1

An Analog VLSI Implementation of the Wake-Sleep Learning Algorithm Using Bi-Stable Synaptic Weights

Guy Lipworth
Dept. of Electrical
and Computer Engineering
University of Florida
Gainsville, Florida 32611
Email: musicguy@ufl.edu

Kyle McMillan
Lane Department of Computer Science
and Electrical Engineering
West Virginia University
Morgantown, West Virginia 26506
Email: kmcmill2@mix.wvu.edu

***Abstract*—Drawing on biological systems for their inspiration, typical supervised neural networks learn to classify features within a set of inputs through repetition. Here, we focus on using an auto-encoder network to memorize each item in a set of inputs rather than to classify them. We have simulated this network in MATLAB using the "Wake-Sleep" learning algorithm proposed by Hinton *et al.* [1] and demonstrated that the algorithm can be used successfully with binary synaptic weights trained in a bistable manner. Working from these simulations, we have designed and simulated a low power analog VLSI synapse circuit with analog but bi-stable weights that can implement the Wake-Sleep algorithm.**

I. Introduction

As we go about our daily lives, we make extensive use of our recognition abilities. When you walk down the hallway toward your office in the morning, you can pick your office out of the collection of all others in the hall. Although all of the offices may have similar characteristics, you know which one is yours because you recognize particular features around it. Perhaps it is a poster in the hallway opposite your office. Perhaps its the burned-out light in the hallway immediately outside the door, or the strange music your co-worker plays when they think no one is listening. Your brain recognizes a number of these unique features and combines them into one thought, "I'm standing right outside *my* office."

Our challenge is to give this same ability to a robot with an extremely low-power implementation. To do so, we must design a system capable of recognizing the unique features of an input and remembering them at any point in time. In this paper we will introduce neural networks, (sections II & III), how to train them to memorize inputs according to the Wake-Sleep Algorithm (sections IV & V), and the several circuits we used and designed to implement the training in hardware (sections VI & VII).

The authors would like thank the following people who contributed time, expertise, motivation and inspiration to this project.
FACULTY ADVISORS:
Timothy Horiuchi
Pamela Abshire
GRADUATE ADVISORS:
Timir Datta
Anshu Sarje

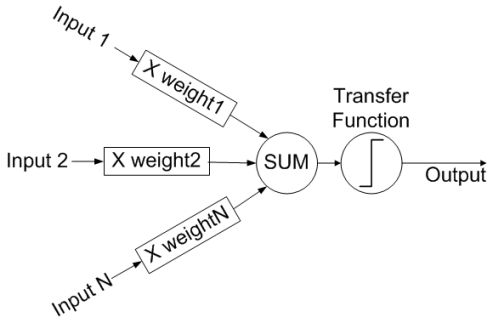


Fig. 1. A Perceptron: inputs are multiplied by their corresponding weights before being summed together and passed through a transfer function to obtain the output. The transfer function shown here is the unit step but others, including the signum, tanh and tansig, can be used.

II. The Perceptron and Synaptic Weights

The biological neuron communicates with other neurons through synapses, or the connection points between neurons. When modeling the neuron in circuits, neuromorphic engineers traditionally break this biological system down into discrete parts and treat the synapse and neuron as independent components. The synapse integrates voltage spikes into current; a higher spike-train frequency produces a larger current. The neuron integrates synaptic currents and produces output voltage spikes when the integration rises above a certain threshold.

A simplified description of the relationship between the synapse and the neuron is shown in the perceptron in Fig. 1. Each of the perceptron's inputs is multiplied by a weight - this part of the perceptron can be thought of as the synapse. The perceptron then sums the results of each multiplication (synaptic outputs) and passes them through a transfer function to produce an output - this is the neuron-like part. In our figure the transfer function is the unit step, but other transfer functions may be used as well. The perceptron demonstrates

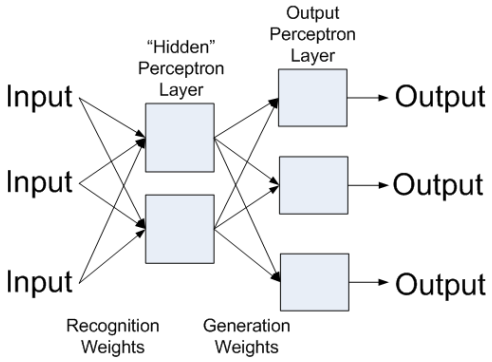


Fig. 2. Simple Auto-Encoder Network with separate Generation and Recognition weights.

how several synaptic outputs are passed to a neuron as inputs to create one output. Biological synaptic weights allow the neuron to discriminate between synapses; since larger weights produce greater synaptic outputs, the weights tell the neuron the importance of each synapse and its associated input.

