# Acknowledgments

I would like to thank my supervisor, Assoc Prof S. N. Piramanayagam, for introducing me to Spintronics technologies,

keeping me on the right track and training me in my presentation skill.

I would like to acknowledge Asst Prof Mohamed M. Sabry for his guidance through each stage of the design process

and defining challenging next steps. I would have appreciated him even more if he answers my emails.

My Phd mentor,Arko, was instrumental in assisting me in setting up the software environment and diagnosing software

technical problem.

I also thankful for HESL lab manager, Mr Chua for providing reliable server access so that I can work remotely from

home on this project particularly in times of pandemic.

Also not forgetting Cadence Design System and foundry partners for making the software and technologies available

for access on the computing platform in the lab.

For these, I am extremely grateful to all of them.

## 1 Introduction

### 1.1 Overview

#### 1.1.1 Artificial Intelligence

Artificial Intelligence (AI) is widely used in today’s industry. It is an emerging field where machines are empowered to mimic human thinking, decisions and actions, to increase industries productivity in the sense of higher level cognitive processing. In the not so distant past where robotics were programmed and used in manufacturing, today robotics incorporate not just programmed logic, but also trained experience via continual exposure to external stimuli, action feedback and judgement adaptation to make decision even on abstract information. For example, in the field of image classification, images are basically 2D array of data. In a traditional programmable logic computing, it is architecturally impossible to recognize its unstructured content as it only deal with well defined, structured data. However, with newAI architecture like neural network, it is possible to leverage the similar and minimalistic structural architecture of a human brain, to efficiently and precisely deal with processing of such data. More complex structure would mean a better ability of extracting subtler and finer details. Based on these details, classification of images can then be made. In medical field, we have classification of images for presence or severity of tumour. In search engine, we have classification of images for objects, scenery and

etc. In a self-driving vehicle, we also have classification of images for objects on road (See Figure 1).

Figure 1: Objects recognised by AI on the road [1]

Beside image classifications, AI is widely used in control and actuation. In a self driving vehicle, AI can process data collected from its surrounding through different types of sensors, not just data from the objects to recognise their presence, but also data from intrinsic measurable, like distance to those objects and their trajectories, to steer the vehicle in the right direction and away from danger. For example, if a child suddenly ran out from nowhere on the road, an AI can force the

vehicle to stop, and effectively avoid an accident.

#### 1.1.2 Biological Neural Network

The biological neural network found in a human brain structure is extremely complex. A human brain contain more than 8 billionneuronsandmorethan100trillionsynapses(connectionbetweenneurons). Aneuronisanervecellthatcommunicate with other cells via nerve impulses. It receives signals from one end known as dendrites, process them via a mechanism known as action potential, and send signal via elongated axon to another end. For the signal to pass from this terminal to the next neuron dendrite, it has to cross between a gap known as synapse. The synapse can grow weaker or stronger overtime,

determining the strength of synaptic connection, known as neural plasticity.

Aneuron maintain a negative voltage (i.e. polarized) across its membrane at *−*70*mV* . It is achieved by complex protein structures sitting on the membrane known as ion channel and ion pump. Whenever there are stimuli from the dendrites, they are integrated over time. As these stimuli are not perfect spikes, their strength die down through time. There are 2 types of stimuli, one is excitatory post-synaptic potential, while the other is inhibitory post-synaptic potential. We can thus think of them as plus signal and minus signal. When these signals of equal strength comes together at the same time, they cancel out each other and the net effect on a neuron is 0. If the sum of stimuli has a net positive effect, they come into the

action potential mechanism picture.

For a neuron to fire, i.e. generate a spike to other neuron downstream, the stimuli strength needs to cross the threshold at *−*55*mV* . Weak stimuli results in failed initiation action potential, whereas a strong stimuli results in depolarisation, triggering its action potential, peaking at about *−*40*mV* before dropping back down, known repolarization. This repolarization process can cause its voltage to overshoot its initial negative voltage, before recovering to its resting state, known

as refractory period, and be ready for the next cycle.

In various stages of the action potential, the permeability of the neuron changes, as there are exchange of ions. These

are the 4 stages of an action potential cycle:

1. Resting State:

Sodium (*Na*2+) and potassium (*K*+) ions have difficulty to pass through the membrane, and the neuron has a net

negative charge internally.

1. Depolarisation:

This happens when the action potential is triggered. The sodium channels are activated, allow sodium ions to flow into the cell through the channel, resulting in a positive potential difference with respect to the extracellular fluid

outside.

1. Repolarization:

This happens when the action potential peak is reached. The sodium channels close and potassium channel open, allowing potassium ions to flow out of the membrane through its channel into the extracellular fluid, reducing the

membrane potential to a negative value.

1. Refractory Period: the voltage-dependent ion channels are inactivated

Memory is stored in the synaptic connection strength [4], subjected to its plasticity. This plasticity exists for both short term and long term. Short term means this type of change only lasts for time below a second before reverting to normal, whereas long term plasticity can last for years. Long term synaptic plasticity involves long term potentiation (LTP), i.e. strengthening or long term depression (LTD) i.e weakening. Intuitively, frequent activity of a synapse results in LTP, i.e. better memory, whereas prolonged infrequent activity result in LTD, i.e. memory loss. Neuron firing timing contributes to LTP and LTD too. For two neurons that are connected together through a synaptic junction, if a pre-synaptic (i.e. previous) neuron fires 20*ms* or less before the post-synaptic (i.e. next) neuron, it results in LTP for the synapse connecting them. However, if a post-synaptic neuron fires 20*ms* or less before pre-synaptic neuron, it results in LTD. Hebbian theory, in particular, states that coincidental activity of synaptically connected neuron leads to lasting change in the effectiveness of synaptic transmission [5]. In other words, it means neurons that fire together, are connected together.

#### 1.1.3 Artificial Neural Network

An artificial neural network (ANN) is an architecture that mimics human brain structure, which consists of synapses and neurons. Similarly, anANN can receive inputs and process those input before reaching its outputs. Output can then be used for classification. For example, in a self-driving car, the input would be from the camera and sensors, and the output would be steering direction and magnitude and acceleration. Input can be in many form, i.e. 1D vector or 2D matrix. Processing would then involve passing these input through many neuron layers, with each deeper hidden layer extracts subtler details (See Figure 2).

Figure 2: Fully connected neural network overview [2]

For example, the input layer (1*st* layer) will have nodes/neurons with its numbers correspond to the number of inputs, i.e. 1D vector with *n* elements would have *n* number of inputs, whereas 2D matrix with *m×n* elements would have *m×n* number of inputs. In a fully connected layer (i.e. a dense layer), each neuron in the previous layer is connected to every neuron in next layer. The connection between neuron A and neuron B is known as a synapse. Each of this synapse carries a weight and a stronger synaptic connection would mean a greater weight. These weight would then act on the input signal from the neuron to extract certain details. Basic processing is done as followed:

Figure 3: A simple artificial neural network illustration

A layer is defined as column of arrows, with/without weight attached, and usually ended with a column of neurons. Suppose in the input layer we have neuron 1 and neuron 2, and layer 2 we have neuron A and neuron B. Neuron 1 and neuron 2 synaptic connection to neuron A would be *w*1*A* and *w*2*A* respectively. Likewise, neuron 1 and neuron 2 synaptic connection to neuron B would be *w*1*B* and *w*2*B* respectively. Let say after

layer 2 is the output layer:

1. Neuron 1 and neuron 2 receive signal *S*1 and signal *S*2
2. Before reaching neuron A, by multiplying with the synaptic weight, the 2 signals become *netA* = *S*1 *×*

*w*1*A* +*S*2*×w*2*A*

1. Before reaching neuron B, by multiplying with the synaptic weight, the 2 signals become *netB* = *S*1 *×*

*w*1*B* +*S*2*×w*2*B*

1. The 2 signals for neuron A can be fed into an activation function *f*(*x*), hence the output for neuron A becomes *outputA* = *f*(*netA*)
2. The 2 signals for neuron B can be fed into an activation function *f*(*x*), hence the output for neuron B becomes *outputB* = *f*(*netB*)

Here, we are assuming that we already have the weight. i.e. the network is in inference mode. If we have defined inputs and desired corresponding outputs but without the weights, such that we need to figure out each weight, we can put the network into training mode to set the weight. A common way to work out the weights would be back propagation technique. In a simple back propagation technique, we would randomly set the weight of each of the synapses. Then, we inject signals into the input layer and measure the output from the last layer. We then compare the difference between the desired output and the output from the network, known as the error. From there, we can use partial differentiation to work out the error dependencies and tune the corresponding weight accordingly. The exact detail for back propagation technique for previous

example is as followed:

1. Calculate error:

*Etotal* = *EoutputA* +*EoutputB* = (*desiredA −outputA*)2+ (*desiredB −outputB*)2

. Here, the purpose for the error for each of the output to appear in a halved and squared form is to simplify

differentiation.

