
Identification of Assembly Code Patterns for Hardware Acceleration

BEng Final Project Report

Computer Systems Engineering 2021/2022

Conducted by:

Elias Abou Farhat (199246766)

Total count: 9452 words

Supervisor

Email

Dr. Jonathan Graham-Harper-Cater jeghc20@bath.ac.uk



UNIVERSITY OF
BATH

May 13, 2022

1 Abstract

In today's world, technological applications require more and more computational power to run. CPU speed increase has seen a decline over the past four years, leaving room for new ideas to accommodate the increasing need for fast processing power

This project aims to study the viability of an automatic acceleration of any pre-compiled assembly code. The breakdown of Dennard scaling and the end of Moore's law has led to heterogeneous parallel processors being proposed to meet the insatiable demand for computing power. A possible implementation is the mixed scalar processor with a co-processor like FPGA capable of parallel computing.

Meaning, at a higher level, the aim is that an FPGA could be automated for hardware acceleration. Programming parallel computers efficiently require a deep understanding of the underline physical hardware and compilers and, therefore, is hard for humans to gain optimal efficiency. This project focuses on finding parallelisable patterns inside a complex assembly code, leading to the easier implementation of acceleration algorithms and processes and, finally, more accessible to the end-user.

An AI convolutional neural network was implemented to classify complex assembly codes. The deep learning model is trained to find loop dependent classes, independent loop classes, or unknown class patterns. The input datasets were created using an in-house data generator system that compiles C applications and then disassembles them. The research led to a great accuracy classification model. However, on heavy testing applications like convolution and sigmoid operation, the system classified the assembly code as unknown, questioning the viability of finding patterns inside the pre-compiled C code or the system's validity. The research needs a lot of future work, starting by expanding the input datasets and finding new possible types of patterns. Future work also needs to consider the generalisation of compilers and optimisation operations.

2 Acknowledgements

I want to extend my deepest gratitude to my supervisor Dr Jonathan Graham-Harper-Cater for his unwavering support, unparalleled knowledge and profound belief in the work. I am extremely grateful for his guidance and support throughout this project, constantly providing me with insightful suggestions.

Table of Contents

1	Abstract	1
2	Acknowledgements	1
3	List of Acronyms	4
4	Introduction	5
5	Background	7
5.1	Hardware Acceleration	7
5.2	Parallelism	7
5.2.1	Dependent Loops	8
5.2.2	Independent Loops	10
5.3	Compilers	11
5.4	Machine learning	11
5.5	Previous Work	16
6	Methodology	17
6.1	Data Generation System	17
6.1.1	Overall Design	17
6.1.2	Control	18
6.1.3	C applications	19
6.1.4	Compiler	20
6.1.5	Testing Applications	21
6.1.6	Disassembly	22
6.1.7	Vectors Generation	22
6.2	CNN	25
6.2.1	Architecture	25
6.2.2	Data	27
6.2.3	Training	28
7	Results	32
7.1	Engineering results	32
7.1.1	Data Generation System	32
7.1.2	CNN	38
7.2	Experimental results	42
8	Discussions	44
8.1	Over fitting	44
8.2	Data hypothesis	45
8.3	Possible Solutions	45
8.4	Future Work	46
9	Conclusion	47

10 Appendices	51
10.1 Appendix 1: FPGA Structure	51
10.2 Appendix 2: PIC12F1840 Data Sheet	52

3 List of Acronyms

GPU	Graphic processing unit
FPGA	Field-Programmable Gate Array
CPU	Central processing unit
MCU	Microprocessor
AI	Artificial Intelligence
ML	Machine Learning
ANN	Artificial Neural Networks
CNN	Convolutional Neural Networks
PIC	PIC microprocessors
FIR	Finite impulse Response
DOP	Degree of Parallelism
ReLU	Rectified Linear Unit
CSV	Comma Separated Values
Microchip	Microchip Technology Incorporated

4 Introduction

In today's world, technological applications require more and more computational power to run. Data management and analysis reached a new stage where technological breakthroughs are needed to accommodate this increasing demand. Ahmdals law describes an upper limit in the possible speedup [1]. Scientists predict the end of Moore's law [2], and the decrease of Dennard scaling [3] which shows the relationship of MOSFET scaling: the decrease of transistor size and a constant power density. A "dark silicon" problem was raised in improving and creating chips: Power consumption cannot scale in parallel with the die shrinking. The issues limited the amount of possible acceleration and improvement using traditional and standard ways like transistor shrinking and increase of CPU speed.

New techniques are needed to accelerate algorithms. A common trend created in the 70s was to use a co-processor to offload some CPU resources and be highly specialised compute resources that can deal better with these loads. The creation of GPUs or graphic cards as co-processors has significantly impacted technologies like Artificial Intelligence and machine learning. Significant speedups at a small cost can be achieved by deploying co-processors such as GPUs. [4].

However, GPUs were initially highly specialised coprocessors but, over time, have become more programmable and more general-purpose. Silicon area has been spent to "expose" the underlying processing resources in a way that makes them usable by a broader range of applications.

GPUs have a limit on their processing power and, in some scenarios, are being replaced by FPGAs [5]. By using techniques like: aggressive pipelining, parallel execution, custom memory hierarchies, wide datapaths, and optimised operators, performance gains could be achieved using an FPGA as a co-processor [6]. FPGAs are also cheaper than GPUs for mass producing. This process is a branch of hardware acceleration, and it is becoming a crucial part of computing.

At a high-level programming language, the use of co-processors is facilitated by libraries. Nevertheless, these methods require the designers require a detailed understanding of the application, the source code and the underlying processing resources.

This research envisions the acceleration of any application with a co-processor implementation. It means the source code will be compiled and analysed. Potential blocks of code (instructions) will be identified for acceleration, and the potential performance gain will be computed. This study focuses on the first part of this complex task and primarily aims to explore metrics by which machine code acceleration candidates may be identified on a RISK CPU.

Machine learning has seen a usage increase in most sectors: Medical, Transportation, Security, etc... Neural networks have the ability to identify complex patterns that might otherwise be difficult (or computationally expensive) to identify analytically. The task will focus on the implementation of a convolutional neural network, where the model could predict which assembly patterns can be accelerated.

This report will consist of introducing the background theory needed for this research. It will

also demonstrate the methods used to allow future work. Finally, the writing will present the results and potential issues with this technique and plan the continuance of this research.

5 Background

5.1 Hardware Acceleration

Hardware acceleration is the process of creating a system to improve application performance. The most common form of hardware acceleration involves using a co-processor to complement the compute resources provided by a CPU. A co-processor is a hardware component generally placed next to the main chip. While the CPU is still the main chip that controls the application, the host offloads some operations to the co-processor, and if correctly implemented, a speed-up could lead to performance gains. This process usually leads to better efficiency and is used in the most common heavy application that requires heavy computing [7]. A co-processor uses a form of parallel computing depending on which type of hardware is used.

GPUs are the most common. They are graphics units originally responsible for accelerating 3D graphics with pixel rendering but then were designed as a general-purpose unit computation accelerator to enter multiple markets. GPUs will be used in this project to accelerate the AI system created and explained later in the report.

FPGA is another type of accelerator implemented as parallel computing. GPUs have a limit on their processing power and, in some scenarios, are being replaced by FPGAs [8]. It is the flexibility offered by FPGAs that makes them attractive: FPGA processing and memory resources can be configured in an application-specific way, and it is this co-design (and tight coupling) of processing and memory resources that makes them so efficient. However, previous research has shown that the use of reconfigurable co-processors can substantially improve performance, for some applications. [6]

A previous paper showed that convolution for neural networks had a 20x speedup using an FPGA compared to a CPU [8].

FPGAs have a hardware structure that supports vector (and array) processing. And even though the clock frequencies that can be achieved using FPGAs are slower than a CPU, it benefits from executing the operation in parallel compared to a standard microprocessor which executes its computation in sequence. [9]

Appendix 1.0 describes the complete structure of an FPGA.

5.2 Parallelism

Concurrency is the process of having different computations in different parts. Parallelism is these computations executing at the same time. As parallelism has been proven to accelerate computation in a heavy application, two primary approaches are used. Task parallelism consists of offloading tasks or threads concurrently to processors to offload CPU resources. Data parallelism consists of offloading concurrently the same tasks but with different data on processors. [10]. In the scope of this project, we focus only on data parallelism.

Loop parallelism is the most obvious form of parallelisable pattern we can predict. Rather than iterating over a complete loop, the data is split on an FPGA for parallel execution.

When classifying a loop, we need to consider dependencies:

Loop can be dependent, meaning the data at each iteration depends on other data. This process leads to complex or non-parallelisable tasks.

There are several types of dependences [11] (Assuming sequential statements S1 and S2):

- **Flow Dependence:** S1 is flow-dependent on S2 if there is a path between S1 and S2 where execution consists of one output of S1 being fed as an input in S2.
- **Anti Dependence:** S1 and S2 are in order (S1→S2) where respective I/O overlaps.
- **Input Dependence:** S1 and S2 have the same input variables or are read from the same memory address.
- **Output Dependence:** S1 and S2 have the same output variables or write in the same memory address.

The dependencies led to Bernstein's conditions [12] which can help with the determination of dependencies in statements. A flow dependence must be preserved at all costs when considering the parallelisation process. This ensures the sequential behaviour of a loop and could prevent the code from having race conditions or other parallel-based errors. An algorithm presents a degree of parallelism or DOP, the number of concurrent instructions that can be done in parallel.

5.2.1 Dependent Loops

The pseudo-code below shows a basic example of a dependency in a loop:

```

i ← 1
n ← [10]
while i < 100 do
    n[i] ← n[i - 1] + 10
end while

```

In the loop, you can see that the next iteration is dependent on the previous iteration. That leads to a $DOP = 0$. The task at hand is non-parallelisable.

There are many applications for loop dependent algorithms. The chosen applications were based on research done on already tried methods [13].

Fibonacci series:

It is a mathematical sequence in which in number is the addition of the two previous ones.

$$F_n = F_{n-1} + F_{n-2}, n > 1 \quad (1)$$

This means if $F_0 = 0, F_1 = 1$ then $F_2 = 1$.

The pseudo-code of a Fibonacci sequence is:

```

n ← fibonacciN

```

```

arr ← [0, 1]
n1 ← 0
n2 ← 1
for i ← i + 1, i ≤ n do
    n3 ← n1 + n2
    n1 ← n2
    n2 ← n3
    arr[i] ← n3
end for

```

The Fibonacci series in code shows the dependency with a $DOP = 0$. The number of operations can be changed randomly to give a different type of operation.

Binomial coefficients:

Binomial coefficients are the results of the binomial theorem describing the expansion of the power of the binomial. The mathematical formula is presented below:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad (2)$$

The pseudo-code code implemented does not use a recursive function as compilers of microchip does not provide recursive compilation:

```

n ← n
k ← k
arr ← [[]]
for i ← i + 1, i ≤ n do
    for j ← j + 1, j ≤ min(i, k) + 1 do
        if j ← 0 Or j ← i then
            arr[
        else
            arr[i][j] = arr[i - 1][j - 1] + arr[i - 1][j]
        end if
    end for
end for

```

The pseudo-code shows the use of a dependent nested loop. Parameters can be varied, leading to different computations.

Dependent Array computation:

Dependent arrays aim to create dependent loops where we add and store the resulting in either one of two input lists or third output lists. This application can be updated to create multiple types of assembly codes.

5.2.2 Independent Loops

The pseudo-code below shows a basic example of an independent loop:

```

i ← 1
n ← [10]
while i < 100 do
    n[i] ← i
end while

```

Each iteration is independent of the other, leading to a $DOP = 1$. This algorithm is parallelisable. The applications for loop independent are also based on research done on already tried methods [13].

Swap/Add Array computations:

These applications perform swapping and addition on arrays meaning there is no direct dependence on elements in a for loop. We are switching or adding the parts of different lists at different indices. However, When storing the result in one of the input lists, we could end up with an anti dependence. The sizes of the lists, for loop indices and lists content, are determined randomly to alternate possible assembly codes. However, the size of the lists is constrained by the CPU's memory.

FIR filter:

The last application for a dependent loop was more complex, hoping to train the AI model in a realistic application. Finite impulse response filters are used in many applications (signal processing...). The mathematical formula to compute an FIR filter is:

$$y[n] = \sum_{i=0}^N b_i * x[n-i] \quad (3)$$

However, there are many ways of implementing an FIR filter in C. The application considers the delay of the samples inside the FIR filter. The output of this function is the convolution of the delay lines by the filter coefficients. The equation presented above proves the dependency in the implemented loop.

5.3 Compilers

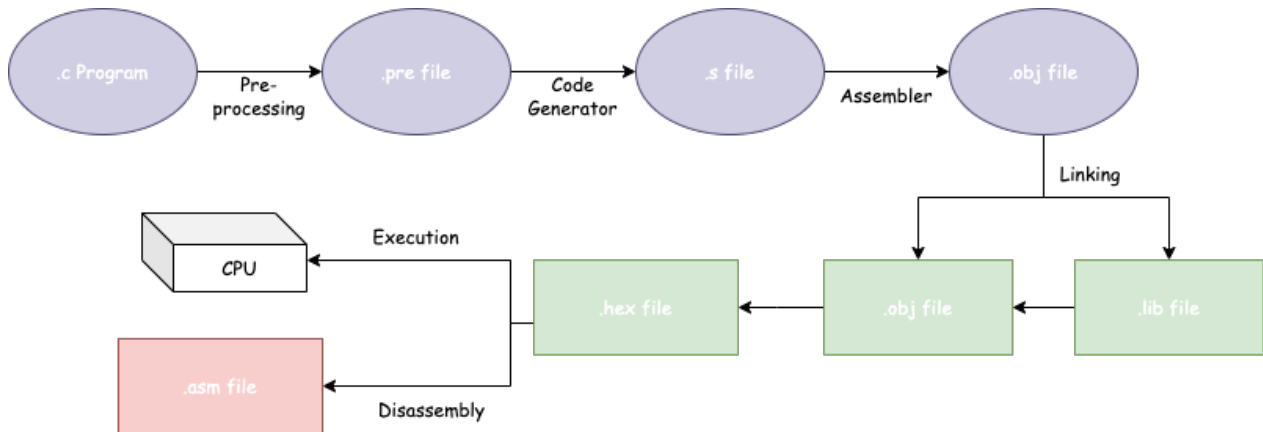


Figure 1: Diagram representing a typical full compiler process

The diagram above represents the entire process of a typical compiler. [14] [15]

- **Pre-processing:** An algorithm or code is passed through the whole system as an input. This report assumes that all code will be given as a .c code. The code is expanded and converted to a .pre or .i file.
- **Code-Generation:** The next step is the assembler, in which the assembly code is generated from the pre-processed file. In general, the compiler has some optimisation levels to increase code efficiency. The classes of optimisation depend on the CPU used and the chosen compiler. Level 0 describes a non-optimisation suitable for debugging and, in our case, analysing. Then, the optimisation increases with the levels available. The channel is from a .pre file to a .s file and finally, an object file .obj, which is the input of the next step.
- **Linking:** Usually, all code uses libraries for functions or different algorithm templates. The compiler takes the object file and links it to all library files to create a final executable file. This file is then converted to a hexadecimal file fed into the CPU program memory for execution or, in our case, supplied to a disassembler.