III. The Auto-Encoder Neural Network

By feeding the output of one perceptron into the input of another we can create a neural network as shown in Fig. 2. Adding more perceptrons in this manner allows us to add layers to the network, or make an existing layer larger. Because the overall outputs of the network are the outputs of the second layer we call the second layer of perceptrons the 'Output Layer' and the first layer the 'Hidden Layer'. We also give each set of weights a different name to be able to distinguish between them: The weights between the network's inputs and the hidden layer are called 'Recognition Weights', while the weights between the hidden and output layers are called 'Generation Weights'.

In these experiments, a perceptron-based auto-encoder neural network was used. This network compares the network outputs to its inputs and trains its synaptic weights according to the perceptron learning rule until the outputs are identical to the inputs:

$$W_{new} = W_{old} + \gamma(In)(E)$$

Where W_{new} is the new synaptic weight, W_{old} is the previous synaptic weight, γ is the learning rate (between 0 and 1), In is the synaptic input, and E is the error, in this case the difference between the desired output and the perceptron output.

There are several ways to train the synaptic weights of a multiple-layered neural network; in our project we focus on the Wake-Sleep Algorithm.

IV. Wake-Sleep Algorithm

To simulate an auto-encoder in MATLAB, we created a two layer auto-encoder with 25 hidden perceptrons. As shown in Fig. 3, inputs to the network are images of 10 pixels wide by 15 pixels long; thus the desired outputs are also 10x15 pixel

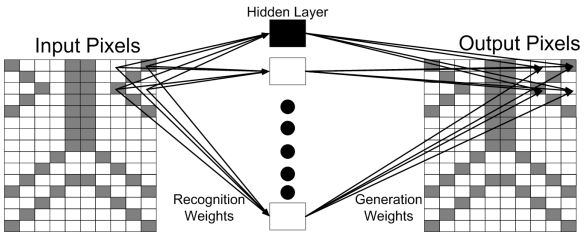


Fig. 3. A simplified block diagram showing 10x15 pixel input and output images connected by a hidden layer and two separate sets of weights.

images and we need 150 neurons in our output layer. Each pixel in our images was set to a value of 1 or -1, and the perceptron transfer function was the signum function (output is +1 if the summation is positive, 0 if the summation = 0, and -1 if the summation is negative).

Initially, all Recognition and Generation weights were normally distributed with zero mean and a standard deviation of one. The Wake-Sleep Algorithm was then used to train the auto-encoder's weights in two cycles. During the Wake cycle, the network passed an input through the Recognition weights to produce an encoded version of the input in the hidden layer. The output of the hidden layer is passed through the Generation weights, which were trained to replicate the original input to the network. During the Sleep cycle, the hidden layer was randomly excited and passed through the Generation weights to form a "fantasy image." This fantasy was passed through the Recognition weights, which were then modified in an attempt to recreate the hidden layer states that resulted from the initial random excitation. This process forms a two step iterative algorithm, with each cycle improving the overall recognition abilities of the network until it converges on an optimal encoding solution (minimum description cost) for a particular set of inputs.

A. Wake Cycle

During the Wake cycle, an image is randomly selected from a pool of 20 test images. Each pixel serves as one input to a perceptron in the hidden layer - thus each perceptron in the hidden layer has 150 inputs. In addition, since each perceptron has one weight associated with each input, we need 3,750 recognition weights (25 hidden perceptrons multiplied by 150 inputs). The hidden states - the outputs of the perceptrons in the hidden layer - are sent through the Generation weights to the output layer. Since each of the 150 output layer perceptrons receives 25 inputs (from the hidden layer) there are 3,750 independent Generation weights.

An error signal for each output pixel is computed by comparing the output image to the input image (our desired output), and the Generation weights are trained in an analog manner according to the perceptron learning rule.

During the wake cycle we repeat the process with the same image for 30 iterations before displaying a new image and retraining the weights. The Wake cycle ends after 500 hundred iterations of the process described above.