1. we want to know how much *w*1*A* contributes to the total error *Etotal*. Applying chain rule, we get:

*∂Etotal ∂Etotal ∂outputA ∂netA*

= *× ×*

*∂w*1*A ∂outputA ∂netA ∂w*1*A*

1. As mentioned before in the 1*st* step, we use squared form to facilitate differentiation, here we have:

*∂Etotal*

= *−*(*desiredA −outputA*) *∂outputA*

1. And, *∂output∂netAA* would depend on the activation function used, we will obtain a numerical value here too.
2. Also,

*∂netA*  = *S*1 *∂w*1*A*

1. After obtaining numerical value for *∂output∂EtotalA , ∂output∂netAA , ∂net∂w*1*AA* , we can finally have the value for *∂E∂wtotal*1*A* .
2. We can then proceed to set *∂Etotal*

*w*1*A*(*new*) = *w*1*A*(*old*)*−η× ∂w*1*A*

where *η* is the learning rate. The smaller the value of *η* (*<* 1), the slower or more gradual the update is to new value of *w*1*A*, to prevent over adjustment. Over adjustment will lead to divergent behaviours, which

is unfavourable especially if we aim to hit the minimum error spot. (See Figure 4).

1. we can repeat the same process above to set for *w*1*B,w*2*A,w*2*B*.

The same technique can be extended to *n* layers fully connected network.

Figure 4: Different learning rate and their contribution to the Cost Function *J*(*θ*) where *θ* is any parameter we want to optimize (e.g. weight *w*) and *J*(*θ*) is the function that computes Error *Etotal* [3]

#### 1.1.4 von Neumann Computing

In a traditional computing, von Neumann architecture is being used. That means processing is only done in the central processing unit (CPU) and there constant shuttling of data between different memory device. There are mainly 2 type of memory devices on a traditional computing system, i.e. CPU cache and random access memory (RAM). CPU cache is made up of expensive, superfast but small memory capacity whereas RAM is made up of slow but huge memory capacity. All neural network data are loaded to RAM before they are moved to the CPU cache for it to work with and the final processing result are stored back to the RAM. This long distance bi-directional information flow between CPU and RAM constitutes

most of CPU instruction cycles, consuming huge amount of time and power (Figure 5).

Figure 5: von Neumann architecture

Neuromorphic computing come into rescue when we have processor specially designed for neural computing. Although information can be analog/digital, but processing is done with analog processing. For example, a typical neural network

involves summing and multiplication. On a von Neumann architecture, the process flow goes like this:

Let say we are summing number *a,b,c*:

1. Load number *a* from RAM to CPU register 1
2. Load number *b* from RAM to CPU register 2
3. Sum *a* and *b* on CPU and store the result in register 3
4. Save value of *a*+*b* in CPU register 3 to RAM
5. Load value of *a*+*b* from RAM to CPU register 1
6. Load number *c* from RAM to CPU register 2
7. Sum the value of *a*+*b* and *c* on CPU and store the result in register 3
8. Save value of *a*+*b*+*c* in CPU register 3 to RAM

We would require so many steps to compute a simple summation, nevermind that CPU has billions of cycles to take care

of these steps.

#### 1.1.5 Neuromorphic Computing

Figure 6: Neuromorphic architecture

In neuromorphic computing (Figure 6), highly specialized circuits are used to do the computation and physical laws like fundamental circuit laws are leveraged. Simple mathematical computation like multiplication and summing that take many processing steps and cycles in von Neumann computing are now irrelevant, because Ohm’s law is used to do the multiplication and Kirchoff’s current law is used to do the summation in massive parallelism, which is an operation known as

in-memory computing.

However, due to the non programmable, non sequential processing and highly specialized nature of the circuit, it is designed solely for this purpose. It cannot run software like von Neumann architecture does, so we can forget about running computer operating system, watching YouTube videos, checking email and etc. Such circuit is known as an applicationspecific integrated circuit (ASIC). One such example is IBM TrueNorth chip, it consumes as little as 1*/*10000*th* the power

of a traditional von Neumann computer.

What it lacks in its functionality, it more than makes up for in performance in terms of speed and power consumption. MultiplestepsinasimplecomputationinvonNeumannarchitecturethatcanscaleeasilyuptomillionsofstepsdependingon thenumberofinputscandeterthenetworkefficiencyanditsresponseinrealtime, becausethetimeinsequentialcomputation all add up to a prohibitively large delay. In contrast, all these sequential steps are reduced to a single step in neuromorphic computing architecture as the circuit laws naturally take care of these computation, less the need for purposeful sequential

processing.

One notable development in neuromorphic computing is spiking neural network (SNN). Unlike Von Neumann architecture, that is always on and all instruction executions are synchronized to the CPU clock cycle, SNN is event driven, i.e. it performs computation and consumes power only when there is external stimuli, and there is no clock cycle to synchronized to. Furthermore, this SNN take in signal in a form of spike, unlike all other neuromorphic network that takes in signal in a constant signal strength. That is how its power consumption is reduced significantly. Its computation method is biologically inspired, in a way that spikes from the external stimuli are received and accumulated in a neuron, and when a threshold is reached, the neuron will generate a spike to other neurons connected to it through a network of synapses.

#### 1.1.6 Spintronics for Neuromorphic Computing

Spintronics make use of electron spin (orientation) to change its resistance. Spin Hall Effect (SHE) is used to write to the spintronic device, whereas Spin Torque Transfer (STT) is used to read value from the device, in a form of current (Figure

7).

(a) Separate write and read path (b) Field free switching write operation

Figure 7: SHE/SOT MRAM

In a SHE write operation, when a current consisting of spin up and spin down electron, flowing through a heavy metal (HM) conductor in either direction, the electric field near the atom deflect spin up and spin down electron (much like sorting) to the edge of the conductor according to the spin. So, we can see spin up electron gather along one side or the conductor, whereas spin down on the opposite side. Adifferent direction of current flow will see swapping side of previously mentioned electron spin (i.e. The side that consists of only spin up electron will now replaced by only spin down, and vice versa). In the event with one side of the conductor is in contact with an electrode (Free layer), the electron spin in the HM can exert

reorientation influence on the spin direction of the free layer.

Similarly, In a STT read operation, current consisting spin up and down electron flow through a normal conductor, but in a perpendicular/orthogonal direction from the top to the bottom. Since fixed layer has its electron spin pinned to one particular direction, when viewed together with electron spin in the free layer, we can observed them being parallel or antiparallel electron spin. Parallel spin result in lower resistance compared to that of antiparallel spin. After passing through and filtered by fixed layer (aligned spins pass through, whereas anti-aligned spin get bounced), electrons have to tunnel through to a thin, insulating oxide layer in between these 2 layers to form a read current. This structure is known as magnetic tunnel junction (MTJ).

Figure 8: Domain wall device with multiple resistance states

In a domain wall (DW) type MRAM (Figure 8), a MTJ has multiple resistance state instead of just 2 states (low and high). For the purpose of this project, we consider 8 steps or equivalently 9 resistance states. These states comprises a fully antiparallel state, 7 intermediate states and a fully parallel state. To traverse between these states, multiple short SHE current pulses are applied, to move domain wall in step like, forward and backward fashion, until a desired resistance state

is achieved.

Simulation in this project only considers the resistance states of a DW MTJ, not a full simulation model (consisting of both writing and reading operation) because it is not available, and only recently under development by a Phd student.