5.4 Machine learning

AI refers to the simulation of human intelligence. It is used in most fields and has been proven to increase problem-solving. Machines are trained to mimic human learning and knowledge to solve more complex situations. In this project, AI will be used to analyse machine assembly code and find potential accelerated patterns. Machine learning is a concept that has become more and more famous over the years with breakthroughs in processing power and parallel programming, and it is a field of Artificial Intelligence as shown in figure 2.

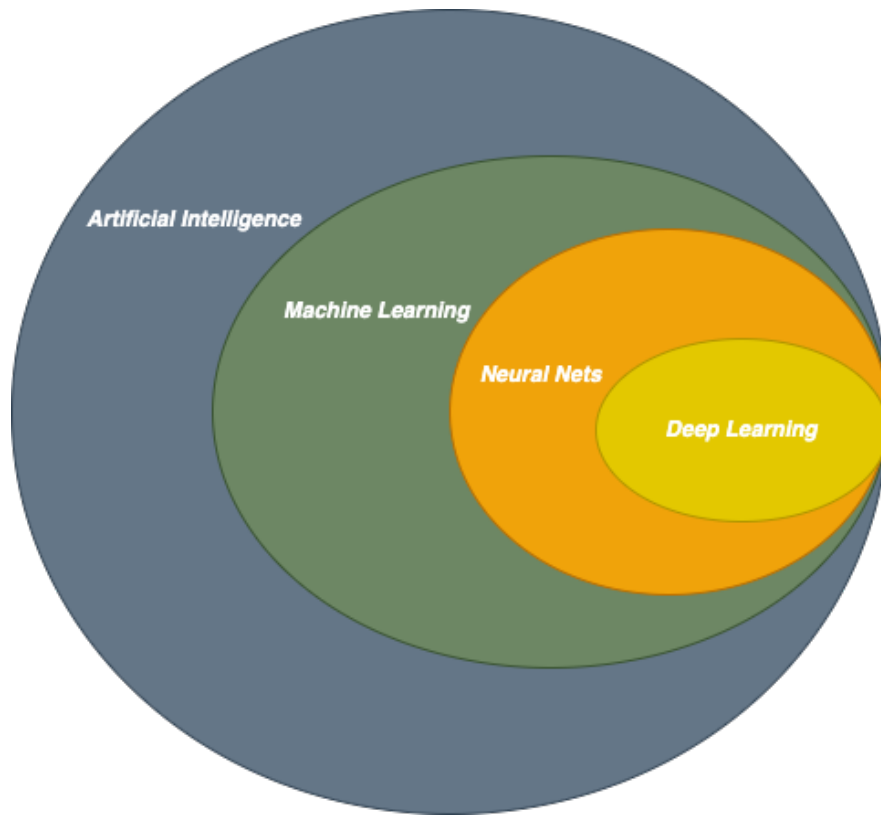


Figure 2: Image representing the ML field inside AI

Machines learn by gaining knowledge of the data sets [16]. The systems analyse the raw data structure under a model and use it to predict the unknown data.

The model used in this project is a neural network in the deep learning field. This model shares characteristics with the human brain and the visual cortex.

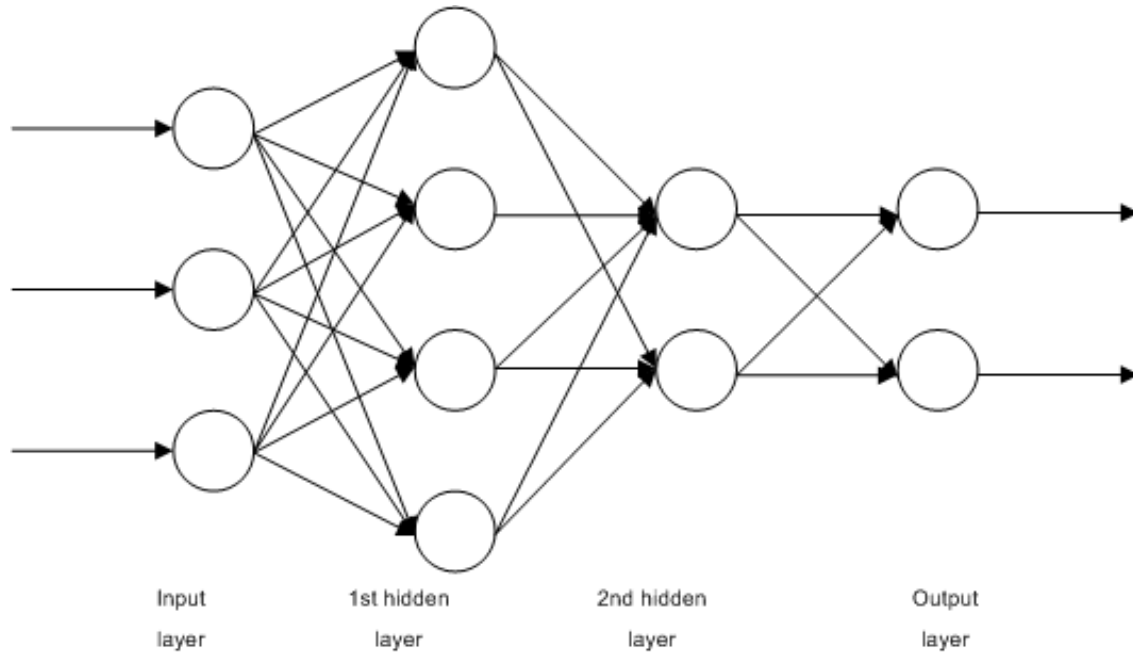


Figure 3: ANN topology structure representation

As shown in figure 3, the network architecture shows neurons and layers connected. Each model has inputs, hidden layers and outputs. ANN uses a sort of weights to determine the result, which means, for example, taking a function $F(Ax + C) = B$, the input matrix A is multiplied by weights to predict the output B . The number of weights varies between models and applications; however, they represent the only memory of the system and are usually in a relatively large number. This structure makes ANN interesting, as they mimic the biological neurons in our mind and even provide an advantage in some applications.

A deep neural network presents more than one hidden layer[17]. There are different structures of ANN available, but they share similar basic components:

- **Parameters:** the defined x in the example equation above. The weights represent parameters. They are trained using an algorithm in the hope of finding optimal values to predict with high accuracy the output vector. Possible algorithms include optimization techniques like the gradient descent algorithm (find a minimum of a function to minimize the loss of a function)
- **Layers:** A layer is one of the largest blocks in ANN. They take input and multiply with some activation function to get an output fed into as input for the following layers.
- **Activation function:** functions that defines if the neuron(node) should fire to the next one. It defined the transformation of an output to an input. Activations function can take many forms like sigmoid or ReLU. Yet, they depend mainly on the type of architecture used and the application.
- **Loss function:** When training a machine learning model, we compare the predicted output with the actual current output. A loss function determines the difference be-

tween the two outputs and is used to adjust the model's weights accordingly.

There are many types of learning and loss functions. The problem is approached in this research using a classification application. The goal is to minimize the loss function result, determining the difference between the predicted class and the actual class. This process is called optimization, finding the optimal best values for the model's weights to minimize the loss function result.

One of the most famous types of model architecture is a **Convolutional Neural Network or CNN**. CNNs are used in deep learning applications because of their image and pattern recognition success. These models extract features of images or graphs using convolution to predict predefined classes. A good CNN model usually uses input datasets with some structure or pattern for the AI machine to learn.

In a CNN architecture, there are multiple layers: Input, learning, and classification layers [16]. The learning layers are from the same configuration shown in figure 4.

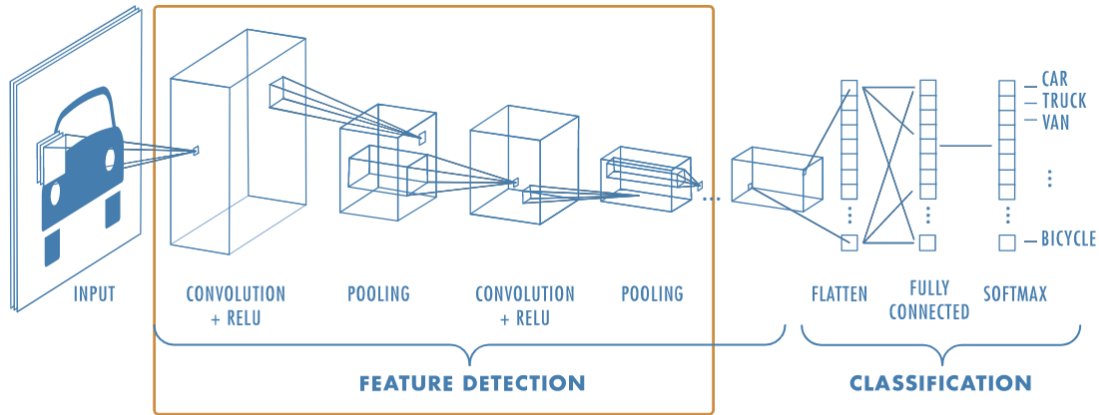


Figure 4: Diagram representing a full CNN configuration [18].

Input Layer

First, the input layer is responsible for storing the data (3D matrix with an H, W and depth representing the colour palettes). This data will be fed to the following layers.

Learning Layers

The learning layers inside a convolutional neural network extract features and characteristics from the input data to give a score for each class. They have multiple channels and are duplicated many times depending on the model. The first layer is convolution representing the feature extractor behaviour of the CNN. As explained above in the report, convolution is a mathematical concept that combines two functions. For image processing, convolution is used with a kernel impersonating filtering.

Convolution in machine learning is done using sort of multidimensional arrays. These structures are called tensors. Tensors are mathematical structures where arrays are defined as a subclass of tensors. The difference between the two structures is that tensors present additional dimensions. Tensors can represent N dimension data; it is like a container that can hold scalars, vectors or matrices. In ML, the data is defined as tensors rather than typical structures. Convolution in multidimensional arrays is mathematically defined as [19]:

$$S(i, j) = (I \otimes K)(i, j) = \sum_m \sum_n I(m, n) K(i - m, j - n) \quad (4)$$

$$S(i, j) = (K \otimes I)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n) \quad (5)$$

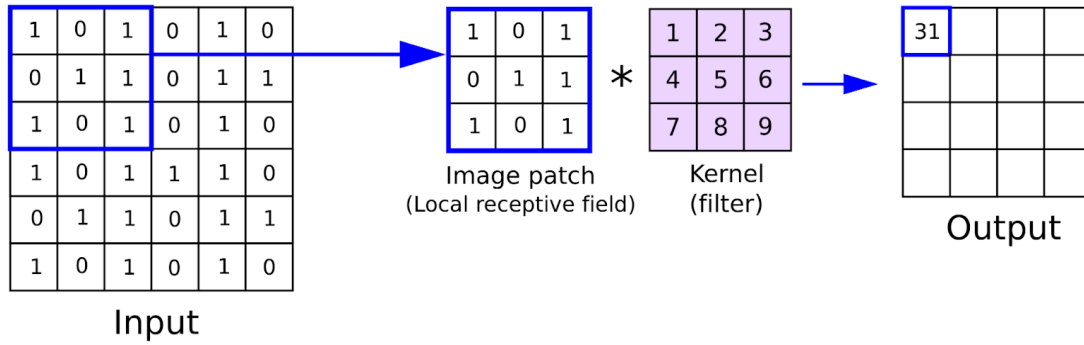


Figure 5: A convolution operation example [20]

Figure 5 shows how the kernel passes through the input data and creates output data. In a CNN, the weights are the filter or kernel coefficients; these weights are updated to reach optimal values in training mode. Convolution uses hyper parameters:

First, the kernel size represents the filter size applied when convoluting the input tensor data. Second, a stride size configures the sliding window moves per application. Applying a higher stride will result in more overlapping and vice versa.

The convolution result is passed through a ReLU activation function which is non-linear. The next stage is pooling, shown in Figure 4, which modifies the output. There are many pooling functions, but in general, pooling replace the output of a net with a summary statistic of nearby output [19]. The outcome is then **invariant** which means any small change will not affect the pooled result. In summary, pooling performs the down-sampling of the output of the previous convolutional layer.

Output Layers

The output layers include the fully connected layers. The output of the feature layers is flattened into one 1D vector, and then for each class, a score is calculated. This layer is connected to each neuron of the CNN, and the output is a tensor 1*1*number of classes.

Training a CNN uses a large dataset where tensors are already classified. The training algorithm uses **stochastic gradient descent**. This means that the model will predict the label output for each sensor and compare it to the actual correct label. The algorithm will reward good guesses and penalize the wrong guesses [16], updating the coefficients of the weights or filters. This is what learning means in terms of CNN, and weights should reach optimal values to predict any input given! The most famous algorithm is backtracking. Backtracking means tracking down the activated neurons to divide the contribution of errors and update the neural network's weights.

5.5 Previous Work

Mr Jalabert did his Master's thesis [13] [21], and its research is on new ways of accelerating sequential compute code using machine learning techniques. However, they produced an in-house compiler for multiple applications where the AI model classifies blocks of code as loop dependent or loop independent. The system was never tested on actual CPU compiled code. Nevertheless, it achieved high accuracy. The relevant part of his work is his innovative graphical representation of an assembly script. He represents the opcode and its operands as one-hot vectors creating a large matrix of vectors. These matrices represented a 2D graph that can be shown as an image. The AI model uses as input these vectors and classifies each image using the mathematical concepts explained in the background section. My work was inspired by this research, where I used the graphical representation technique for my AI model.