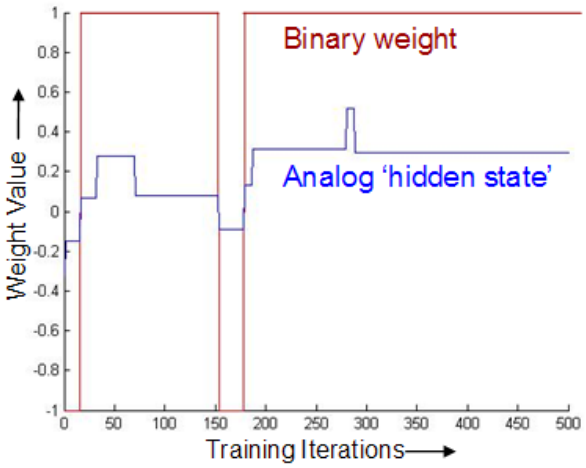


Fig. 4. Time vs Synaptic Weight of a perceptron. As the "hidden" analog weight is trained, the digital weight used in computation is snapped high or low as the analog weight crosses a threshold, here set to zero Volts.

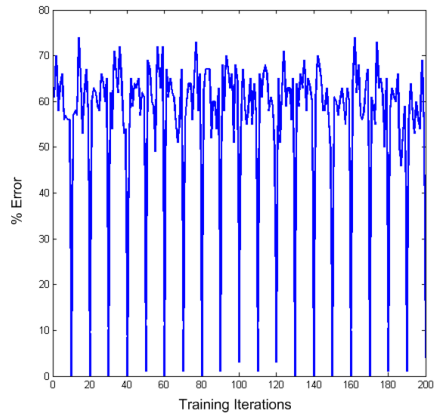
B. Sleep Cycle

During the Sleep cycle, our input is no longer a 10x15 image. Instead, we randomly excite the hidden layer to create 20 hidden states with values of +1 or -1. These hidden states are sent through the Generation weights to create a 10x15 matrix which is then passed through the Recognition weights to produce a 20 element output matrix. Next, we compare the output to the randomly excited hidden states which served as inputs, compute an error signal, and train the Recognition weights. As was done in the wake cycle, we repeat this process 500 times with different randomly generated inputs. Notice that the sleep cycle is similar to a wake cycle except that it 'begins' at the Hidden layer; the 10x15 matrix generated during the sleep cycle does not serve as an output in the sleep cycle, but is analogous to the output image generated in the wake cycle.

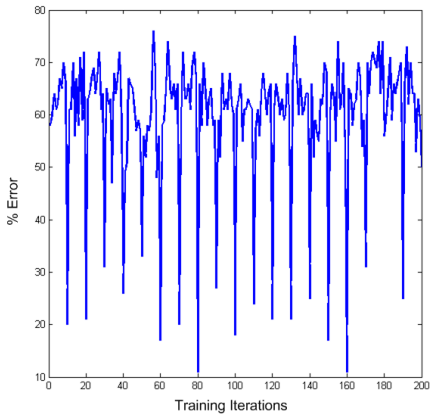
To train the weights we followed each wake cycle with a sleep cycle and repeated the process 50 times. The last sleep cycle, which trained the recognition weights with randomly created inputs, was then followed by one more wake cycle which trained the recognition weights again to ensure proper weighting.

C. Binary Weights

Weights trained in an analog manner mimic the way synaptic weights are biologically trained. However, quantizing the weights in a binary manner will facilitate storage in electronic & possibly biological hardware over long periods of time. To quantize the weights, we modified the auto-encoder: for each weight (Recognition and Generation) we added a 'hidden weight' (unrelated to the hidden layer). The hidden weights were trained as above, in an analog manner (with a limited range of analog values). At the end of each cycle, however, each weight was 'snapped' to -1, 0, or +1 by taking the signum of the hidden weight, shown in Fig. 4.