### 1.2 Background

There has been numerous development in neuromorphic computing and spintronics.

For neuromorphic computing:

In 1957, there was invention of perceptron. Then there was publication of very large scale implementation (VLSI) architecture for implementation of neural network after 29 years. In 1997, we saw the third generation of neural network - spiking neurons being published. There there were more neuromorphic silicon chip invented, mainly by IBM (TrueNorth chip in 2014) and Intel (Loihi chip in 2018).

For spintronics:

In 1989, GMR effect is discovered in thin film, GMR application of hard disk head by IBM in 1997. Around the same period, there was advent of STT. Until recently, we have field assisted MRAM and STT MRAM.

#### 1.2.1 Literature Review

Different research papers were read and cross referenced to have a better idea of this project. Concretely, they comprised of around 8 neuromorphic computing related papers, a spintronic paper, a neuromorphic spintronics review paper, and a domain wall based neuromorphic computing paper, with the latter most closely related to this project, supplied by both

supervisor and co-supervisor.

A few papers touched on the material and device of crossbar array, involving physical metal oxide (MO) resistive ram (RRAM)by S.Yu et al, a simulation model of MO RRAM by M.K.F.Lee et al, physical Conductive-Bridge RAM (CBRAM) by M. Suri et al, memristor by X. Liu et al, domain wall MRAM by D. Kaushik et al and spintronics by J. Grollier. They

demonstrated the viability of such material and device use in a synapse.

A handful deal with spiked based neuromorphic computing, involving its architectures by G. Indiveri et al, conversion of recurrent neural network (RNN) to its spike-based form by P.U. Diehl et al, its application in object recognition by Y.Q.

Cao et al.

A few involves employing existing on chip system to realize full application, like

All papers assume the reader to have some level of foundation in training a neural network and testing its efficacy.

### 1.3 Motivation

Neuromorphic computing offers huge application in AI Computer Vision by mimicking human brain neuron and synapse network. It consumes very low power compared to von Neumann Computing and in memory processing and hardware based implementation of neural circuit make it very efficient for visual object recognition. It can be designed to have multiple

deeper layer network that extract subtler visual details and its final network output can be fed for image classification.

Spintronic devices are originally used in hard disk drive read/write head. They are involved in:

1. Spin Transfer Torque (STT)
2. Spin Orbit Torque (SOT, Spin Hall Effect)
3. Tunneling Magnetoresistance (TMR)
4. Giant Magnetoresistance (GMR)

for magnetoresistive application like Magnetoresistive Random-access Memory (MRAM). They are non-volatile (information retention during power loss) like solid state drive (SSD) unlike metal oxide semiconductor (MOS) and capacitors. Similar to SSD, magnetic tunnel junction (MTJ) uses electron tunnelling. The main motivation of using MTJ is it has smaller footprint comapred to that of CMOS. But, MTJ uses electron spin orientation instead of electron presence in SSD. Charge retention in a typical SSD changes its threshold voltage, causing wear out. In contrast, MTJ offer further advantage in higher read/write speed, lower power through flipping of electrons’spin rather than the charge motion that tend to result in Joule heating and no wear out (Unlimited read/write cycle).

### 1.4 Problem Statement

Neuromorphic network is an efficient way to realise neural network, together with domain wall based MRAM with benefits like ultra fast switching, ultra low power consumption and excellent memory retention, therefore, it is very desirable to combine these two. An ideal neuromorphic network involves infinite range of resistance, vastly simplifying the design of the network, however, with the limited range resistance, additional circuit is needed to overcome the shortcoming. In this study, we create and simulate model of neuromorphic network, taking into account the resistance range of domain wall based spintronic MRAM and compare the performance between neuromorphic model and traditional software implemented neural network model.

### 1.5 List of Contribution

In this final year project report:

1. We show a basic construction of a neuron with circuit theory and derivations.
2. We demonstrate Cadence Software implementation of an extended *n* by *m* scale neuromorphic circuit.
3. We bridge the gap between theory and practical implementation.
4. Weshowedapractical, foundrymanufacturable, componentlevelimplementationofneuromorphiccircuitfromactual

third party foundry supplied 65nm process design kit, without resorting to ideal model.

1. We illustrate 4 versions of neuromorphic circuit.
2. We compare and contrast the practical and ideal model with practical limitation.
3. We explain sources of error in practical model with detailed diagram and equation.
4. We demonstrate the training of different neural network models for a case study - MNIST handwritten digit database

using open source deep learning TensorFlow, Keras Library.

1. We cherry pick the best model most suited for transferring to neuromorphic network, with high accuracy.
2. We use computer vision library - OpenCV to demonstrate the result of neuromorphic circuit output for our case study.

## 2 Design ofA Single Neuron

### 2.1 Theory

(b) Non-ideal

(a) Ideal

Figure 9: Operational amplifier (opamp) characteristics

In an ideal operational amplifier (opamp), it has 2 input channel (inverting *−* and non-inverting +). When DC signal flow in to these 2 channels, the difference (*V*+ *−V−*) get amplified with infinite gain (i.e. infinite gradient), only to be limited by supply voltage, reaching saturation voltage *±*Vsat. In non-ideal case, the gain is limited, which means we see finite

gradient.

Figure 10: Ideal inverting summing opamp

In an ideal inverting summing opamp (Figure **??**), non-inverting input is grounded. inverting input exhibit a critical virtual ground behaviour. Virtual ground here means, it is not actually grounded, but its voltage is 0*V* , and also, no current flows into it. So, based on Kirchoff’s Current Law, which states that sum of all current flowing in to a node equals to sum of current flowing out of a node. *I*1, *I*2, and *I*3 flowing into a node marked “x” in the diagram, since they cannot flow into non-inverting input, all of them flow out as *IF* into *RF*. Since voltage at node x is 0, and a current only flow from low potential to higher potential, we expect *V*out to be negative, where the magnitude of *V*out equal sum of all current multiplied by *RF*. Concretely,

*V*1 *V*2 *V*3

*V*out = *−RF* + +

*R*1 *R*2 *R*3

Given 3 weights of *w*1, *w*2 and *w*3, we would want:

*V*out = *w*1*V*1+*w*2*V*2+*w*3*V*3

So, given *RF* = 1, *w*1, *w*2 and *w*3 correspond to *R*11, *R*12 and *R*13, it infers that:

1 *w ∝ R*

Now, we have:

*0 −* *V*1 + *V*2 + *V*3

*V*out =

*R*1 *R*2 *R*3

To obtain a positive *V*out, we can go through the same procedure again (attach another opamp to the existing opamp output):

*−RF0 RV*outout*00*  (1)

*V*out =

When we set *RF0* = *R*out*0* , we obtain below as needed:

*−V 0* 1 1 1

*V*out =

out = *R*1*V*1+ *R*2*V*2+ *R*3*V*3 (2)

Of course, there are many practical issues when we implement the theory above to simulation circuit. This is be covered in Challenges section later on.

### 2.2 Simulation Environment

Figure 11: Cadence Design Environment

Cadence environment is very powerful for us to build and simulate circuit, even for very large scale simulation. For instance, we can build silicon level layout, fill in the physics details and represent it in a symbolic form. This symbolic form can be used as circuit component in a schematic circuit. Also, we can represent a simple schematic circuit in a symbolic form too, and use it as circuit component in an even larger schematic circuit. The 8 opamps as shown in 11 are represented with triangle symbols with input and output pins. Besides, it is entirely possible to simulate a component with just programming

its behaviour using scripting language with mathematical and physics description, like spiceText, verilogA, and etc.

Cadence has its own scripting language for automating schematic circuit diagram design, known as Skill Script. Automation functionality extends to component placement by specifying coordinates, rotation, vertical and horizontal flipping, instance id, resistance value, methodical wiring routing and etc. Asubset of Skill Script is Ocean Script, meant for automating the testing of the design. For example, in Skill script, we can set the resistance value, input voltage signal to variables (i.e. not hardcoded), so during simulation, we can use Ocean Script to manipulate these variables to automate sequence of

simulations.

