UNIVERSITY OF CALIFORNIA

SANTA CRUZ

DESIGN AND TRAINING OF MEMRISTOR-BASED NEURAL NETWORKS

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL AND COMPUTER ENGINEERING

by

Xiaoyang Jia

June 2020

The Thesis of Xiaoyang Jia
is approved:

Professor Sung Mo "Steve"Kang

Professor Shiva Abbaszadeh

Professor Yu Zhang

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

Abstract

Design and Training of Memristor-based Neural Networks

by

Xiaoyang Jia

Modern Artificial Neural Network(ANN) is a kind of nonlinear statistical data modeling tool, which can be optimized by a learning method based on mathematical statistics. Therefore, it is a practical application of mathematical statics. The ANN can get abilities to make simple decisions and judgments, just like a human brain. It is superior to formal logical implication.

Since the advancement of the ANN study significantly depends on the expansion of networks in-depth, a massive amount of vector-matrix multiplications is required [3]. Thus, energy efficiency is a key factor in evaluating the performance of ANNs. Since researches on vector-matrix multiplications have made great achievements, large and deep ANNs have been used to handle complex tasks and process massive data. The memories utilized by conventional ANNs, such as Static Random Access Memory (SRAM) and Flash memory are charge-based, which is not efficient in view of energy consumption during ANN computation since it cannot directly implement vector-matrix multiplication in a crossbar array structure [3]. Over the past decade, the Nonvolatile Memory (NVM) crossbar array has shown its superiority on improving the energy efficiency. Unlike the conventional memory, NVM is current based. NVM crossbar array can calculate matrix multiplication in a single step by sampling the current flowing, therefore, it was utilized as the analog vector-matrix multiplier for on ANN [3]. However, the

nonlinear I-V characteristics of NVM put hard constrains on critical design parameters, such as the read voltage and the range of weight, which causes substantially reduced accuracy.

In this work, we built an ionic floating-gate (IFG) memory unit device model on Cadence based on two presented models, ENODe and ionic floating-gate memory (IFG) memory [2]. The IFG model consists of a polymer redox transistor connected to a conductive-bridge memory (CBM) [2]. We would like to apply this IFG memory unit device to build Memristor-based Neural Networks (MNN) and explore the performance of the networks. In the MNNs, the selective and linear programming of a redox transistor array is executed in parallel by overcoming the bridging threshold voltage of the CBMs [2], which can improve the performance (accuracy). This thesis is mainly about the design and training of the MNNs. Since we would like to apply the MNNs to perform digital number recognition, we designed the architecture that is available to recognize Modified National Institute of Standards and Technology (MNIST) handwriting digits and trained the MNN with MNIST train dataset, then tested the performance (accuracy) by MNIST test dataset. In addition, we tried to make some handwriting images by ourselves and used them to test the trained network to consolidate our conclusion. We also attempted to resize the original MNIST dataset by an image data pixel converting algorithm and make the newly created train dataset applicable to train an MNN with smaller scale. We used both the MNIST test dataset and the handwritten image data newly created to test the network and compared the performance with that of the

original one. We hope that the IFG memory unit device can impose a significant impact on the MNN design.

Acknowledgements

I want to thank my parents for giving me financial support during my study in the University of California Santa Cruz, they also give me sufficient love and patience during my growing process.

I would like to thank Professor Steve Kang for providing me to the software design and training steps of our project, and for the sufficient reference he provided and his unwavering patience. He always gives us helpful suggestions dealing with challenging problems. Best wishes to you in the future work, and I hope we can have a chance to see each other again someday.

I also want to thank Professor Shiva Abbiszadeh and Professor Yu Zhang for serving on the thesis reading committee and providing suggestions for improvement.

To my colleagues, Donguk Choi, Yinghao Shao, for helping me with the research, for giving me favors during difficult time and for the friendship.

Chapter 1
Introduction and Background

1.1 Motivation

The research on Artificial Neural Network started in 1943, and a computational model was created by Warren McCulloch and Walter Pitts. The model is based on threshold logic algorithms [1]. Their work laid the foundation for further research on Artificial Neural Network (ANN) to be classified into two approaches; one focuses on biological process, the other mainly devote to the application of ANN to Artificial Intelligence. ANN is a mathematical calculation model of the biological neural network, which can perform the estimation and approximation of functions. Because of the long history and development, ANN computation has grown into a massive multidisciplinary analysis field. It has many functional models that can handle nonlinear problems in a way quite similar to brains. The promotion of ANN study heavily depends on the expansion of depth in neural networks. Thus, in-depth research has been performed in the past ten years to yield great achievement, enabling ANN to handle many complex problems such as pattern recognition, artificial intelligence (AI), estimation, biology, medical, and economy. ANN demonstrated excellent intelligence features, and the performance

is shown to be superior to that of general-purpose computers. However, much remains to be done to remove restrictions on ANN.

Memristor can be considered as a kind of non-volatile memory (NVM), which is in the nano-scale and has low power consumption. With its memory property, memristor can remember the connection weight between nodes. Besides, its scalability and energy efficiency make it applicable to huge networks. Therefore, memristive crossbar arrays showed a great feasibility for implementing intelligent neuromorphic operations. It is conceivable to implement ANN in a highly integrated circuit chip with a density similar to the human brain. Inspired by a proposed ionic floating-gate (IFG) memory array based on a polymer redox transistor connected to a conductive-bridge memory (CBM), we designed IFG device arrays using Cadence tools based on the reference [2]. We can read out the synaptic weight with currents <10 nano amperes by diluting the conductive polymer with an insulator to decrease the conductance, and it can endure >1 billion write-read operations and support >1-megaherz write-read frequencies [2]. We would like to explore the performance of networks based on IFG device. My work is to design the MNNs on software and get the optimized weights, and then give them to my colleagues Donguk and Yinghao so that they can map the weights to the conductance of each unit in the IFG arrays and do the hardware simulation.\

1.2 Artificial Neural Network (ANN) Fundamentals

To illustrate the ANN model, we need to introduce some basic concepts. Artificial neural networks (ANN) are nonlinear and self-adapting computing systems inspired by the biological neural networks of animal brains [4]. ANN can learn, with or without training, from examples to find the pattern to perform tasks without being programmed in specific rules. An ANN consists of a collection of connected units or nodes called artificial neurons, which is similar to the neurons of the biological brain. Each neuron node is connected to other nodes via links that correspond to biological axon-synapse-dendrite connections in biological brain. Each link has a weight, which determines the strength of one node's influence on another [5]

1.2.1 Artificial Neurons and Neural Layers

ANNs are composed of artificial neurons, which retain the biological concept of neurons. They receive inputs and combine the inputs by utilizing an activation function with an optional threshold value [6]. If the combined input is not large enough, the effect of the activation threshold function will suppress the output. On the other hand, if it is large enough, the effect will fire the neuron and then produce an output [6], as shown in Figure 1.1. The output of each neuron is computed by some nonlinear activation function of the weighted sum of its inputs.

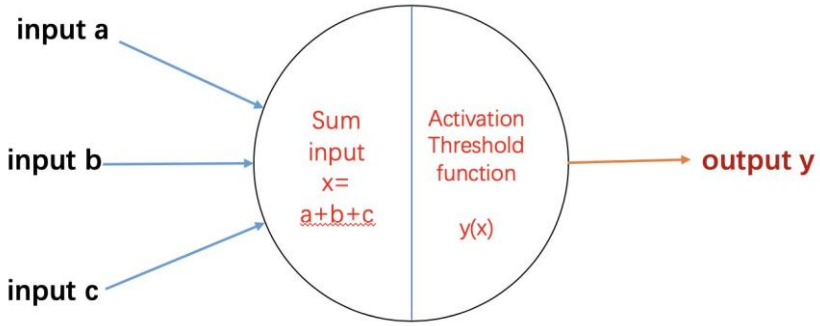


Figure 1.1: Artificial neuron, the circular node represents artificial neuron and arrows represent the input and output.

In order to replicate the biological mechanism presented above and make ANN functional, artificial neurons are typically organized into multiple layers, especially in deep learning. [6] Figure1.2 illustrates this idea. Each neuron or node of one layer connects to every node in the preceding and next layers in a fully connected network. The layer that receives external data is the input layer, and the layer that produces the ultimate result is the output layer. In between are hidden layers. The nodes between two layers can also be not fully connected; they can be pooling, where a group of neurons in one layer connect to a single neuron in the next layer, thereby reducing the number of neurons in that layer [7].

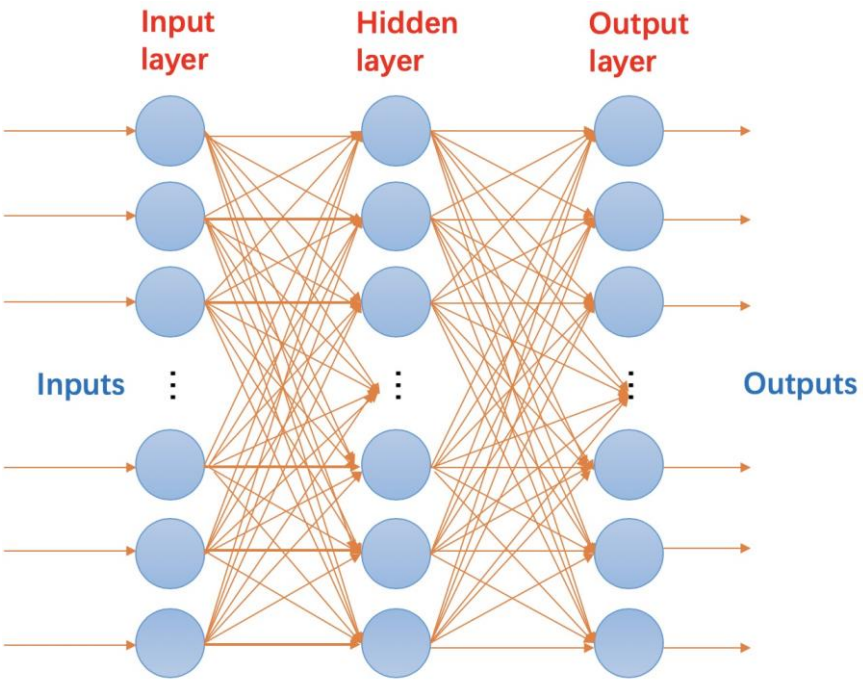


Figure 1.2: ANN layer model, an interconnected group of nodes, circular nodes represents artificial neurons, and arrows between two nodes represent the connections.

1.2.2 Connections and Weights

For ANN's implementation, each of the connections between two layers has a parameter named weight, which represents the relative importance of its corresponding connection [5]. When the output generated by nodes of the preceding layer transmits to the nodes of the next layer, they multiply the weight of their corresponding connection, and the results act as the input to the next layer, as shown in Figure 1.3. Therefore, the value of weights can increase or decrease the strength of a signal at connections to adjust the learning process of ANN

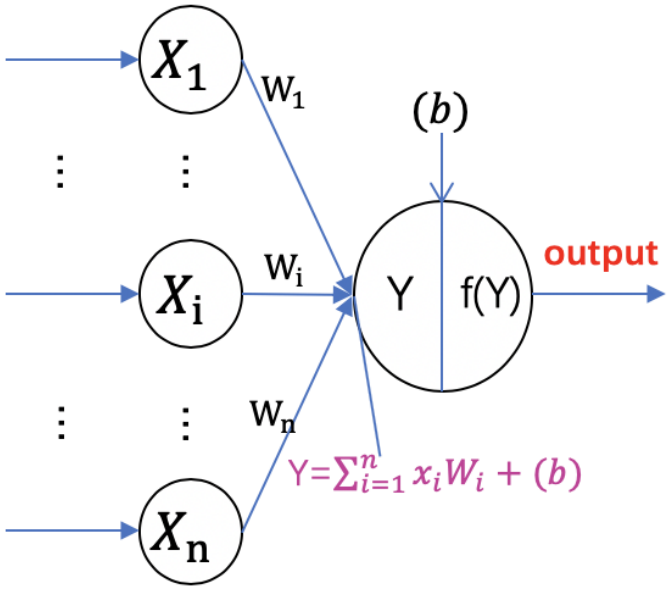


Figure 1.3: Theoretical weight model of ANN. x_i denotes the output signal of node X_i, W_i denotes the weight value of the connections, Y denotes the combined signal. f denotes the activation function, f(Y) is the output singal, and b denotes the bias. The bias can modify the range of the input of Y. But it is not necessary, and we did not apply it to our project.

1.2.3 Activation Function

The activation function is the function running on the neurons, shown in Figure1.3. It is responsible for mapping the inputs of a neuron to the output. It is of great significance for the ANN model since it gives an important nonlinear characteristic to the network, which enables the ANN model to compute complex and nontrivial problems. Without nonlinearity, the output of each layer is merely the linear function of the inputs of the preceding layer. That is, no matter how many layers the ANN has, the outputs are always the linear combination of original inputs, which will restrict the performance and the application field.

There are several common activation functions, such as sigmoid function, tanh (hyperbolic tangent) function, and ReLu (Rectified Linear unit) function. Their expressions and graphs are shown in Figure 1.4. For our project, we chose the sigmoid function as the activation function. Sigmoid function is easy to work with and it has a fixed output range. Although it can cause the vanishing gradients problem when x value is large, it still has the whole nice properties of activation functions.

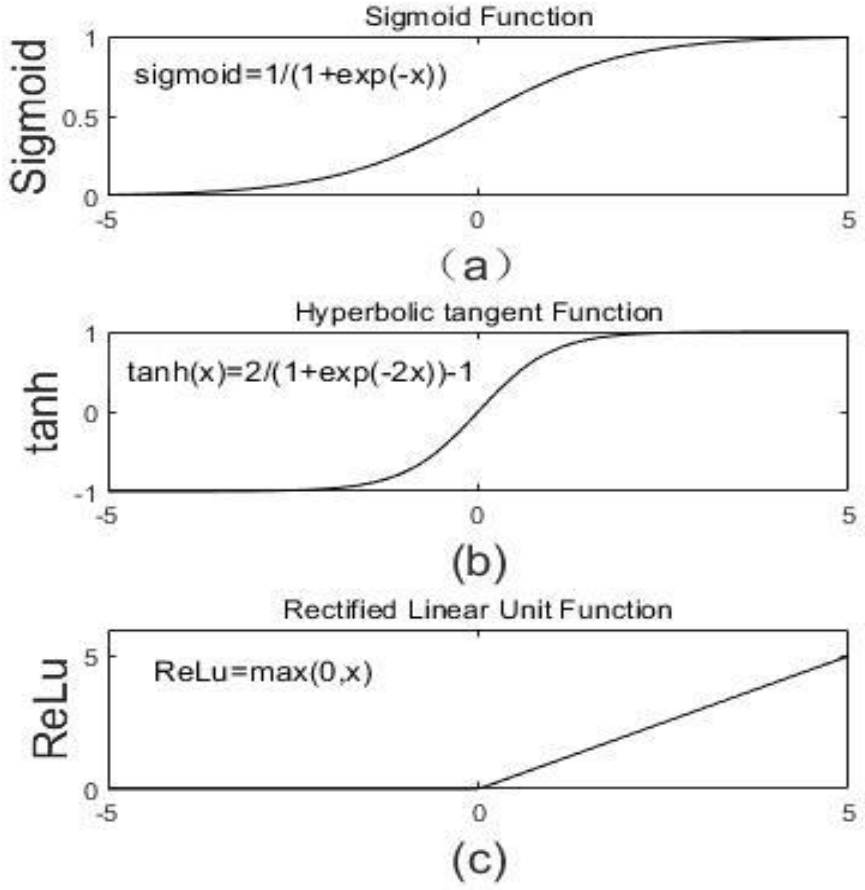


Figure 1.4: Common activation functions. (a)Sigmoid function. (b) tanh. (c) ReLu.

1.2.4 Loss Function

In optimization, the function applied to evaluate candidate solutions is usually called an objective function. Researchers may devote themselves to optimize (minimize or maximize) the objective function to find a candidate solution that has the highest or lowest score respectively. When it comes to ANNs, we concentrate on minimizing the error, which is the difference between the target output and the actual output. In this case, the objective function is usually referred as a loss function and the value calculated by it is simply the error. Loss function is applied to optimize the parameter values (weights) in the ANN model and achieve the optimal performance for specific tasks. The process we use to minimize the error of the ANN's output is referred as training.

1.2.5 Matrix Multiplication

Matrix Multiplication is quite useful for doing the ANN computation. Assume that we have a 2-layer ANN with 2 or 3 nodes in each layer. In this case, we can do the calculation manually since the network is simple. However, imagine that we need to do the same thing for an ANN with 5 layers and 150 nodes in each. Merely performing all the necessary calculations will be a huge challenge. The combination of combining signals, multiplied by the right weights, applying the activation function, for each node, each layer [6], which can be enormously overwhelming. Even though we can do the calculation by some algorithm, the large

amount of coding work is still challenging. Matrix operation can compress all such calculations into a straightforward form. As is well-known, a matrix is just a table or a rectangular grid of numbers, so matrix multiplication enables us to express all the work of ANN computation concisely and easily [6]. Besides, many computer programming languages like Python and Matlab can recognize matrices and do appropriate calculations. Therefore, we can utilize the computer program to do all the calculation work accurately and efficiently. The matrix multiplication between two layers can be simply expressed as

$$\boldsymbol{X} = \boldsymbol{W} \cdot \boldsymbol{I} \tag{1.1}$$

$\boldsymbol{I}$ denotes the vector of inputs, and $\boldsymbol{W}$ denotes the matrix of weights between two layers. $\boldsymbol{X}$ is the resultant vector of combined signals to be adjusted by the second layer. [6] Then the output of the second layer can be expressed as

$$\boldsymbol{O} = Sigmoid(\boldsymbol{X}) \tag{1.2}$$

$\boldsymbol{O}$ is the output vector, which consists of all the output signal from the second layer. If we have more than two layers, like 3 layers, we just simply do the matrix multiplication again, treating $\boldsymbol{O}$ as the input matrix to the third layer, and of course, there is bound to be another weight matrix containing weights between the second and third layer, which is responsible for combining and moderating the input into the third layer [6]. This is not difficult to understand and encode an algorithm to

efficiently solve all of these calculations. Therefore, matrix multiplication is an efficient and powerful tool to implement ANN.