(a) Network Testing With Analog Weights



(b) Network Testing With Quantized Weights

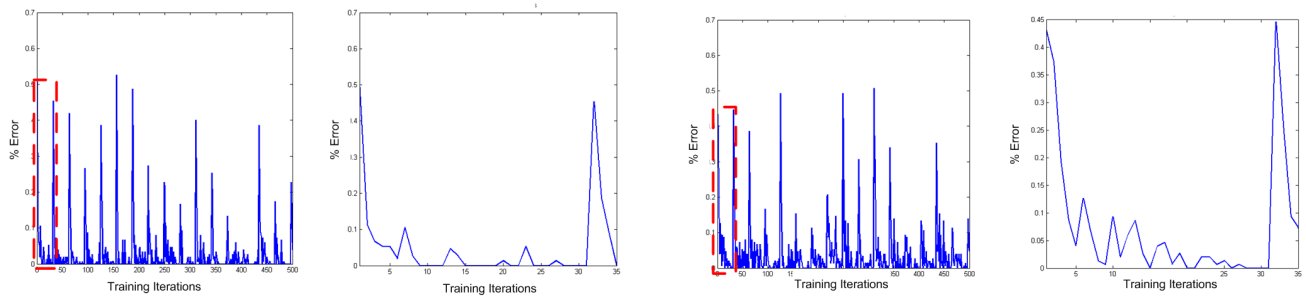
Fig. 5. MATLAB simulations of an auto-encoder network trained with analog weights (a) and binary weights (b). After the network has been trained on 20 images over 50 Wake / Sleep cycles of 500 iterations each, learning is stopped and the network is presented with normally distributed noise. On every 10th exposure, an image the network has been trained to recognize is presented and the percentage error drops dramatically, indicating that the network "recognizes" the image.

At the end of each cycle we also 'snap' the 'hidden weights' to -1, 0, or +1 using the same method to prevent them from increasing or decreasing to unrealistic values. This is done after finding the values of the quantized weights.

V. MATLAB Simulation Results and Analysis

Figures 6(a) 6(b) and show Error vs. Training Iterations in MATLAB during the first wake cycle for the auto-encoder trained with analog weights and quantized weights, respectively. Error is high when a new image is displayed, and decreases while the same image is repeatedly displayed and the weights are trained. Similar patterns are observed for the Sleep cycle. Typically, the Wake Cycle error for the analog weighted network drops to 0.5% and the error for the binary weighted network drops to 3% within the 30 iterations during which the same image is displayed.

For post-training simulation of the network, we selected



(a) Training With Analog Weights

(b) Training With Quantized Weights

Fig. 6. MATLAB simulation of an auto-encoder network training with analog 6(a) and quantized 6(b) weights. The figure on the right part of 6(a) is an enlarged section of the left part of 6(a), indicated by the dotted rectangle. (The same is true for the two images in 6(b)) A new image is presented to the network and displayed 30 times. The initial exposure results in high percentage error, but as the image is repeatedly displayed the error decreases (error increases after reaching zero due to random noise added to the inputs). This pattern of exposure is repeated every 30 iterations.

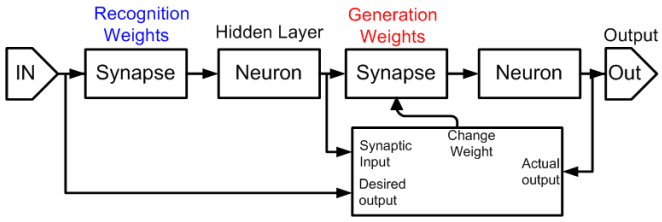


Fig. 7. System Architecture for the Wake Cycle. The perceptron used to create the simulated systems in MATLAB has been replaced with a synapse and neuron pair, and the learning rule that implements the auto-encoder network is provided by the Weight Modification Circuit.

a 10x15 image as the input and calculated the error by comparing it to the output (similar to the wake cycle but without modifying the weights). The error graphs for the analog-weighted and quantized-weighted auto-encoders are shown in Figures 5(a) and 5(b), respectively. To test if network training actually occurred we displayed a series of images that the system was not trained on, but every 10th iteration displayed an image that the network *was* trained on. The error is high for images the network had not seen before, and spikes of low error are produced for images the network had seen during training.

Our simulation results confirm that an auto-encoder network with quantized synaptic weights can approach the performance of a network with analog weights and the same complexity. While the resolution of the individual synapse is compromised, a large network with many synapses can overcome this limitation and produce accurate outputs.

VI. Hardware Design

Fig. 7 shows the overall system design broken down into basic circuit blocks: neurons, synapses, and the Weight Modification block. For our neuron circuit we chose to use a neuron previously described by T. Horiuchi [2]. The synapse circuit has been designed to implement a bi-stable synaptic weight, and the Weight Modification block implements the auto-encoder network learning rules. Each block will be covered in the sections below.

