MANNIX

FC + ACTIVATION + SW

2020

Authors: Dor shilo & Eliyahu levi

Date: December 30, 2020

Document Information

Document Title:	File Information:
MANNIX	**File Name:** Dor_shilo_Eliyahu_levi_report **Last time saved:** December 30, 2020 **Saved by:** Dor Shilo, Eliyahu levi
Keywords:	
Abstract:	
This document is the Specification of the FC + ACTIVATION layers and the SW part of the project.	

Contents

Introduction To Neural Network:

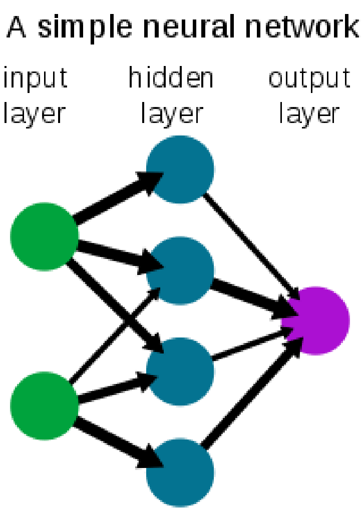
Neural Networks

A Neural Network is built from groups of neurons that are divided into layers.
Each and every neuron stores data and is connected to the neurons on the different layers close to his own in a well-defined connection structure.
Neural Networks resemble the human brain neurons – Our own human "AI" – a series of electrical connections that connect one neuron to another in a process called synapses. Those transitions are the key factor of us being able to process information correctly!
Nowadays, Neural Networks idea is being used to solve problems using computers – harnessing computers calculation abilities to process information in the same way human brain's neurons does!
The connection between biological Neural Networks to AI Neural Networks is modeled into weighs.
Weight, a number that if positive, reflects an excitatory connection, while negative values mean inhibitory connections. Using those relations, this activity is referred to as linear combination of data and weights granting the computer to process information.

$$Output = weights \cdot input + bias$$

But how does the computer learn?

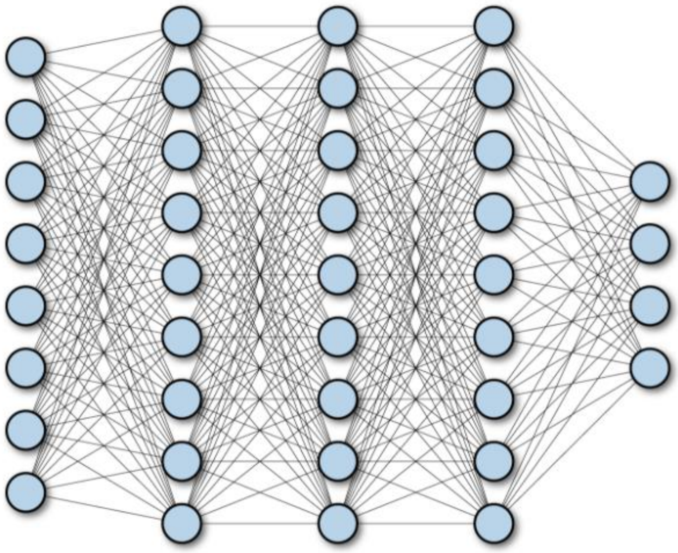
One of the most common human behavior is being able to learn from our mistakes and improve.
This feature is implemented using predictive modeling – The AI start to process the information given to him by "guessing the result" (the computer only controls the weighs values) and by comparing the outcome of his calculation to the correct result provided by the user – the computer learns and adapt in order to be more precise in the next calculation.



Fully Connected Neural Networks

Neural Networks have different types of architecture that can be used in order to get a different behavior of the network – a faster one, a more precise and any combination of both.
One of the most basic Neural Network architecture is the Fully Connected Neural Network.

A Fully Connected Neural Network consists of a series of fully connected layers that connect every neuron in one layer to every neuron in the next layer.



The major advantage of such architecture is that every piece of data is being analyzed – if the network input consists of pictures than every single pixel is passing through the network and Taken into account.
The liability of such architecture is that it's very big in size (Memory) and usually have a weaker performance.