Figure 12: Single summing neuron implemented in Cadence

In Figure 12, we implemented single summing neuron with 3 voltage source inputs. The diagram was hand drawn initially before we moved to programmatic schematic design generation. To ensure minimal errors, all resistance values are hardcoded and the expected neuron output (*V*out) was calculated manually. Then, Cadence Virtuoso Analog Design Environment (ADE) simulator was used to feed in the voltages (supply voltages inclusive) and measure the output. ADE also allowed us to specify any wiring to measure its voltage and specify endpoint of any circuit schematic component to measure its current. Current flowing into the specified endpoint is rated positive, whereas the current flowing out is rated negative. Of course, the read out value contain error, not exactly as calculated. We will cover these errors in our challenges

section.

## 3 Extension to Full Macro

Large scale full macro extension involves Skill Script schematic circuit design automation, scaling it up to arbitrary size, because it is not practical and almost impossible to have hand drawn circuit at large scale. 4 version iterations were undergone to overcome design challenges (See Challenges section) and adding new features. All testing wise are automated. Generally, *V*in are randomly generated. Each of the *n* random *V*in has a value 0*V ≥ V*in *≥ Vn*sat so that sum of all these values will not exceed *V*sat and get clipped. Weights *w* are randomly generated and the output of all neuron are monitored and compared against the ideal values through finding the mean and standard deviation of all neurons. **3.1 1st version**

Figure 13: 4 by 3 generated summing neuron design implemented in Cadence

Figure 13 is simply a scaling up of Figure 12. Since the weight is between 0 *≥ w ≥* 1, the corresponding resistance is *∞ ≤ R ≤* 1, but for practical consideration, *∞* resistance is limited at 0*.* which should suffice when

compared to high weight (*w* = 1) or low resistance of (*R* = 1Ω).

Figure 14: Optimal *RF* value determination for large scale circuit (for parameters data, see Appendix Table 8)

From Equation 1, the full equation for a single neuron output is:

*RF*2 *V*1 *V*2 *V*3 *Vn*

*V*out = *− −RF*1 + + +*...*+

*R*out*0 R*1*|RF| R*2*|RF| R*3*|RF| Rn|RF|*

We could have set *R*out*0*, *RF*2, *RF*1 to 1Ω but this is not possible due to practical challenges (see challenges section), so an *|RF|* magnitude is add to every *Rn* such that:

*R* = *|R*out*0|* = *|RF*2*|* = *|RF*1*|* = *|RF|* (3)

to maintain the same Equation 2 and thus the physical *R*phy = *Rn|RF|*.

In a matrix form for *m* neuron outputs, we have:

In Figure 14, average percentage of measured *V*out value is plotted against *n* number of input *V*in and magnitude *R*. When *n* and *R* are small, percentage average is predictable, as we can see the surface plot fit nicely, but towards the extreme ends, the values get unpredictable as the errors got out of hand (see Challenges section), as the limit of scaling is reached.

### 3.2 2nd version

Figure 15: Complementary metal oxide semiconductor (CMOS)

Complementary metal oxide semiconductor (CMOS) is involved as we are looking for voltage controlled switches to isolate and program physical resistances *Rphy*. We assume the resistances are programmable when when they are isolated and extreme voltages (outside of normal operating range) are applied and caused a physical lasting change in their resistances. There are a few similar type of resistive memory pertaining to our assumption here (conductive-bridging RAM (CBRAM),

and phase-change memory, etc.) but they are out of our scope of discussion and we will not explore them here.

In Figure 15, we have n channel mosfet (NMOS) and p channel mosfet (PMOS). These 2 mosfets, together are known as CMOS. PMOS sit in n-well, with p-type diffusion on n type substrate. For PMOS to operate, source S voltage *Vs* is held at high, gate G voltage *Vg* lower than *Vs*, and drain voltage at the lowest among all. When *Vg* = *Vs*, PMOS does not operate. However, when we lower *Vg* to a point where *Vgs ≤Vt* (*Vgs* being negative and *Vt* is the threshold voltage), PMOS

switches on (p channel is formed from source to drain and current flow with holes as the majority carrier).

Similarly and yet being different, NMOS operation involves source S voltage *Vs* held at low, gate G voltage *Vg* higher than *Vs*, and drain voltage at the highest among all. When *Vg* = *Vs*, NMOS is at cut off region. However, when we increase *Vg* to a point where *Vgs ≥ Vt*, NMOS switches on (n channel is formed from source to drain and current flow with electrons

as the majority carrier).

Both NMOS and PMOS body terminal voltage are tied to their source voltage to maintain their *Vt*.

Figure 16: Complementary metal oxide semiconductor (CMOS) inverter with PMOS on top and NMOS below

A simplest device by PMOS and NMOS is the inverter. PMOS is marked with a hollow circle above whereas NMOS is situated below. A is the input and Q is the output. *V*dd is held at high voltage and *V*ss is held at low voltage. When input A is high (*V*dd), PMOS is off and NMOS is on, hence Q is low (*V*ss). Conversely, if input A is low (*V*ss), NMOS is off and PMOS is on, hence Q is high (*V*dd). Hence, we are assuming ideal short circuit on and off behaviour of CMOS.

However, practically, PMOS and NMOS have finite resistance. Their ADE measured on resistance is at *k*Ω range, and their off resistance is at *G*Ω range. For domain wall based MRAM resistance range of (1*k*Ω to 3*k*Ω), we have trouble

differentiating the MRAM resistance and CMOS on resistance, thus they affects our result negatively.

|  |  |  |
| --- | --- | --- |
| MOSFET Type | On Resistance (Ω) | Off Resistance (Ω) |
| pch\_18 | 3.43k | 1.316G |
| pch | 2.32k | 24G |
| nch | 1.2k | 3.44G |
| nch\_18 | 1.5k | 785.8M |

Table 1: Table of MOSFET type supplied by process design kit (PDK) from foundries and their on/off resistance

PMOS and NMOS based design were tested as shown in Figure 17 (only showing PMOS). Circuit execution wise is

the same as that of the 1st version. The only difference is the circuit weights are made programmable through isolation.

In MOSFET execution mode, all MOSFETs are set to on, such that all the programmable resistors (weights) array is connected to the summing opamps neuron. In programming mode, resistors are programmed row by row because we have *V*en\_n, wires routed to the MOSFETs in series with the resistors row by row. MOSFETs of the current row of interest are all turned on, while the remaining are turned off, thus isolating all other rows from the programming action. *V*in or *V*i\_n pins on the left of the row of interest is set as reference voltage whereas *V*in,set or *V*is\_n (from the top) of the individual resistances

on that row is set individually.

NMOS performance is better than PMOS, but not by a wide margin. With *n* = 10 number of inputs and *m* = 20 number of neuron outputs and *R* = 50*k*Ω (Condition 3), NMOS circuit performance (average measured percentage in relation to ideal) vs that of PMOS is 80*.*3*±*3*.*0% vs 67*.*7*±*4*.*0%. One reason is NMOS majority charge carriers (electrons) have higher mobility as compared to that (holes) of PMOS.

Figure 17: 4 by 4 generated summing neuron with pmos design implemented in Cadence

### 3.3 3rd and 4th versions

A proposal has been put forward by co-supervisor about mimicking simplified MRAM resistance property (with limited 8 step 1*k*Ω*−*3*k*Ω), while preserving the actual weight range (also see Case Study section for additional detailed explanation). Resistance and weight relationship in Table 2 is used when designing the 3rd and 4th versions neuromorphic circuit. The physical resistance *rp* is what actually went into the circuit and they represent compressed weight *wc*, additional circuit are

added to restore the original weight representation (decompressed weight *wd*).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Index | Physical Resistance *rp* (Ω) | Resistance *r* (Ω) | Compressed Weight *wc* | Decompressed Weight *wd* |
| 0 | 1000 | 1.0 | 1.0 | 1.0 |
| 1 | 1250 | 1.25 | 0.8 | 0.7 |
| 2 | 1500 | 1.5 | 0.6667 | 0.4999 |
| 3 | 1750 | 1.75 | 0.5714 | 0.3571 |
| 4 | 2000 | 2.0 | 0.5 | 0.25 |
| 5 | 2250 | 2.25 | 0.4444 | 0.1667 |
| 6 | 2500 | 2.5 | 0.4 | 0.1000 |
| 7 | 2750 | 2.75 | 0.3636 | 0.0454 |
| 8 | 3000 | 3.0 | 0.3333 | 0 |

Table 2: Table of resistances and weights

Here, we set *wc,*0 = 1 and *r*0 = 1*/wc,*0, where the subscript number after comma is the index.

