88</td>
</tr>
<tr>
<td>20</td>
<td>70.83</td>
<td>0.134</td>
<td>71.62</td>
<td>0.92</td>
</tr>
<tr>
<td>30</td>
<td>104.42</td>
<td>0.174</td>
<td>105.21</td>
<td>0.96</td>
</tr>
<tr>
<td>40</td>
<td>138.03</td>
<td>0.214</td>
<td>138.82</td>
<td>1.00</td>
</tr>
<tr>
<td>50</td>
<td>171.64</td>
<td>0.254</td>
<td>172.43</td>
<td>1.04</td>
</tr>
</tbody>
</table>

where 50 was the largest possible number of neurons for implementing with this µC due to memory restrictions. As its data memory is limited to 8 kilobytes and the weights $W_1$ and $W_2$ are stored in a 4 Bytes floating point variable each, the maximum size possible would be $P = 2$. This is due to the fact that there are $N = 785$ inputs for $P = 2$ neurons in the hidden layer, which implies $(P \times N) = 2 \times 785 \times 4\text{Bytes} = 6,2\text{kilobytes}$. That is, 76.66% of data memory occupancy would be occupied by $W_1$ and $W_2$. This would make it impossible to embed an ANN with a larger topology than the one proposed in this work.

By storing data in the program memory of the AVR µC and accessing it in execution time, it was possible to store 159 kilobytes of $W_1$ and $W_2$ in the program memory, summing a total of 172.43 kilobytes together with the implementation instructions, which totals 65.8% of the total data memory available. On the other hand, 1.04 kilobytes, that is, 12.7% of data memory was occupied for $P = 50$ and $M = 10$. It is important to note that 8 kilobytes of the program memory is used by the Arduino bootloader to allow programming the µC without any additional programming hardware apart from the Arduino Mega itself. All of this data can be seen in $P = 10, 20, 30, 40, 50$ in Table 3.2. The memory occupation results for after compiling the implementation in section .data were not shown in Table 3.2 because 788 Bytes were constant for all cases of $P$, as expected.

One important point to notice is the linear behavior observed in the execution times from Figures 3.5 and 3.6. They represent the times required to process the feedforward step of a hidden layer ($t_{FFM-1}$) and the time required for the same process in the output layer ($t_{FFM-2}$), respectively. Similar results can be seen in Figure 3.8 regarding data memory, and in Figure 3.7 regarding the amount of program memory occupied in the µC. This has already been discussed in (Vilar et al. 2018) which showed that, for devices with greater memory capacity, the time required to run an ANN topology with more neurons...
Figure 3.5: Feedforward time in the hidden layer, $t_{FFM-1}$

can be estimated.

Another important point is the time required to carry out the classification. In Table 3.1, it is possible to see that for the largest ANN topology used ($P = 50$), the maximum time to perform a classification was $\approx 1.61$ seconds, which represents a reasonable time for applications that do not require a high response speed, such as (Deshmukh & Moh 2018, Yu & Chen 2010, Fuady et al. 2017, Farooq et al. 2010, Saad Saoud & Khellaf 2011b, Cotton et al. 2008, Mancilla-David et al. 2014) and (Huang et al. 2017).

Concerning timing results, comparing with the linear behavior of increasing time with increasing topology from (Vilar et al. 2018), it is possible to affirm that the final topology here presented ($N = 785$, $P = 50$ and $M = 10$) achieved a results compatible with previously cited reference applications, even for a 8-bit 16MHz $\mu$C, with $\approx 2.26$ milliseconds for classification per $P$ and $M$, as noted in Table 3.1.
3.5. RESULTS

Figure 3.6: Feedforward time in the output layer, $t_{FFM-2}$
Figure 3.7: Occupancy of program memory for $P = \{10, 20, 30, 40, 50\}$ and $M = 10$
Figure 3.8: Occupancy of data memory for $P = \{10, 20, 30, 40, 50\}$ and $M = 10$
3.6 Conclusion

The present work demonstrated that it is possible to implement ANN topologies larger than a few neurons in low-memory devices such as the ATmega-2560 µC. Execution times close to 1 second have been demonstrated for a complete numeric handwritten digits classification using the MNIST Dataset (LeCun & Cortes 2010) with an externally trained MLP-ANN containing 50 neurons in the hidden layer and 10 neurons in the output layer. Future works point out for other optimizations such as the storage of activation functions in the form of Look Up Tables (LUTs) in an effort to reduce the processing time with native C language libraries. Furthermore, it is possible to reduce the resolution of synaptic weights from 32-bit to 16-bit floating point, in addition to the exploration of compression techniques. Finally, it was demonstrated that 8-bit µCs can be used with embedded AI applications that require larger network topologies, enabling lower cost and lower final energy consumption for such applications.
This work presents implementations of Multilayer Perceptron Artificial Neural Networks on a microcontroller for embedded real-time AI applications. Inference time duration, training convergence and source code memory occupation were analyzed in regards to the variation in hyperparameter values. Two scenarios were analyzed and discussed taking advantage of embedded training and inferencing, and also the usage of excess program memory to store more knowledge representation in the form of synaptic weights.