A. Neuron Circuit

Horiuchi's neuron provides low power consumption, an adjustable refractory period, and an Address - Event Representation (AER) interface. The specific features of this circuit are not essential to the operation of our system; any of the myriad Integrate-And-Fire Neuron circuits available in literature can be used. The only requirement is an adjustable refractory period to allow for circuit and system tuning.

B. Synapse Circuit

The most basic function of the synapse circuit is to integrate a series of voltage spikes into a current output. The synaptic output should also depend on an adjustable "weight" that represents the importance of the neuron in the overall neural network. The synapse in Fig. 8 modulates the output current by varying the gate voltage (the synaptic weight) of transistor *M9*, which changes the amount of current that can pass through the transistor's channel. The synapse accomplishes the voltage spike integration using a Current Mirror Integrator (CMI) [3], located on the right side of the circuit. For a full explanation of the operation of the CMI, see [3]. The synaptic weight, stored as a voltage across capacitor $C1$, can be modified by two systems: the Bi-Stability circuit and the Weight Modification Circuit through the node labeled inc/dec_weight.

1) Bi-Stability: This circuit draws heavily from the work of Indiveri *et al.* [4]. In this synapse circuit, the Bi-Stability section is found in the left-hand part of Fig. 8. A thorough description of the Bi-Stability circuit's operation is provided in [3], so here we will give a high-level view of the circuit's characteristics.

The purpose of the Bi-Stability circuit is to adjust the weights over long time periods in response to the leakage currents through capacitor $C1$. Recognizing that leakage currents will modify the voltage across $C1$, the Bi-Stability circuit slowly pulls the weight towards a high or low value by comparing the synaptic weight to a voltage threshold V_{wt_thr}. If the weight is higher than V_{wt_thr}, the Bi-Stability circuit slowly drives the output even higher. Over time, this output voltage will reach a limit set by V_{wt_high}. If the weight is lower

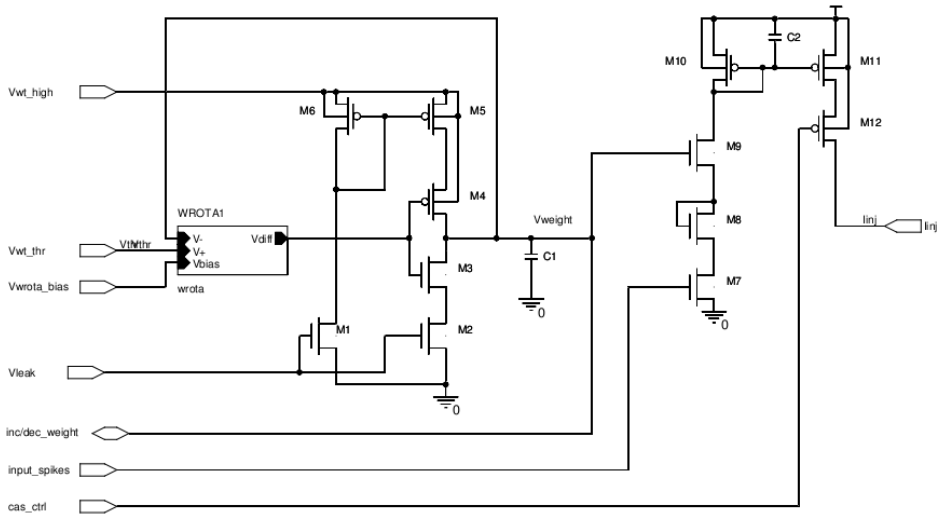


Fig. 8. Synapse Circuit composed of a Bi-Stability section and a Current Mirror Integrator

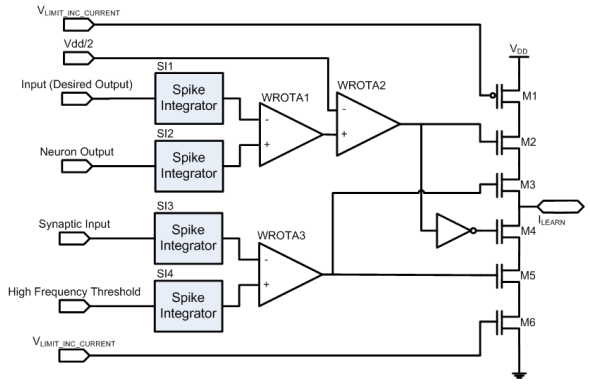


Fig. 9. Weight Modification Circuit. The blocks $SI1$ through $SI4$ are spike integrator circuits, shown in Fig. 10. These circuits integrate the spiking neuron signal from the Input (the Desired Output), Neuron Output and Synaptic Input and produce slow changing voltage outputs that are used to determine how to modify the synaptic weight via I_{LEARN}

than V_{wt_thr}, the circuit will drive the output even lower until the synaptic weight is zero Volts. The rate at which this occurs is set by V_{leak} and should be much slower than the effect of the Weight Modification Circuit.