6 Methodology

6.1 Data Generation System

6.1.1 Overall Design

When creating a convolutional neural network, the main concern is generating enough data for guided or supervised learning (training the model using large input datasets). Usually, data sets are created by big institutions, but in the case of the type of images we will use, the need for a data generation system is critical. The system is a python script which calls several stages, as shown in Figure 6. All components of this diagram will be explained in more details in sections to follow.

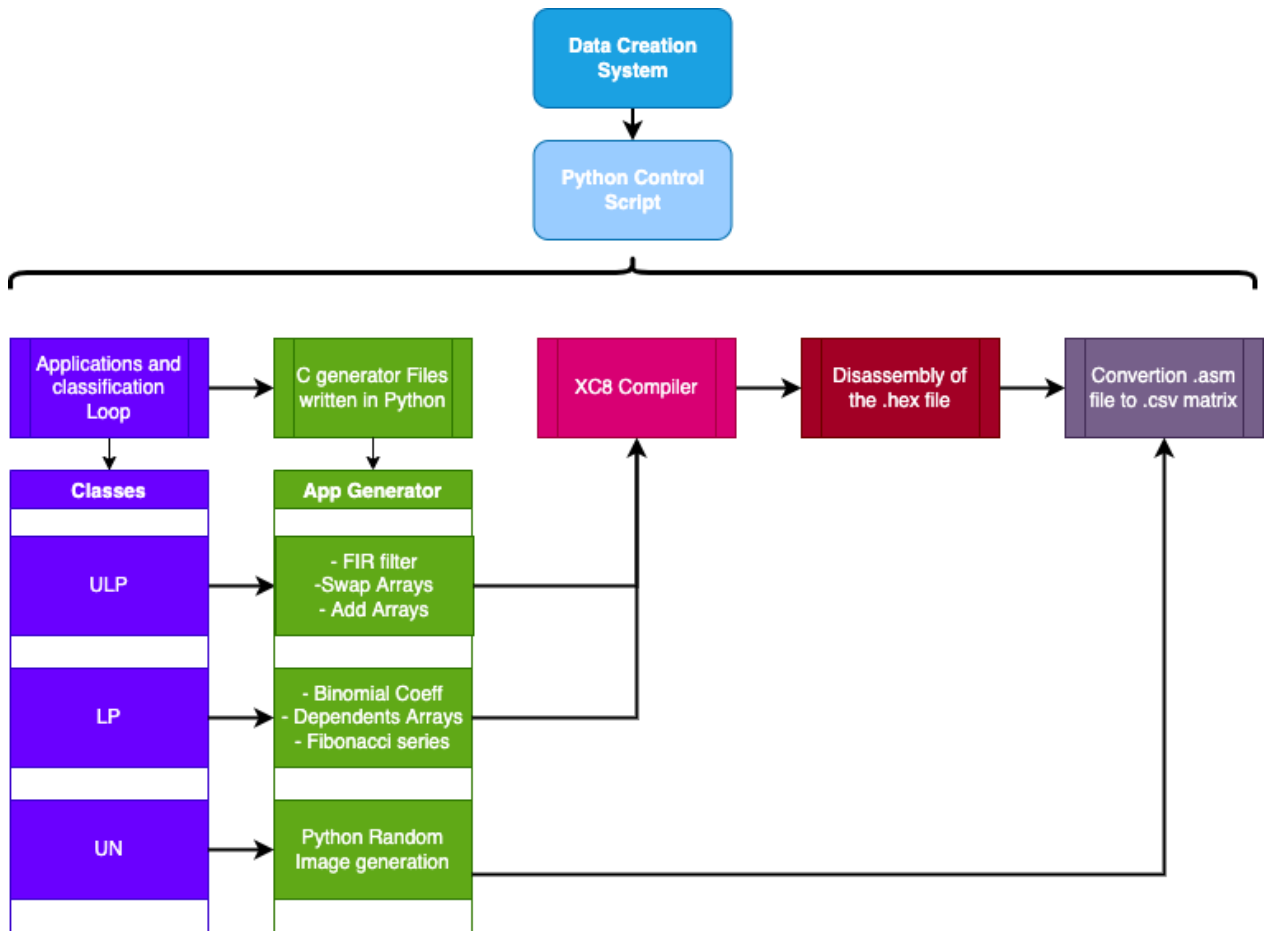


Figure 6: Diagram representing the full data generation system

The main point is to assign a number of images(matrices) per possible application and then generate balanced data sets for our neural network. The plan of the system is defined as:

- have built a code generator (written in Python) to generate short C programs/applications with different characteristics. All are listed in Figure 6, for example “Add Arrays”, and “Fibonacci Series” (and were identified by Jalabert in [13])

- the programs are then compiled using the XC8 compiler, targeting the PIC12F1840
- these compiled programs are then disassembled
- each instruction in the assembly code (i.e. .asm file) produced by the disassembler is then translated into a special vector
- the vectors for all instructions in the program are then combined to form a matrix, which is stored in .csv format for subsequent processing
- each .csv file represents one example in the training dataset used for a particular application

Each subsystem presented above and in the diagram above will be explained individually.

6.1.2 Control

The main subsystem is the control script, which orders around other subsystems. Creating a balanced dataset for the AI model is essential to prioritise each class equally and avoid prediction problems. [22] An unbalanced dataset leads to some classes being represented more often than others, and therefore, when predicting an output, the model risks being biased towards (and applying an inappropriate weighting to) these “preferred” classes. This means that the dataset contains an equal number of each class. The pseudocode 1 representing the Python control script in figure 6 below summarizes the generation process:

Algorithm 1 Main Data Generation System

```

classes ← ULP, LP, UN
n ← number of codes per application
for class ← classes do
    for application ← application_per_class do
        for i ← 1, n do
            Generate C code
            Compile C code
            Disassemble compiled C code
            Convert to a matrix
            Store in .csv format
        end for
    end for
end for

```

So, in summary, the system will loop through our three classes. In each class, we have different types of applications. Finally, we will run the steps above a pre-defined number of times (*n*) for each application creating a different balanced dataset that could be used for training.

6.1.3 C applications

The problem was creating a system where the compiled assembly (generated by the XC8 Compiler, see Section 6.1.4) would be different for each application, leading to different images. A CNN model (explained in section 6.2) will adapt its weights based on the prediction of the class of an image. If the vectors of each application are the same, the AI model would not be able to generalise, i.e. it would not be able to accurately classify vectors/images that it hadn't already seen. More details on the vectors generation process will be given in section 6.1.7. The subsystem C applications consist of different C generator scripts written in Python. The parameters given as input in the . files are randomised in Python.

The application **add arrays** (one of the set of applications selected for analysis and described in Section 5.1.5 taken from [13]) is explained below. However, all applications follow a similar process but different implementations to adapt their purpose.

First, we need to generate different random parameters to feed the c code. This process needs to be as random as possible and gives each time different variables. In this case, the variables and arrays have different sizes and content.

```

1 a = str(random.randint(0,50))
2 b = str(random.randint(0,50))
3 c = '0'
4
5 for i in range(random.randint(2,8)):
6     tempa = str(random.randint(0,50))
7     tempb = str(random.randint(0,50))
8
9     a += ','
10    a += str(tempa)
11
12    b += ','
13    b += str(tempb)
14
15    c += ',0'
16
17 test = random.randint(0,2)

```

The CNN can now adapt its weights to deal with any given length of lists and content for better prediction. The second step is to use these variables and create a .c file:

```

1 with open('12f1840/datasets/cfiles/addarr.c','w+') as file:
2
3     file.write("#include<stdio.h>\n")
4     file.write("void main() {\n")
5
6
7     file.write("int a["+ str(len(a)) + "] = {" + a + "};\n")
8     file.write("int b["+ str(len(b)) + "] = {" + b + "};\n")
9     file.write("int c["+ str(len(c)) + "] = {" + c + "};\n")
10
11    file.write("for (int i=0;i<" + str(len(a)) + ";i++) {\n")
12    if test == 0:
13        file.write("a[i]=a[i] + b[i]; } }\n")

```

```

14     elif test == 1:
15         file.write("b[i]=a[i] + b[i];} }\n")
16     else:
17         file.write("c[i]=a[i] + b[i]; } }\n")

```

We use a file write function to write basic c applications differently and with different variables. This process shown and explained above is repeated for all available applications except the random UN generator, described later in this section.

6.1.4 Compiler

In terms of a compiler, Microchip provides an XC8 compiler supporting all types of PIC microprocessors. XC8 can be invoked on the command line, allowing the Python Control Script to manage the assembly and linking of the generated C files. With the documentation [23] help, turning off optimization to a level 0 helps generate the wanted hexadecimal file (.hex). This compiler is also a cost-effective solution, as the MCU can be implemented as a simulator and no need for an actual hardware component.

Selecting an appropriate MCU for the project required several factors to be considered. First, to ease computation, it is vital to find a low number of memory addresses. An 8-bit PIC family is the right decision as they provide a good memory size for more complex applications and are constrained by the number of addresses. Microchip provides excellent documentation, community and IDE, which help with the compilation of the applications. Table 1 below summarizes the possible 8-bit MCU PIC family architectures [24] and their characteristics.

Table 1: Table summarizing the 8-bit PIC families' characteristics

Family	Bits	Instructions	Maximum Program Memory	Compatibility
Baseline	8-bit	33	3KB	XC8
Mid-Range	8-bit	35	14KB	XC8
Enhanced Mid-Range	8-bit	49	28KB	XC8
PIC18	8-bit	83	128KB	XC8

Theoretically speaking, using the lowest RISK CPU will lead to faster computation and more straightforward analysis. However, due to the nature of the application, like filtering, convolution or sigmoid operations, the need for relatively large program memory is essential when choosing an MCU. As the table shows, there is a trade-off between program memory size and the size of the instruction set. The size of the instruction set directly affects the length of the vectors created from each command. The choice of an enhanced mid-range MCU ensures enough program memory size to accommodate the need of the application while maintaining a relatively small instruction set. **The MCU chosen is the PIC12F1840.** According to its datasheet found in [25], the MCU presents 49 basic instructions and 2 optimization instructions. It also offers a program memory of 7KB or 4000 words.

Based on the datasheet provided for this specific MCU, we can create Python variables that will be fed in as input for the next step of the generation system. This task is the only manual

required duty of the system as the instructions are only available as a table in the datasheet. The C Code Generator uses a database describing the nature of each of the instructions in the instruction set, including its ID and which of the three operand types (f, d and b) accompanies it. The possible operands for each opcode the system work with are:

- f: File Register address (7 bits)
- d: Destination select (1 bit; determines where the result of an instruction is stored)
- b: Bit address (3 bits; identifies a specific bit within an 8-bit File Register)

The system disregards other operands like literals (k) as it complicates the process and lead to large vectors. Decrementation of registers is done using a built-in function (DECF), and therefore, literals are only assigned theoretically in a random pattern by the end-user. The destination address controls the result's storing in either the working register W (0) or a file f (1).

Each instruction has a specified operands combination and is stored in a list as shown below:

```

1 instruEncode_dict = {
2     'ADDWF': 1,
3     'ADDWFC': 2,
4     'ANDWF': 3,
5     ...
6 }
7
8 instru_dict = { #[opcode,f,d,b]
9     1: [1,1,1,0], #ADDWF
10    2: [1,1,1,0], #ADDWFC
11    3: [1,1,1,0], #ANDWF
12    ...
13 }
```

For example, the command "ADDWF" with ID:1 takes a file address and a destination bit control as operands.

6.1.5 Testing Applications

Convolution:

Convolution is a mathematical concept that expresses how much a function overlaps with another when shifted. It is presented by the equation below:

$$(f \circledast g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \quad (6)$$

Convolution is a heavy operation used in many applications that require significant computational power. Possible applications are image processing and signal processing, and as explained in the next chapter, convolution is used in machine learning, especially in deep neural networks. However, a compiled C convolution operation does not provide enough

information to access if there are some parts of the assembly script that contain patterns which can be accelerated. For this reason, convolution will be used as a testing script to predict any possible loop dependent patterns in the compiled assembly.

Sigmoid Operation:

The sigmoid curve is a non-linear activation function used in many applications, especially in machine learning and neural networks. The curve of the Sigmoid has an 'S' characteristic and is presented by the function below:

$$S(x) = \frac{1}{1 + \exp^{-x}} \quad (7)$$

Nevertheless, a heavy application needs accelerating, and as for the convolution function, the sigmoid will be used for testing to potentially find any patterns inside the compiled c assembly .

6.1.6 Disassembly

Creating a disassembler is a complex and time-consuming task. An already developed and tested utility [26] was used to convert a .hex file back to an assembly script. This disassembler supports multiple PIC MCUs and, more specifically, the 12F1840 microprocessor used in this project. GNU PIC disassembler can be called from the command line using:

```
1 gpdasm -p p12f1840 -c sigmoid.hex > sigmoid.asm
```

6.1.7 Vectors Generation

The last part of the system is the vector generation based on a complete assembly script. This process was based on the technique described by Jalabert [13] (see Section 5.2)

One hot encoding is a technique in machine learning used to represent data. It is a vector where all components are assigned a low bit (0), and the i th component is given a high bit (1) [27]. The Vector Generator generates a conventional 2D matrix where each row of the matrix represents one instruction. Each row of the matrix is a vector that is derived from the instruction's opcode and operands. As we have 51 instructions, the one-hot vector length of an opcode will be 51, where the position of the high bit determines the instruction ID. Similarly for the operands, the file register address will be translated to a vector $2^5 = 128bits$ in length, d requires a $2bits$ vector, and finally, and b requires $2^3 = 8bits$ vector.

The vectors are then appended, resulting in a final vector of fixed length 189. Figure 7 shows how each row of the matrix is constructed.