Hence,

*rp,i*

*ri* =

1000

*wc,i* = 1*/ri* (4)

*wc,i −wc,*8

*wd,i* = (5)

*wc,*0*−wc,*8

The decompression mechanism for circuit implementation is derived as follows:

We need to achieve:

*V*out = *wd,i*1*V*1+*wd,i*2*V*2+*...*

and we know from:

1. Equation 4:

Compressed weights *wc,i* are directly related to resistance *r* and thus the physical resistances *rp*.

1. Equation 5:

The relationship between compressed weight *wc,*1 and decompressed weight *wd,*1.

Hence,

*wc,i*1 *−wc,*8 *wc,i*2 *−wc,*8

*V*out = *V*1+ *V*2+*... wc,*0*−wc,*8 *wc,*0*−wc,*8

and

1 *−w* 1

*r*1 *c,*8 *r*2 *−wc,*8

*V*out = *V*1+ *V*2+*... wc,*0*−wc,*8 *wc,*0*−wc,*8

and finally for a single neuron output:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 1 1 | | | | | | | | (6) |
| *V*out = *wc,*0*−wc,*8 | | *V*1+ *V*2+*... −*(*wc,*8*V*1+*wc,*8*V*2+*...*) | | | | | |
| *r*1 | *r*2 |  |  |  |  |
| for *m* neuron outputs: |  |
|     *V*o1             *V*o2 1     =    ...  *wc,*0*−wc,*8             *V*on | 1  *R*11  1 *R*12  ...  1  *R*1*m* | 1  *R*21  1 *R*22  ...  1  *R*2*m* | *···*  *···*  *···* | 1 *Rn*1  1 *Rn*2  ...  1  *Rnm* |    *V*i1        *wc,*8      *V*i2*−* ...       ...          *wc,*8     *V*in | *···*  *···* |     *V*i1    *wc,*8       ... *V*...i2    *wc,*8    *V*in | (7) |

The term *r*11*V*1+ *r*12*V*2+*...* can be implemented by normal summation using the 1st row opamp, which results in

*−r*11*V*1+ *r*12*V*2+*...*. Bygoingthroughaopampagainwithafixedscalefactor *wc,*0*−*1*wc,*8, theterm(*wc,*8*V*1+*wc,*8*V*2+*...*) can be sum up separately, together with the previous term, and an inverting sign, yields Equation 6 as needed.

Table 2 was examined further by manipulating the scale factor *s* of compressed weight i.e. 31*s,* 1*s*, with the boundary plotted in Figure 18. When *s* = 1, we obtain *wc* boundary *w*c,0 = 1*.*0 and *w*c,8 = 0*.*3333 same as that of Table 2. As scale

factor *s* increases, the boundary tends towards 0 and compressed weight can no longer be represented faithfully. This figure is mainly used in our case study weight processing functions and their influence on our result (see more in that section). Figure 19 shows clearly the inverse relation between compressed weight *wc* and resistance *r*. Ideally, we would want uniform steps in *w*. However, due to this inverse relationship, what initially uniform in terms of *r* results in non uniformity

of *w*. This non uniformity of *w* can distort our represented compressed weight and affect our result negatively.

Figure 18: Graph of range of values for compressed weight *wc* with scaled R

Figure 19: Graph of relationship between uniform R and non-uniform compressed weight

#### 3.3.1 3rd version

A design put forward by my supervisor and D. Kaushik et al showed that programming 8 bit DW MRAM doesn’t require isolation because read and write operation in DW MRAM are independent of each other. Hence, the problem encountered in 2nd version where MOSFET on resistance in the range of MRAM resistance range can be side stepped. Hence, all MOSFET are removed.

Figure 20: 4 by 4 generated summing neuron with weight decompression circuit implemented in Cadence

#### 3.3.2 4th version

This version design is put forward by co-supervisor due to disappointing performance of very large scale (784 by 20 synapses) implementation of last version. Last version has extremely good performance in closeness in percent relation to ideal, with average scaled up to 100%, and error rate (S.D. = 2*.*17%) in small scale (20 by 10 synapses) but S.D. *≈* 54% in

large scale.

Therefore, the idea was to break up large scale implementation to equivalent 2 stage small scale implementation to improve the performance. Interestingly, the idea doesn’t work, as we still see S.D. *≈* 54% in this version too. For more

info, see Challenges section.

Figure 21: 9 by 3 dual stage with 3 in 1st stage and weight decompression circuit generated summing neuron implemented in Cadence

## 4 Challenges

In previous sections of large scale circuit implementation, there are a few challenges, but all boils down to non ideal opamp behaviour. The non ideality characteristics arise in every practical devices, but the opamp implemented before has worse condition. This is especially the case when the internal circuit design of a LM741 opamp (Figure 22) is obtained from Texas Instrument but implemented using TSMC 65nm process design kit (PDK). High precision opamp design are proprietary

and therefore not available on the manufacturer’s website. The model provided us with 2 null input pins to calibrate opamp

Figure 22: Internal circuitry of TI LM741 operational amplifier (OpAmp)

performance if there are device mismatches (transistor pairs from different vendor, collector and emitter resistors, etc.) due to manufacturing imperfection. In a non ideal case in Figure 9, if *V*+ = *V−* = 0, *Vo* = 0. However, with mismatches, there is a finite *Vo*. In Figure 23, we can add resistors in parallel with *R*0 and *R*2 as shwon in Figure 22. These resistors are adjustable, together known as potentiometer as *R*0 and *R*1 in Figure 23 add up to a fixed resistance. If adjustment is successful, we get *Vo* = 0 as needed. However, this is not the case. It doesn’t matter if we swap out the potentiometer for 2 independent resistor, it still does not work as it makes only tiny difference to the output. Figure 24 show the effective resistance we can achieve if we keep adding resistance in parallel with *R*0 and *R*2 in Figure 22. We observe that the limit of 1*k*Ω (resistance of *R*0 and *R*2 where *R*0 = *R*2 = 1*k*Ω) is reached when parallel resistance tends to infinity.

The formula for the graph plot is:

*R// ×R*

*R*effective =

*R//* +*R*

Figure 23: TI LM741 OpAmp null offset

Figure 24: Graph of effective resistance vs resistance added in parallel with existing resistance *R* = 1kΩ

Out-of-the-box thinking was required for opamp to work. Hence, both 2 null input pins were removed and the 2 1*k*Ω resistors in Figure 22 were tuned independently and it worked. The final resistance setting for those 2 resistors are 1*k*Ω and

1*.*138*M*Ω for left and right resistors respectively.

In Figure 25, we want to identify optimal operation condition of opamp with minimal error. Initially, we fixed *R*0 = *R*1 as shown in Figure 26 as increased their value in tandem. In an ideal opamp summing application, a single *V*in should yield *V*in = *−Vo*. It works as expected for *R*1 = 50*k*Ω,*R*0 = 50*k*Ω and *R*1 = 500*k*Ω,*R*0 = 500*k*Ω where *Vo* decreases linearly until it reaches *−V*sat. However, opamp exhibits erratic behaviour for *R*1 = 2*k*Ω,*R*0 = 2*k*Ω where *Vo* decreases initially

2 before rising again unpredictably. This is suspected as caused by high power consumption. Using *VR*in and assume virtual ground at 0*V* , for the same *V*in, taking *V*2in*k*2 as reference, for *R*1 =50*k*, the power is reduced by 25 times, and for *R*1 =500*k*, the power is further reduced by 10 times.

Figure 25: Tuning output resistance *RF* (*R*0) for input resistance (*R*1) between 1kΩ and 3kΩ

(a) virtual ground voltage with different resistor pair (b) Output voltage with different resistor pair

Figure 26: Non ideal opamp characteristic (based on Figure 25)

For virtual ground voltage, ideally, we expect the voltage to be 0*V* regardless since it is a critical condition for our computation. However, this condition is not met for the high power case (*R*1 = 2*k*Ω,*R*0 = 2*k*Ω). The virtual ground

voltage is low initially and when *V*in increases, virtual ground condition breaks and its voltage increases too.