1.2.6 Training

When we start with a complete ANN model, we randomly initialize the value of weights. In this case, when we apply the input to the network model, the output would not be correct. Training of the network, namely, the network's learning process to evaluate the influence of the parameters (weights w_{ij}) and modify them so that the output can approach the target output, which is the most genuine part of deep learning. This learning process can be regarded as an iterative process of "going and return" of information by the layers. The "going" and "return" are respectively the forward propagation and back propagation of the information in the ANN. The information forward propagated through the ANN is the training datasets, and the information backpropagated is the loss (error) information. We can summarize all the training process by Figure 1.5.

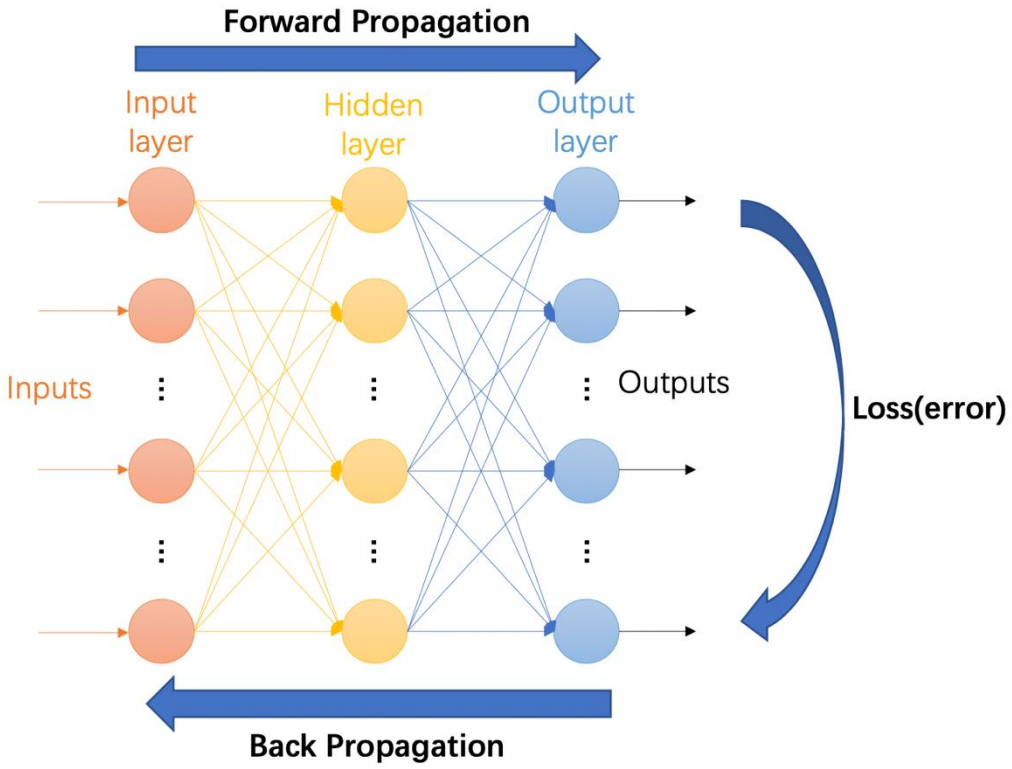


Figure 1.5. Visual scheme of the training process

Forward propagation is the first phase of the training. It occurs when the training datasets are applied to the network and cross the whole neural network for their corresponding outputs to be calculated. During this process, all the inputs pass through the ANN in a way that all the neurons perform transformation to the information they receive from the preceding layer and transmit the processed information to the next layer. The resultant outputs can be compared to the expected outputs in the training data.

In the next phase, the loss function comes into play. In a supervised learning environment, we have the target output value of each input data in the training

datasets. The loss function is applied to estimate the loss (error), which is the divergence between the target value and our actual output value. Ideally, our purpose is to eliminate the loss (error) after training, but in practice, as the ANN model is being trained, it will adjust the value of the weights of the interconnections between neurons of different layers to approach the target value until the actual output is close enough to the desired output.

Once the loss (error) of each node in the output layer has been worked out, the loss information will be propagated backward to guide the refinement of weights inside the network, which is referred as backpropagation. The diagram of the backpropagation is shown in Figure 1.6. Here a simple ANN has three layers, each layer with two nodes. Starting from the output layer, two errors e_{o1} and e_{o2}, each denoting the error of two output nodes, are propagated to all neurons in the hidden layer to calculate the errors (loss) of nodes in the hidden layer, which are denoted as e_{h1} and e_{h2}. Each of the two hidden nodes has two links connecting them to the two output nodes. The error of each output node is split in a way that is proportional to the weights of links connecting to it as shown in Figure 1.6. We can recombine link errors of each two links emerging from a hidden node and form the hidden error for this node. This process is repeated layer by layer until every neuron in the ANN model has received its corresponding error, which is its relative

contribution to the whole error. The following expressions show the error of each node in the network shown in Figure 1.6

$$e_{h1} = \frac{w'_{11}}{w'_{11}+w'_{21}} \cdot e_{o1} + \frac{w'_{12}}{w'_{12}+w'_{22}} \cdot e_{o2} \quad (1.3)$$

$$e_{h2} = \frac{w'_{21}}{w'_{11}+w'_{21}} \cdot e_{o1} + \frac{w'_{22}}{w'_{12}+w'_{22}} \cdot e_{o2} \quad (1.4)$$

$$e_{i1} = \frac{w_{11}}{w_{11}+w_{12}} \cdot e_{h1} + \frac{w_{12}}{w_{12}+w_{22}} \cdot e_{h2} \quad (1.5)$$

$$e_{i2} = \frac{w_{21}}{w_{11}+w_{12}} \cdot e_{h1} + \frac{w_{22}}{w_{12}+w_{22}} \cdot e_{h2} \quad (1.6)$$

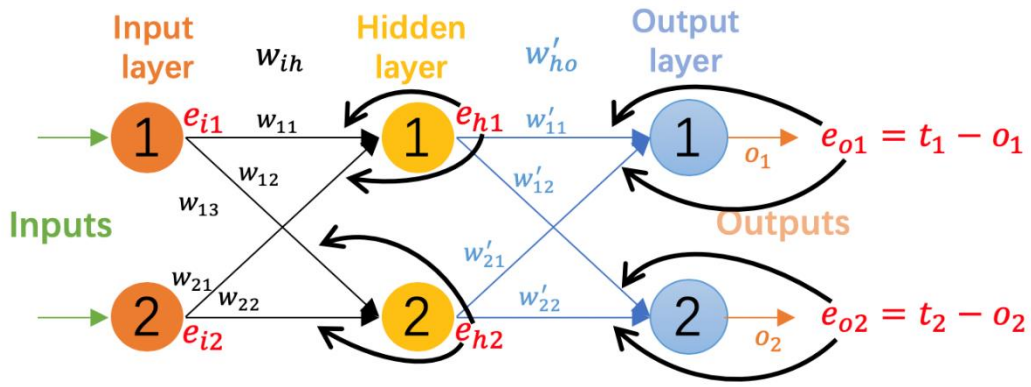


Figure 1.6: Diagram of backpropagation. e_{o1}, e_{o2} are the error of output nodes, e_{h1}, e_{h2} are the error of hidden nodes, e_{i1}, e_{i2} are the error of input nodes, w_{ih} denotes the weight between input and hidden layer, w'_{ho} denotes the weight between hidden and output layer.

We can simplify all the laborious calculations above and make the backpropagation more concise by utilizing matrix multiplication. At first, we need to construct the matrix for output error (Eq.1.7), then we can have the error matrix for the hidden layer (Eq.1.8).

$$error_{output} = \begin{pmatrix} e_{o1} \\ e_{o2} \end{pmatrix} \quad (1.7)$$

$$error_{hidden} = \begin{pmatrix} \frac{w'_{11}}{w'_{11}+w'_{21}} & \frac{w'_{12}}{w'_{12}+w'_{22}} \\ \frac{w'_{21}}{w'_{11}+w'_{21}} & \frac{w'_{22}}{w'_{12}+w'_{22}} \end{pmatrix} \cdot \begin{pmatrix} e_{o1} \\ e_{o2} \end{pmatrix} \quad (1.8)$$

By looking at the Eq. 1.8, we can find that the most significant part is the multiplication of the output error with its connected weight. The output error propagated back to the hidden layer is proportional to the value of the linked weight. Thus, the denominator is a kind of normalizing factor. Therefore, we can simplify the matrix again, as shown in Eq. 1.9 by ignoring the normalization factor. Obviously, the weight matrix in Eq.1.9 is the transpose of the original weight matrix $\boldsymbol{W_{ho}}$. We can normalize the matrix approach of backpropagation by Eq. 1.10.

$$error_{hidden} = \begin{pmatrix} w'_{11} & w'_{12} \\ w'_{21} & w'_{22} \end{pmatrix} \cdot \begin{pmatrix} e_{o1} \\ e_{o2} \end{pmatrix} \quad (1.9)$$

$$\boldsymbol{error_{hidden} = W_{hidden\_output} \cdot error_{output}} \quad (1.10)$$

Now that the loss (error) has been successfully propagated back, we next consider how to update weights to make the loss (error) approach zero. To achieve this, we need to utilize a technique called gradient descent, which modifies the weights in small increments by the calculation of the gradient of the loss function. The gradient refers to the slope of the points on the curve of the loss function. Since the initial weights are randomly set, the starting point on the loss function curve is not fixed, which indicates that the direction to descend towards the minimum of

loss can be both positive and negative. The gradient descent is performed in sequence in the successive epochs of all the training data that we apply to the ANN in each epoch so that the weights would be modified to enable the network to render the desired output. The scheme of gradient descent is shown in Figure 1.7

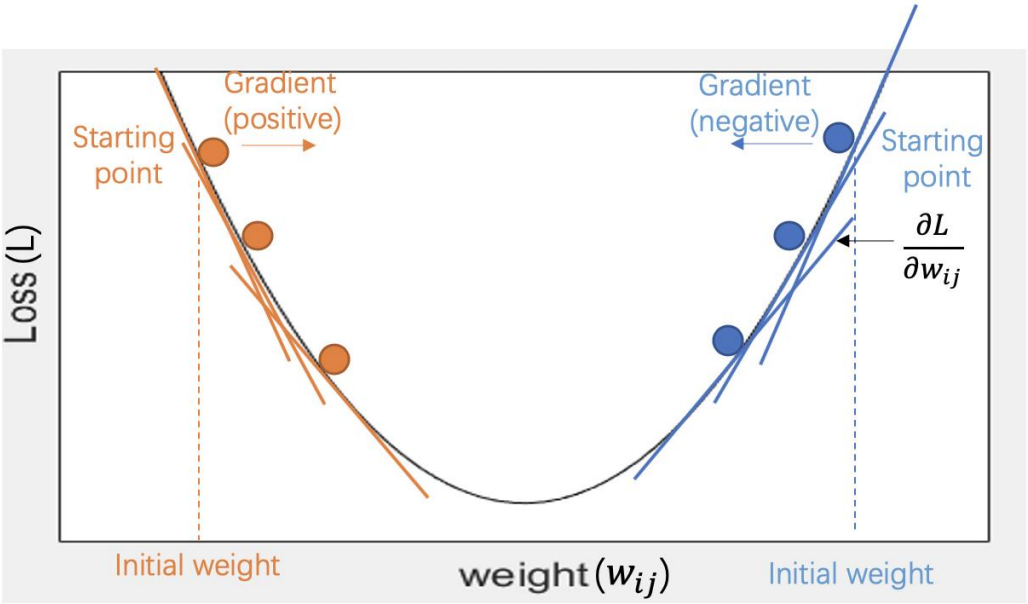


Figure1.7: Scheme of gradient descent

Let L denotes the loss function, the slope of the loss function for the weights can be expressed as Eq.1.11, and we can update the weights by following the rule shown in Eq.1.12. α is the learning rate, and it is applied to moderate the strength of the weight changes and avoid overshooting.

$$\frac{\partial L}{\partial L w_{ij}} = -(L_j) \cdot \ activation(\sum_i w_{ij} \cdot O_i) \cdot (1 - activation(\sum_i w_{ij} \cdot O_i)) \cdot O_i \quad (1.11)$$

$$\Delta w_{ij} = \alpha \cdot L_j \cdot activation(O_j) \cdot (1 - activation(O_j)) \cdot O_i^T \quad (1.12)$$

1.2.7 Applications

Generally speaking, the original purpose of the ANN model was to handle things in the same way as that of the human brain. For image recognition, ANN is able to identify images of handwritting digital numbers after getting trained by a massive amount of example images with different writing styles. ANNs can also be applied to do identify images that contain other things, like dogs. After learning from example images labeled as “dog” or “no dog”, ANNs can identify arbitrarily selected images and judge whether the images contain dogs. However, over time, researches paid more attention to performing specific tasks, which led to deviations from biology. Today, ANN has been used for a variety of tasks, including computer vision, speech recognition, machine transition, social network filtering, playing board, and video games, medical diagnosis, painting [5] and even for solving business problems, like market forecasting, data validation, and risk management. In this thesis, we use the MNN (Memristive Neural Network) for digital number image recognition and analysis of the performance.

1.3 Memristor-based Neural Network (MNN)

The following is intended as a brief review of memristor fundamentals and the introduction of both the IFG device and memristive crossbar array structure we utilized to build MNN.

1.3.1 Memristor Fundamentals

Memristor is the concatenation of “memory resistor”, which was first introduced in 1971 [8]. It is a two-terminal circuit element that constitutes the relationship between flux and charge, the missing pair link among the four basic circuit variables [8], as shown in Figure 1.8. Several years later, Chua and Kang [9] introduced to the scientific community the generic properties of a broad generalization of the memristor to an interesting class of nonlinear dynamical devices, called memristive devices. [10]

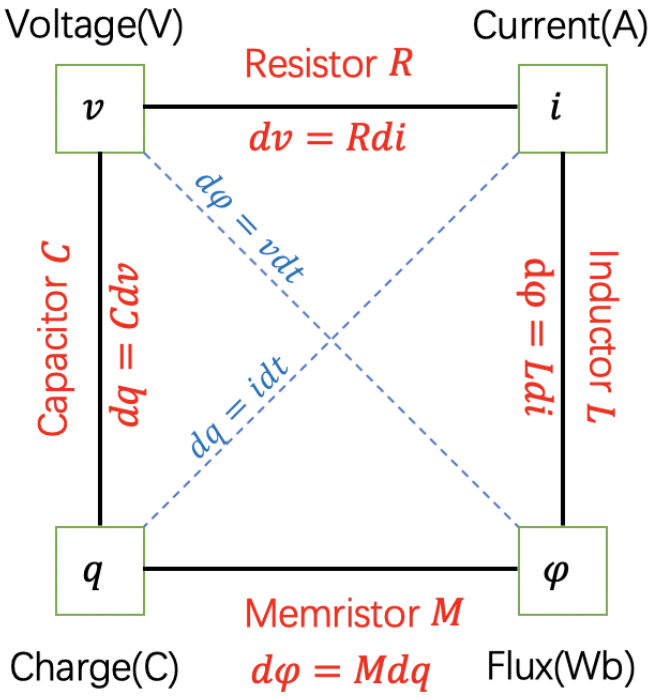


Figure 1.8: The four basic electrical elements: resistor, inductor, capacitor and memristor, and their constitutive relationship in terms of four fundamental circuit variables.

Normally, the memristor is modeled by a state-dependent Ohm's Law. There are two types of time-invariant memristors depending on the type of the input signal. The memristor with a current source as the input signal is named as the current-controlled memristor, while the memristor with a voltage source as the input signal is called voltage-controlled memristor. In a broader sense, any electrical device with two terminals can be called a memristor if the behavior of it can be described by a nonlinear constitutive relation between the voltage drop at its terminals v and the current flowing through the device i as shown below: [10]

Current-controlled memristor:

$$v = R(x)i \tag{1.13}$$

It has the state equation as follows:

$$\frac{dx}{dt} = f(x, i) \tag{1.14}$$

Voltage-controlled memristor:

$$i = G(x)v \tag{1.15}$$

It has the state equation as follows:

$$\frac{dx}{dt} = g(x, v) \tag{1.16}$$

R(x) and G(x) are named as memristance (memory resistance) and memductance (memory conductance), respectively, and they have units Ω (Ohm) and S (Siemens) [10]. The x is a state-vector, which has n (n≥1) components $x_1, \ldots x_n$ called state-variables. These state-variables represent internal physical parameters and do not depend on any external variables like voltages or currents [10].