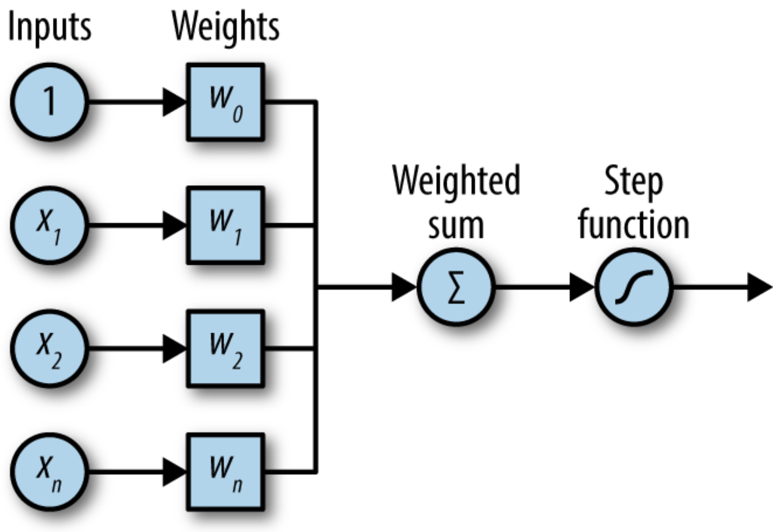
Fully Connected layer – Concepts

Let's define some basic Concepts:
In order to make these concepts more understandable we'll follow along with an example –

<u>Let's us try to create a network that deletes every spam email we receive</u>:

Concept	Definition	Example
Label	The correct outcome out net is trying to predict.	Is the email we received is spam or not.
Feature	an individual measurable property or characteristic of a phenomenon being observed.	• Irregular words • Irregular email address • Time of day
Example (concept wise)	A single input: • Input with label • Input without a label	An email: • Email I know if spam or not • A new Email The net knows nothing about
Hidden layers	Neural network consists of layers – all the layers between the input and output called "Hidden layers"	

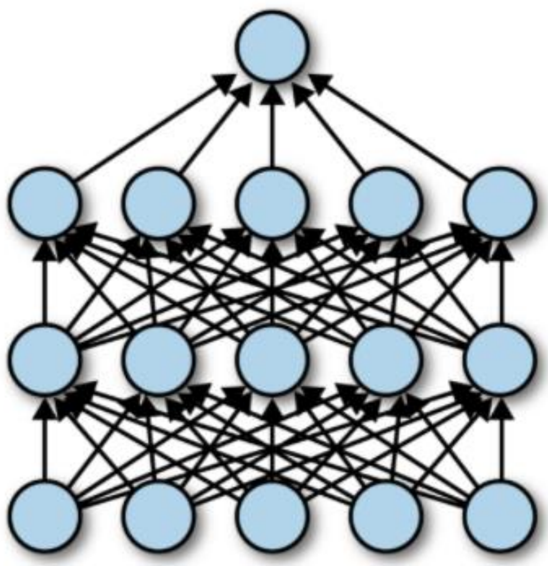
So now we'll have a look inside a Fully connected layer and understand the way it works:



This is the schematic diagram of a simple Fully Connected layer.
As we can see – our data (for example every pixel of a $\sqrt{n}$ by $\sqrt{n}$ picture - n pixels overall) enter the network.
The next step is to multiply every pixel in it's corresponding weight value. All those multiplications are being summed to one value that is being normalized using the "Activation function" – we'll discuss on this function shortly.

Another concept called **bias** is added to this scheme – The bias is a single value completing the linear function of the Fully connected layer.

$$\boldsymbol{y = Activation\,(\sum_{i=0}^{n} x_i \cdot w_i + bias)}$$



This action happens **in every single neuron** – every neuron in the next layer has it's own weights values corresponding to each neuron in the previous layer.

This suggest that in the output of a layer with M neurons there will be **M different y values**.

Finally We get to the Final layer – where we figure out if the network answer is correct and adjust the weight properly (depends on the amount of different outcomes our system can receive – for example, if the system needs to distinguish between 10 different objects, we will have 10 neurons in the final layer).

$$y_{final} = \max\{y_{final_i}\}$$

Activation layer:

The Activation layer is usually incorporated in the end of a fully connected layer – either as an independent layer in the net or as part of the fully connected layer that came before.

what is this Activation layer and what is it's role?

As we saw in the previous chapter – when data enters a neuron we receive :

$$\boldsymbol{y = Activation\ (\sum_{i=0}^{n} x_i \cdot w_i + b)}$$

Notice the value in the parenthesis can range from $-\infty$ to ∞ - the neuron does nothing to bound the value (it's mandatory to bound values - especially in hardware!).