Table 3 follows that of Figure 25, where different value for *R*1 and *R*0(*RF*) were tested. Range of *R*1 is between 1*k*Ω to 3*k*Ω and hence only these 2 extreme values of *R*1 are examined with different *RF* values to find out the optimal setting through varying *V*in. The best case is having the lowest value for *V*virtual gnd,min and *V*virtual gnd,max to closely approximate virtual ground condition. When *R*1 =1*k*Ω, the optimal *RF* is almost indiscernible. One notable phenomenon in this table is higher *V*virtual gnd,max corresponds to lowest *I*in,max and therefore lowest *I*out,max due to reduced potential difference between

*V*in and *V*virtual gnd.

A way to calibrate *V*virtual gnd has been discovered though adjusting +*V*sat and *−V*sat. This is possible due to the effect

of Voltage Divider Rule.

In a voltage divider rule, *V*virtual gnd is determined by:

+*RV*0sat + *−RV*1sat

*V*virtual gnd = 1 1

+ *R*0 *R*1

In a case with 2 *V*in, *V*virtual gnd becomes:

+*V*sat + *−V*sat + *V*1 + *V*2

*V*virtual gnd = *R*01 1*R*3 1*R*1 1*R*2

+ + +

*R*0 *R*3 *R*1 *R*2

We can expected many more *V*in for a larger scale circuit. Since we can cannot tune individually the resistors, we stick to tuning *±V*sat. However, in a very large scale circuit where we have many terms, tuning 1 or 2 terms has insignificant effect, which explains why we have huge, similar *S.D.* error for large scale 3rd version and 4th dual stage version neuromorphic

circuit.

All these suboptimal performance (i.e. non-zero *V*virtual gnd and low output current) leads to poor gain. To counter this

effect, different resistor pertaining to opamp scaling (not resistor array linked to weights) are tuned.

Concretely, Equation 6 is modified to the form:

*RFo*1  1 1 1 1

*V*out = *RFo*2 *V*1+ *V*2+*... − V*1+ *V*2+*... RFi |RF|r*1 *|RF|r*2 *RaFi RaFi*

where *RaFi* = 1000*wc,*8*s*, *RFi* = 1000*s* , *RFo*2 = *wc,*10000*−wsc,*8 and *s* is a scale factor to scale the summing neuron output to ideal value.

1. Virtual ground voltage (b) Virtual ground voltage with many *V*in

Figure 27: Voltage Divider Rule

Simulation time is also one of the challenges. Simulation time grow proportionally with network size. There are 10000 test samples in our MNISTtest data set (See Case Study Section). For a network size of 10 by 20, the simulation time foreach sample take about 1 second. However, for the same test dataset with a network size of 20 by 784, the simulation time for a sample takes around 2 hours. So, to tune a parameter and to see its result before proceeding to the next adjustment of the same parameter is very time consuming, not to mention tuning for all other parameters. It is mostly a waiting game. The parameters tuned for a small network is also incompatible for a larger network due to practical non ideality.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *RF*(kΩ) | *I*in,min(A) | *I*in,max(A) | *I*out,min(A) | *I*out,max(A) | *V*virtual gnd,min(V) | *V*virtual gnd,max(V) | |
| 1 | -730.89n | 313.18u | -2.2189u | 310.71u | 730.9u | | 86.86m |
| 2 | -727.74n | 312.09u | -2.2157u | 309.61u | 727.7u | | 87.91m |
| 4 | -721.438n | 309.897u | -2.2093u | 307.4u | 721.4u | | 90.1m |
| 8 | -708.83n | 305.39u | -2.1965u | 302.85u | 708.8u | | 94.61m |
| 16 | -684.22n | 295.73u | -2.1716u | 293.14u | 684.2u | | 104.3m |
| 32 | -636.65n | 261.518u | -2.12332u | 258.761u | 636.7u | | 138.5m |
| 64 | -541.745n | 135.729u | -2.03317u | 132.789u | 547.7u | | 264.3m |
| 128 | -391.155n | 70.1832u | -1.87439u | 67.2269u | 392.1u | | 329.8m |
| 256 | -142.562n | 36.8194u | -1.62232u | 33.8592u | 142.6u | | 363.2m |
| 512 | -194.7n | 19.971u | -1.2803u | 17.009u | -197.4u | | 380.0m |
| 1024 | 566.9n | 11.495u | -902.95n | 8.5331u | -566.9u | | 388.5m |
|  |  |  | (a) *R*1=1kΩ | |  | |  |
| *RF*(kΩ) | *I*in,min(A) | *I*in,max(A) | *I*out,min(A) | *I*out,max(A) | *V*virtual gnd,min(V) | *V*virtual gnd,max(V) | |
| 1 | -257.14n | 362.95u | -1.7457u | 360.31u | 771.4u | 111.2m | |
| 2 | -256.26n | 362.14u | -1.7448u | 359.49u | 768.8u | 113.6m | |
| 4 | -254.47n | 360.45u | -1.7429n | 357.78u | 763.4u | 118.6m | |
| 8 | -251.21n | 356.71u | -1.7395u | 353.98u | 753.6u | 129.9m | |
| 16 | -244.09n | 347.28u | -1.7321u | 344.46u | 732.2u | 158.2m | |
| 32 | -230.374n | 236.374u | -1.7178u | 233.866u | 691.1u | 90.88m | |
| 64 | -203.673n | 117.933u | -1.68998u | 115.848u | 611.0u | 46.2m | |
| 128 | -152.904n | 69.0335u | -1.63708u | 66.15u | 458.7u | 192.9m | |
| 256 | -60.6307n | 36.5236u | -1.54094u | 33.5749u | 181.9u | 290.4m | |
| 512 | 94.008n | 19.889u | -1.3798u | 16.931u | -282.0u | 340.3m | |
| 1024 | 321.43n | 11.471u | -1.1429u | 8.5107u | -964.3u | 365.6m | |

1. *R*1=3kΩ

Table 3: Table of *R*1, current *I*in flowing from *R*1 into virtual ground , current *I*out flowing from virtual ground into *RF* and voltage *V*virtual gnd of virtual ground (Optimal performance is highlighted in green)

## 5 Case Study

### 5.1 MNIST

The neuromorphic network designed earlier on is applied to computer vision application of MNIST(Figure 28) dataset. This dataset has 60000 samples available for training and 10000 samples available for testing. For training wise, TensorFlow python library and Keras framework were used. For computer vision prediction result visualisation as shown in Figure 43, OpenCV library was deployed. These libraries and framework is briefly described in Table 4. Their description are adapted

from their official websites.

Figure 28: Modified National Institute of Standards and Technology (MNIST) Handwritten digit database

|  |  |
| --- | --- |
| Tools | Description |
| TensorFlow | Google’s open source library for machine learning training and development.  [6] |
| Keras | TensorFlow’s high-level application interface for deep learning models developments. It’s used for fast prototyping, state-of-the-art research, and production. [7] |
| OpenCV | Open Source Computer Vision Library (OpenCV) is an open source computer vision and machine learning software library. OpenCV provides a common infrastructure for computer vision applications and speeds up the commercial use of machine perception. [8] |

Table 4: Table of tools of trade

As discussed before, each neuron output is fed to an activation function like shown in Figure 29 and Figure 30. Without an activation function, our network is just a linear regression model. That is, its continuous output is as a result of linear function. Linear functions cannot properly fit complex non-linear dataset. Therefore, we need these non-linear activation function as follows. Although ReLU looks like linear activation function, it is in fact non-linear due to changing gradient at *x* = 0. For softmax activation function, let say we have 3 neuron with output *a*, *b* and *c* respectively. the final output of

*ea eb ec*

Figure 29: Rectified linear unit (ReLU) activation function

Figure 30: Sigmoid function, a special case of Softmax activation function with 2 outputs and one output set to 0