AI applications are the state-of-the-art on almost all fields nowadays and its applications requires high computational processing power and massive and fast data storage solutions. This requires fairly expensive devices and powerful devices, making applications that require fast-inferencing with low-power usage, low-cost and low-size unable to benefit from this technology.

Due to its low-cost, low-size and high availability, µCs are SoCs found on almost all devices, industries and markets. However due to its limitations on resources, such as computational power and low internal memory capabilities, their applications are somewhat limited to specific scenarios that do not require more of those resources.

The first work here discussed focused on implementing an MLP with a XOR gate, showing that it is possible to model and implement this ANN on a device with an 8-bit processor and high constraints on working frequency and small internal memories. Next, while increasing the hyperparameter values of the MLP architecture, a linear behavior was observed. This behavior showed that with this implementation, even if a limit is reached because of resource constraints, the model architecture can be expanded of devices with more resources. Also, the inference duration and maximum MLP architecture can be predicted to fit different applications with embedded training or only embedded inference. This also showed that compared with other works and implementations, the proposed system has an inference time compatible with application requirements seen in the industry discussed in this paper.

The second work conversed on how to make bigger MLP architectures with more nodes feasible into a commonly available µC. The proposed implementation focused only on inferencing with the notorious MNIST 10 digit classification model with an MLP. The MNIST model usually requires hundreds of units into a single fully-connected layer to classify the 10 handwritten digits with an accuracy close to 99.9%. The implementation used a technique to packed 50 units of a hidden layer and 10 units on the output into the program memory, with an memory usage of approximately 160 KBs, and using only
1.04 KBs of work memory, which would be completely full over only 10 units, under a common implementation.

4.1 Future Works

This work proposed to implement and validate an MLP on a microcontroller 8-bit platform and validated it using two use cases. Further, in the third chapter, a new implementation extended the capabilities of the proposed system with more room for artificial synaptic weights and even more complex use cases, usually solved using CNNs. Although the presented results showed that the expected outcomes were achieved, there certainly is more work to be done to extract more performance from this platform.

A few techniques can be easily mentioned for better utilization of the available memory. Pruning can be used to remove the unnecessary intermediary neurons, synaptic weights, and even layers on RNA applications. Another technique is weights quantization, at which the binary precision can be reduced to reduce the memory usage. Also, on reducing memory usage, synaptic weights compression is another possibility, even if it may incur into loss of accuracy.

The field of AI and RNA algorithms is constantly improving and growing. Many other methods to extract more performance from these algorithms appear each year and such a flexible proposal, as the one from this work, is liable to benefit from it. This better illustrates how easily modifiable this proposal can be to many RNA applications and its relevance throughout the years in its field.

Finally, this work presented two papers implementing embedded inferencing and training on an 8-bit µC with time, accuracy, and memory occupation with regards to different sizes of an MLP architecture. The results showed that µCs are a viable platform for embedded real-time AI applications, using MLP ANNs, that are present in almost all modern ML and DL implementations.
Bibliography


Dey, S. & A. Mukherjee (2018), Implementing deep learning and inferencing on fog and edge computing systems, em ‘2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)’, pp. 818–823.


Gajski D., Daniel, Samar Abdi, Andreas Gerstlauer & Gurnar Schirmer (2009), *Embedded Systems Design: modeling, synthesis, and verification*.

URL: http://proceedings.mlr.press/v97/gural19a.html


Krizhevsky, Alex, Vinod Nair & Geoffrey Hinton (2008), ‘Cifar-10 (canadian institute for advanced research)’.
URL: http://www.cs.toronto.edu/kriz/cifar.html


LeCun, Yann & Corinna Cortes (2010), ‘MNIST handwritten digit database’.
URL: http://yann.lecun.com/exdb/mnist/


URL: http://arxiv.org/abs/1709.02260

URL: http://doi.acm.org/10.1145/3302506.3310390