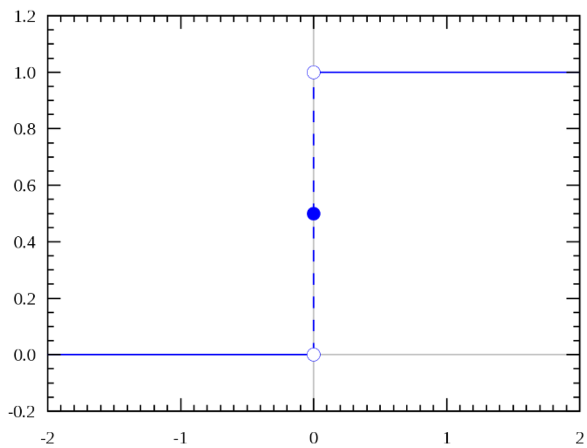
In order to bound this value and normalize all the values, we add **Activation function!**

Here are some examples for different Activation function:

1) **Step function -**

   This is probably the first function that comes to mind – if Y is below some threshold value, consider it as zero –

$$\begin{aligned} &if\ (Y < active\_threshold) \\ &\qquad Y = 0; \\ &else \\ &\qquad Y = normalize(Y); \end{aligned}$$



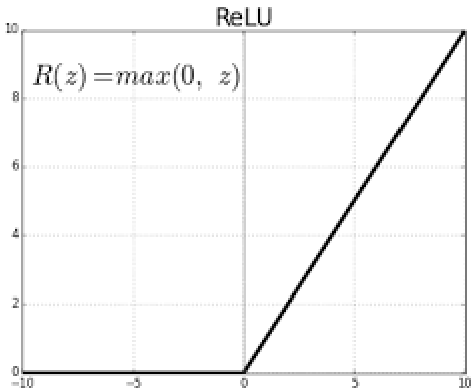
   But as you can tell – this ideal function only lives in the dreams of engineers and we must be more realistic.

2) **Relu function –**

   Relu function is the "linear" way to solve this problem. Using this function we get a range of values, hence it's not binary activation like the step function –

$$Y = C \cdot \max\ (0, Y)$$

   Where C is the slope.



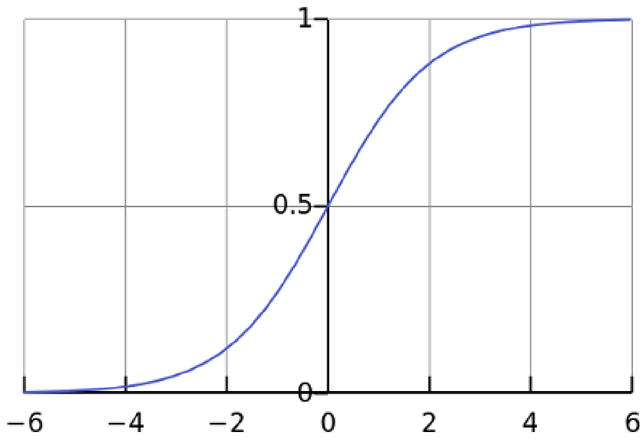
Once again we face the non Continuous problem but in a more gentle way. notice we can still explode to infinity.
On the other hand this function can be a decent option because while eliminating all the negative values, it's still a Hardware buildable function.

3) Sigmoid function

Sigmoid function is a well-defined function –

$$Y = \frac{1}{1 + e^{-x}}$$

Which gives the plot:



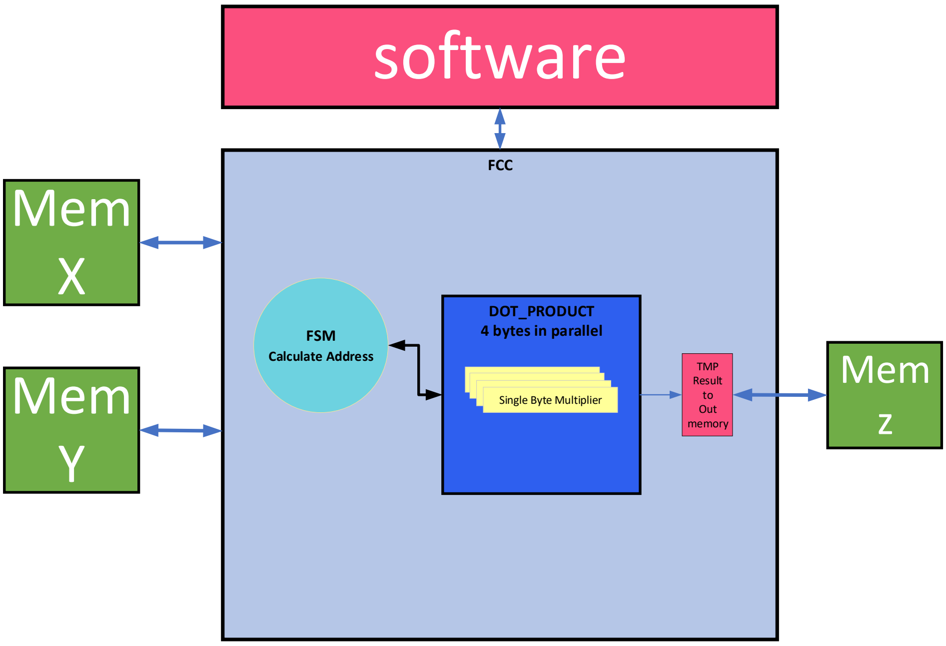
This function does looks smoother and more "step function like" and we get rid of all non-continuous problems – so it's prefect isn't it?!
Well … no! in software it's may be easy – but in MANNIX we deal with bit's so it's harder to define a continues function like sigmoid (it's possible using LUT but it's still on debate in our project).

Implementation - General Description:

The FC block is a part of HW accelerator that makes convolutional Neural Network.
It is an implementation of the algorithm that was explained in the introduction above.
It uses a mathematical calculation of dot-product in order to calculate the output matrix.
FC has an interface to a memory and it's able to ask for data according to the address that has been chosen by the software.

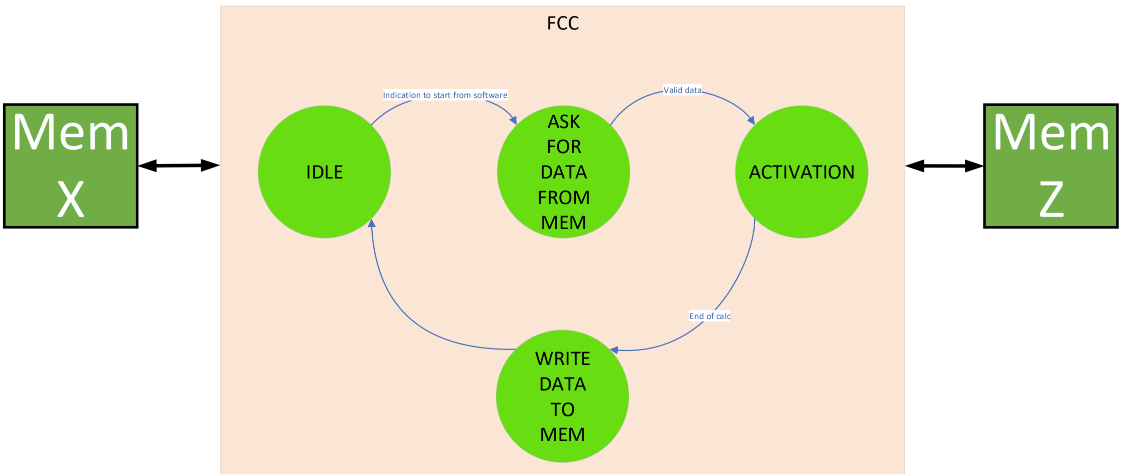
Block Diagram:

FC:



Activation:

As of right now – The activation will be a part of the FC module (the multiplication + sum +bias and activation will be part of the ACTIVATION state):



Interfaces:

FC:

Name	I/O	Comment
General		
clk	I	clock
rst_n	I	reset negative
Memory Interfaces		
mem_intf_read_pic	IF	Read interface from memory
mem_intf_read_wgt	IF	Read interface from memory
mem_intf_write	IF	Write interface to memory
Software Interface		
fc_sw_busy_ind	O	An output to the software - 1 – FC unit is busy 0 – FC is available (Default)
sw_fc_if_vld	I	SW registers can be used/ there was a data change in registers
fc_addr_x	I	FC Data FIRST address
fc_addr_z	I	FC return address
fc_xm	I	FC data matrix num of rows
fc_ym	I	FC weight matrix num of rows
fc_yn	I	FC weight matrix num of columns

Activation:

As of right now, the activation will be part of the fully connected module.