We can consider the ideal case of memristor, when the state equations (Eqs. 1.14 and 1.16) can be expressed as $f(x, i) = i$, and $g(x, v) = v$. So, we can integrate both sides of the equations and get the following equations:

$$x(t) = \int_{-\infty}^{t} i(\tau)d\tau = q(t) \tag{1.17}$$

$$x(t) = \int_{-\infty}^{t} v(\tau)d\tau = \varphi(t) \qquad (1.18)$$

Then we can substitute the above equations for x in Eqs. 1.13 and 1.14 respectively, and integrating both sides, [10] which gives:

$$\varphi(t) = \int_{-\infty}^{t} v(\tau)d\tau = \int_{-\infty}^{t} R(q(\tau))\frac{dq(\tau)}{d\tau}d\tau = \int_{-\infty}^{q(t)} R(q)dq = \hat{\varphi}\big(q(t)\big) \quad (1.19)$$

$$q(t) = \int_{-\infty}^{t} i(\tau)d\tau = \int_{-\infty}^{t} G(\varphi(\tau))\frac{d\varphi(\tau)}{d\tau}d\tau = \int_{-\infty}^{\varphi(t)} G(\varphi)d\varphi = \hat{q}\big(\varphi(t)\big) \quad (1.20)$$

The Eqs. 1.19 and 1.20 indicate that, in this case, Eqs. 1.13and 1.14, which defines the current-controlled memristor, are equivalent to a defined by a single equation as shown below, which defines a charge controlled memristor:

$$\varphi = \hat{\varphi}(q) \qquad (1.21)$$

Also, Eqs. 1.15, 1.16 defining the voltage-controlled memristor are equivalent to the following equation, which defines a flux-controlled memristor.

$$q = \hat{q}(\varphi) \qquad (1.22)$$

Later on, these are precisely described as the fourth constitutive relationship between the charge q and flux φ in Figure 1.8, which defines the memristor by an axiomatic approach in which q and φ need not have any physical significance. We can also differentiate Eqs. 1.21 and 1.22 and obtain two new equations:

$$v = \frac{d\varphi}{dt} = \frac{d\hat{\varphi}(q)}{dq}\frac{dq}{dt} = R(q)i \qquad (1.23)$$

$$i = \frac{dq}{dt} = \frac{d\hat{q}(\varphi)}{d\varphi}\frac{d\varphi}{dt} = G(\varphi)v \qquad (1.24)$$

These two equations show that the charge-controlled memristor is equivalent to a charge-dependent Ohm's Law in which the $R(q)$ is merely the slope of the curve $\varphi = \varphi(q)$ at q and the flux-controlled memristor is equivalent to a flux-dependent Ohm's Law where the $G(\varphi)$ is the slope of the curve $q = q(\varphi)$ at φ [10]. Since the ideal memristor relation is quite rare to do serve in experiment, normally, we model the memristive device using the state-dependent Ohm's Law presented above. A comprehensive and detailed review is given in [10].

1.3.2 Memristor Crossbar-based Nonvolatile Memory

Memory technologies can be divided into volatile and nonvolatile on the basis of the ability to retain data with and without power. Today, nonvolatile memory (NVM) is universally acknowledged, since it provides key advantages on data storage and the NVM crossbar array is used for implementing matrix multiplication, which is essential to ANN implementation. Therefore, emerging NVM technologies, including Phase-Change Random Access Memory (PCRAM), Resistive Random Access Memory (ReRAM), Conductive-Bridge Random Access Memory (CBRAM), and Spin-Transfer-Torque Magnetic Random Access Memory (STT-RAM) have been widely studied as next-generation memories [11]. Unlike the conventional memories that are charge-based, emerging NVM is current-based, it enables NVM crossbar array to calculate matrix multiplication in a single step by

sampling the current flow in each column [3], which opens up an opportunity to apply it for ANN acceleration. And emerging NVM represents its states with different resistance values. This type of storage device is normally memristive.

The crossbar is the most prominent and well-documented memristive framework in literature and is among the most promising candidates to implement memristor-based ReRAM arrays [12]. The two-terminal structure enables memristors to be integrated into crossbar networks, thus crossbar array architectures have drawn much attention from researchers in nanoelectronics and NVM. The structure provides massive advantages such as pattern regularity, manufacturing flexibility, CMOS compatibility, defect-tolerance, and the highest device density [13], which may enable low-cost fabrication and products. Figure 1.9 shows the diagrammatic representation of a crossbar array structure with m wordlines (WLs) and n bitlines (BLs) [14]. As shown in the figure, R_j is the selected device for illustration, and devices that share a line with R_j (R_m, R_n) are named as “half-selected” devices. [14].

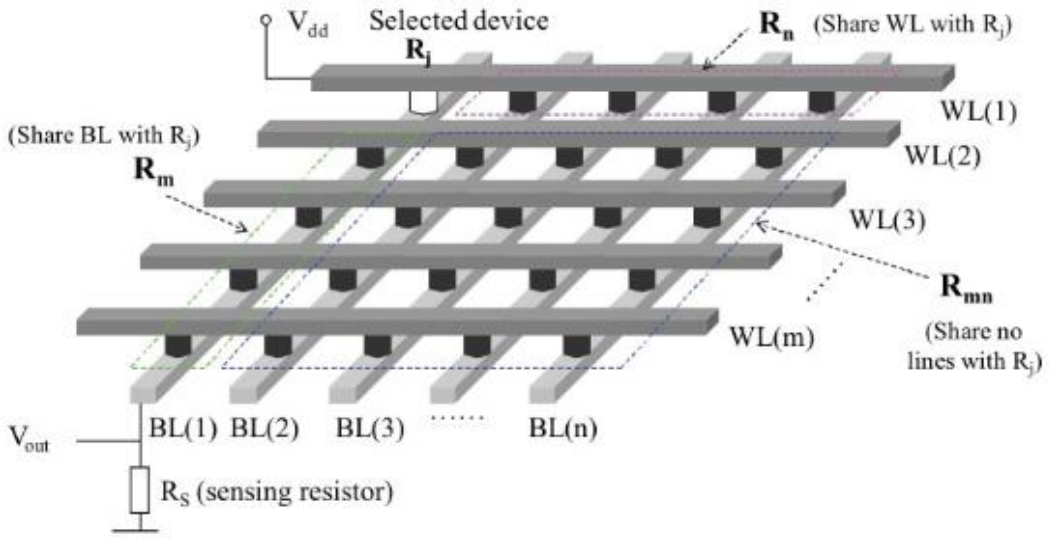


Figure 1.9: [14] Crossbar array with m WLs and n BLs. R_j at the upper left corner is selected. R_n, R_m, and R_{mn}: three groups of unselected devices sharing WL, BL, and no line with R_j, respectively.

We can depict the stable-state electrical characteristics of a memristive crossbar array by a set of voltage variables, which are the voltages at every cross-point cell. Basically, there are two voltages for each cell, V_{WL} and V_{BL}, and we can derive other electrical parameters (applied voltages, access resistance, and line resistance) from the set of these 2mn voltage variables. More detailed reviews of the crossbar array structure are described in [14].

1.3.3 Memristive IFG Model

The IFG model we used in our MNN is on the basis of an electrochemical neuromorphic organic device (ENODe), which is a memristive device with three terminals. It uses ionic currents to control the oxidation state of a semiconducting

polymer channel [15-17], and its high-density of ionic doping sites in the conductive polymer enables continuous analog conductance tuning [19]. The low switching energy and numerous resistance states make ENODes a promising candidate for low-power neuromorphic computing [19]. As shown in Figure 1.10 (a), an ENODe device contains an electrochemically active gate electrode, which is applied to actuate ion exchange between an electrolyte and a doped semiconducting polymer channel [20]. The gate voltage is responsible for modulating the channel conductance (G_{ENODe}) by controlling the electronic carrier concentration, and the low energetic barrier for ion migration between the electrolyte and the channel contributes to low minimum programming energies (e.g., $390pJ.mm^2$) [19]. The corresponding equivalent circuit is shown in Figure 1.10(b), in which R_{limit} is a limit resistor aiming at preventing discharge between programming pulses, R_{el} is the electrolyte resistance, C_{ENODe} denotes the mutual capacitance between gate and channel, and R_{ct} represents the equivalent resistance describing faradaic current from unexpected redox reactions at the channel/electrolyte interface [20]. The value of R_{limit} is ideally to be set high enough to impede self-discharge of the device; however, a high R_{limit} value will also result in a low writing efficiency, which is passively impacting the computation of ANN after training. To solve this problem, we can replace R_{limit} by nonlinear selectors as shown in Figure 1.10 (c), which serves as an electronic switch using voltage threshold filament formation

between two metal electrodes to switch between OFF state (no filaments) and ON state (filament forming a conductive bridge between electrodes) [18] like shown in Figure 1.10 (d). The fabrication and analysis of the equivalent circuit are described in detail in [20].

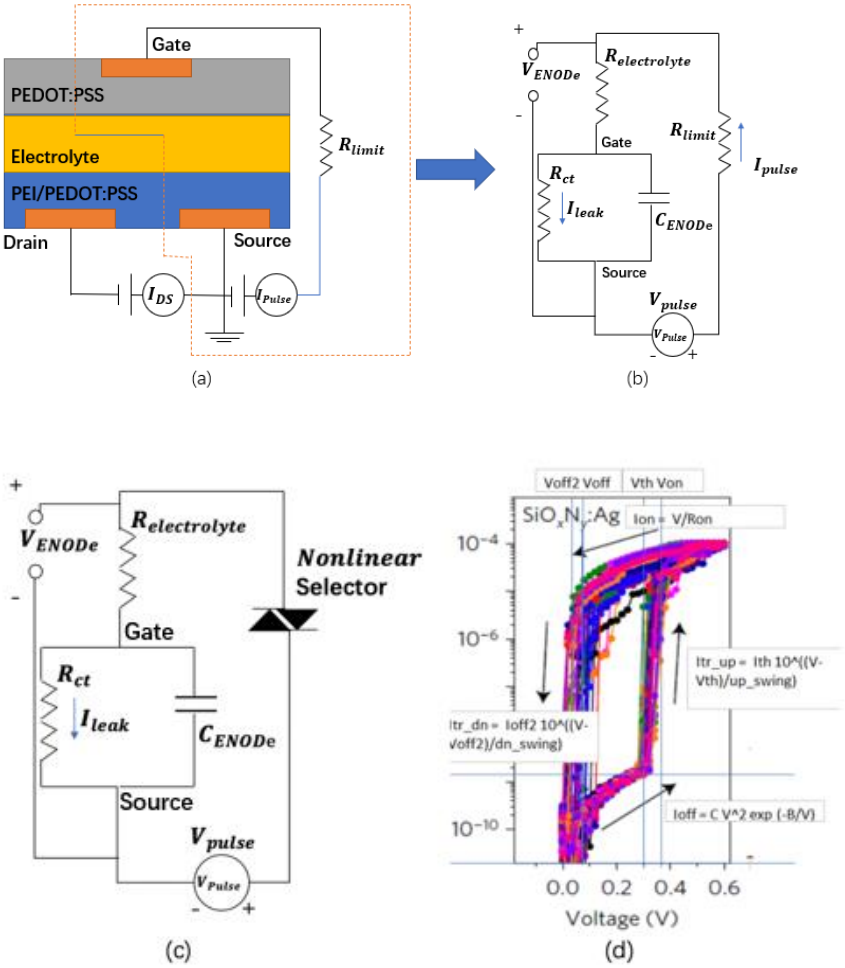


Figure 1.10: ENODe circuit model: (a) ENODe measurement setup[20]. (b) ENODe equivalent circuit model [20]. (c) ENODe circuit with a nonlinear selector [20]. (d) I-V characteristic of a nonlinear selector

A redox-transistor memory was developed to circumvent existing memristor technology limitations. The write and read operations can be decoupled by a three-terminal redox transistor, which uses a "gate" electrode to tune the conductance state through electrochemical reactions involving Li^+ and H^+ ion injection into the channel electrode through a solid electrolyte [18]. Based on the above theories, a polymer-based redox transistor [21] integrated with a volatile conductive bridge memory (CBM) was produced and named as ionic floating-gate memory (IFG), which is a nonvolatile, addressable synaptic memory. Its three-terminal design enables the channel to be engineered for ultralow-current without sacrificing analog performance through diluting the conductive polymer in a polymeric insulator. [18]. Figure 1.11 illustrates the IFG device concept. The weight of connections between artificial neurons of two layers can be mapped into the source-drain conductance of the corresponding synaptic device (transistor). More details about IFG characteristics are described in [18].

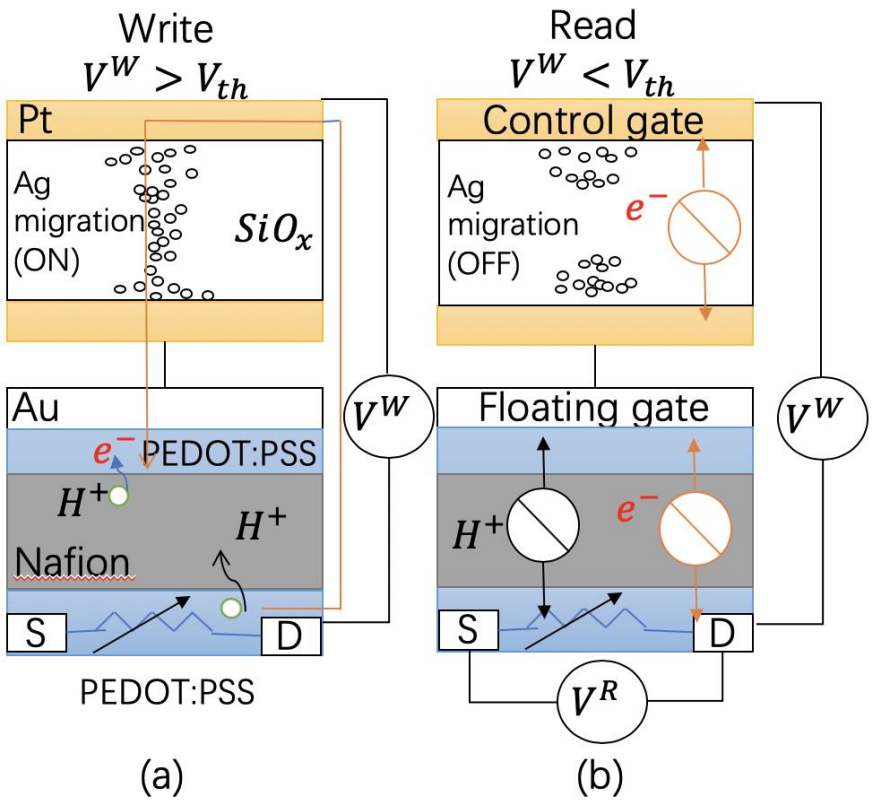


Figure 1.11: IFG memory device. (a) Write operation of an IFG cell. (b) Read operation of an IFG cell

Inspired by the two models presented above, we made some modifications and designed a compact IFG unit named IFG_r6 that described by Verilog, then we import the Verilog file to Cadence Virtuoso and create the IFG_r6 unit[23], the equivalent circuit, and the model created on Cadence are shown in Figure 1.12. we changed the position of the selector to the middle of the gate terminal and the electrolyte [23]. In this case, the conductance change is proportional to the flux of gate voltage whose absolute value is larger than the threshold voltage with the selector, and we can adjust the threshold gate to source voltage. We applied the

IFG_r6 cell to crossbar array structure and encoded a weight (conductance) update by voltages applied along the rows and columns of the array, which enables us to execute parallel writes to IFG memory during the network training.

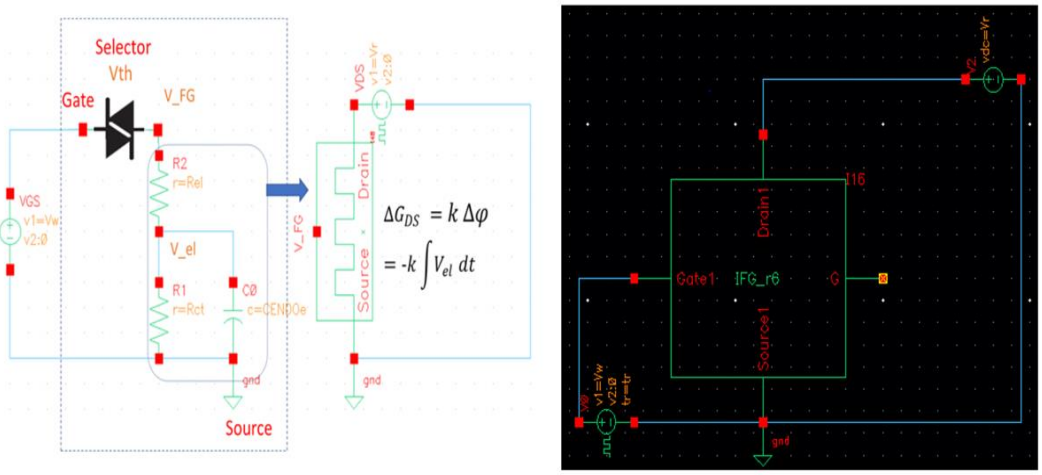


Figure 1.12: IFG model. (a) equivalent circuit (b) Cadence Virtuoso model.

Chapter 2
Memristor-based Neural Network (MNN) Design

2.1 Data Collection and Processing

2.1.1 MNIST Database

Data collection and processing is the first step of neural network design since we need massive data to train the neural network to set the proper weights. And we also need some test data to assess the performance. In this thesis, we plan to build two MNNs with different sizes and apply them to recognize the handwritten digits, then compare the performances of the two networks. Thus, we need to collect massive handwritten digit images to train and test the neural network. For specific numbers (0~9), there should be different images with different writing styles, which is time-consuming and cumbersome. Fortunately, there is an existing database called MNIST, which consists of a training dataset with 60,000 images and a testing dataset with 10,000 images. Every image in the MNIST dataset is an anti-aliasing grey-scale map of digital number from 0 to 9 and normalized to 28×28 pixels. These images are taken two random-selected groups; one is the Census Bureau employees, the other is high-school students. In brief, the MNIST data set is a huge set of handwritten digits with different writing styles, which perfectly meets our requirements.

However, the original format of the MNIST database is difficult to work with, and it is inconvenient for humans to view in text editors and also difficult to work with by computers. Some professionals have created MNIST data files of a simple format called CSV files, which is easy to view and process with some programming language like Python. Figure 2.1 shows a section of the MNIST test csv file loaded into a text editor.