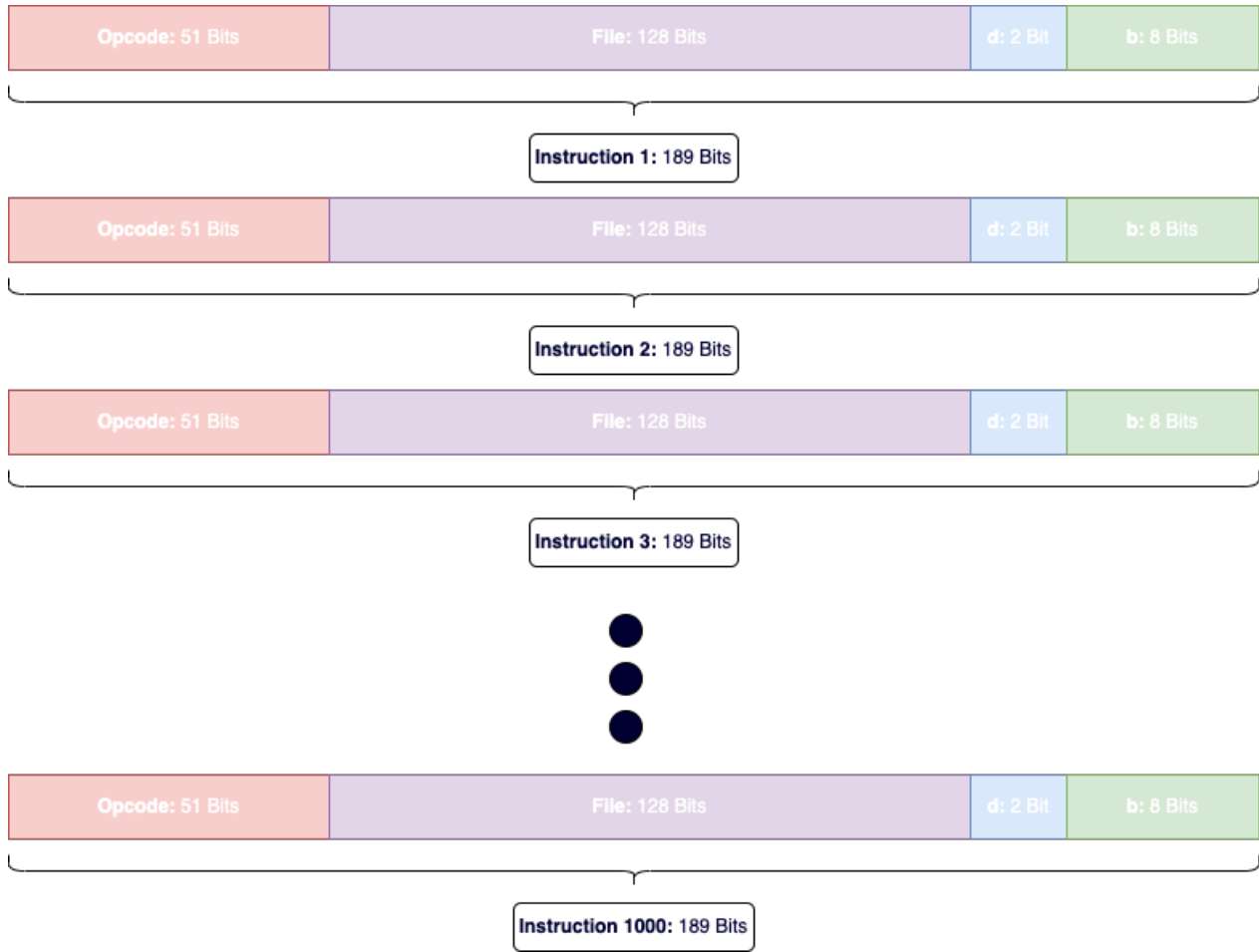


Figure 7: Diagram representing a typical matrix of one-hot encoded instruction vectors

All vectors are grouped to form a 2D matrix, which can be considered to be an "image" representing the assembly code. (The term image is used here to refer to these 2D matrices). Each image will be classified based on our application (0: LP, 1: ULP). These graphs will create the part of the complete dataset used to train our convolutional neural network or CNN.

The code for the process is shown below:

```

1      # (Opcode, f, d, b)
2      vector_list = [51,128,2,8]
3
4      f = np.zeros(vector_list[1],dtype=int)
5      d = np.zeros(vector_list[2],dtype=int)
6      b = np.zeros(vector_list[3],dtype=int)
7
8      # Creating onehot vector for the f if it is available for this
      instruction
9      if instru_dict[keyInstruc][1] == 1:
10         f[int(temp3[counter],0)] = 1
11         counter += 1

```



```

12
13     # Creating onehot vector for the d if it is available for this
instruction
14     if instru_dict[keyInstruc][2] == 1:
15         if int(temp3[counter],0) == 0:
16             d = [0,1]
17         else:
18             d = [1,0]
19         counter += 1
20
21     # Creating onehot vector for the b if it is available for this
instruction
22     if instru_dict[keyInstruc][3] == 1:
23         b[int(temp3[counter],0)] = 1
24         counter += 1
25
26     # Joining the vectors
27     # lineImage = np.append(opcode,f,d,b)
28     for bits in opcode:
29         lineImage.append(bits)
30
31     for bits in f:
32         lineImage.append(bits)
33
34     for bits in d:
35         lineImage.append(bits)
36
37     for bits in b:
38         lineImage.append(bits)

```

The third class of the classification system is UN which stands for unknown. Random images of the exact sizes are created. Nevertheless, the exact configuration of one-hot vectors is implemented in the 2D matrices. The algorithm is presented below:

```

1     # generate a random opcode
2     opcode = random.randint(1,vector_list[0])
3     config = instru_dict[opcode]
4     # opcode_Vector = np.zeros(vector_list[0],dtype=int)
5     opcode_Vector = [0 for _ in range(vector_list[0])]
6     opcode_Vector[opcode-1] = 1
7
8     temp = opcode_Vector.copy()
9
10    for i in range(1,len(config)):
11
12        if config[i] == 1:
13            encode = random.randint(1,vector_list[i])
14            # encode_Vector = np.zeros(vector_list[i],dtype=int)
15            encode_Vector = [0 for _ in range(vector_list[i])]
16            encode_Vector[encode-1] = 1
17
18            for entry in encode_Vector:
19                temp.append(entry)
20

```

```

21     elif config[i] == 0:
22         # encode_Vector = np.zeros(vector_list[i],dtype=int)
23         encode_Vector = [0 for _ in range(vector_list[i])]
24         for entry in encode_Vector:
25             temp.append(entry)

```

The process is done by randomly picking a number between 1 and 51, which imitates an instruction. Then depending on the operands of this specific instruction, a vector is created with a randomly assigned high bit for each one-hot vector operand when appropriate. This algorithm is repeated 1000 times to create the final matrix.

6.2 CNN

The machine learning neural network was implemented using Pytorch, an open-source framework that uses the Torch library [28]. This framework is one of the most famous Python machine learning APIs used in all related applications: Natural language processing, computer vision, and ANNs. It benefits from comprehensive documentation and an active community that helps developers understand and create these new algorithms.

6.2.1 Architecture

The architecture of the convolutional neural network for this application was defined as Figure 8.

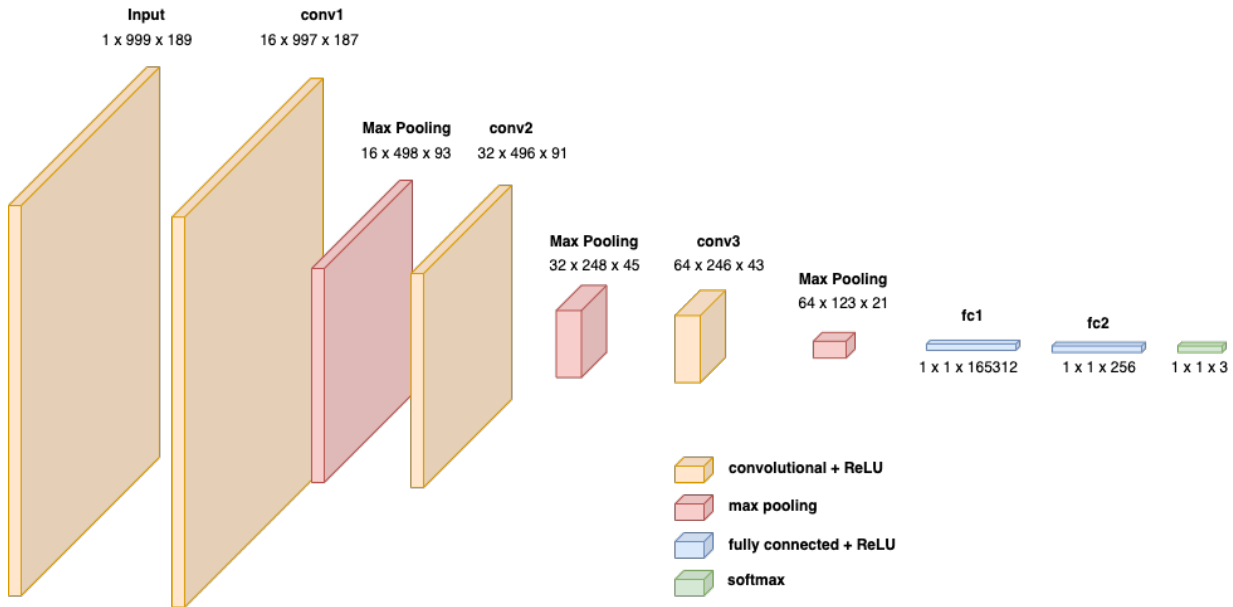


Figure 8: Diagram representing the full architecture of the CNN

The model has three different convolutional networks followed by an activation function and pooling. The number of layers was determined based on a typical application that uses three

conv layers [29]. The parameters of each layer were selected using an iterative process with the goal of achieving the best possible training accuracy.

Pytorch benefits from creating the model at a high level and simplifying the use of machine learning models.

The first part is to create the model layers and definitions. A class in object-oriented programming defines a model. This model inherits from the PyTorch basic ANN model. The code below shows the architecture implemented in Python:

```

1 class CNN(nn.Module):
2
3     def __init__(self):
4         super(CNN, self).__init__()
5
6         self.sizeToLinear = None
7
8
9         self.conv1 = nn.Conv2d(1,16,3)
10        self.conv2 = nn.Conv2d(16,32,3)
11        self.conv3 = nn.Conv2d(32,64,3)
12
13        self.maxPool = nn.MaxPool2d(2)
14
15        x = torch.randn(999,189).view([-1,1,999,189])
16        self.cnn(x)
17
18
19        self.fc1 = nn.Linear(self.sizeToLinear,256)
20        self.fc2 = nn.Linear(256,3)
21
22    def cnn(self,x):
23
24        x = self.maxPool(F.relu(self.conv1(x)))
25        x = self.maxPool(F.relu(self.conv2(x)))
26        x = self.maxPool(F.relu(self.conv3(x)))
27
28        if self.sizeToLinear == None:
29            self.sizeToLinear = x[0].shape[0]*x[0].shape[1]*x[0].shape[2]
30
31        return x

```

The convolutional layer uses the function **conv2d** from PyTorch, which accommodates the need for a 2D image.

First, the layers have as hyperparameters the in and out channels. We can see that the in channels of the next conv layer are the output of the precedent layer. Next is the kernel size. For this application, the kernel is a 2D array with a shape of 3*3, which means that a sliding window of 3*3 will perform the convolution. By default, the stride size will be 1*1.

Generally, the activation function used for a CNN is a **ReLU function** mentioned in section 5.4.

The pool function used was **MaxPool** for a 2D input. The pooling is responsible for determining the maximum of a feature map and downsampling.

As seen in Figure 8, there are two linear layers. They are responsible for flattening the results of the learning feature layers. Yet, when initially initialising the model, it's hard to determine the output of the convolutional and pooling layers. So, the hyperparameter of the FC layer was automated: We created a random tensor with the same size of the data inside the initialisation of the class. Then, we passed this structure to the convolutional layers to mimic a dataset. The input length of the linear layer can be determined by multiplying each component in the shape of the resulting tensor. Finally, this value is updated once, and the linear layer is created. This whole process is done only once at instantiation to improve efficiency. The second layer takes 256 channels and outputs three channels representing the score for each class. Eventually, the last layer will be a **softmax** function to predict the classification.

```

1  def forward(self,x):
2
3      x = self.cnn(x)
4      x = x.view(-1,self.sizeToLinear)
5      x = F.relu(self.fc1(x))
6      x = self.fc2(x)
7      x = F.softmax(x,dim=1)
8
9      return x

```

The forward function shown by the code above is responsible for passing the data in order through the layers. We use the computed convolutional layers and then pass the result by the linear FC layers and the softmax function.

6.2.2 Data

One of the most crucial parts of supervised learning of a CNN is the datasets. Unfortunately, because of the originality of the research, there are no datasets that fit the model's need, and it needs to be created using the data generation system explained above. The downside of the in-house system developed is the heavy computing power it requires. With no other available resources than a standard computer, a dataset of 900 images was created: 300 NumPy matrices were generated in a balanced way for each class, which means an equal amount of images per application and between classes.

A testing script was produced to test the generated data and ensure that no matrix is empty or incomplete. This script is run until the dataset is complete and error-free. In the case of a bug, the algorithm count how many matrices needs to be recreated.

Dealing with the data is implemented using a class called codeGraphs(). This class will present a function that reads the CSV files generated from the Vector Generator 7. The code below provides an example of the function used:

```

1  def TrainingData(self):
2
3      for file in tqdm(os.listdir(self.pathTraining)):

```

```

4      # Get the training data
5      if 'csv' in file:
6          pathFile = os.path.join(self.pathTraining, file)
7          data = pd.read_csv(pathFile).values.astype(int)
8
9      # Get the expected output (classification)
10     if file[0] == '0':
11         output = self.labels['LP']
12         self.counting['LP'] += 1
13
14     if file[0] == '1':
15         output = self.labels['ULP']
16         self.counting['ULP'] += 1
17
18     if file[0] == '2':
19         output = self.labels['UN']
20         self.counting['UN'] += 1
21
22     self.trainingData.append([data,output])
23
24
25     # Print balances to show that the data set is balanced
26     print(self.counting)
27
28     # Shuffle and save training data
29     np.random.shuffle(self.trainingData)
30     np.save("training_data.npy", self.trainingData)

```

Using multiple libraries like NumPy¹ and CSV, we first read all the CSV files, which convert them to NumPy 2D arrays. It is important also to read the title of the file, which is labelled with the class:

- 0: "LP" or Dependent Loops
- 1: "ULP" or Non-dependent Loops
- 2: "UN" or Unknown

For each class, the number of images is computed for report purposes and to test the balanced property of the dataset. The matrix is added together with its label in a multidimensional NumPy array, and the whole is appended to the final dataset array. Eventually, the array is shuffled and stored as a NumPy document to be read by the training functions.

6.2.3 Training

The last step of the project is to train the model to predict with good accuracy potential accelerated patterns inside any compiled assembly code. Training a CNN requires vast computational power and usually runs on GPUs rather than a CPU. GPUs offer the compute resources to perform many of the complex mathematical operations required to train a CNN in parallel. Pytorch ease the use of GPUs as it is already implemented in the library.