2) Current Mirror Integrator: The Current Mirror Integrator (CMI) [5] integrates a spiking input on the gate of transistor $M7$ across $C2$. This changing voltage across the capacitor produces a current through $M11$, which is pushed through the cascode transistor $M12$ to produce a current at the node labeled I_{inj}.

C. Weight Modification Circuit

The Weight Modification (WM) circuit shown in Fig. 9 implements the actual network learning rules. This circuit performs the calculations necessary to implement the auto-encoder by comparing the output of the system to the input of the system as illustrated in Fig. 7.

Before we begin explaining the WM circuit, we must note that unlike our MATLAB simulations, our hardware design includes only excitatory synapses. As a result, our synapses can not have negative weights; this affects the way we train our weights. In addition, please note that Fig. 7 shows only one WM circuit. To implement the entire algorithm, two are needed: one to train the Generation weights in the Wake Cycle and one for the Recognition weights in the Sleep Cycle. In the following discussion we explain the circuit with its inputs as shown in Fig. 7, which corresponds to the Wake Cycle.

The Weight Modification circuit modifies the synaptic weight after deciding if the weight is too high or low. This is done by comparing the network output and input (its desired output) through Spike Integrator (Fig. 10) circuits. If the actual output signal is lower than the desired output signal, the synaptic weight is apparently too low and should be increased. Alternatively, if the actual output signal is higher than the desired output signal the weight should be decreased. The WM circuit also checks the synaptic input before adjusting the weight: since the synapse would have no noticeable output if the synaptic input is low, the circuit will not modify the synaptic weight. The circuit accomplishes weight modification by sourcing or sinking current through I_{LEARN}, which is connected to the synaptic weight capacitor $C1$ in the Synapse circuit. When the output is too high (and the synaptic input is high) transistors M4 and M5 are on, sinking current from the synaptic weight and thus lowering its value. When the output is too low (and the synaptic input is high), M2 and M3 are on and current is sourced to the synaptic weight, raising its value. Transistors M1 and M6 are used to limit the amount of current flowing to/from the weight in order to set the learning rate.

1) Spike Integrator: The network input (desired output), output, and synaptic input are all voltage spike trains of varying frequencies. Since these spike trains are asynchronous,

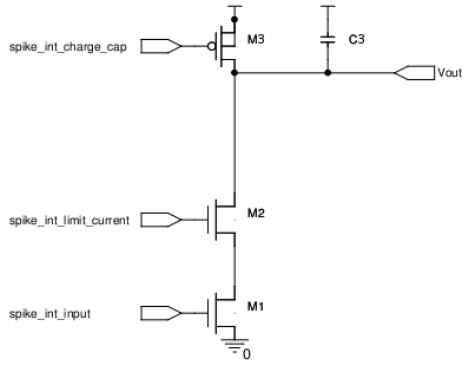


Fig. 10. The Spike Integrator circuit. A spiking voltage at the gate of $spike\_int\_input$ produces a current through transistor $M3$, which changes the voltage across $C1$.

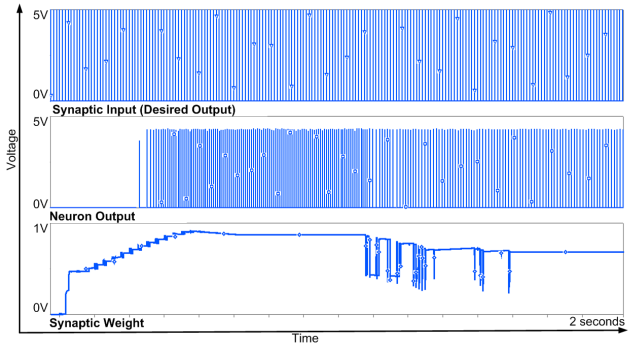


Fig. 11. Simulation results of a Synapse / Neuron pair controlled by a Weight Modification circuit. Initially, the synaptic weight is zero and the output neuron spiking frequency is zero. As time increases, the synaptic weight is pushed higher by the Weight Modification Circuit. When the output neuron spikes too quickly, the Weight Modification circuit decreases the synaptic weight until the output neuron spiking frequency matches the input neuron spiking frequency.