7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8
4,185,159,151,60,36,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,222,254,254,254,254,241,198,19
8,198,198,198,198,198,198,170,52,0,0,0,0,0,0,0,0,0,0,0,0,0,0,67,114,72,114,163,227,254,225,254,
254,254,250,229,254,254,140,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,17,66,14,67,67,67,59,21,236,254
,106,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,83,253,209,18,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,22,233,255,83,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,129,254,238,44,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,59,249,254,62,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,133,254,187,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,9,205,248,58,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,126,254,182,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,75,251,240,57
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,19,221,254,166,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,3,203,254,219,35,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,38,254,254,77,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,31,224,254,115,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,13
3,254,254,52,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,61,242,254,254,52,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,121,254,254,219,40,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,121,25
4,207,18,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,116,125,171,
255,255,150,93,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,169,253,253,253,253,253,253,218,30,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,169,253,253,253,213,142,176,253,253,122,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,52,250,253,210,32,12,0,6,206,253,140,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,77,251,210,25
,0,0,0,122,248,253,65,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,31,18,0,0,0,0,209,253,253,65,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,117,247,253,198,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
76,247,253,231,63,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,128,253,253,144,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,176,246,253,159,12,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,25,23
4,253,233,35,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,198,253,253,141,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,78,248,253,189,12,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,19,200,253,
253,141,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,134,253,253,173,12,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,248,253,253,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,248,253,253,
43,20,20,20,20,5,0,5,20,20,37,150,150,150,147,10,0,0,0,0,0,0,0,0,0,248,253,253,253,253,253,2
53,253,168,143,166,253,253,253,253,253,253,253,123,0,0,0,0,0,0,0,0,0,174,253,253,253,253,2
53,253,253,253,253,253,253,249,247,247,169,117,117,57,0,0,0,0,0,0,0,0,0,0,118,123,123,123,
166,253,253,253,155,123,123,41,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0

Figure 2.1: Section of MNIST test csv file.

As shown above, it looks like a messy code, but actually, everything is fine. For the data list of each image, it is too long to be shown as merely one line on the window of the text editor, so it is wrapped around several times. Helpfully, there is some space between two image data lists, and we can see two whole lines of image

data in Figure 2.1. For each data list, the first value is the label, which is the exact digit that the handwriting is supposed to represent. The subsequent comma-separated values are the pixel values of the digit. Since the pixels of the image is 28×28, there are 784 values after the label. For example, the second data list, the first record represents number 2, as shown by the first value; thus, the rest 784 values are the pixel values of someone's handwritten digit 2. That means we can randomly pick a line from the MNIST data file and get the label for the image data by looking at the first number of the line.

2.1.2 MNIST Database Resizing

Additionally, we exercised another interesting hypothesis that how the performance would change if we trained and tested an MNN with handwritten digit images with smaller pixel size. Thus, we needed another train database and test database handwritten digit images of lower pixels. To make a great comparison, we can collect 60,000 handwritten digit images database for training and that of 10,000 for testing, but it would take a huge amount of time to collect all these by ourselves. Instead, we directly resized the original MNIST database to our desired pixel size so that we can use the same images to train a new MNN, and then analyze the performance of both two networks, which makes the simulation result more convincing and reasonable.

We can simply process the CSV file by Pandas, which is a Python data analysis library. It is a powerful and flexible data analysis tool based on Numpy. We can use Pandas to read the MNIST database CSV files row by row. Once we read a row of data, we first convert the data list to an array and separate the label value and pixel values. We resize the pixel values by an algorithm shown in Figure 2.2 and combine the processed pixel data with the original label value. Finally, save the combined data array to a 2D array. It is easy for us to repeat the operation by applying a "for" loop until we transfer all rows of data in one CSV file. All the processed image data rows are saved line by line in the 2D array mentioned, and we can easily save the 2D array as a new CSV file, which is the new converted MNIST database available for small pixel size experiment.

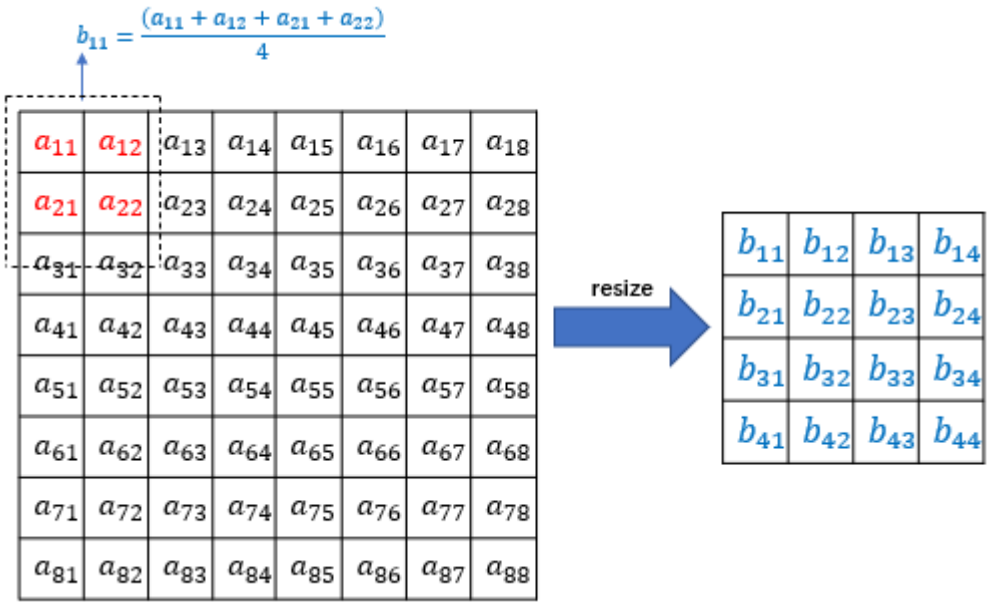


Figure 2.2: The diagram of image data pixel converting algorithm

We finally decided to convert the original pixel size (28×28) to 7×7, and we successfully got both of the two new training CSV file and testing CSV file. Figure 2.3 shows a few lines of the converted data. We can see that the data is displayed in the same way as the original database. The length of data lines changes from 785 to 50, and pixel values are also different. Now, we can easily pick a line of image data from either the original MNIST database or our converted one, and the first number will tell us the correct value. But it is still hard to see how the subsequent pixel values make up an image of someone's handwritten number. We can easily use Python to plot the image by these pixel values, and Figure 2.4 shows the image of the number 7 in the first line in both two databases. The method of visualizing the image data will be introduced in the latter part.

```
7.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,16.8125,25.375,0.0,0.0,0.0,0.0,0.0,41.0625,99.75,124.
5625,156.125,75.1875,0.0,0.0,0.0,0.0,0.0,0.0,145.875,7.9375,0.0,0.0,0.0,0.0,0.0,59.6875,88.125,0.
0,0.0,0.0,0.0,0.0,10.25,153.5,2.1875,0.0,0.0,0.0,0.0,0.0,65.8125,81.125,0.0,0.0,0.0,0.0
2.0,0.0,0.0,0.0,15.0625,51.9375,5.8125,0.0,0.0,0.0,0.0,8.0625,177.125,149.375,83.375,0.0,0.0,0.0,0.0,
0.0,3.0625,166.5,36.8125,0.0,0.0,0.0,0.0,0.0,91.5625,111.625,0.0,0.0,0.0,0.0,0.0,0.0,194.625,14.5
625,1.875,22.3125,19.1875,0.0,0.0,0.0,144.0,176.1875,149.75,122.8125,57.5,0.0,0.0,0.0,0.0,0.0,0.0
,0.0,0.0,0.0
1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,2.8125,99.5,0.0,0.0,0.0,0.0,0.0,0.0,57.5625,60.12
5,0.0,0.0,0.0,0.0,0.0,0.0,121.4375,5.3125,0.0,0.0,0.0,0.0,0.0,0.0,131.1875,0.0,0.0,0.0,0.0,0.0,0.0,39
.375,99.625,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,20.0625,210.375,36.5625,0.0,0.0,0.0,0.0,2.0,185.5,193
.25,184.875,29.125,0.0,0.0,15.6875,167.75,1.25,74.0625,143.0625,0.0,0.0,0.0,54.3125,134.875,7
3.1875,215.5625,60.9375,0.0,0.0,0.0,9.0,184.0625,224.0625,93.8125,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
,0.0,0.0,0.0
4.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,66.0,0.0,0.0,19.625,26.125,0.0,0.0,0.0,5.25,117.875,0.0,0.0,6
0.8125,63.375,0.0,0.0,0.0,56.0,62.5625,0.0,145.875,4.125,0.0,0.0,0.0,13.125,109.8125,99.875,181.8
75,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,154.6875,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,15.3125,0.0,0.0,0.0
1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,6.25,116.1875,0.0,0.0,0.0,0.0,0.0,0.0,84.6875,113
.4375,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,164.3125,24.8125,0.0,0.0,0.0,0.0,0.0,0.0,7.125,175.25,0.0,0.0,0.0,0.0,0.0,0
.0,41.0625,114.6875,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.625,16.5,0.0,0.0,0.0,0.0
4.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.25,100.5,2.0,0.0,0.0,68.875,0.0,0.0,0.0,67.375,79.625,0.0,0.0,8
1.5625,59.0,0.0,0.0,0.0,44.4375,149.375,115.3125,173.75,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,86.625,51.75,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,158.625,58.4375,0.0,0.0,0.0,0.0,0.0,0.0,0.0,23.6875,1.8125,0.0,0.0,0.0
```

Figure 2.3: section of the converted MNIST test csv file

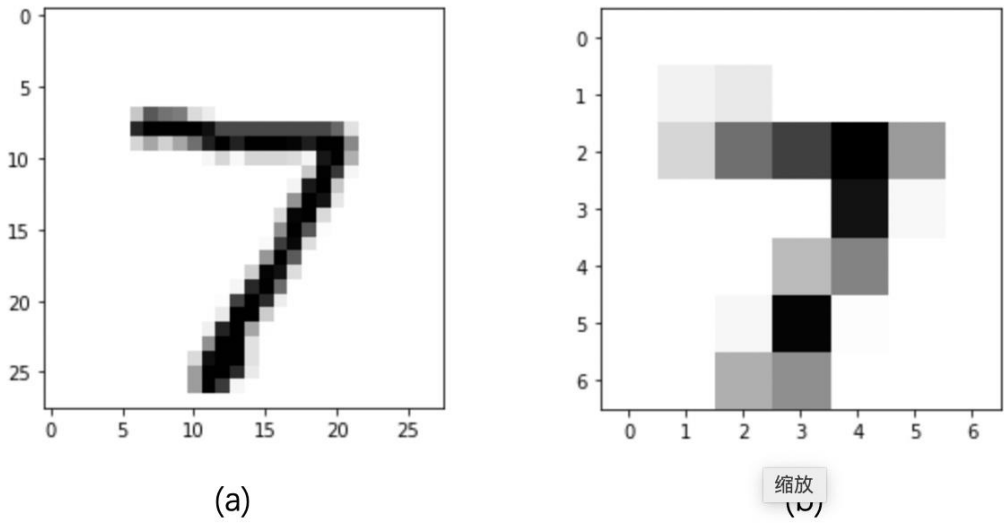


Figure 2.4: Image of number 7 in the first line of two MNIST test csv files. (a) Original MNIST database. (b) Converted MNIST database.

2.2 MNN Architecture Design

Now, we can start to design our MNNs; at first, we need to define the architecture of our MNNs. As mentioned in Chapter 1, a neural network consists of a series of layers of neurons, and all the neurons in each layer connect to neurons of the next layer. Basically, an ANN should at least have two layers, one input layer, and one output layer, but in this case, the network works just like linear regression, since the input nodes merely bring in the input signals applied to the input layer and the output nodes only push out the answer of the ANN. This will restrict the learning ability of the network. Therefore, to make our MNN more intelligent, we need to add hidden layers to our network.

The hidden layer is quite common in ANNs; it is located between the input and output layers as mentioned in Chapter 1. It multiplies weights to the inputs and produces outputs through a weighted summing and an activation function, which gives nonlinear characteristics to the network. Briefly speaking, hidden layers are layers designed to produce outputs specific to serve as input to the output layer aiming at getting intended results. Normally, one hidden layer is sufficient for ANNs to solve a majority of problems, and we need our MNNs to recognize handwritten digits, which is not a very complicated task. Thus we merely need to add one hidden layer to our MNNs, thus, the MNNs for our research are designed with three layers.

Now, we should determine the number of nodes in each layer. To set up MNNs and enable them to recognize MNIST handwritten images, the pixel values of each image should be inputs for input layers, and the ten labels should be the outputs. Since the original MNIST images contain 784 pixels and resized MNIST images contain 49 pixels, we should set one MNN with 784 input nodes, and the other one with 49 input nodes. Since we would not like to increase the amount of work on IFG circuits simulation, we at first set the number of hidden nodes of both two networks as 10, if the accuracy is unsatisfactory, we would add more hidden nodes to the hidden layer and perform simulation again. For the number of output nodes, recall that we are asking our MNNs to classify the MNIST handwritten

image and response with correct label, which is one of ten numbers from 0 to 9. Therefore, both of the two MNNs have an output layer of 10 nodes. The architecture of the two networks we designed are shown in Figure 2.5 and Figure 2.6. Both of the two networks are fully connected and feedforward network with backpropagation to auto-recognize and train.

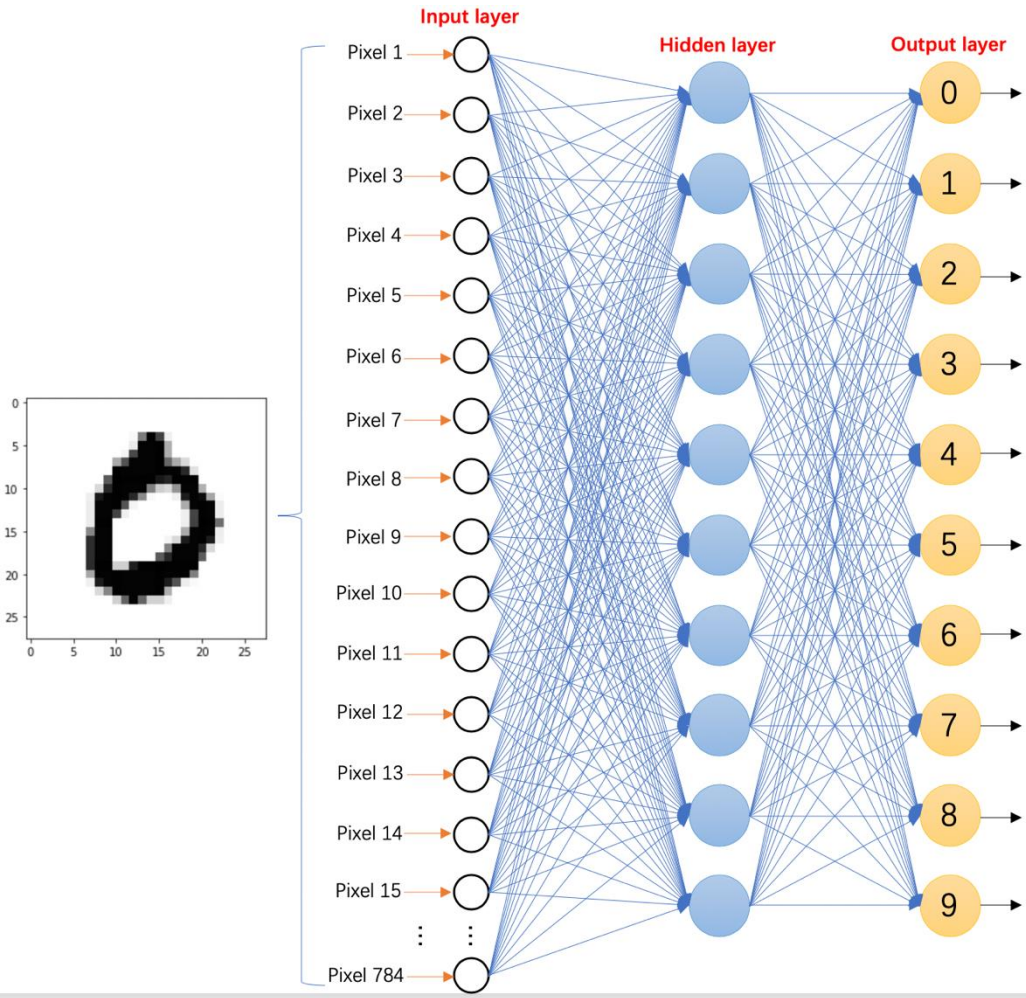


Figure 2.5: MNN architecture for original MNIST database. The pixels (784) of each image are the inputs for the network and the output nodes are ten correct labels.

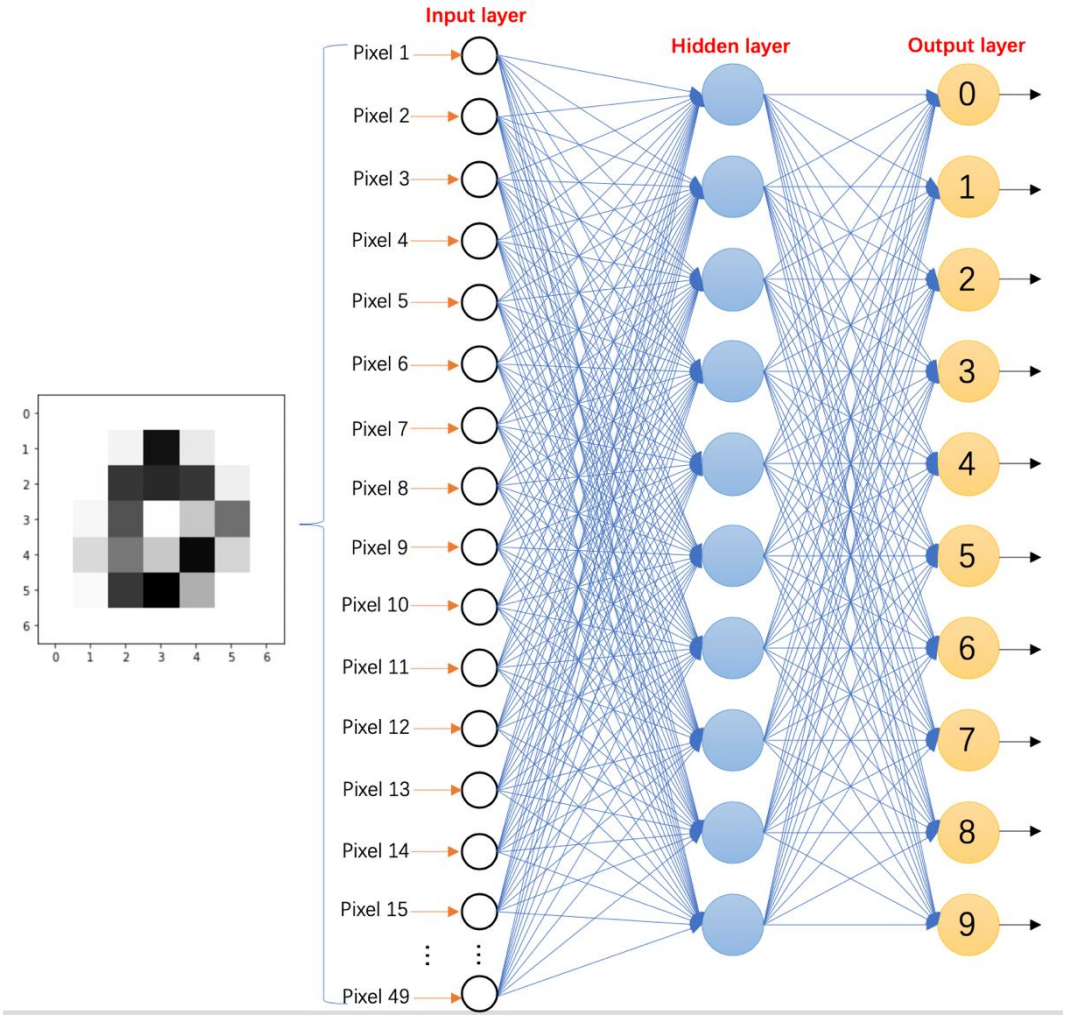


Figure 2.6: MNN architecture for resized MNIST database. The pixels (49) of each image are the inputs for this network and the output nodes are ten correct labels.