¹NumPy is an open-source Python library for manipulating and performing mathematical operations on arrays.

```
1 if torch.cuda.is_available():
2     device = torch.device("cuda:0")
3     print("Running on the GPU")
4 else:
5     device = torch.device("cpu")
6     print("Running on the CPU")
```

Because of limited resources and cost-free research, Google offers a cloud service that gives GPUs access to some applications. Google Colab [30] is an online editor based around Jupyter notebooks where we use GPUs and CPUs to train the model.

First, the NumPy data is either loaded or created if it's the first time. The loading function is defined below and is used from the NumPy library:

```
1 dataCnn = np.load("training_data.npy", allow_pickle=True)
```

The parameter `allow_pickle` is by default `false` for security reasons when loading an object from a NumPy file.

It is crucial to formulate a training and testing plan. Using our data, it is common to split the dataset. The NumPy function `train_test_split()` automatically performs this.

```
1 training_dataCnn, testing_dataCnn = train_test_split(dataCnn, test_size
    =0.25, random_state=42)
```

We give as inputs to the function: our dataset and a split size. The model's training represents 75%, and the testing represents 25% of the data. We also ensure that the data is shuffled using a random state one more time. Mixing the images is crucial to ensure a generalization and to reduce variance. An unshuffled dataset will advantage some classes and not learn properly, resulting in poor accuracy. It will not truly represent the natural distribution of the data.

As mentioned above, machine learning does not deal with typical structures like arrays and scalars but deals with tensors. A conversion from NumPy arrays to tensors is required and done as shown by the code below:

```
1 X_training = torch.Tensor([entry[0] for entry in training_dataCnn]).view
    (-1, rowsIn_shape, columnsIn_shape)
2 Y_training = torch.Tensor([entry[1] for entry in training_dataCnn])
3
4 X_testing = torch.Tensor([entry[0] for entry in testing_dataCnn]).view(-1,
    rowsIn_shape, columnsIn_shape)
5 Y_testing = torch.Tensor([entry[1] for entry in testing_dataCnn])
```

Regarding the code above, the X of training and testing represents the actual images, and the Y represents the labels. The tensors are not shuffled anymore in this part of the code: Images and classes share the same positions.

A common practice to increase the efficiency and speed of training is to group the training images in **batches**. The program will not have to store errors for all the data, only for the batches. Another term to understand is **epochs**. One epoch represents a successful pass of the algorithm over the training data. The optimal epoch value is vital to find the model with

the lowest error in the loss function. The testing of these values is presented in the results section of the report.

After successfully assigning the number of epochs needed and the batch size, choosing an optimizer function and a loss function is required.

```
1 optimizer = optim.Adam(cnn.parameters(),lr=0.001)
2 lossCriteria = nn.MSELoss()
```

Common choices for these functions are:

- **Optimizer:** A gradient descent function like Adam optimizer is commonly used in CNNs. It adapts the weights to minimize the loss of function. Its defined as a hill and the end goal is to move the coefficient downhill representing a lower error. The gradient directs the descent and is defined with a learning rate of 0.001. The weights will be updated by 0.001 each time in either direction [16].
- **Loss function:** Because we are using one-hot vectors in a sort of order, a mean squared error loss function is a good fit. The error between prediction and actual label is square and then averaged [16].

The training is done performed over the number of epochs we have, iterating over the data and incrementing by the number of batch sizes:

```
1     for i in tqdm(range(epochs)):
2         for j in range(0,len(X_training),batchSize):
3
4             # Batches of images
5             batchX = X_training[j:j+batchSize].view([-1,1,rowsIn_shape,
columnsIn_shape]).to(device)
6             batchY = Y_training[j:j+batchSize].to(device)
7
8             cnn.zero_grad()
9             prediction_CNN = cnn(batchX)
10            loss_prediction = lossCriteria(prediction_CNN,batchY)
11            trainingLoss.append(loss_prediction)
12            loss_prediction.backward()
13            optimizer.step()
```

To use the GPUs, the data is sent to the graphic units's memory using the toDevice attribute. TQDM is a library to help track the progress of training. It outputs a progress part showing us a percentage of computation and also allows us to track the timing required for training.

The following steps are:

1. Create the batches
2. Reset all gradients of the optimizer to none
3. Pass the data through the CNN
4. Compute the loss
5. Compute the gradient

6. Update the parameters

The training was done using 75% of the data. The rest was used for validation to check the model's accuracy and its ability to predict correctly. The algorithm to validate the model is straightforward, indicates the class, and compares it to the actual label. All the results are stored in variables.

```
1  with torch.no_grad():
2      for i in tqdm(range(len(X_testing))):
3
4          outCNN = cnn(X_testing[i].view([-1,1,rowsIn_shape,
5              columnsIn_shape])).to(device)
6
7          prediction = torch.argmax(outCNN[0])
8          correctLabel = torch.argmax(Y_testing[i]).to(device)
9
10         if prediction == correctLabel:
11             validationDict["correct"] += 1
12         else:
13             validationDict["wrong"] += 1
14
15         validationDict["total"] += 1
```

As seen in the code above, all the validation is done using a no_grad function where the gradient is none. The function argmax() outputs the label's maximum score, which represents the final prediction.

7 Results

7.1 Engineering results

7.1.1 Data Generation System

The overall data generation system was called to create a complete data-set of 1000 images. In this part of the report, the outputs of each section are presented to understand the testing strategy implemented and to show the accurate analysis of a compiled assembly.

Pattern Matching:

MPLAB IDE provided a disassembly function on compiled C code. This IDE was used for manual testing between the disassembly file from the system and the IDE function. However, it is also used to justify the use of this data generator system. At the beginning of the research, a manual implementation of the disassembly process was done using the IDE and then analysed using a python script. The analysis report is shown in Figure 9.

```

4  |
5  | ['XORLW', 'SUBWF', 'BTFS']    [3, {'lines': [[93, 96], [126, 129], [288, 291]]}]
6  |
7  | ['ADDLW', 'BTFS', 'GOTO']    [2, {'lines': [[159, 162], [180, 183]]}]
8  |
9  | ['LSLF', 'ADDWF', 'MOVWF']    [2, {'lines': [[162, 165], [183, 186]]}]
10 |
11 | ['MOVWF', 'LSLF']            [2, {'lines': [[212, 214], [238, 240]]}]
12 |
13 | ['BTFS', 'GOTO']            [6, {'lines': [[58, 60], [128, 130], [132, 134], [160, 162], [290, 292], [294, 296]]}]
14 |
15 | ['MOVF', 'SUBWFB']           [2, {'lines': [[44, 46], [74, 76]]}]
16 |
17 | ['XORLW', 'SUBWF', 'BTFS', 'GOTO', 'MOVF', 'SUBWF', 'BTFS', 'GOTO', 'GOTO'] [2, {'lines': [[126, 135], [288, 297]]}]
18 |
19 | ['ADDLW', 'MOVWF', 'MOVLW'] [7, {'lines': [[9, 12], [48, 51], [78, 81], [84, 87], [111, 114], [117, 120], [138, 141]]}]
20 |
21 | ['BTFS', 'GOTO', 'GOTO', 'GOTO', 'MOVLB', 'MOVF'] [2, {'lines': [[132, 138], [294, 300]]}]
22 |
23 | ['MOVLW', 'ADDWFC']          [5, {'lines': [[50, 52], [80, 82], [86, 88], [140, 142], [282, 284]]}]
24 |
25 | ['ADDWF', 'MOVLW']           [2, {'lines': [[258, 260], [262, 264]]}]
26 |
27 | ['RLF', 'MOVF']              [2, {'lines': [[216, 218], [242, 244]]}]
28 |
29 | ['ADDWFC', 'MOVWF', 'MOVF'] [7, {'lines': [[6, 9], [12, 15], [51, 54], [81, 84], [87, 90], [114, 117], [120, 123]]}]
30 |
31 | ['MOVF', 'SUBWF']            [2, {'lines': [[130, 132], [292, 294]]}]
32 |
33 | ['GOTO', 'MOVF', 'SUBWF']    [3, {'lines': [[96, 99], [129, 132], [291, 294]]}]
34 |
35 | ['SUBWFB', 'MOVWF', 'MOVF', 'ADDLW', 'MOVWF'] [2, {'lines': [[45, 50], [75, 80]]}]
36 |
37 | ['XORLW', 'SUBWF', 'BTFS', 'GOTO', 'MOVF', 'SUBWF'] [2, {'lines': [[126, 132], [288, 294]]}]
38 |

```

Figure 9: Screenshot of the pattern matching report showing potential repetitive pattern and there location

This output shows repetitive potential patterns that could be accelerated. Each repetition of a possible pattern where computation instructions exist is inserted into a dictionary and displayed in the report with its position. The sequences of instructions represent also an opportunity for pipelining. Regardless, analysing the memory addresses patterns and the instructions patterns in this application to predict accelerated candidates is near impossible. And therefore the idea of using an automated data generation system with an ML model.

C code generator and compilation:

The first part of the system was the C generator. A python script computes a C file with different parameters for each application. Figure 10 shows the binomial coefficient C script for the loop dependent classification.

```
1  #include<stdio.h>
2  void main() {
3  int i, j, n, k,result, min, c[6][6]={0};
4  n = 5 ;
5  k = 5 ;
6  if(n >= k) {
7  for(i=0; i<=n; i++) {
8  min = i<k? i:k;
9  for(j = 0; j <= min; j++) {
10 if(j==0 || j == i) {
11 c[i][j] = 1;
12 } else {
13 c[i][j] = c[i-1][j-1] + c[i-1][j];
14 } } }
15 result = c[n][k];
16 } }
17
```

Figure 10: Screenshot of random generated C code for a binomial coefficient application.

In this random state, the values of variables n and k are equal to 5. The resulting arrays also have a size of $5*5$ with 0 as coefficients. Each time, these values would be different, and you can see from this code that the python script produces an actual C algorithm. Each application was tested multiple times to ensure a change in the compilation and bug-free codes. Table 2 shows the summary of the unit testing.

Table 2: Table summarizing the unit testing of the C generation files

Application	Random Variables	Functionality	Accuracy (%)	Possible-Issues
Fibonacci (LP)	size / i	Yes	100	Same-assembly generation
Binomial coefficients (LP)	n / k / size	Yes	100	
Dependent Arrays (LP)	Arrays	Yes	74	Memory full
Swap Arrays (ULP)	Arrays / Storing	Yes	83	Memory full
Add Arrays (ULP)	Arrays / Storing	Yes	85	Memory full
FIR filter (ULP)	Arrays / Coeff / Storing	Yes	94	Memory full

For each application, the variables generated randomly are stated in the table above. Each application presents a functionality statement which tests for syntax errors inside the c code. All applications passed this requirement. However, some other issues were seen. The unit testing presents the accuracy of all applications created, showing how many compilations failed. The errors were summarized in the last column of the table.

The c files are converted to a hexadecimal file that the CPU simulator can read during compilation. The screenshot 11 below shows an example of a .hex file generated:

```

1  :08000000803102288731F02F46
2  :1001B400FE0012001E00FE0BDB28003472087104DE
3  :1001C4007004031908008030F2060800F300F201FD
4  :1001D400731CF0287008F1007108F2070310F00D89
5  :1001E4000310F30C7308031DEA2872080800F401D5
6  :1001F400F501701C01297208F4077308F53D0130FC
7  :10020400F235F30D890B02290130F136F00C890B1C
8  :10021400072970087104031DFB287508F100740890
9  :10022400F000080039083804370403191B2980300A
10 :10023400B9063A08AA003B08AB003C08AC003708F2
11 :10024400AD003808AE003908AF008531F4252A081E
12 :10025400B7002B08B8002C08B9000800003400349B
13 :10026400403400340A344234CD34543441340034FC
14 :1002740040344134CD349F34423433343534423401
15 :10028400C334ED344034CD348C343F340034B8348A
16 :100294004234003404344234FD01F91F5829F8096A
17 :1002A400F909F80A0319F90AFD01FD0A7808F000B2
18 :1002B4007908F100F2018E30FB007B08F3007D0821
19 :1002C400FC007C08F400833103237008F8007108F3
20 :1002D400F9007208FA000800F21F7D297008003C3A
21 :1002E400F0007108031C710F003CF1007208031C3C
22 :1002F400720F803CF200F51F8C297308003CF30058

```

Figure 11: Screenshot of .hex file which represents one of the output of the compilation process

Nevertheless, some issues with some applications were raised, and the main one was that for some c files, the memory used passed the limit. It means the compilation could not free space for some variables! Decreasing the size of possible random variables will lead to a minor variance between applications and, therefore, less generalization in training.

```

Memory Summary:
Program space      used  748h ( 1864) of 1000h words ( 45.5%)
Data space        used   B1h (  177) of  100h bytes ( 69.1%)
EEPROM space      used    0h (    0) of  100h bytes (  0.0%)
Configuration bits used    0h (    0) of    2h words (  0.0%)
ID Location space used    0h (    0) of    4h bytes (  0.0%)

```

Figure 12: Screenshot of program memory statement which represents one of the output of the compilation process

Figure 12 shows the output of the XC8 command-line compiler, which defines the percentage of memory used.

```

8  0: (500) undefined symbols:
9  |   _free(convolution.obj) _calloc(convolution.obj)
10 (908) exit status = 1
11 (908) exit status = 1

```

Figure 13: Screenshot of errors statement which represents one of the output of the compilation process

With the combination of the error statement shown in Figure 13, an algorithm was added to count failures and regenerate failed applications until the demanded number of files per application is generated without any errors. The trade-off was the processing power and timing, as this could be an expensive process for the CPU resources.