it is difficult to compare them directly. To accomplish this, our Spike Integrator circuits (Fig. 10) translate the spike frequency into slowly-changing DC voltages by discharging capacitor $C1$ with every spike. Each spike turns transistor M1 on for a short time period, during which current flowing from C1 through M1 lowers the capacitor voltage Vout. Transistor M2 limits amount of current that flows through the drain of M1, allowing us to control the rate at which Vout decreases. Between spikes, current flowing through M5 increases Vout slowly. Therefore, a higher frequency discharges the capacitor faster than a lower frequency. By comparing Vout using WROTAs (Wide Range Operational Transconductance Amplifiers) in the rest of the Weight Modification Circuit, the circuit can determine which frequency is higher. The output of WROTA2 is a digital signal and indicates which input is spiking faster than the other.

VII. Circuit Test Results and Analysis

Our synapse and Weight Modification circuits were tested using a single synapse neuron pair. Performance was observed by setting the input (also the desired output) to a fixed 100 Hz neural spiking frequency. The output neuron's spiking frequency was observed and compared to the desired input frequency to determine the synaptic weight.

An example trial is shown in Fig. 11. At time = 0, the synaptic weight (Voltage) is low and the neuron output is low. As time passes, the WM increases the weight in an effort to make the output neuron spike faster. The weight continues to increase until the neuron output is too high (here, at time = 0.5 seconds). This condition is detected by the WM circuit and the synaptic weight is decreased until the desired output frequency is equal to the actual output frequency at time = 1.2 seconds.

VIII. Conclusions

Through the MATLAB circuit simulations, we have demonstrated that an autoencoder network can operate with binary weights. Though there is a significant loss in the resolution of the individual synapses, the auto-encoder network is still capable of "remembering" input patterns.

Our successful simulations led to the design of a simple synapse and the Weight Modification Circuit. By using these elements with an existing neuron design, we have demonstrated that a synapse / neuron pair can be trained to modulate its output signal to match an input signal and assume a binary state over long periods of time.

IX. Future Work

Since we simulated our circuit with only one synapse-neuron pair, we were not able to demonstrate that the synaptic weight can assume a binary state over long periods of time (due to the high synaptic accuracy needed when using only one synapse). In the future, we would like to implement networks with many more synapses in the hope that the decreased cost associated with binary storage will allow for an increase in network complexity and match the overall functionallity of a purely analog system.

It is our belief that the circuit elements presented in this paper can be used to make a robust auto-encoder in silicon. By using separate WM circuits for the Recognition and Generation weights, it is also possible to fabricate a true hardware implementation of the Wake-Sleep algorithm. By combining both of these structures into a monolithic design, the groundwork presented here can be transformed into a practical learning circuit.

Acknowledgments

The authors would like to thank Professors Timothy Horiuchi and Pamela Abshire for their inspiration and guidance, Timir Datta and Anshu Sarje for their advice and dedication, and the National Science Foundation for making this project possible.

References

[1] G. Hinton, P. Dayan, B. Frey, and R. Neal, "The "wake-sleep" algorithm for unsupervised neural networks," *Science*, vol. 268, p. 1158, May 1995.

[2] T. Horiuchi, "A neural model for sonar-based navigation in obstacle fields," in *Proceedings of the International Symposium on Circuits and Systems*, 2006, pp. 4543–4546.

[3] G. Indiveri, E. Chicca, and R. Douglas, "A vlsi array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity," *Neural Networks, IEEE Transactions on*, vol. 17, no. 1, pp. 211–221, Jan. 2006.

[4] S. Mitra, S. Fusi, and G. Indiveri, "A vlsi spike-driven dynamic synapse which learns only when necessary," *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pp. 4 pp.–, May 2006.

[5] S. Liu, J. Kramer, G. Indiveri, T. Delbruck, and R. Douglas, *Analog VLSI: Circuits and Principles*. The MIT Press, 2002.