Now we need to add an appropriate activation function to hidden and output layers to introduce nonlinearity characteristics into the two networks and elevate them beyond the capabilities of a simple perceptron. The common activation functions have been introduced in Chapter 1. The sigmoid function is chosen for our networks, since it requires less calculation and is already defined in the scipy Python library, in which the sigmoid function is called expit(). It is now quite

convenient for us to quote the sigmoid function during programing the networks by Python.

2.3 Training

2.3.1 Data Preprocessing

We have all the data for training and testing, but we still need to think about preprocessing the data before we apply it to our MNN. When we apply the data to the network, the pixel values of each handwritten digital image will be treated as inputs to the input layer. As we can see in Figure 2.1 and Figure 2.3, the pixel values are in the range of 0 to 255. The large input value would make the sigmoid function quite flat, which is problematic when we perform gradient descent during the training process. The flat portion of the sigmoid function has tiny gradients, as shown in Figure 2.7. Tiny gradient results in limited learning ability [5]. This is referred as saturating a neural network. Therefore, we need to rescale the input pixel values to a smaller range 0.01-1.0, and we chose 0.01 as the lower bound to avoid zero-valued inputs since it can annihilate the weight updating [5]. Besides, the sigmoid function is not able to produce a value above 1. Figure 2.7 illustrates the sigmoid function, which indicates that the sigmoid function is asymptotically approaching 1 [5]. We should restrict the target values to match the range of the activation function, since if the target values are set in the outside range of

activation function, the network will also be saturated in this case. We chose the range of 0.01 to 0.99 for our MNNs.

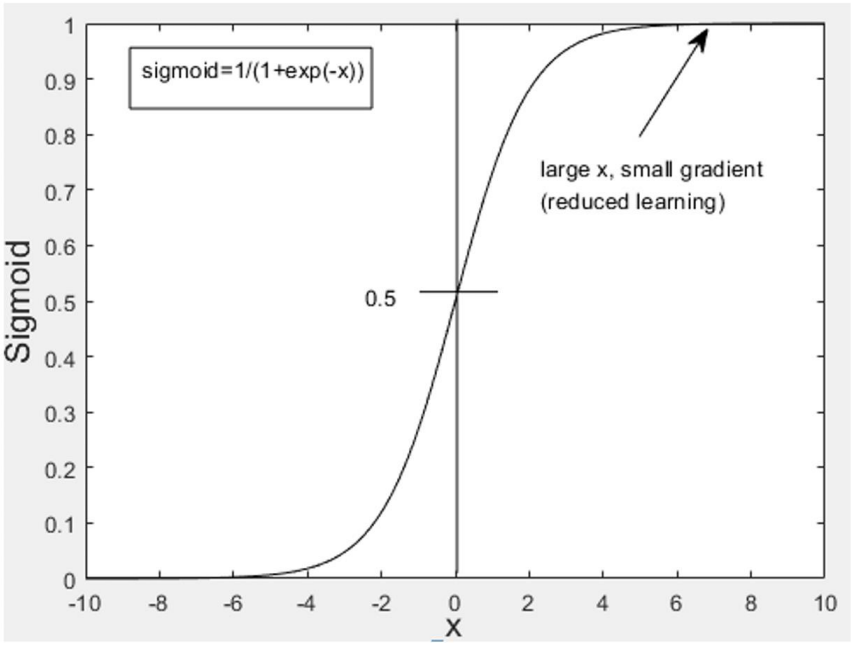


Figure 2.7: The graph of sigmoid function.

The same argument applies when we initialize the weights, we should avoid large initial weight values since they would result in large inputs to the sigmoid function, leading to the saturation problem [6]. Besides, we should also avoid initializing the weights as the same constant value, especially not zero, it is troublesome since all the nodes in the network would receive the same signal and all the outputs of these nodes would also be same, which would result in equal weight updates during training. We will finally get a set of updated weights with the same value. Normally, a properly trained ANN should have unequal weights. Zero weights are even worse, since the incoming signals would be entirely zero,

which will deactivate the learning ability of the ANN. There is a sophisticated rule that initializing the weights by randomly sampling from a range that is approximately the inverse of the square root of the number of links into a node [6]. By following this rule, we initialize the weights of the first MNN by randomly sampling from $-\frac{1}{\sqrt{784}}$ to $\frac{1}{\sqrt{784}}$, and respectively initialize the weights of second one by randomly sampling from $-\frac{1}{\sqrt{49}}$ to $\frac{1}{\sqrt{49}}$.

2.3.2 Training Programming

As mentioned in Chapter 1, the training process has two phases, the first one is propagating forward the training data and calculating the outputs, then compare the outputs with our target outputs to get the loss (error).The second phase is backpropagating the loss (error), which can guide the MNNs to update their weights. Therefore, we can draw the scheme of the training algorithm framework as Figure 2.8. It is now quite clear how to program by Python.

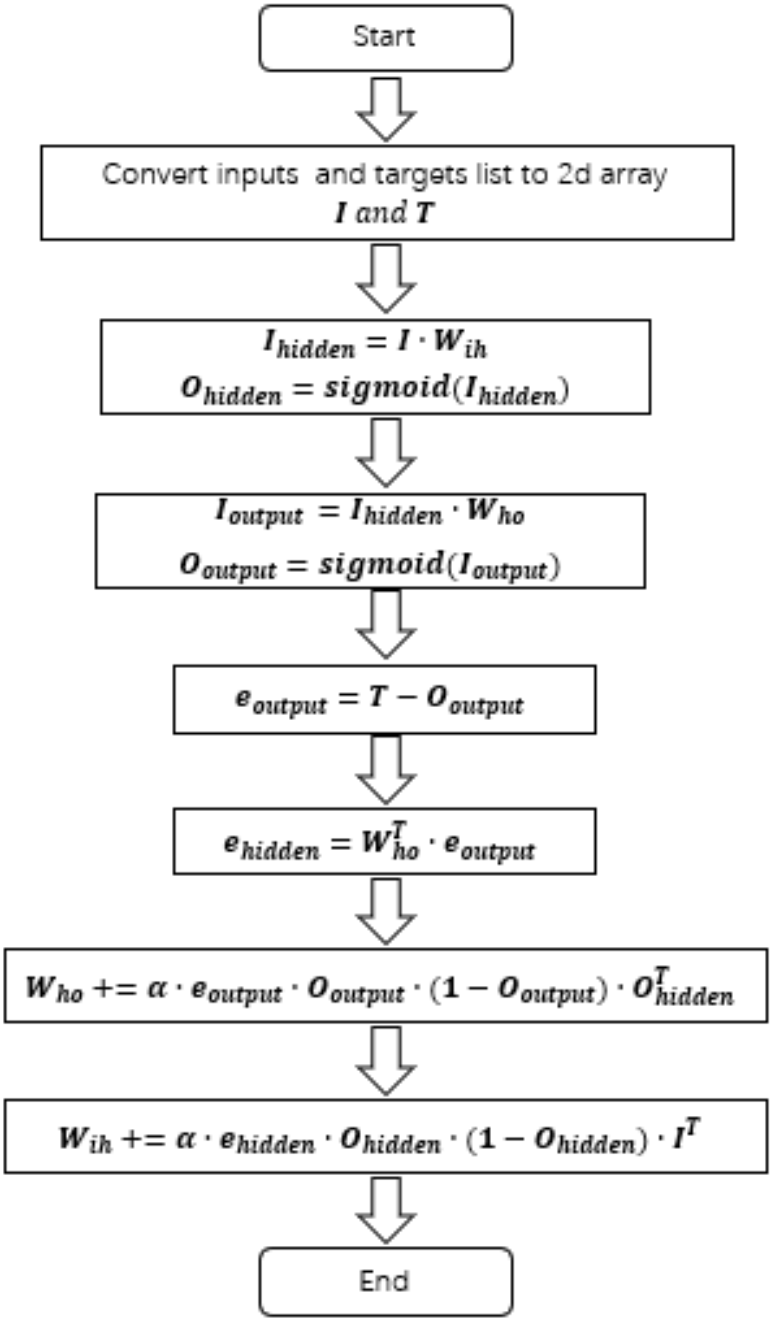


Figure 2.8: Framework of training program

2.4 Testing

Since we have trained the two MNNs and got proper weights, we can now test whether the two networks work well. Before programming the testing part of the MNNs, we need to figure out what the output is supposed to be because we need the correct outputs of all the 10 numbers to justify the answer of the MNNs. As mentioned earlier, both of the two MNNs have an output layer of ten nodes, one for each of numbers from 0 to 9. If we input an image data of number 6, the corresponding node in the output layer should fire while all other nodes are silent. Figure 2.9 illustrates this scheme and gives example outputs. This example shows that the network justifies that the image we gave is number 5, since the node labeled with 5 gives largest output signal. The second example in Figure 2.9 is more interesting, the node corresponding to number “9” has generated the largest output signal while the node labeled with number “4” has a moderately big output signal. In general, we always go with the largest output signal, but the moderately big signal indicates that the network wonders the answer might also be 4. This is not necessarily incorrect, sometimes a weird handwriting style makes it difficult to have a confirmed answer, this kind of doubt does happen with neural networks, even for those of human brain. We are supposed to treat it as a valuable insight into how another answer was also a contestant [6], which makes the neural network be closer to that of human brain.

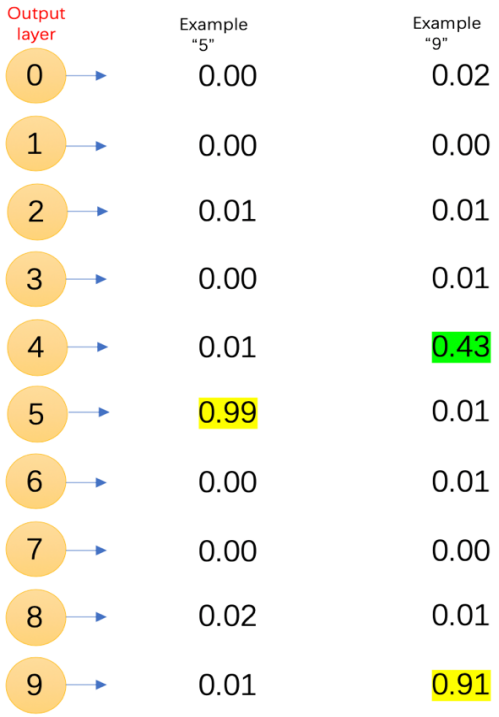


Figure 2.9: Scheme of how to get the answer from the output signal

2.4.1 MNIST Testing

Now, we can think about the testing work, in this case, the MNIST test dataset comes into play. We can easily write lines of Python code to get the test records from the MNIST test csv file, which is quite similar to that applied to acquire the training data. As presented, the test dataset is not a part of training dataset, which means the trained network hasn't seen these images. Therefore, the main mechanism of testing is going through all the 10,000 test records and check the correctness of the outputs. We can easily write a Python code to see the performance of the networks against the whole MNIST test set. Python has a quite helpful numpy function called "numpy.argmax()", which can easily pick up the

largest signal in the output signal array and return the corresponding position, which can be referred as the node generating the largest signal. Also, we can apply a scorecard, which is updated after running each record, to assess the accuracy. If the value returned by the argmax function is equal to the correct label, we append 1 to the scorecard, respectively, we append 0 to it if the network makes incorrect judgement. In this case, the accuracy is fraction of correct answers.

During the testing process, since we are able to directly see the performance (accuracy) of the networks, some parameters can be modified to improve the performance score. The first one is the learning rate, it was at first set as 0.5 when programing the MNNs by Python, and we are not able to check whether 0.5 is the optimum value and experiment with other values until we finish the testing part, the process of experimenting with different values of learning rate will be shown in the next chapter.

Another improvement we can do is to repeat the training several times against the training dataset [5]. In the Python code, we name each training run as an epoch. Thus, when we set the epochs as 5 in the training session, it means the training session of the program will run through the whole MNIST training dataset 5 times. And we can experiment with different values of epoch and compare the performance (accuracy), and it is easy for us to find the optimum one. But we need to realize that since the size of the MNIST training set is quite large, so if we set the

value of epochs too high, like 20 or 50, it will take up to 20 or 30 minutes for the computer to finish the training session. We should also take this issue into consideration. The experiment of finding the optimum epoch value will be described in Chapter 3.

2.4.2 Own Handwriting Image Testing

Now that we have figured out how to apply the MNIST test set to assess the performance of our MNNs, a remaining interesting question would be what if we test our trained MNNs with our own handwriting digits? If they can recognize most of these images, the test results will be more convincing.

In order to create the test dataset with our own handwriting digits, we can use the paint board on the computer or other image painting and editing software, we can even write digits on paper and take the photos of them by cell phone or camera. But the images are required to be square and be saved as PNG format, also, the pixels of these images should be rescaled to the size matching our MNNs (28×28 and 7×7). The key point of the testing work is how to transfer these image files to the data, which is similar to MNIST dataset. Python has a quite helpful library named “scipy.mimc” that can effectively read out and decode the data from image files such as PNG or JPG files. Therefore, we can easily write Python code

to perform our own handwriting digits test. The scheme of the testing work can be seen in figure 2.10.