Disassembly:

```

10 00df: 3400 retlw 0x00
11 00e0: 0872 movf 0x72, 0x0
12 00e1: 0471 iorwf 0x71, 0x0
13 00e2: 0470 iorwf 0x70, 0x0
14 00e3: 1903 btfsc 0x03, 0x2
15 00e4: 0008 return
16 00e5: 3080 movlw 0x80
17 00e6: 06f2 xorwf 0x72, 0x1
18 00e7: 0008 return
19 00e8: 00f3 movwf 0x73
20 00e9: 01f2 clrf 0x72
21 00ea: 1c73 btfss 0x73, 0x0
22 00eb: 28f0 goto 0x00f0
23 00ec: 0870 movf 0x70, 0x0
24 00ed: 00f1 movwf 0x71
25 00ee: 0871 movf 0x71, 0x0
26 00ef: 07f2 addwf 0x72, 0x1
27 00f0: 1003 bcf 0x03, 0x0
28 00f1: 0df0 rlf 0x70, 0x1
29 00f2: 1003 bcf 0x03, 0x0
30 00f3: 0cf3 rrf 0x73, 0x1
31 00f4: 0873 movf 0x73, 0x0
32 00f5: 1d03 btfss 0x03, 0x2
33 00f6: 28ea goto 0x00ea
34 00f7: 0872 movf 0x72, 0x0
35 00f8: 0008 return

```

Figure 14: Screenshot of random 25 lines of the disassembly produced by the system

The hexadecimal machine code was disassembled to produce a compiled code. To check the system's accuracy, as mentioned before, a comparison was made between the assembly file created by the system and the disassembler function provided by the MPLAB IDE with the same configuration applied. Figure 14 provides a screenshot of the disassembler file. Figure 15 delivers a comparison made with Visual Studio Code, the code editor used in this project.

2- 07ee: 3538	lsf	0x38, 0x0	2+ 07E1 3538	LSLF i, W
3- 07ef: 3e20	addlw	0x20	3+ 07E2 3E20	ADDLW 0x20
4- 07f0: 0086	movwf	0x06	4+ 07E3 0086	MOVWF FSR1
5- 07f1: 0187	clrf	0x07	5+ 07E4 0187	CLRF FSR1H
6- 07f2: 0834	movf	0x34, 0x0	6+ 07E5 0834	MOVF n3, W
7- 07f3: 3fc0	movwi	.0[1]	7+ 07E6 3FC0	MOVWI 0[FSR1]
8- 07f4: 0835	movf	0x35, 0x0	8+ 07E7 0835	MOVF 0x35, W
9- 07f5: 3fc1	movwi	.1[1]	9+ 07E8 3FC1	MOVWI 1[FSR1]

Figure 15: Comparaision between two disassembly file produce by the system and the MPLAB IDE

As you can notice, there are some differences between the disassembler. Yet, this difference does not mean a different functionality. The addresses assigned might be different, but the pattern in which they occur should be the same. Line addresses are given in various formats: hexadecimal and integer. Almost all instructions happen in the pattern type with a slightly different order depending on out-of-order implementation. Overall, the system's performance is proved to have relatively reasonable accuracy.

Vectors generation:

The comma-separated variables produced after conversion to vector sequences are plotted using pyplot from the matplotlib library.

```
1 plt.imshow(image.T, cmap='gray')
2 plt.show()
```

The plot shown in Figure 16 is the transpose matrix of the image in a grayscale colour map.

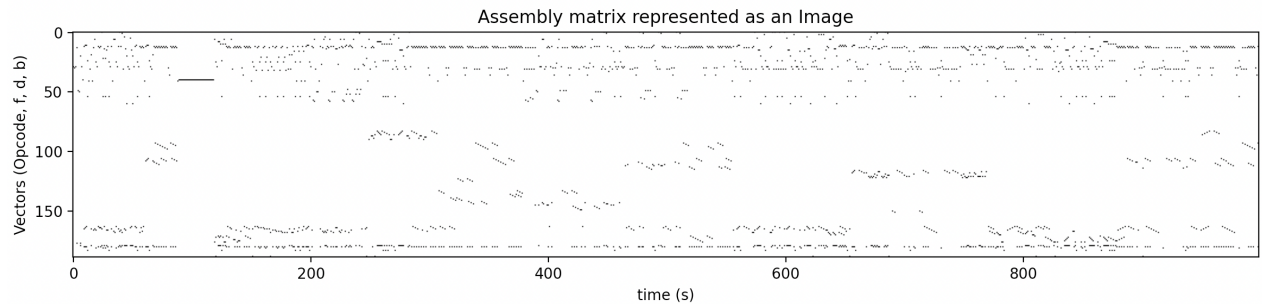


Figure 16: Plot of a graphical representation of sigmoid assembly matrix application

The x-axis represents the timeline of the instruction. The y-axis represents the vectors, as explained in the methods section. As stated in the previous work [13], we can see that the analysis of this graph might be optimal using a CNN as they are proven to perform well with 2D black and white images. Some patterns can be visualised manually, but it is pretty hard to determine some potential accelerated candidates and yet the use of new technologies like machine learning.

The last graph 17 shows a similar graphical representation, but in this case, it is a random pattern generated by the algorithm to train the CNN on random states.

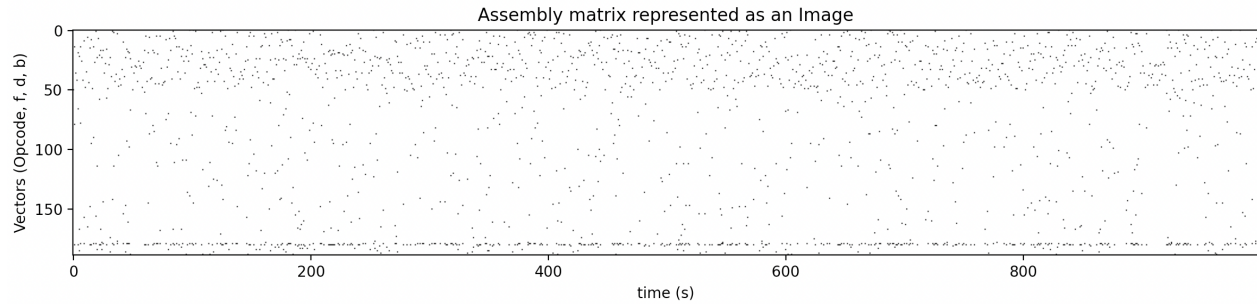


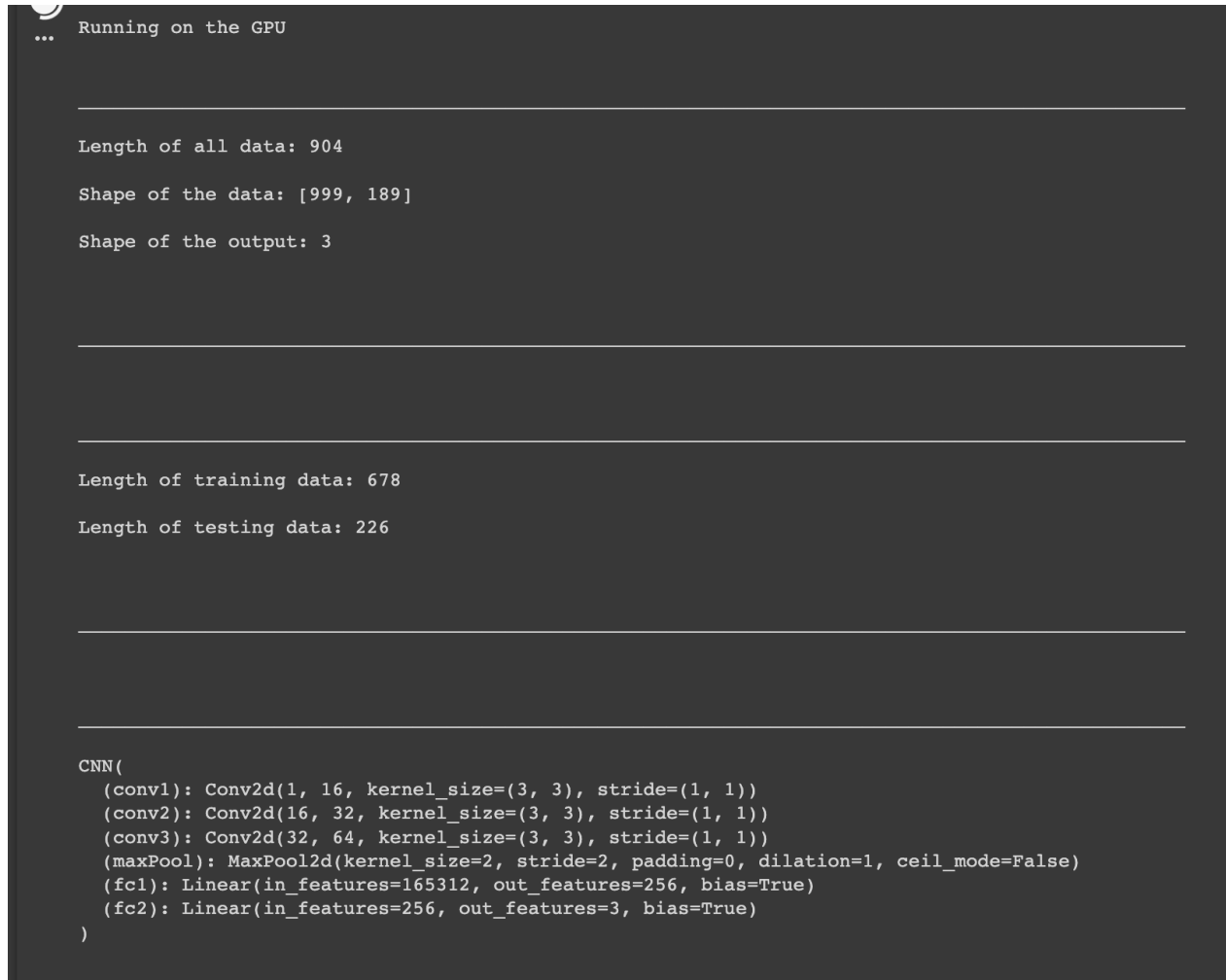
Figure 17: Plot of a graphical representation of random assembly matrix generated

These random patterns were designed to train the CNN to assign an unknown state to any pattern it does not recognize.

In summary, the whole system was created and tested to provide balanced, relevant datasets as inputs for the CNN training.

7.1.2 CNN

The CNN model was trained using a google cloud service cost-free with access to GPUs. After creating the Jupyter notebooks, the model's training was performed using several EPOCHS and batch size numbers. A complete report of the operation is seen in Figure 18.



```

... Running on the GPU

Length of all data: 904

Shape of the data: [999, 189]

Shape of the output: 3

Length of training data: 678

Length of testing data: 226

CNN(
  (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (maxPool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=165312, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=3, bias=True)
)

```

Figure 18: Screenshot reporting the initial configuration of the CNN model

First, the training is run on the GPUs decreasing running time and improving efficiency. This statement will only be displayed if a GPU is available. Then, the data is split between training data and testing data. Training data = $\frac{678}{904} = 0.75$ or 75%, Moreover Testing data a = $\frac{226}{904} = 0.25$ or 25%. The shuffling behaviour was tested manually to ensure random shuffling between data. Finally, the CNN presents the same expected structure illustrated in Figure 8 and implemented in the methods section. The input shape of (999,189) is expected. However, the output shape is said to be 3. The one-hot vector encodement can explain this behaviour.

- [0,0,1] -> represents the position 0 or "LP"
- [0,1,0] -> represents the position 1 or "ULP"
- [1,0,0] -> represents the position 2 or "UN"

This ensures a simple comparison between the predicted and the real class when training and validating.

After training, the validation using 25% delivered a log file shown in Figure 19.

A screenshot of a terminal window with a dark background and light gray text. The text displays the final results of a CNN's testing. The output is as follows:

```
Validation:  
  
Correct Prediction:226  
  
Wrong Prediction:0  
  
Total Prediction:226  
  
Accuracy:1.0
```

Figure 19: Screenshot reporting the final result of the CNN's testing using 25% data

The model took 11.3 minutes to train using GPUs in the cloud service compared to 17.7 minutes on traditional CPUs in a standard computer. This result shows the efficiency of using GPUs for machine learning and neural networks.

In a total of 226 predictions, there were an unexpected 0 wrong predictions and 226 correct predictions, leaving the model with an accuracy of 100%. Further explanation is given in the discussion section.

To analyse the performance and accuracy of the model, two graphs were created:

First, Figure 20 highlights the loss function result against the number of EPOCHS.

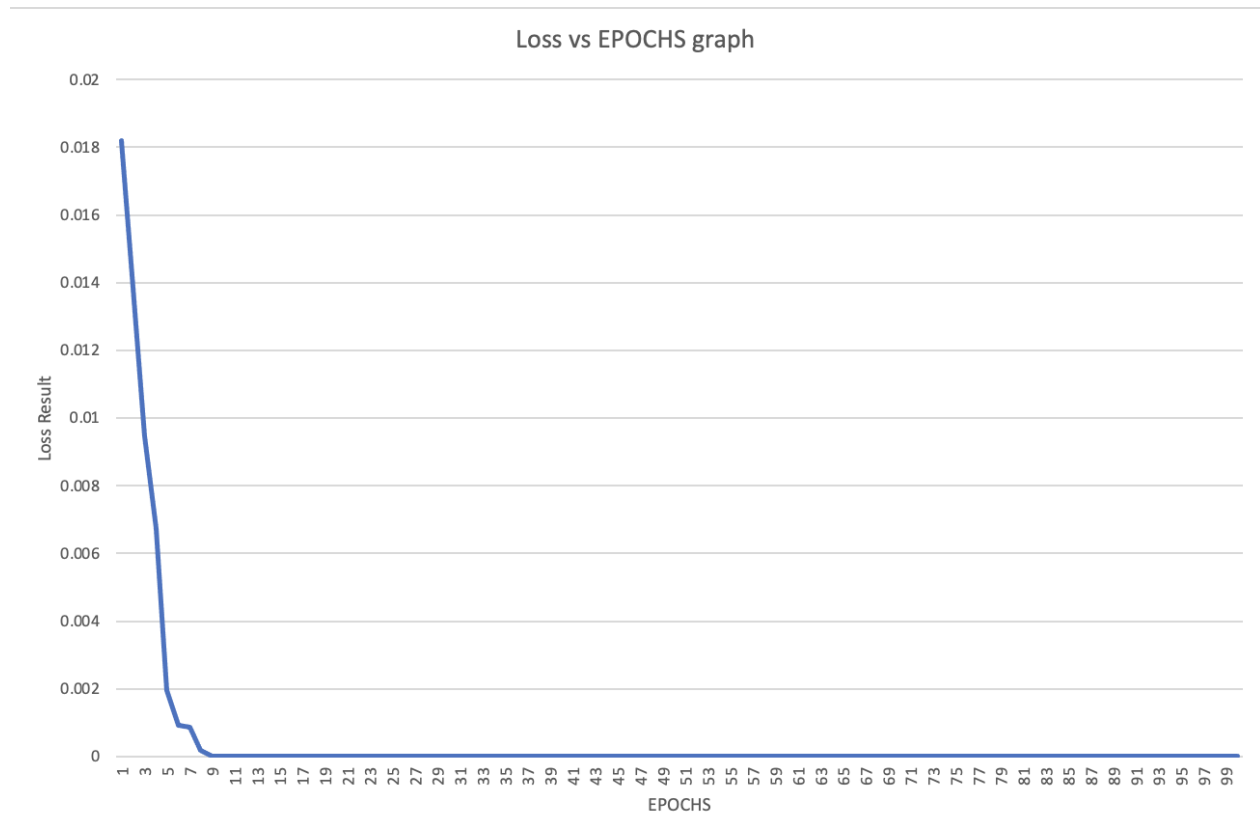


Figure 20: Graph of the loss function results against the number of EPOCHS

The model is training and adapting the weights accordingly to reduce the loss function results (difference between predicted and actual classification). The plot shows the decrease of the loss function when training the data. It reaches a constant minimum at around 9 EPOCHS leaving the choice of 100 EPOCHS to be more than necessary and risk over-fitting.