A few network architecture has been attempted to test the performance of previous activation function (Table 5). FCN 100 here means fully connected network, i.e. all input neurons in input layer connect to each of the 100 neuron in the hidden layer. The table shows ReLU activation function is meant for hidden layers with softmax function meant for output layer. Techniques such as maxpooling (Figure 31) and 2D convolution (Figure 32) are used. Maxpooling reduces matrix

|  |  |
| --- | --- |
| Network Architecture | Accuracy |
| Input *→* FCN 100 *→* output | 11.78% |
| Input *→* FCN 100 with ReLU activation *→* output | 11.35% |
| Input *→* FCN 100 with ReLU activation *→* output with ReLU activation | 9.8% |
| Input *→* FCN 100 with ReLU activation *→* output with softmax activation | 96.91% |
| Input *→* FCN 100 with softmax activation *→* output with softmax activation | 54.97% |

Table 5: Table of network architecture with different activation function variation and their accuracies (highest accuracy is highlighted in green)

to smaller size with only the largest values. Furthermore, multi layers of convolution matrix being used to extract different features, i.e. curvature, strokes and etc. It is achieved by summing the multiplication of individual elements between large data matrix and small convolution matrix of the overlapped region. For these 2 techniques, we have a small matrix sliding across a large data matrix to perform operation. Visually, it means sliding from left to right, from 1st on top to last row at the bottom. The sliding motion is in a form of 1 step or few steps at once. For maxpooling, the sliding step size is 2, whereas

for 2D convolution matrix the sliding step size is 1.

Figure 31: Maxpooling

(a) 2D convolution at the start (b) 2D convolution at the end

Figure 32: 2D convolution operation

Unlike the architecture ofTable 5, a higher performing one involves more than just FCN. It consists of multiple repetitive units of 2D convolution, followed by maxpooling, before it is flatten from 2D matrixes to 1D vector, and finally reaching the FCN and output layer (See Figure 33). Each deeper repetitive unit of 2D convolution and maxpooling extracts subtler features. Let’s look at the 2 repetitive units in greater detail. An output of convolution layer is known as feature map. If we have *n*1 convolution matrixes working on a 2D input, there will be *n*1 feature maps, which look like a 3D stack. These feature maps will go through maxpooling to reduce the its 2D dimension, retaining its 3D height. The next convolution layer that has *n*2 convolution matrixes will have each of its matrices working on the 3D stack individually. The way it work on the 3D stack is a slightly modified version of Figure 32. In the figure, we have 1 matrix working on 1 feature map of the previous repetitive unit. Now, we have *n*1 feature map in a form of 3D stack, hence, the matrix will work on the intersected volume (*n*1 height), not just intersected area. Let’s say the top most feature map matrix operation of top left area result is 4 as shown in Figure 32, the intersected lower layers we have value *a,b,c,...*, therefore the output (top left corner unit of a 2D feature map) will be 4+*a*+*b*+*c*+*...* (*n*1 terms of them). A complete operation yields a complete 2D feature map. Do note that this is just 1 2D feature map of the 3D matrix operation from 1 of the *n*2 matrices. All *n*2 3D matrix operations yield *n*2 layers of 3D feature map stack. Flattening will simply be extracting rows by rows of a 3D volume into a 1D vector.

The rest of FCN network is already explained in subsection 1.1.3.

Figure 33: An example convolutional neural network sequence

A few types of network architecture have been attempted to discover the network characteristic by tuning the number

of neuron in FCN, introducing convolutional layers, and varying the number of convolutional network matrices.

|  |  |
| --- | --- |
| Abbreviated Title | Full Description |
| *n* FCN | Input Layer *→* Flattening Layer *→ n* Neurons Fully Connected Layer *→* 10  Neurons Fully Connected Layer *→* Softmax Activation Layer |
| *n* Conv Maxpool 10 FCN | Input Layer *→ n* 2D Convolutional Layers with 5*×*5 Kernel Size *→ n* ReLU Activation Layers *→ n* 2D Max Pooling Layers with 2*×*2 Window Size & 2  Strides *→* Flattening Layer *→* 10 Neurons Fully Connected Layer *→* Softmax Activation Layer |
| (20 Conv Maxpool) *×*2 10 FCN | Input Layer *→* 20 2D Convolutional Layers with 5*×*5 Kernel Size *→* 20  ReLU Activation Layers *→* 20 2D Max Pooling Layers with 2*×*2 Window  Size & 2 Strides *→* 20 2D Convolutional Layers with 5*×*5 Kernel Size *→* 20  ReLU Activation Layers *→* 20 2D Max Pooling Layers with 2*×*2 Window  Size & 2 Strides *→* Flattening Layer *→* 10 Neurons Fully Connected Layer *→* Softmax Activation Layer |
| 20 Conv 50 Conv Maxpool 10 FCN | Input Layer *→* 20 2D Convolutional Layers with 5*×*5 Kernel Size *→* 20  ReLU Activation Layers *→* 20 2D Max Pooling Layers with 2*×*2 Window  Size & 2 Strides *→* 50 2D Convolutional Layers with 5*×*5 Kernel Size *→* 50  ReLU Activation Layers *→* 50 2D Max Pooling Layers with 2*×*2 Window  Size & 2 Strides *→* Flattening Layer *→* 10 Neurons Fully Connected Layer *→* Softmax Activation Layer |

Table 6: Table of legends for TensorFlow trained graphs

A few weight post processing function were used on trained weight in Tensorflow and Keras, to prepare our trained neural network for neuromorphic circuit weight transfer in Cadence. Following are the weight processing function name

and their description:

1. step:

Originally, wehavediscrete8steps*wc* inTable2. InFigure18, wehave*s*(*x*axis), ascalefactorandthecorresponding compressed weight value between upper bound and lower bound. This is computed by *wc/s*. There is no whatsoever compressing being done here although we talked about those compressed weight values as we are merely using them. With these 8 steps or 9 valued *wc/s*, any input *w* below the lower bound gets clipped to the lower bound value, whereas those above upper bound get clipped to upper value and those within the boundary get rounded to nearest *wc/s*.

1. stepUniform:

Unlike previously where we used non-uniform values from Table 2, we process the input directly. We assumed the input *w* to be 0 *≤ w ≤* 1, for given *n* steps, or *n*+1 levels with uniform spacing, input *w* is rounded to one of these *n*+1 levels that *w*processed is still 0 *≤ w*processed *≤* 1. In the event our assumption of 0 *≤ w ≤* 1 is false, where *w* can be *>* 1, *w* is still rounded to one of the uniformly spaced *n*+1 levels with *w*processed *>* 1.

1. lim:

Similar to “step” function before, we still have the upper bound and lower bound with values from *wc* which set the clipping value in case our input *w* exceeds them. The difference here is we have continuous, unprocessed, original

input *w* within the boundary, not rounded to nearest step values.

1. compdecomp:

There are actual compression and decompression done here. For example, we want to map *w*, a.k.a. *wd*, within [0*,*1] to *wc* within [*a,b*] for compression, and vice versa for decompression, since decompression is just an inverse function

of compression . We can derive the decompression function as follows:

For boundary value *wc* = *b*, *bb−−aa* = 1 as needed. For boundary value *wc* = *a*, we get *ab−−aa* = 0 as expected.