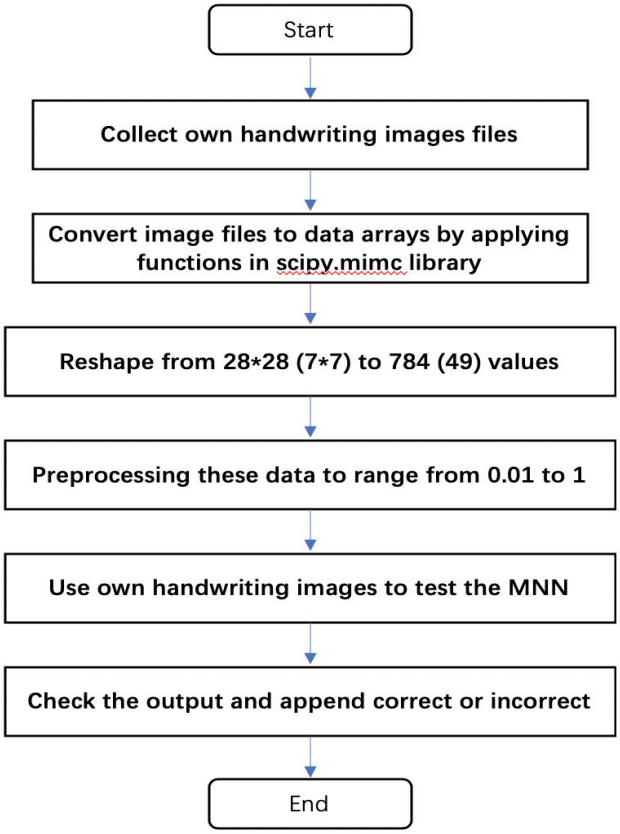


Figure 2.10: Scheme of own handwriting digits testing program

Chapter 3

Results and Discussion

3.1 MNN with 784 Input Nodes

3.1.1 MNIST Test Dataset

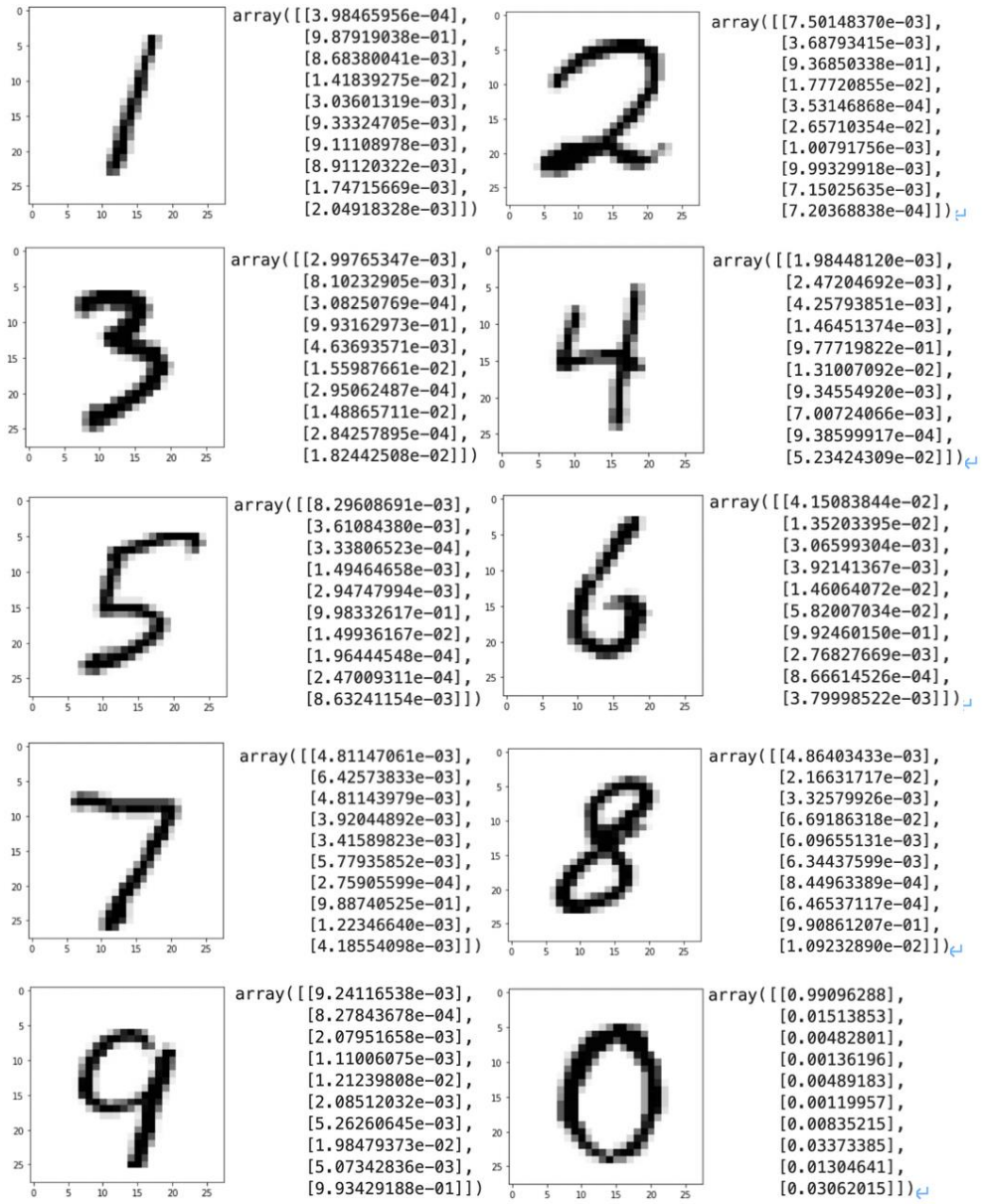


Figure 3.1: Some ideal cases of MNIST test results

Figure 3.1 shows the result of ideal cases of all the 10 digits in MNIST test dataset. As we can see the output signal of the node corresponding to correct label is quite close to 1, while that of other nodes are close to 0. This means the MNN can recognize the label in these images clearly. There are some correct cases, shown in Figure 3.2, causing some doubt because of handwriting. As mentioned above, there is another moderately big output signal from one node other than the correct one, which means the network think the label of the node generating relatively big output signal can also be the answer. The general character of these cases is that the handwriting is not neat. Some weird handwriting makes the digit ambiguous, like the 9 in the second image and the 4 in the third image, they look quite similar, it is reasonable for the MNN to feel doubtful, even for me, I still cannot make undoubtful judgements under this circumstance.

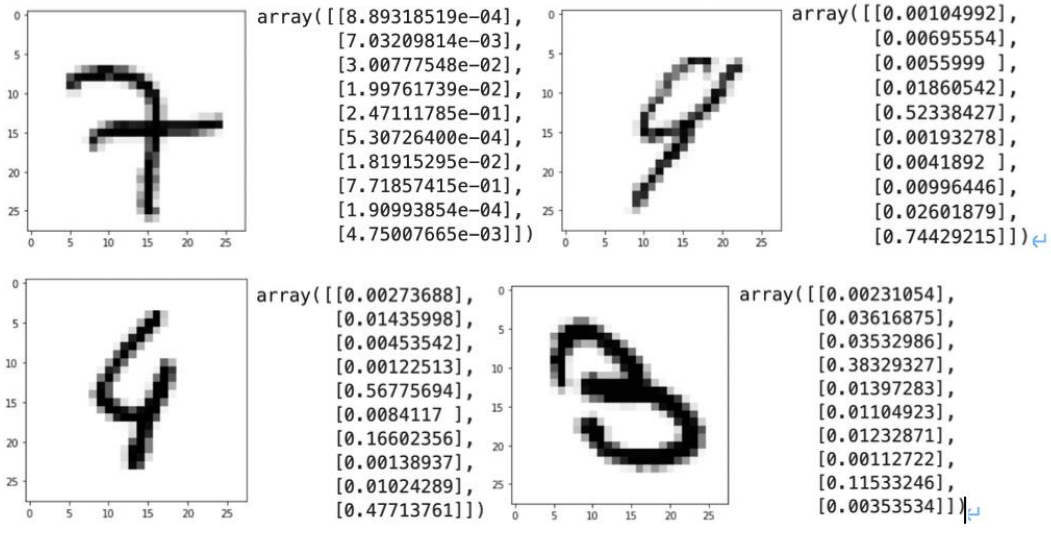


Figure 3.2: Some special correct cases

Now that, untidy handwriting can make the MNN make doubtful judgement, it can also be illegible enough to prompt the MNN to make mistakes, the following Figure 3.3 shows some incorrect cases. The correct labels of these images are respectively supposed to be 5, 5, 7, and 2. As we can see, if not being told in advance, we cannot correctly recognize them. Therefore, it is understandable for the MNN to make wrong judgements in this case.

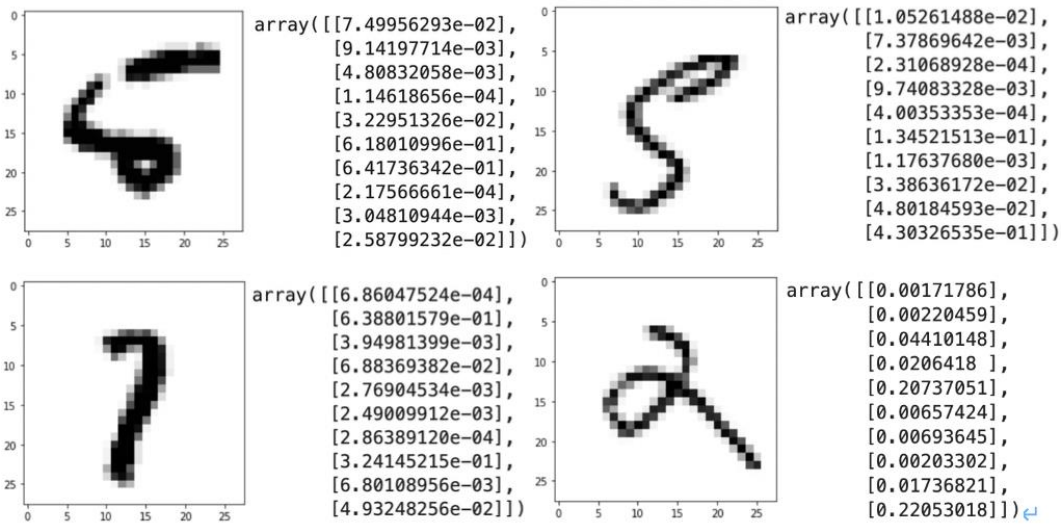


Figure 3.3 Some incorrect cases.

As mentioned in Chapter 2, we can enable the Python code to go through all the test record and calculate the performance (accuracy), which is referred as the fraction of correct answers. In this phase, we can modify two parameters to improve the performance, learning rate and epochs. To experiment with different values of learning rate, we are supposed to keep the value of epochs fixed. In our experiment on learning rate, we set the epochs as 5 and experiment with several values of

learning rate at the range from 0 to 1. We plot a graph of the results as shown in Figure 3.4. The plot suggested that the learning rate of 0.1 gives us the optimum performance.

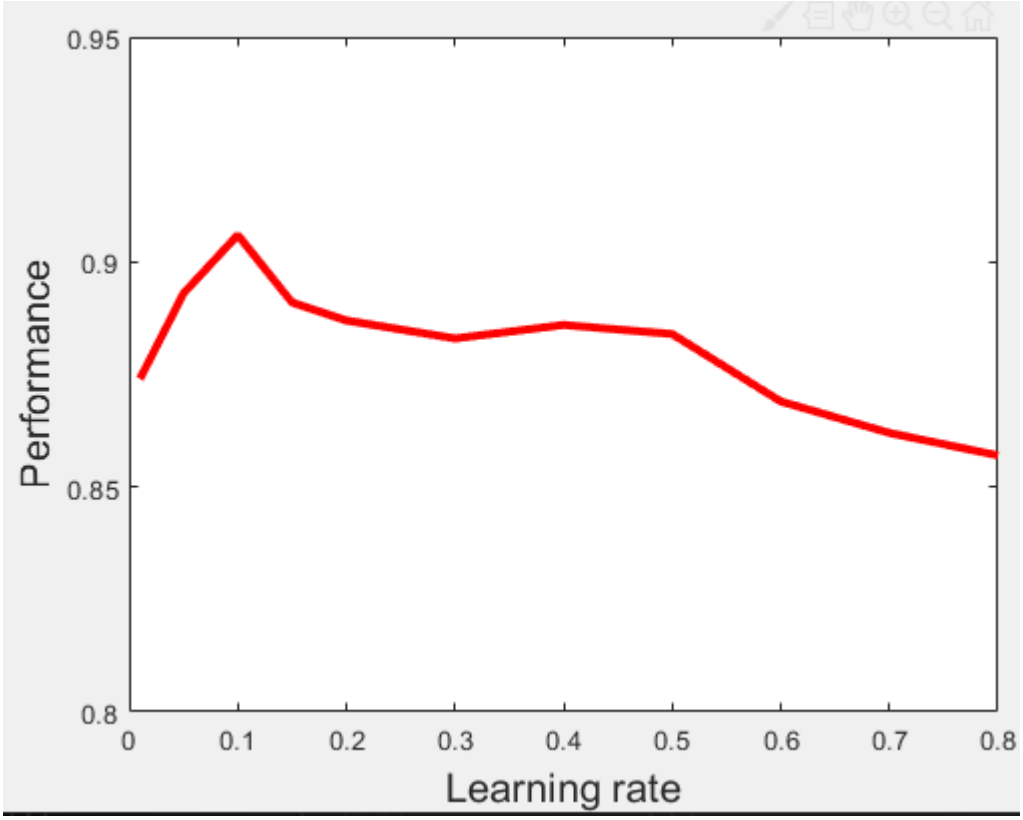


Figure 3.4: Learning rate vs. Performance (28×28)

Next, we move on to examine the impact of epochs on the performance. Now that we have found the optimum learning rate of 0.1, we merely need to keep it fixed and experiment with values of epochs from 0 to 10, then compare the performances. We also plot the relationship between the value of epochs and

performance in Figure 3.5. We can see the optimum performance occurs when the value of epoch is 5, and beyond that the performance degrades, which may attribute to overfitting. But the performance increased again after 8 epochs, and the performance is even quite close to the optimum one that corresponding to 5 epochs, although a new optimum performance might be found if we continue to increase the value of epochs, it would take a quite long time for the computer to finish running the code. Therefore, we still treat 5 as the epochs value of optimum performance. However, there is still one thing we need to keep in mind that, the approach we used to analyze the effect of learning rate and value of epochs is not convincingly scientific, since we failed to repeat the experiments for a large number of times to minimize the impact of randomness and bad journeys down the gradient descent [5]. It is still helpful to understand the general idea that there is an optimal values for learning rate and the number of epochs.

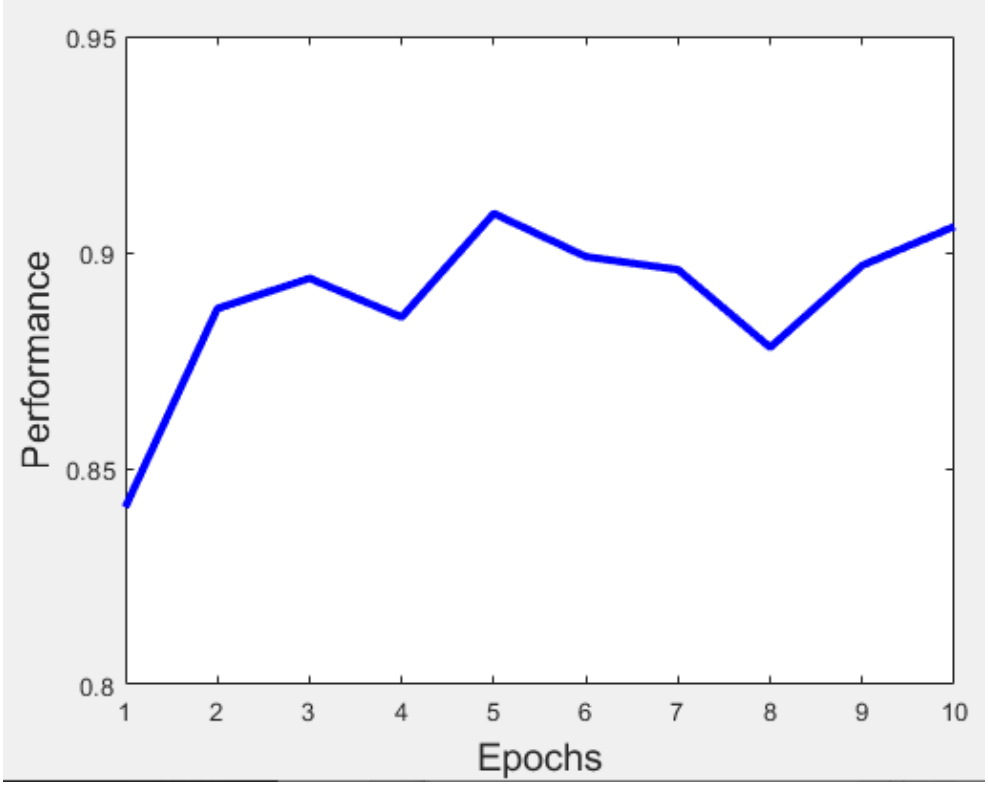


Figure 3.5 Epochs vs. Performance (28×28)

3.1.2 Own Handwriting Image

We collected our handwriting image of all 10 digits, and then performed the testing operation similar to that of MNIST test set. The results are shown below. It can be seen that our MNN can make correct judgement for most of images as shown in Figure 3.7, only the images of number 7 and number 1 result in incorrect result as shown in Figure 3.8. The MNN can make correct judgement for most cases. However, the two incorrect cases are unexpected, we still can't figure out the reason by now.

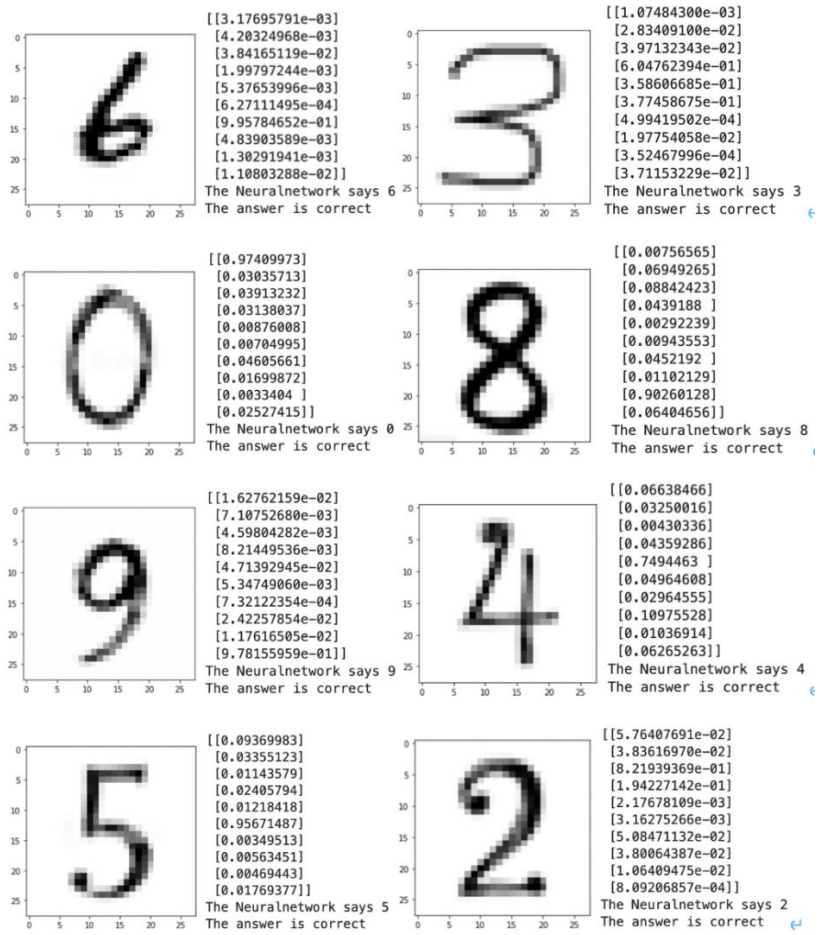


Figure 3.6 Correct cases of own handwriting images test.

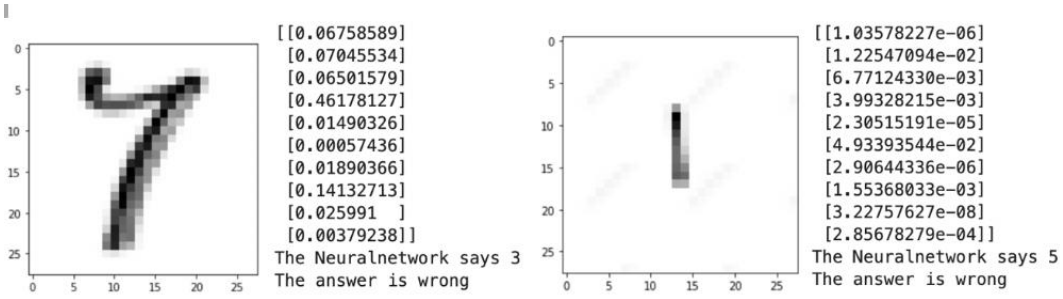


Figure 3.7: Incorrect cases of own handwriting images test.