Second, Figure 21 highlights the accuracy of the data fed through the model). An average around the first 20 EPOCHS was taken to give the graph a clear view.

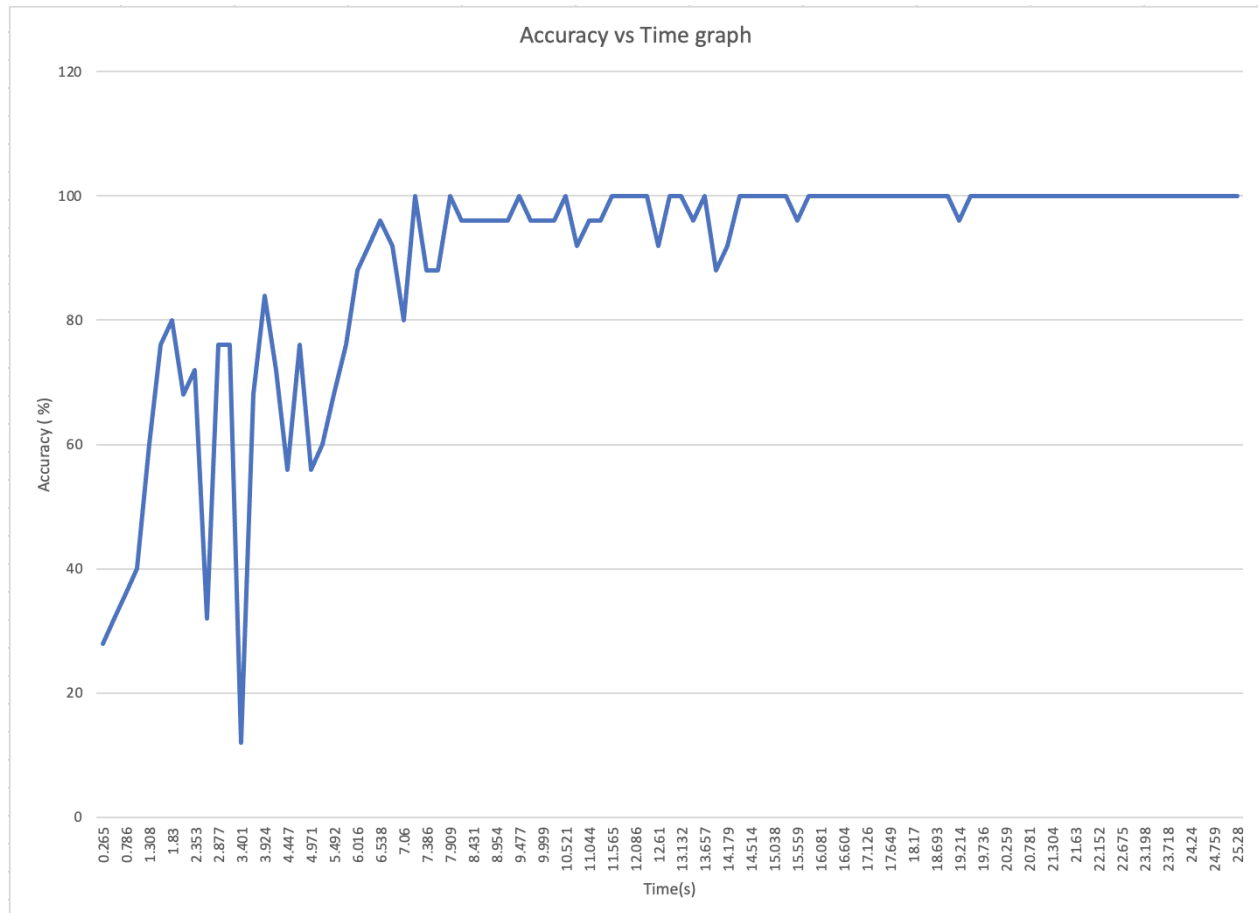


Figure 21: Graph of the accuracy in training vs Time in timestamps

Analysing the graph, a rapid random change in accuracy in the first five timestamps shows the model training on multiple classes. However, the overall trend is still upwards, leaving us to deduct an improvement in prediction. The graphs reach 100% accuracy in around 15 timestamps.

This model was then saved as .pt file to test some heavy applications, as mentioned before with a torch method. All weights, structure and entire model are saved.

```
1 torch.save(cnn, 'model.pt')
```

7.2 Experimental results

The engineering results proved a working algorithm with a data generation system.

After training, the model is loaded,

```
1 cnn = torch.load("model.pt",map_location=torch.device('cpu'))
2 cnn.eval()
```

and it is manually tested on:

- Unseen data, however, these matrices are the same application that the model is trained.
- New heavy applications, like convolution and sigmoid operation. The labels of these applications are unknown.

Table 3, provides a summary of the testing of this data on our trained best model.

Table 3: Table summarizing the experimental testing on the trained model

Application	Repetitions	Accuracy (%)	Score 0	Score 1	Score 2	Output
LP	10	100	$9.9996e^{-01}$	$3.6805e^{-05}$	$3.0551e^{-09}$	tensor(0)
ULP	10	100	$8.0333e^{-04}$	$9.9920e^{-01}$	$2.8624e^{-07}$	tensor(1)
UN	10	100	$1.3469e^{-36}$	$3.9983e^{-31}$	$1.0000e^{+00}$	tensor(2)
Convolution	7	NaN	$2.1190e^{-27}$	$1.2603e^{-23}$	$1.0000e^{+00}$	tensor(2)
Sigmoid	9	NaN	$1.7192e^{-28}$	$4.6e^{-25}$	$1.0000e^{+00}$	tensor(2)

The table shows that manual testing of 10 images per application for the three classes gave the correct prediction and an accuracy of 100%. The highest average score represents the prediction outputted by the model. This shows that the model works on these data and predicts and analyses to classify images.

On the unseen data, heavy applications like Convolution and Sigmoid, multiple images were also produced using the data generation system. The data was passed through the trained model, and an unknown state was predicted for all graphs of convolution and sigmoid with a high score of 1. It means that the model assigns the assembler process of the C application as unknown patterns and could not spot any acceleration pattern. Figure 22 shows the plot of the convolution graphical representation.

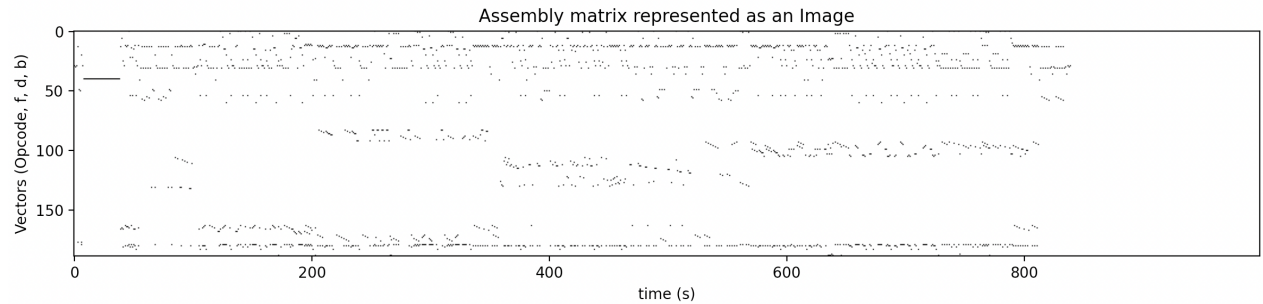


Figure 22: Plot of a graphical representation of convolution assembly matrix application

This experimental result of the research shows that the model could not find any loop dependent or independent inside an unknown application. Even though CNN has proven a promising analysis tool for assembly patterns, it could not find any potential accelerated patterns in the newly compiled assembly. The possible explanation for this behaviour is explained in the following discussion section.

8 Discussions

This section highlights the possible issues that led to the behaviour seen in the experimental and engineering results. Also, we introduce a plan and potential future work research to continue the project.

8.1 Over fitting

To explain the model's behaviour, a possible most likely issue is "fitting".

While all algorithms try to implement models and training with optimisation techniques to rule out underfitting, after analysis of the graphs presented in the results, the most likely reason for high accuracy of 100% and a prediction of unknown states in never seen data is "overfitting".

To explain **overfitting**, Figure 23 shows the likely scenario where this error occurs.

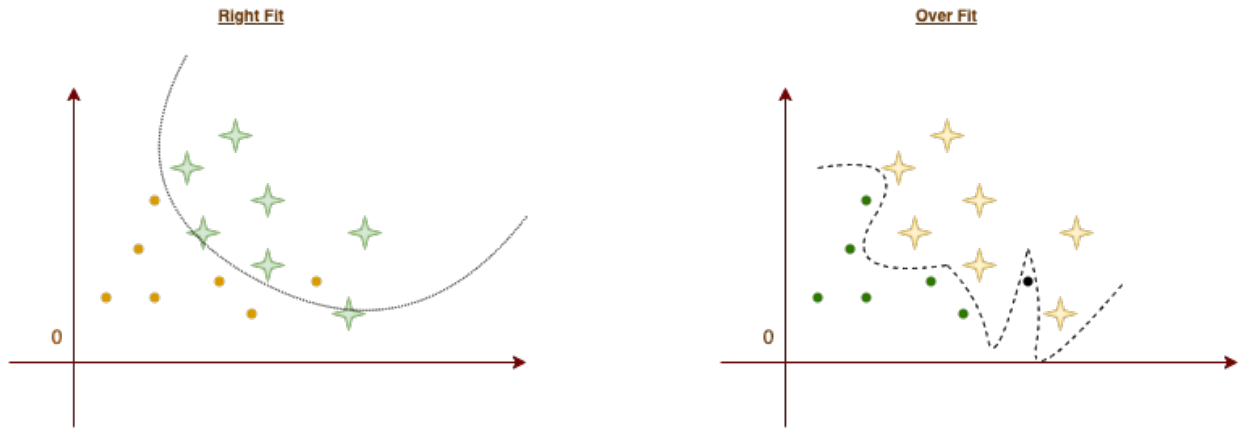


Figure 23: Graph representing the overfitting scenario in ML

This scenario is caused when the data fits too well. In the figure on the left, a quadratic function fits the data and generalises well to new unseen data [19]. In comparison, the function on the right overfits the data causing a lack of possible generalisation to new data. The model might have a low result in the loss functions and high accuracy, but the model does not generalise well to the new population of images as expected. The model performs excellently on the training and validation data but poorly on the testing data (Convolution and Sigmoid). This overfitting occurs when there is a gap between the training and test errors, as seen in the results section. The model [31] has learned and updated its weights in a manner that is applicable for only the training data.

The paper referenced here [32] explains some possible reasons for oversitting. The possibility of noise inside the data can cause overfitting. In this project, random behaviours can represent noise by the compiler. The memory address assignment for the registers and the optimisation techniques are examples of this process (out-order-executions). It also states the trade-off between accuracy and consistency. Having a relatively great accuracy in the training

dataset leads to the potential to believe that consistency in a different dataset like the heavy applications are low and therefore unrecognisable pattern with a high score of 1.

8.2 Data hypothesis

So, in summary, one hypothesis is that the data generation process causes this whole behaviour. There is two main assumptions usually in a dataset:

- **Example on the datasets are independent**
- **Data is identicly distributed**

The point of training this CNN using multiple applications was to improve pattern prediction in unseen data. The hypothesis was that the loops inside basic standard applications represent the generalisation of loop patterns inside any complex assembly code.

First, the restrictions (processing time) on data generation lead to small datasets of 900 images. Also, after analysis, the one hypothesis drawn is that some applications' assembly code generation changes are not enough to assume independence between the data. The applications are too specific and do not show a general pattern aspect. Indeed, the compiler produces an almost similar instructions pattern between two applications leading to a lack of generalization.

Another view is that a large portion of the c files is storing the results into variables or arrays. If the model learns about storing and assignment behaviour rather than computation, that could lead to a problem when predicting other applications.

8.3 Possible Solutions

There are three possible solutions to fixing and improving the generalization process of the ML machine. **Data augmentation** [33] is a necessity in this project. Producing more data and adding them to the training dataset can overcome overfitting. The data also needed to be more independent of each other. Tuning the data generation system to produce C files with different implementations but the same application will let the compiler assemble completely different matrices. This data expansion should be accompanied by hyperparameters sweep [32] to fine-tune the CNN model. In the data expansion, it is also possible to include noise in the data or give multiple versions of the same C file where memory addresses are different. This will hopefully let the CNN adapt to the randomness of the compiler.

"Early stopping" implementation in the algorithm [32]. When training the model, a stop system can be implemented to stop overtraining when accuracy remains almost constant, or loss of function reduction is smaller than a predefined value.

Other solutions stated in the paper [32] are network reduction to reduce noise and regularization, which means having fewer features and simplifying the model. Nevertheless, these methods cannot be implemented in this project as they are not relevant or possible to the current issues raised above.

8.4 Future Work

The conclusion of the research done with an in-house compiler [13]: "CNN failing to detect head and tail of a Loop" combined with assignment of unknown state in this project using a real-time compiler, led to the belief that this analysis is more complex than it looks. However, an optimistic view can be assumed as machine learning in both researches has proven to be highly efficient in graphical analysis of the assembly compiled code. Moreover, at first view of an assembler's process, patterns do exist and hopefully can be accelerated using an FPGA as a coprocessor. In future work, I will recommend implementing the possible solutions to overfitting. Also, expanding the datasets by finding a new type of patterns and application would add great benefits and even leads to a generalization of patterns for the CNN training.