Schedule:

TASK	Owner	15-20-נוב	22-20-נוב	29-20-נוב	06-20-דצמ	13-20-דצמ	20-20-דצמ	27-20-דצמ	03-21-ינו	10-21-ינו	17-21-ינו	24-21-ינו	31-21-ינו	07-21-פבר	14-21-פבר	21-21-פבר	28-21-פבר	07-21-מרץ	14-21-מרץ	21-21-מרץ	28-21-מרץ	04-21-אפר	11-21-אפר	18-21-אפר	25-21-אפר	02-21-מאי	09-21-מאי	16-21-מאי	23-21-מאי	30-21-מאי	06-21-יונ	13-21-יונ	20-21-יונ	27-21-יונ
											מבחנים	מבחנים	מבחנים	מבחנים																				
Initial Design Spec writing	Dor																																	
Initial skeleton implementation	Dor																																	
Base Version implementation	Dor																																	
Final Design spec update	Dor																																	
Final version implementation	Both																																	
ASIC feasibility check (synthesis)	Dor																																	
Productizing and documenting	Both																																	
In parallel to all - continues verification	Both																																	

Project Software

1.0 Introduction

The MANNIX HW accelerator has software tools. The reasons for that are:

1. **Profiling** – writing a model in C or other high-level language to find the bottleneck that we want to accelerate.
2. **Verification** – the system needs to be bit accurate, which mean that the output of the accelerator must be equal (byte resolution) to the output of a software program that simulates the same operation.
3. **Modularity** – We want the project to be modular and flexible to change. The best way to do so in minimal cost is to manage it with software program.

Those three demands will affect the way the program will be written.

2.0 Mannix software manager

Following the introduction, we decided to use this model:

```
#define ADDRX /*BASE POINTER OF THE ADDRESS WHERE THE IMAGE WILL BE LOADED */
#define ADDRY /*BASE POINTER OF THE ADDRESS WHERE THE WEIGHTS WILL BE STORED*/
#define ADDR_CONV_RES /*BASE POINTER OF THE ADDRESS WHERE THE CONVOLUTION LAYER RES
ULT WILL BE STORED*/
#define ADDR_ACTIVATION_RES /*BASE POINTER OF THE ADDRESS WHERE THE ACTIVATION LAYE
R RESULT WILL BE STORED*/
#define ADDR_PULL_RES /*BASE POINTER OF THE ADDRESS WHERE THE MAX PULL LAYER RESULT
 WILL BE STORED*/
#define RESULT /**/
define COUNTER /*SAVE THE NUMBER OF IMAGE */

void MANNIX_NN(image_ptr*, weights*, bias*) {
    load(image_ptr, ADDRX); load(weights, ADDRY); load(bias, ADDRY + B);
    MANNIX_convolution_layer       (ADDRX,         ADDRY,         ADDR_CONV_RES);
    MANNIX_non_linearity_activation(ADDR_CONV_RES, ADDRY, ADDR_CONV_RES);
    MANNIX_pull_layer              (ADDR_CONV_RES, ADDRY, ADDR_PULL_RES);
    MANNIX_fully_conneted          (ADDR_PULL_RES, ADDRY, RESULT);
}
```

2.1 Loading data to the memory:

Description:

Since we are working on a FPGA device which is much slower then ASIC, the first step will be to allocate a memory area to the incoming data.

2.2 MANNIX_convolution_layer

Description:

The convolution layer - The program must send to the processing unit the starting address of data (ADDRX), the start address of the weights (ADDRY) and the return address to save the output (ADDRZ). It also must send the data length (Xm), it's width (Xn), the window length (Ym) and it's width (Yn).

2.3 MANNIX_pull_layer

Description:

The software must send to the processing unit the start address (ADDRX) and the return address (ADDRZ). It is also must send windows length (Xm), windows width, (Xn),the resulting matrix length (Pm) and the width of the matrix (Pn).

2.4 MANNIX_non_linearity_activation

Description:

The unit must send the return address (ADDRZ), this address is also the input address. The software must send the window length (Xm) and window width (Xn). The method of execution will be ReLU.

2.5 MANNIX_fully_conneted

Description:

The software must send to the processing unit the start address (ADDRX), the weights address (ADDRY), the base address (ADDRB) and the return address (ADDRZ). It is also must send the input vector length (Xn), weight vector length (Ym), weight vector width (Yn) and base length (Bn).