3.2 MNN with 49 Input Nodes

3.2.1 MNIST Test Dataset

As mentioned in Chapter 2, our resizing algorithm is not carefully designed, and thus some of the resized MNIST handwriting digits are not quite clear. Also, the grayscale is not uniformly distributed, which makes the test result of even correct ones not quite ideal. Thus, we just displayed some correct cases in the Figure 3.9. It is easily seen that the output arrays are not as ideal as that of the MNN with 784 input nodes. The images of number 8 and 9 are too blurry to recognize the label in the image. The results might become better if we collected better image data or optimized our algorithm for resizing.

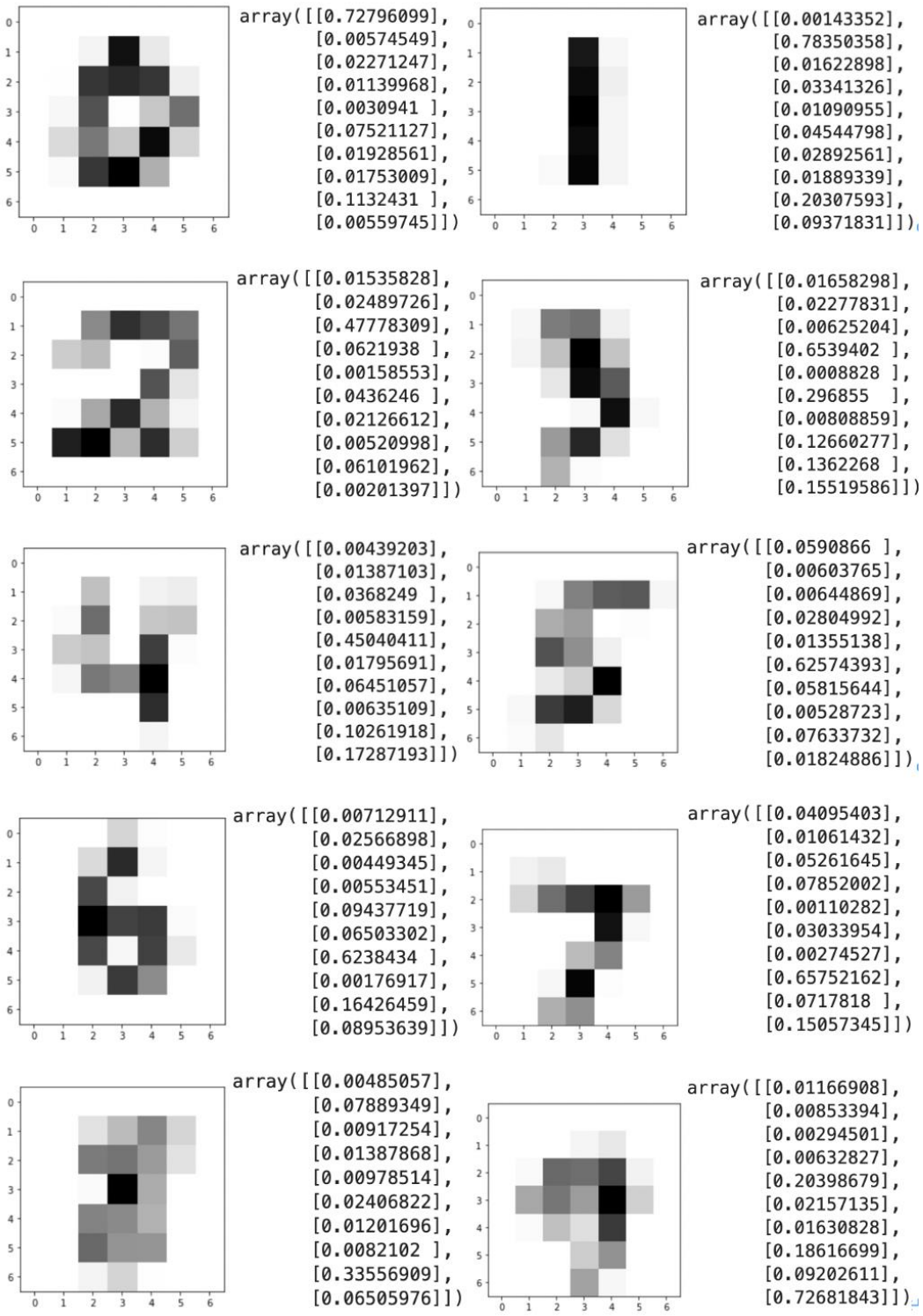


Figure 3.8 Some correct cases of resized MNIST test results

The images with weird handwriting would be worse due to the lower dimension pixel. We can see some cases in figure 3.10. What we can see is merely some strange label. Our MNN make wrong judgements for these images. This is reasonable because they are quite difficult even for human to recognize, not to mention the artificial one.

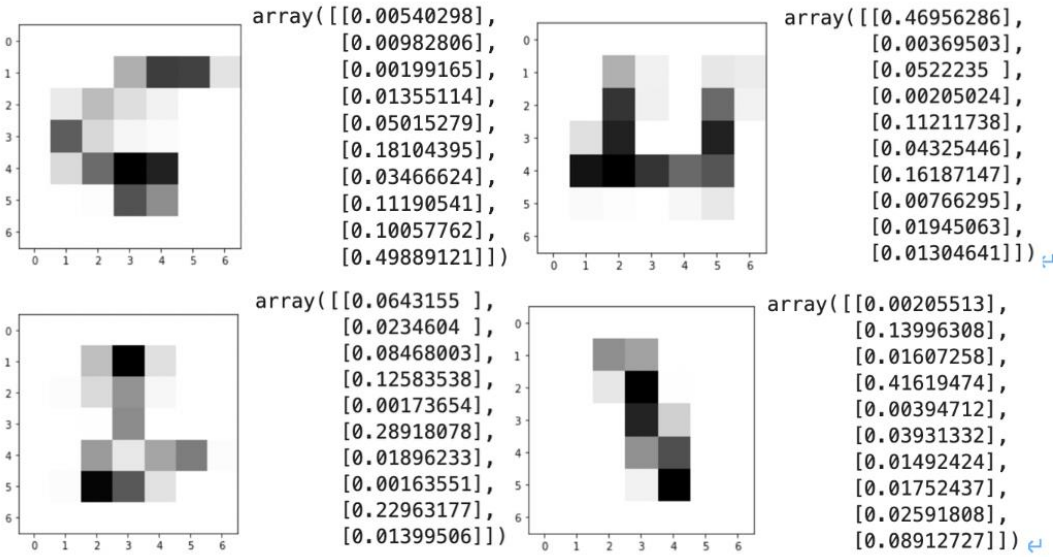


Figure 3.9: Some incorrect cases of resized MNIST test results

We did the same experiment on the impact of learning rate and epochs as that of MNN with 784 input nodes. At first, we still kept the value of epoch as 5 and experimented with the same values of learning rate. The corresponding performances of these values are shown in Figure 3.11.

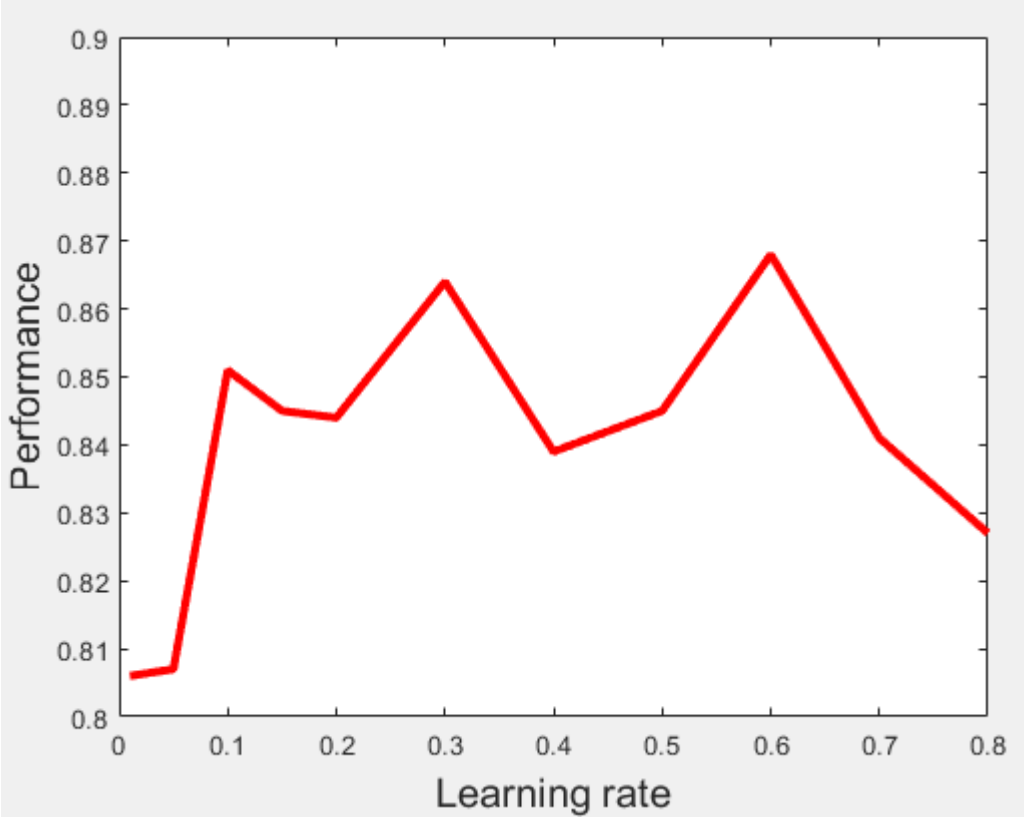


Figure 3.10: Learning rate vs. Performance (7×7)

In Figure 3.10, we can see that the optimum learning rate is 0.6, which is different from that of the former MNN. This could attribute to uneven pixel value of images of resized MNIST test set.

Also, we utilize the same experiment setting of epochs as we used in the former MNN, this time we kept the learning rate as 0.6, which is the optimum value we found. We can see the experimental consequence in Figure 3.11. We can see that the optimum epoch value is also different from that of the former network, in this case the optimum value of epochs is 9. Actually, the performance of MNN with 49 input nodes is better than we expected. Admittedly, when we resize the original

MNIST database, the algorithm presented above is not convincingly scientific, and the pixel values are not as neat as that of original ones, that is the reason why some images of digits are too blurry to recognize. In all these cases, we were pessimistic on evaluating the performance, but the actual consequences were beyond our expectation.

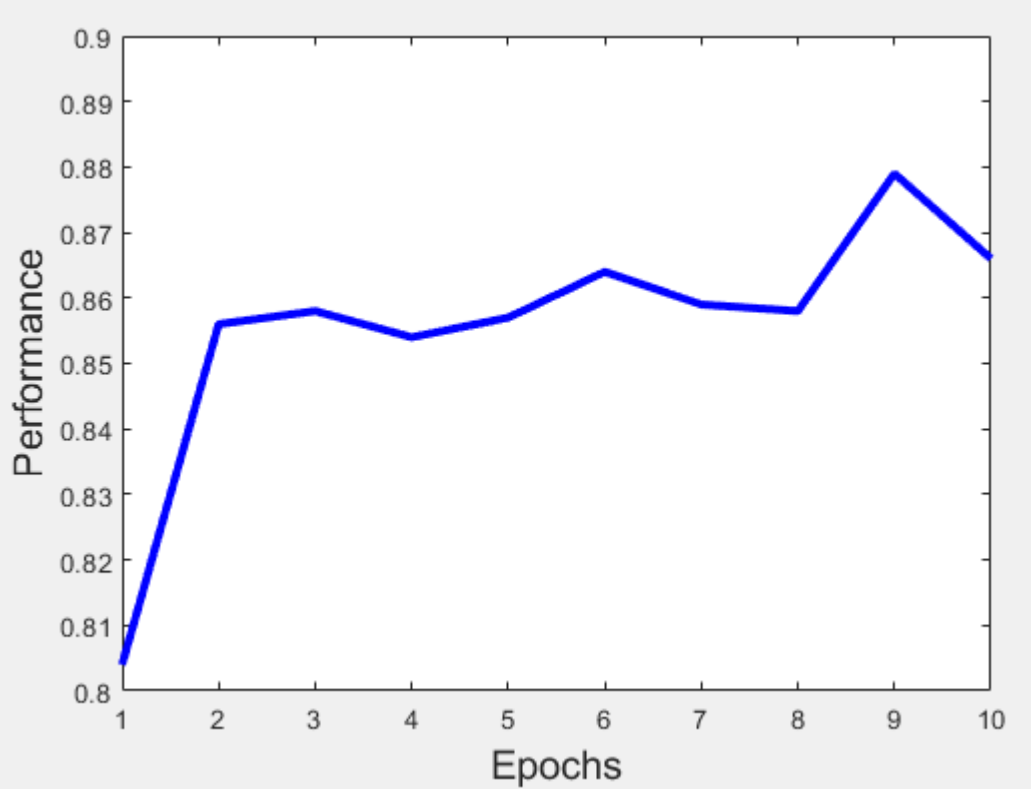


Figure 3.11: Epochs vs. Performance (7×7)

3.2.2 Own Handwriting Image

In this phase, we used some image editing software to resize the images we made to test the former work to the pixel size that is available to the MNN of 49 input nodes. But when putting them into the MNN, none of them gave correct

answers. And we tried many methods to do some optimization, but all of them were not effective.

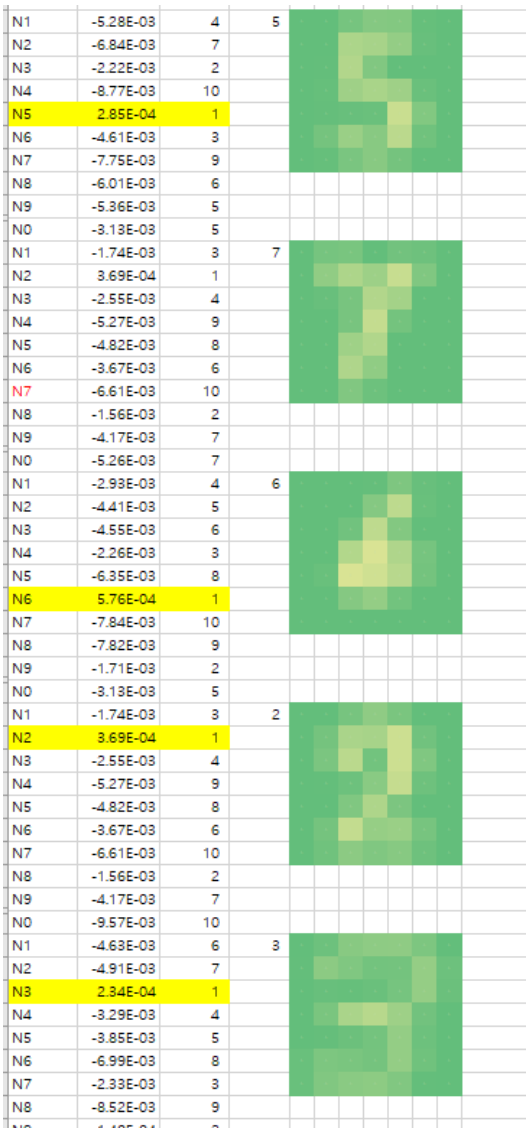
3.3 Hardware

The IFG networks on Cadence Virtuoso were designed and simulated by colleagues [22] [23]. The partial simulation result of MNN with 784 input nodes is shown in Figure 3.12. And the hardware experimental performance (accuracy) is 93.8% for 1800 test sets, which indicates that the IFG memory unit shows great feasibility for artificial neural networks.

input		rank	image
N0	-5.12E-03	8	
N1	-6.44E-03	9	
N2	-4.12E-03	4	
N3	-4.86E-03	6	
N4	-4.65E-03	5	
N5	-3.40E-03	2	
N6	-4.99E-03	7	
N7	5.14E-03	**1 7**	
N8	-6.50E-03	10	
N9	-3.47E-03	3	
input	1		
N0	-2.85E-03	2	
N1	-4.68E-03	5	
N2	4.99E-03	**1 2**	
N3	-7.79E-03	10	
N4	-6.30E-03	8	
N5	-4.66E-03	4	
N6	-6.08E-03	7	
N7	-5.41E-03	6	
N8	-3.85E-03	3	
N9	-6.74E-03	9	
input	2		
N0	-7.81E-03	8	
N1	4.22E-03	**1 1**	
N2	-4.42E-03	4	
N3	-5.43E-03	5	
N4	2.51E-03	2	
N5	-7.57E-03	7	
N6	-5.55E-03	6	
N7	1.93E-03	3	
N8	-8.79E-03	9	
N9	-1.26E-02	10	

Figure 3.12: Partial results of hardware simulation (28×28).

However, the Cadence simulation of the MNN with 49 input nodes shows somewhat poor consequences, the accuracy is merely 78.8% for 2000 test sets. Unlike the computer software simulation, the partial simulation result is shown in Figure 3.13. There are several factors that can influence the hardware simulation result. We have tried various methods to optimize the simulation, but none of them was effective. We need to investigate the causes in the future and seek to improve the hardware simulation of this MNN.



N1	-5.28E-03	4	5
N2	-6.84E-03	7	
N3	-2.22E-03	2	
N4	-8.77E-03	10	
N5	2.85E-04	1	
N6	-4.61E-03	3	
N7	-7.75E-03	9	
N8	-6.01E-03	6	
N9	-5.36E-03	5	
N0	-3.13E-03	5	
N1	-1.74E-03	3	7
N2	3.69E-04	1	
N3	-2.55E-03	4	
N4	-5.27E-03	9	
N5	-4.82E-03	8	
N6	-3.67E-03	6	
N7	-6.61E-03	10	
N8	-1.56E-03	2	
N9	-4.17E-03	7	
N0	-5.26E-03	7	
N1	-2.93E-03	4	6
N2	-4.41E-03	5	
N3	-4.55E-03	6	
N4	-2.26E-03	3	
N5	-6.35E-03	8	
N6	5.76E-04	1	
N7	-7.84E-03	10	
N8	-7.82E-03	9	
N9	-1.71E-03	2	
N0	-3.13E-03	5	
N1	-1.74E-03	3	2
N2	3.69E-04	1	
N3	-2.55E-03	4	
N4	-5.27E-03	9	
N5	-4.82E-03	8	
N6	-3.67E-03	6	
N7	-6.61E-03	10	
N8	-1.56E-03	2	
N9	-4.17E-03	7	
N0	-9.57E-03	10	
N1	-4.63E-03	6	3
N2	-4.91E-03	7	
N3	2.34E-04	1	
N4	-3.29E-03	4	
N5	-3.85E-03	5	
N6	-6.99E-03	8	
N7	-2.33E-03	3	
N8	-8.52E-03	9	

Figure 3.13: Partial results of hardware simulation (7×7).