In this project, the algorithms focuses only on data parallelism and, more specifically, loop-level parallelism. The need to enforce task parallelism as a possible pattern type would also be beneficial to expanding the datasets.

This whole work also opens new questions about the behaviour of compilers and optimizers. Would patterns available on one type of compiler perform or execute in the same manner as the one used in this research? Without this, the study would only apply to one compiler type and, therefore, less impact on possible acceleration. Further analysis is required about the generalization of compilers.

9 Conclusion

In summary, a software system based on Convolutional Neural Networks (CNNs) capable of learning how to recognise and correctly classify sequences of machine code instructions was created. The AI system produces a classification which can be used to identify candidates for implementation using a reconfigurable coprocessor, with the ultimate goal of accelerating system performance. It takes as input a graphical representation of any pre-compiled assembly. A Data Generation System capable of producing labelled datasets suitable for the training of CNNs, and a Control System to manage the necessary supervised learning were also produced alongside the neural network. Using this system, 900 synthetics "images" representing a range of different applications were built and fed into the model for training. Despite the limitations of available compute resources, training the CNN using 75% of the synthetic dataset led to a 100% classification accuracy on the remaining (previously-unseen) 25%. We also used the system to analyse applications not represented in the original training set, exposing the need for training using much larger datasets than those constructed. The lack of data led to overfitting and a lack of generalisation of the new pattern population. Solutions like "early stopping" and data augmentation were presented alongside a plan for future work to continue this study. An idea includes the use of loop parallelism and task parallelism. That means training the CNN on a broader range of applications.

This report has proven machine learning as a powerful tool for analysis and potentially hardware acceleration. Despite the incompleteness and uncertainty, it supplies a promising beginning in a new hardware acceleration process. The background explains the importance of this research in today's world and the methods used to highlight the significance of AI in hardware acceleration.

References

- [1] János Véghe. “How Amdahl’s Law limits the performance of large artificial neural networks.” In: *Brain Informatics* 6.1 (2019). ISSN: 2198-4018. DOI: 10.1186/s40708-019-0097-2.
- [2] Elie Track, Nancy Forbes, and George Strawn. *The End of Moore’s Law*. 2017. DOI: 10.1109/MCSE.2017.25.
- [3] Adrian McMenamin. *The end of Dennard scaling*. 2013. URL: <https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/>.
- [4] Esraa Shehab, Alsayed Algergawy, and Amany Sarhan. “Accelerating relational database operations using both CPU and GPU co-processor.” In: *Computers and Electrical Engineering* 57 (2017). ISSN: 00457906. DOI: 10.1016/j.compeleceng.2016.12.014.
- [5] Lin Bai, Yiming Zhao, and Xinming Huang. “A CNN Accelerator on FPGA Using Depthwise Separable Convolution.” In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.10 (2018). ISSN: 15497747. DOI: 10.1109/TCSII.2018.2865896.
- [6] Stephen Charlwood, Jonathan Mangnall, and Steven Quigley. “System-level modelling for performance estimation of reconfigurable coprocessors.” In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 2438 LNCS. 2002. DOI: 10.1007/3-540-46117-5_59.
- [7] HEAVY.AI. *Hardware Acceleration*. 2022. URL: <https://www.heavy.ai/technical-glossary/hardware-acceleration>.
- [8] Lin Bai, Yiming Zhao, and Xinming Huang. “A CNN Accelerator on FPGA Using Depthwise Separable Convolution.” In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.10 (2018). ISSN: 15497747. DOI: 10.1109/TCSII.2018.2865896.
- [9] SparkFun. *USING FPGAs*. 2020. URL: <https://www.sparkfun.com/fpga>.
- [10] Arnab Chakraborty. *Data parallelism vs Task parallelism*. 2019. URL: <https://www.tutorialspoint.com/data-parallelism-vs-task-parallelism>.
- [11] Wikipedia. *Loop-level parallelism*. 2022. URL: https://en.wikipedia.org/wiki/Loop-level_parallelism#cite_note-Solihin-1.
- [12] GeeksforGeeks. *Bernstein’s Conditions in Operating System*. 2020. URL: <https://www.geeksforgeeks.org/bernsteins-conditions-in-operating-system/#:~:text=Bernstein’s%20Conditions%20are%20the%20conditions,still%20produce%20the%20same%20result..>
- [13] R. Luis Jalabert. “Deep Learning Based FPGA-CPU Acceleration.” In: (2018). URL: https://github.com/LuisJalabert/Deep-Learning-Based-FPGA-CPU-Acceleration/blob/master/Deep_Learning_Based_CPU_Acceleration_-_Luis_Jalabert_December_2018.pdf.
- [14] Javatpoint. *Compilation process in c*. 2020. URL: <https://www.javatpoint.com/compilation-process-in-c#:~:text=The%20compilation%20is%20a%20process,it%20generates%20the%20object%20code..>

- [15] Microchip. *MPLAB® XC8 C Compiler User's Guide*. 2020. URL: <http://ww1.microchip.com/downloads/en/devicedoc/50002053g.pdf>.
- [16] Adam Gibson Josh Patterson. *Deep Learning*. O'REILLY, 2017.
- [17] Christian Janiesch, Patrick Zschech, and Kai Heinrich. "Machine learning and deep learning." In: *Electronic Markets* 31.3 (2021). ISSN: 14228890. DOI: 10.1007/s12525-021-00475-2.
- [18] MathWorks. *Introducing Deep Learning with MATLAB*. URL: <https://uk.mathworks.com/campaigns/offers/next/deep-learning-ebook.html>.
- [19] Aaron Courville Ian Goodfellow Yoshua Bengio. *Deep Learning*. Massachusetts Institute of Technology, 2017.
- [20] VIJAYSINH LENDAVE. *What Is A Convolutional Layer?* 2021. URL: <https://analyticsindiamag.com/what-is-a-convolutional-layer/>.
- [21] R. Luis Jalabert. "Deep Learning Based FPGA-CPU Acceleration." In: *EMECS* (2019). URL: <https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2627032/no.ntnu%5C%3Ainspera%5C%3A39095155%5C%3A46487432.pdf?sequence=1&isAllowed=y>.
- [22] Qiong Wei and Roland L. Dunbrack. "The Role of Balanced Training and Testing Data Sets for Binary Classifiers in Bioinformatics." In: *PLoS ONE* 8.7 (2013). ISSN: 19326203. DOI: 10.1371/journal.pone.0067863.
- [23] Microchip Technology Inc. *MPLAB® XC8 C Compiler User's Guide for PIC® MCU*. 2021. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB_XC8_C_Compiler_Users_Guide_for_PIC_50002737.pdf.
- [24] Microchip Technology Inc. *8-Bit Microcontroller Summary*. 2022. URL: <https://microchipdeveloper.com/8bit:summary#:~:text=The%5C%208%5C%2Dbit%5C%20family%5C%20has,14%5C%2Dbit%5C%20wide%5C%20program%5C%20memory>.
- [25] Microchip Technology Inc. *PIC12F1840*. 2022. URL: <https://www.microchip.com/en-us/product/PIC12F1840>.
- [26] James Bowman Craig Franklin. *GNU PIC Utilities*. URL: <https://gputils.sourceforge.io/>.
- [27] DeepAI. *One Hot Encoding*. URL: <https://deepai.org/machine-learning-glossary-and-terms/one-hot-encoding>.
- [28] Pytorch. *PYTORCH DOCUMENTATION*. 2019. URL: <https://pytorch.org/docs/stable/index.html>.
- [29] sentdex. *Convolutional Neural Network Model - Deep Learning and Neural Networks with Python and Pytorch p.6*. 2019. URL: <https://pythonprogramming.net/convnet-model-deep-learning-neural-network-pytorch/?completed=/convolutional-neural-networks-deep-learning-neural-network-pytorch/>.
- [30] Google. 2022. URL: https://colab.research.google.com/?utm_source=scs-index.

- [31] Simukayi Mutasa, Shawn Sun, and Richard Ha. *Understanding artificial intelligence based radiology studies: What is overfitting?* 2020. DOI: 10.1016/j.clinimag.2020.04.025.
- [32] Xue Ying. “An Overview of Overfitting and its Solutions.” In: *Journal of Physics: Conference Series*. Vol. 1168. 2. 2019. DOI: 10.1088/1742-6596/1168/2/022022.
- [33] Soumya Joshi et al. “Issues in Training a Convolutional Neural Network Model for Image Classification.” In: *Communications in Computer and Information Science*. Vol. 1046. 2019. DOI: 10.1007/978-981-13-9942-8_27.
- [34] Robert Cottrell. *FPGA Coprocessors: Hardware IP for Software Engineers*. URL: <https://www.design-reuse.com/articles/6733/fpga-coprocessors-hardware-ip-for-software-engineers.html>.

10 Appendices

10.1 Appendix 1: FPGA Structure

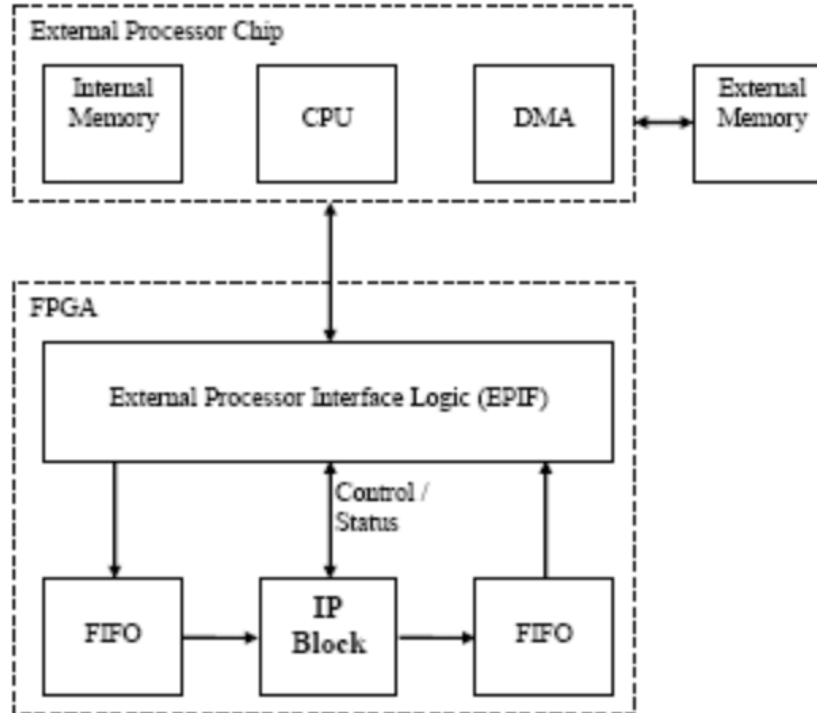


Figure 24: Diagram representing the use of an FPGA coprocessor taken from [34]

10.2 Appendix 2: PIC12F1840 Data Sheet

TABLE 29-3: PIC12(L)F1840 INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode				Status Affected	Notes
				MSb		LSb			
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C, DC, Z	2
ADDWFC	f, d	Add with Carry W and f	1	11	1101	dfff	ffff	C, DC, Z	2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	2
ASRF	f, d	Arithmetic Right Shift	1	11	0111	dfff	ffff	C, Z	2
LSLF	f, d	Logical Left Shift	1	11	0101	dfff	ffff	C, Z	2
LSRF	f, d	Logical Right Shift	1	11	0110	dfff	ffff	C, Z	2
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z	2
CLRW	—	Clear W	1	00	0001	0000	00xx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	2
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	2
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	2
MOVWF	f	Move W to f	1	00	0000	1fff	ffff		2
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C, DC, Z	2
SUBWFB	f, d	Subtract with Borrow W from f	1	11	1011	dfff	ffff	C, DC, Z	2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	2
BYTE ORIENTED SKIP OPERATIONS									
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1, 2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1, 2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		2
BIT-ORIENTED SKIP OPERATIONS									
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		1, 2
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		1, 2
LITERAL OPERATIONS									
ADDLW	k	Add literal and W	1	11	1110	kkkk	kkkk	C, DC, Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLB	k	Move literal to BSR	1	00	0000	001k	kkkk		
MOVLW	k	Move literal to PCLATH	1	11	0001	1kkk	kkkk		
MOVLW	k	Move literal to W	1	11	0000	kkkk	kkkk		
SUBLW	k	Subtract W from literal	1	11	1100	kkkk	kkkk	C, DC, Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Figure 25: Screenshot of the instruction set summary (1)

Mnemonic, Operands		Description	Cycles	14-Bit Opcode				Status Affected	Notes
				MSb		LSb			
CONTROL OPERATIONS									
BRA	k	Relative Branch	2	11	001k	kkkk	kkkk		
BRW	—	Relative Branch with W	2	00	0000	0000	1011		
CALL	k	Call Subroutine	2	10	0kkk	kkkk	kkkk		
CALLW	—	Call Subroutine with W	2	00	0000	0000	1010		
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
RETFIE	k	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	0100	kkkk	kkkk		
RETURN	—	Return from Subroutine	2	00	0000	0000	1000		
INHERENT OPERATIONS									
CLRWDT	—	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}, \overline{PD}$	
NOP	—	No Operation	1	00	0000	0000	0000		
OPTION	—	Load OPTION_REG register with W	1	00	0000	0110	0010		
RESET	—	Software device Reset	1	00	0000	0000	0001	$\overline{TO}, \overline{PD}$	
SLEEP	—	Go into Standby mode	1	00	0000	0110	0011		
TRIS	f	Load TRIS register with W	1	00	0000	0110	0fff		
C-COMPILER OPTIMIZED									
ADDFSR	n, k	Add Literal k to FSRn	1	11	0001	0nkk	kkkk	Z	2, 3
MOVIW	n mm	Move Indirect FSRn to W with pre/post inc/dec modifier, mm	1	00	0000	0001	0nmm		
MOVWI	k[n]	Move INDFn to W, Indexed Indirect.	1	11	1111	0nkk	kkkk	Z	2, 3
	n mm	Move W to Indirect FSRn with pre/post inc/dec modifier, mm	1	00	0000	0001	1nmm		
	k[n]	Move W to INDFn, Indexed Indirect.	1	11	1111	1nkk	kkkk		

Figure 26: Screenshot of the instruction set summary (2)

The full data sheet can be found at:

<https://ww1.microchip.com/downloads/en/DeviceDoc/40001441F.pdf>