2.6 Variable memory allocation

In the previous paragraph we mentioned many different variables that must be allocated to memory.

<u>As of right now – the allocation is described in the following table:</u>

MANNIX Name	SOFTWARE Size	REGISTER Address (offset)	DOCUMENT Operation
CNN LAYER			
CNN_ADDRX	31:0	0x0000	CNN Data window address
CNN_ADDRY	31:0	0X0004	CNN weights window address
CNN_ADDRZ	31:0	0x0008	CNN return address
CNN_XM	31:0	0x000B	CNN input length
CNN_XN	31:0	0x0010	CNN input width
CNN_YM	31:0	0x0014	CNN weights length
CNN_YN	31:0	0x0018	CNN weight width
PULLING LAYER			
PL_ADDRX	31:0	0x001B	PULL Data window address
PULL_ADDRZ	31:0	0x0020	PULL return address
PULL_XM	31:0	0x0024	PULL input length
PULL_XN	31:0	0x0028	PULL input width
PULL_PM	31:0	0x002B	PULL result length
PULL_PN	31:0	0x0030	PULL result width
ACTIVATION LAYER			
ACTIV_ADDRX	31:0	0x0034	ACTIVATIN Data window address
ACTIV_XM	31:0	0X0038	ACTIVATION weights window address

ACTIV_XN	31:0	0x003B	ACTIVATION return address
FULLY CONNECTED			
FC_ADDRX	31:0	0x0040	FULLY CONNECTED Data vector address
FC_ADDRY	31:0	0X0044	FULLY CONNECTED weights window address
FC_ADDRB			FULLY CONNECTED bias vector address
FC_ADDRZ	31:0	0x0048	FULLY CONNECTED return vector address
FC_XM	31:0	0x004B	FULLY CONNECTED input vector length
FC_YM	31:0	0x0050	FULLY CONNECTED weights window length
FC_YN	31:0	0x0054	FULLY CONNECTED weights window width
CNN_BN	31:0	0x0058	FULLY CONNECTED bias vector length

Note - this is the temporary table! Changes will be made as we move along with the project!

3.0 software timestamp

The software program will be divided to two parts:

1) pure software
2) writing to the processing unit.

3.1 Step One - Pure software

Description:

Building software that simulates the behavior of a MANNIX accelerator. The program will fit the software structure mentioned in the introduction. The behavior of the software will be serial, that is - no unit will start work before it's predecessor has finished. In addition, each function will simulate the behavior of one of the processing unit's in such a way that the input and output of every function will be identical to the processing unit which it replaces.

Goal:

- Creating a memory management software shell
- building a model that is easier to test and integrate.
- Modularity.

Expected date: 12/02/21

Notes:

- The model will be written in C in windows operating system in order to facilitate the transition to the RISC-V code later on.
- The chosen dataset is fashion emnist – this is temporary and progress dependent.

3.2 Step Two - Managing a Basic Operating System (Software)

Introduction:

Instead of waiting for it's processing to be completed, new data is sent once it is possible. When we come to do this, we encounter two problems:

1. **Memory allocation management** – for Each image or data that arrives the program must know where it is located.
2. **Address Management for Processing Unit's** - Each processing unit handles information independently of the other unit's. This creates a problem when several images are waiting for the same unit.

The solution to the first problem would be to track an address index. The solution to the second problem would be to manage an address queue for each unit.

Goal:

Creating a memory management model in such a way that it can be replaced by a hardware mechanism.

Notes:

- This step will also be managed in the software only.
- Adding parallel software components (threads) in order to simulate the hardware mechanism. The intention is to create a situation where several unit's are waiting for the same function.
- Once we have finished processing the image, it's place in memory will be vacated.

Expected date: Passover 2021.

3.3 Third stage - integration of processing unit's within the hardware

introduction:

in order to use mannix accelerator we need to write to the gpp. We want to create functions that do so and replace the software functions.

Goals:

- Implement each hardware module built into the software system separately.
- Convert program code to RISCV code.

Notes:

- This phase may be parallel to stage 2 depending on the pace of progress of the construction of the processing unit's.

Expected date: two weeks after Passover.

3.4 Step Four - Create a Python Shell for Code / * Optional * /

References

FC Intro:

[1] Adi teman course in BIU – "From HW to DL"

[2] https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html

Activation:

[3] https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0