Chapter 4
Conclusion and Future Work

The purpose of this thesis project was to examine how well the Memristive IFG device can improve the learning efficiency and inference in memory. I was responsible for training the networks and determining the proper weights. Based on the software simulation, the two networks we built both showed fairly good performances. We also performed the experiments on seeking the influence of learning rate and epochs and found that the optimum values of the two parameters for two MNNs, which enable them to reach optimal accuracy. The MNN with 784 input nodes is able to reach the optimal accuracy of 90.9%, and the smaller one presented the optimal accuracy of 87.9%. The MNN trained with original MNIST dataset also showed a good performance on both our own handwriting digits recognition and Cadence virtuoso hardware model simulation with quite excellent learning efficiency. For some reported NVM + selector crossbar arrays, the training (generalization) accuracy is merely 82.9% [24]. Therefore, our IFG array simulation result is fairly good. However, when we did the same with the smaller MNN hardware, the simulation results were unsatisfactory, which made it meaningless to evaluate the improvement on learning efficiency and inference in memory.

There are several possible factors that may have caused the relatively poor performance of MNN with 49 input nodes. The first one is the resizing algorithm. The images in the resized MNIST dataset are of poor quality. Some is quite difficult for even members in our team to recognize. Even though the result of software simulation is good, it is still disappointing without witnessing simulation results to be functional in hardware. Since the MNIST training and testing dataset have 70,000 images, replicating and resizing such huge amount of data without any loss is quite a difficult task, this calls for some novel and precise algorithms that we have not yet found. The bad performance may also attribute to the low pixel size. As shown in Chapter 3, only the digits with quite neat handwriting can be displayed in images of good quality with this pixel size. Therefore, in future work, it is recommendable to try to do experiments on images of higher pixel array, such as 10×10 or 14×14.

Bibliography

[1] McCulloch, Warren and Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". *Bulletin of Mathematical Biophysics*. 5(4):115-133. Doi:10.1007/BF02478259. PMID 2185863.

[2] Elliot J. Fuller, Scott T. Keene, Armantas Melianas, Zhongrui Wang, Sapan Agarwal, Yiyang Li, Yaakov Tuchman, Conrad D. James, Matthew J. Marinella, J. Joshua Yang, Alberto Salleo and A. Alec Talin. “Parallel programming of an ionic floating-gate memory array for scalable neuromorphic computing” Science 364 (6440), 570-574 DOI:10.1126/science.aaw5581 originally published online April 25, 2019

[3] Hyungjun Kim, Taesu Kim, Jinseok Kim, and Jae-Joon Kim. “Deep Neural Network Optimized to Resistive Memory with Nonlinear Current-Voltage Characteristics” arXiv:1703.10642

[4] Chen, Yung-Yao; Lin, Yu-Hsiu; Kung, Chia-Ching; Chung, Ming-Han; Yen, I.-Hsuan (January 2019). "Design and Implementation of Cloud Analytics-Assisted Smart Power Meters Considering Advanced Artificial Intelligence as Edge Analytics in Demand-Side Management for Smart Homes". Sensors. **19** (9): 2047. doi:10.3390/s19092047. PMC 6539684. PMID 31052502.

[5] Retrieved April 27, 2020 from Wikipedia

https://en.wikipedia.org/wiki/Artificial_neural_network

[6] Tariq Rashid. Make your own neural network. CreateSpace Publishing (2016).

[7] Ciresan, Dan; Ueli Meier; Jonathan Masci; Luca M. Gambardella; Jurgen Schmidhuber (2011). "Flexible, High Performance Convolutional Neural Networks for Image Classification" . Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence-Volume Volume Two. **2**: 1237–1242. Retrieved 17 November *2013*.

[8] L.O. Chua, Memristor—the missing circuit element. IEEE Trans. Circuit Theory 18(5), 507–519 (1971)

[9] L.O. Chua, S.M. Kang, Memristive devices and systems. Proc. IEEE 64(2), 209–223 (1976)

[10] Ioannis Vourkas · Georgios Ch. Sirakoulis. Memristive-Based Nanoelectronic Computing Circuits and Architectures. Springer (2016).

[11] S. Yu and P.-Y Chen. “Emerging memory technologies: recent trends and prospects” IEEE Solid-State Circuits Magazine, vol.8, no.2, pp. p43-p56, 2016.

[12] S. Mondal, J.-L. Her, F.-H. Chen, S.-J. Shih, T.-M. Pan, Improved resistance switching characteristics in Ti-Doped Yb2O3for Resistive nonvolatile memory devices. IEEE Elec. Dev. Lett. 33(7), 1069–1071 (2012)

[13] J.R. Heath, P.J. Kuekes, G.S. Snider, R.S. Williams, A defect-tolerant computer architecture: opportunities for nanotechnology. Science 280(5370), 1716–1721 (1998)

[14] An Chen. “A Comprehensive Crossbar Array Model With Solutions for Line Resistance and NonlinearDevice Characteristics” IEEE TRANSACTIONS ON ELECTRON DEVICES, VOL. 60, NO. 4, APRIL 2013

[15] Gkoupidenis P, Schaefer N, Garlan B and Malliaras G G. “Neuromorphic functions in PEDOT:PSS organic electrochemical transistors”. Adv. Mater. 27 7176–80

[16] Gkoupidenis P, Schaefer N, Strakosas X, Fairfield J A and Malliaras G G “Synaptic plasticity functions in an organic electrochemical transistor Appl”. Phys. Lett. 107 263302

[17] Qian C et al. “Artificial synapses based on in-plane gate organic electrochemical transistors ACS Appl. Mater. Interfaces 8 26169–75”

[18]. Z. Wang et al., Nat. Mater. 16, 101–108 (2017).

[19] Van de Burgt Y et al 2017 “A non-volatile organic electrochemical device as a low-voltage artificial synapsefor neuromorphic computing” Nat. Mater. 16 414-8

[20] Scott T Keene et al. 2018 J. Phys. D: Appl. Phys. 51 224002

[21] Y. van de Burgt et al., Nat. Mater. 16, 414–418 (2017).

[22] Yinghao Shao. “Application of Memristive Device Array for Pattern Recognition”. MS. Thesis, UC Santa Cruz June 2020

[23] DongUk Choi, private communication on compact model of IFG.

[24] Geoffrey W. Burr, Robert M. Shelby, Severin Sidler, Carmelo di Nolfo, Junwoo Jang, Irem Boybat, Student, Rohit S. Shenoy, Pritish Narayanan, Kumar Virwani, Emanuele U. Giacometti, Bülent N. Kurdi, and Hyunsang Hwang. “Experimental Demonstration and Tolerancing of a Large-Scale Neural Network (165000 Synapses)

Using Phase-Change Memory as the Synaptic Weight Element” IEEE TRANSACTIONS ON ELECTRON DEVICES, VOL. 62, NO. 11, NOVEMBER 2015

Appendix

Python code of 28×28 MNN:

```
import numpy as np
import scipy.special
import pylab
import imageio
class neuralNetwork:
    def __init__ (self, inputnodes, hiddennodes, outputnodes, learningrate):
        #initialize the nodes of each layer
        self.inodes=inputnodes
        self.hnodes=hiddennodes
        self.onodes=outputnodes

        #initialize the weights wih and who
        self.wih=np.random.normal(0.0, pow(self.hnodes, -0.5), (self.hnodes,
self.inodes))
        self.who=np.random.normal(0.0, pow(self.onodes, -0.5), (self.onodes,
self.hnodes))

        #initialize the learning rate
        self.lr=learningrate

        #activation function
        self.activation_function=lambda x: scipy.special.expit(x)
        pass

    #train the network
    def train (self, inputs_list, targets_list):
        #convert the inputs and targets lists to 2D array
        inputs=np.array (inputs_lists, ndmin=2).T
        targets=np.array (targets_lists, ndmin=2).T
        #propagation forward
        #the signals into hidden layer
        hidden_inputs=np.dot (self.wih, inputs)
        #the signals emerging from hidden layer
        hidden_outputs=self.activation_function(hidden_inputs)

        #the signals into final output layer
```

```
            final_inputs=np.dot (self.who, hidden_outputs)
            final_outputs=self.activation_function (final_inputs)
            #the final output layer error analysis by backpropagation
            output_error=targets-final_outputs
            #update the weights between the hidden and final output layers
            self.who+=self.lr*np.dot ((output_errors*final_outputs*(1.0-
final_outputs)), np.transpose(hidden_outputs))
            #the hidden layer error analysis by propagation
            hidden_errors=np.dot (self.who.T, output_errors)
            #update the weights between the input and hidden layers
            self.wih+=self.lr*np.dot ((hidden_errors*hidden_outputs*(1.0-
hidden_outputs)), np.transpose(inputs))
            pass

        #query the network
        def query (self, inputs_lists):
            #convert the input list to 2D array
            inputs=np.array(inputs_list, ndmin=2).T
            hidden_inputs=np.dot (self.wih, inputs)
            hidden_outputs=self.activation_function(hidden_inputs)
            final_inputs=np.dot (self.who, hidden_outputs)
            final_outputs=self.activation_function(final_inputs)
            return final_outputs

#number of input hidden output nodes
input_nodes=784
hidden_nodes=10
output_nodes=10

learning_rate=0.1
n=neuralnetwork= (input_nodes, hidden_nodes, output_nodes, learning_rate)
#load the mnist training data CSV file into a list
training_file=open (“mnist_train_csv”, ‘r’)
training_list=train_file.readlines()
training_file.close()

#train the neuralnetwork
epochs=5
for e in range(epochs)
   #go through all the records in the training data set
```

```
    for record in training_list:
        #split the record by the ‘,’ commas
        all_values=record.split(‘,’)
        #scale and shift the inputs
        inputs=(np.asfarray(all_values[1:])/255.0*0.99)+0.01
        #create the target output values (all 0.01, except the desired label which is
0.99)
        targets=np.zeros(output_nodes)+0.01
        #all_values[0] is the target label for this record
        targets[int(all_value[0])]=0.99
        n.train(inputs, targets)
        pass
pass
np.savetxt(‘win.csv’, n.wih)
np.savetxt(‘who.csv’, n.who)

#load the mnist test data CSV file into a list
test_file=open (“mnist_test.csv”, ‘r’)
test_list=test_file.readlines()
test_file.close()

#test the neural network
scorecard= []

#go through all the records in the test data set
for record in test_list:
    #split the record by ‘,’ commas
    all_values=record.split(‘,’)
    #correct answer is the first value
    correct_label=int(all_values[0])
    #scale the shift the inputs
    inputs=(np.asfarray(all_values[1:])/255.0*0.99)+0.01
    #query the network
    outputs=n.query(inputs)
    #the index of the highest value corresponds to the label
    label=np.argmax(outputs)
    #append correct or incorrect to list
    if(label==correct_label):
        #network’s answer matches correct answer, append 1 to scorecard
        scorecard.append(1)
```

```
    else:
        #network's answer doesn't match correct answer, append 0 to scorecard
        scorecard.append(0)
        pass
    pass
scorecard_array=np.asarray(scorecard)
print ("performance=", scorecard_array.sum()/scorecard_array.size)
def mytest (self, "my_written_x.png"):
    im=imageio.imread(my_written_x.png, as_gray=True)
    im=im.resize((28,28))
    tmp=np.array(im)
    vec=tmp.ravel()
    for i in range(len(vsc)):
        if vec[i]==0:
            vec[i]=255
        else:
            vec[i]=0
    my_inputs=(np.asfarray(vec[0:])/255*0.99)+0.01
    my_outputs=n. query(my_inputs)
    label=np.argmax(my_outputs)
    print (my_outputs)
    if (label==correct_label):
        print ("The answer is correct")
    else:
        print ("The answer is wrong")
```

Python code of 7×7 MNN:

```
import numpy as np
import scipy.special
import pylab
from PIL import Image

class neuralNetwork:
    def __init__ (self, inputnodes, hiddennodes, outputnodes, learningrate):
        #initialize the nodes of each layer
        self.inodes=inputnodes
        self.hnodes=hiddennodes
        self.onodes=outputnodes

        #initialize the weights wih and who
        self.wih=np.random.normal(0.0, pow(self.hnodes, -0.5), (self.hnodes,
self.inodes))
        self.who=np.random.normal(0.0, pow(self.onodes, -0.5), (self.onodes,
self.hnodes))

        #initialize the learning rate
        self.lr=learningrate

        #activation function
        self.activation_function=lambda x: scipy.special.expit(x)
        pass

    #train the network
    def train (self, inputs_list, targets_list):
        #convert the inputs and targets lists to 2D array
        inputs=np.array (inputs_lists, ndmin=2).T
        targets=np.array (targets_lists, ndmin=2).T
        #propagation forward
        #the signals into hidden layer
        hidden_inputs=np.dot (self.wih, inputs)
        #the signals emerging from hidden layer
        hidden_outputs=self.activation_function(hidden_inputs)

        #the signals into final output layer
        final_inputs=np.dot (self.who, hidden_outputs)
```

```
            final_outputs=self.activation_function (final_inputs)
            #the final output layer error analysis by backpropagation
            output_error=targets-final_outputs
            #update the weights between the hidden and final output layers
            self.who+=self.lr*np.dot ((output_errors*final_outputs*(1.0-
final_outputs)), np.transpose(hidden_outputs))
            #the hidden layer error analysis by propagation
            hidden_errors=np.dot (self.who.T, output_errors)
            #update the weights between the input and hidden layers
            self.wih+=self.lr*np.dot ((hidden_errors*hidden_outputs*(1.0-
hidden_outputs)), np.transpose(inputs))
            pass

        #query the network
        def query (self, inputs_lists):
            #convert the input list to 2D array
            inputs=np.array(inputs_list, ndmin=2).T
            hidden_inputs=np.dot (self.wih, inputs)
            hidden_outputs=self.activation_function(hidden_inputs)
            final_inputs=np.dot (self.who, hidden_outputs)
            final_outputs=self.activation_function(final_inputs)
            return final_outputs

#number of input hidden output nodes
input_nodes=49
hidden_nodes=10
output_nodes=10

learning_rate=0.6
n=neuralnetwork= (input_nodes, hidden_nodes, output_nodes, learning_rate)
#load the mnist training data CSV file into a list
training_file=open ("new_train_csv", 'r')
training_list=train_file.readlines()
training_file.close()

#train the neuralnetwork
epochs=9
for e in range(epochs)
   #go through all the records in the training data set
   for record in training_list:
```

```
        #split the record by the ‘,’ commas
        all_values=record.split(‘,’)
        #scale and shift the inputs
        inputs=(np.asfarray(all_values[1:])/255.0*0.99)+0.01
        #create the target output values (all 0.01, except the desired label which is
0.99)
        targets=np.zeros(output_nodes)+0.01
        #all_values[0] is the target label for this record
        targets[int(all_value[0])]=0.99
        n.train(inputs, targets)
        pass
pass
np.savetxt(‘win_new.csv’, n.wih)
np.savetxt(‘who_new.csv’, n.who)

#load the mnist test data CSV file into a list
test_file=open (“new_test.csv”, ‘r’)
test_list=test_file.readlines()
test_file.close()

#test the neural network
scorecard= []

#go through all the records in the test data set
for record in test_list:
    #split the record by ‘,’ commas
    all_values=record.split(‘,’)
    #correct answer is the first value
    correct_label=int(all_values[0])
    #scale the shift the inputs
    inputs=(np.asfarray(all_values[1:])/255.0*0.99)+0.01
    #query the network
    outputs=n.query(inputs)
    #the index of the highest value corresponds to the label
    label=np.argmax(outputs)
    #append correct or incorrect to list
    if(label==correct_label):
        #network’s answer matches correct answer, append 1 to scorecard
        scorecard.append(1)
    else:
```

```
        #network's answer doesn't match correct answer, append 0 to scorecard
        scorecard.append(0)
        pass
    pass
scorecard_array=np.asarray(scorecard)
print ("performance=", scorecard_array.sum()/scorecard_array.size)
def mytest (self, "my_written_x.png"):
    im=imageio.imread(my_written_x.png, as_gray=True)
    im=im.resize((7,7))
    tmp=np.array(im)
    vec=tmp.ravel()
    for i in range(len(vsc)):
        if vec[i]==0:
            vec[i]=255
        else:
            vec[i]=0
    my_inputs=(np.asfarray(vec[0:])/255*0.99)+0.01
    my_outputs=n. query(my_inputs)
    label=np.argmax(my_outputs)
    print (my_outputs)
    if (label==correct_label):
        print ("The answer is correct")
    else:
        print ("The answer is wrong")
```

MNIST resizing algorithm:

```
import numpy as np
import pandas as pd
import csv
data=pd.read_csv(“filename.csv”, header =None)
res= []
for j in range (0, # of data lists):
   a=data.loc[j]
   a=np.array(a)
   d=a[0]
   a=a[1:]
   b=a.reshape((28,28))
   b= (np.split(b, 7, axis=1))
   for i in range (0, 7):
      b[i]=np.mean (b, 7, axis=1)
      b[i]=b[i].reshape(7,4)
      b[i]=np.mean(b[i])
      pass
   e=np.vstack((b[0], b[1], b[2], b[3], b[4], b[5], b[6]))
   e=e.T
   e=e.reshape(49)
   e=np.hstack((d,e))
   res.append(e)
with open (‘resized filename’, ‘w’, newline=’’) as csvfile:
     writer =csv.writer(csvfile, delimiter=’,’ , quotechar=’””,
     quoting=csv. QUOTE_MINIMAL)
     for line in res:
          writer.writerow(line)
```