Hence, the actual decompression function is:

*wc −a*

*f*(*wc*) = *b−a*

and compression function is:

*f*(*wd*) = *wd*(*b−a*)+*a*

For actual compression and decompression, *a* and *b* are 31*s* and 1*s* respectively where *s* is a scale factor like before. Steps taken to implement this function starts with compression to range [31*s,* 1*s*], “step” function to round off to nearest *wc* level within the same range (see “step” function), and finally decompression to scale back to range [0*,*1].

|  |  |  |
| --- | --- | --- |
| Figure | Figure Description | Weight Processing Function Used |
| Figure 36 | 0 *≤* weights *≤* 1 (MRAM 8 steps non-uniform scaling with lossy  rounding and capping) | step |
| Figure 37 | 0 *≤* weights *≤* 1 (Continuous scaling with lossy capping) | lim |
| Figure 38 | 0 *≤* weights (Uniform scaling with *n* steps lossy rounding) | stepUniform |
| Figure 39 | 0 *≤* weights *≤* 1 (MRAM 8 steps non-uniform lossy rounding with  lossless compression/decompression) | compdecomp |
| Figure 40 | 0 *≤* weights *≤* 1 (Continuous scaling with lossy capping) | limit |
| Figure 41 | 0 *≤* weights *≤* 1 (Uniform scaling with *n* steps lossy rounding) | stepUniform |

Table 7: Table of figures and weight processing function used

Those 6 figures inTable 7 were evaluated in the following pages. First 3 figures have their weights ranging *>*1 due to a false assumption of TensorFlow constraint in training that positive weight is never *>* 1, whereas last 3 figures are all between 0 and 1. There are a few values, falling slightly to *≈ −*0*.*1 (See Figure 34 and these values will be clipped) due to gradual implemention of constraint in training the network so as to achieve optimal learning rate as shown in Figure 4. In general sense, factors that contribute significantly to low network performance are weight clipping and large size of network. The former limits the range of weight that can be represented and therefore distort the trained model (architecture + weight), whereas the latter amplifies the distortion when the network scales and complexity increases. Worst case for all the network accuracy is at around 10%, i.e. the network gets 1/10 correct by getting stuck to a particular digit prediction for all 0-9 digits in uniform distribution. Clipping/Capping is very bad for weight value that fall outside the boundary. Where clipping has failed, weight compression and decompression comes to rescue by flat lining throughout (Figure 39). The flat lining effect is due to the process compression and decompression is lossless and the only lossy process is rounding to nearest step. Nevertheless, the performance still drop for weight processed with “compdecomp” due to larger network size (errors from rounding to nearest step and clipping of weight falling slightly below 0 got amplified). Other than 8 steps weight from MRAM, *n* steps weight are also experimented so that we can get the optimal *n* steps for future MRAM without sacrificing too much network performance. Those figures with uniform scaling (Figure 38 and Figure 41) with *n* steps converge to extremely high accuracy. Another phenomena we observed is a network performance is the lowest for highest scaled R due to a few important and influential (on the outcome) high weight *w* got filtered out (outside the boundary), leaving many

low weight *w* lying around within the boundary, forming the model.

Finally, 20 FCN model (highlighted in green in Figure 39) was chosen for its attributes such as very reasonably high accuracy of 80.24% and significantly smaller scale and simpler implementation of network, minus away complex convolution and maxpooling function. These 2 functions are good in neural network, but they complicate simple neuromorphic network task of recognizing handwritten digits by demanding for more complex circuit, and asking for more errors.

Figure 34: Original and modified (rounded) weight distribution for 20 FCN with “compdecomp” weight processing function

Figure 35: Figure 34 (zoomed). Note those higher but sparser weight distribution towards the right.

In Figure 34, most weight cluster around 0, with some extending slightly before 0 as discussed earlier. These weight will be clipped to the leftmost modified weight at the extreme end of boundary (orange). It is quite lucky in a sense that the distribution of 8 steps modified weight of mram (orange) closely mimicks the distribution of weight (blue) due to its non linear spacing. By that it means, denser/tighter spacing of modified weight (orange) is observed towards the left end where the lower weight (blue) has the highest distribution towards the end (at 0, ignore those *<*0), and increasingly sparser spacing of modified weight (orange) towards the right where the higher/more influential weight (blue) has the lowest distribution

(see zoomed Figure 35).

(a) 10 FCN (b) 1000 FCN

(c) 20 Conv Maxpool 10 FCN (d) 100 Conv Maxpool 10 FCN

(e) (20 Conv Maxpool)*×*2 10 FCN (f) 20 Conv Maxpool 50 Conv Maxpool 10 FCN

Figure 36: 0 *≤* weights *≤* 1 (MRAM 8 steps non-uniform scaling with lossy rounding and capping)

(a) 10 FCN (b) 1000 FCN

(c) 20 Conv Maxpool 10 FCN (d) 100 Conv Maxpool 10 FCN

(e) (20 Conv Maxpool)*×*2 10 FCN (f) 20 Conv Maxpool 50 Conv Maxpool 10 FCN

Figure 37: 0 *≤* weights *≤* 1 (Continuous scaling with lossy capping)

(a) 10 FCN (b) 1000 FCN

(c) 20 Conv Maxpool 10 FCN (d) 100 Conv Maxpool 10 FCN

(e) (20 Conv Maxpool)*×*2 10 FCN (f) 20 Conv Maxpool 50 Conv Maxpool 10 FCN

Figure 38: *≤* weights (Uniform scaling with *n* steps lossy rounding)

(a) 20 FCN (b) 1000 FCN

(c) 20 Conv Maxpool 10 FCN (d) 100 Conv Maxpool 10 FCN

(e) (20 Conv Maxpool)*×*2 10 FCN (f) 20 Conv Maxpool 50 Conv Maxpool 10 FCN

Figure 39: 0 *≤* weights *≤* 1 (MRAM 8 steps non-uniform lossy rounding with lossless compression/decompression)

(a) 20 FCN (b) 1000 FCN

(c) 20 Conv Maxpool 10 FCN (d) 100 Conv Maxpool 10 FCN

(e) (20 Conv Maxpool)*×*2 10 FCN (f) 20 Conv Maxpool 50 Conv Maxpool 10 FCN

Figure 40: *≤* weights *≤* 1 (Continuous scaling with lossy capping)

(a) 20 FCN (b) 1000 FCN

(c) 20 Conv Maxpool 10 FCN (d) 100 Conv Maxpool 10 FCN

(e) (20 Conv Maxpool)*×*2 10 FCN (f) 20 Conv Maxpool 50 Conv Maxpool 10 FCN

Figure 41: 0 *≤* weights *≤* 1 (Uniform scaling with *n* steps lossy rounding)

Figure 42: 20 FCN for neuromorphic circuit

Figure 43: Prediction of last 40 handwritten digits from intermediate output/input of 10000 MNIST test data (predicted in top left corner with true in green and false in red)

In previous chosen 20 FCN model for neuromorphic network implementation, we make the ReLU activation function redundant by processing the weight to *≥ wc ≥* 1, and with our input 0 *≥ V*in *≥* 1, all the outputs are naturally *>* 0, which are already the same as outputs of a ReLU function even when it is implemented. Softmax function are not implemented

in neuromorphic network due to the inherent complexity, so only post processing for Softmax are used.

After taking the architecture of 20 FCN, trained weights of 10*×*20 dense layer, and intermediate output/input from 20*×*768 dense layer (Figure 42) from TensorFlow and transferring it to neuromorphic network in Cadence, the simulation onCadencewasruntotesttheperformanceofoffchiplearningfor1000MNISTtestdataset. OpenCVwasusedtovisualised

the last 40 samples of test dataset and their predicted values (Figure 43).

The performance for small scale 10*×*20 dense layer was astonishing. It achieved an accuracy of 81*.*02% vs 80*.*24%

predicted by TensorFlow. The rounding errors may be very helpful in improving the network performance by chance.

Full simulation of 20 FCN was too conducted with a miserable accuracy of around 10% due to the reasons in Challenges section (section 4) as all outputs are stuck at predicting 0. Basically, an access to better performing opamp model is needed to pull feat off.

## 6 Conclusion and Future Work

Both neuromorphic network and spintronics have been gaining traction in the field of artificial intelligence and power efficient, high performance memory device. The combination of these two is evolutionary in the future development and advancement in the field. This project therefore explores the practical side of implementation and migration of off chip trained data to neuromorphic circuit, and compare the performance between the two, and highlight the challenges and shortcoming. Small scale neuromorphic network has been performing phenomenally with MNIST dataset at 81.02% accuracy, sometimes even better than off chip Tensorflow performance at 80.24%, but large scale performance suffers due to non-ideality of opamp. Therefore, immediate future work include sourcing a higher precision opamp, or developing an opamp capable of sustaining its virtual ground voltage, by being immune to both supply voltage and varying input voltage. Further work can include completing the simulation of large scale neuromorphic network once the opamp shortcoming is sorted, full domain wall synapse software Cadence simulation model implementation and simulation, and finally explore the horizon of domain wall spiked based neuromorphic network, encouraged by benefits such as ultra low power consumption, as pointed out by the reviewed literatures.