8-2019

Whetstone Trained Spiking Deep Neural Networks to Spiking Neural Networks

Jiajia Zhao
University of Tennessee, jzhao29@vols.utk.edu

Conversion of Whetstone Trained Spiking Deep Neural Networks to Spiking Neural Networks

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Jiajia Zhao

August 2019

Abstract

A deep neural network is a non-spiking artificial neural network which uses multiple structured layers to extract features from the input. Spiking neural networks are another type of artificial neural network which closely mimic biology with time dependent pulses to transmit information. Whetstone is a training algorithm for spiking deep neural networks. It modifies the back propagation algorithm, typically used in deep learning, to train a spiking deep neural network, by converting the activation function found in deep neural networks into a threshold used by a spiking neural network. This work converts a spiking deep neural network trained from Whetstone to a traditional spiking neural network in the TENNLab framework. This conversion decomposes the dot product operation found in the convolutional layer of spiking deep neural networks to synapse connections between neurons in traditional spiking neural networks. The conversion also redesigns the neuron and synapse structure in the convolutional layer to trade time for space. A new architecture is created in the TENNLab framework using traditional spiking neural networks, which behave the same as the spiking deep neural network trained by Whetstone before conversion. This new architecture verifies the converted spiking neural network behaves the same as the original spiking deep neural network. This work can convert networks to run on other architectures from TENNLab, and this allows networks from those architectures to be trained with back propagation from Whetstone. This expands the variety of training techniques available to the TENNLab architectures.

Table of Contents

List of Figures

Chapter 1

Introduction

There are a variety of neural networks that are currently under focus by the Artificial Intelligence community. Deep Neural Networks (DNNs), based on traditional, non-spiking, feed-forward neural networks, have been impressively successful in performing image recognition tasks. Spiking neural networks (SNNs) are receiving a great deal of attention because they more closely mimic the human brain, and they can perform powerful computations with less power consumption than DNNs. Recently, researchers from Sandia have designed an algorithm called Whetstone [1], which allows one to train DNNs, and then "sharpen" them into Spiking Deep Neural Networks (SDNNs). These are similar to SNNs, but contain some enhanced functionality beyond SNNs.

The original goal of this project was to leverage the Whetstone methology and incorporate it into the TENNLab Exploratory Neuromorphic Computing framework [2]. This framework supports a variety of SNN implementations, both in software and in hardware. In performing this conversion, we discovered a debilitating problem, in that the converted SNNs were enormous in size. As a result, we designed a second conversion methodology that trades time for space, and drastically reduces the size of the SNN networks.

We have implemented both conversion modules, and incorporated them into two of the TENNLab SNN architectures:

1. An architecture called “WHETSTONE,” whose neuron/synapse properties match the SDNNs, but without the enhanced functionality. This architecture exists in simulation only.

2. An architecture called DANNA2, which is a two-dimensional, integer-based SNN architecture, which has software, FPGA and VLSI implementations [3].

We describe the conversion modules in detail, and show the results of running classification tasks on the two TENNLab architectures.

Chapter 2

Related Work

This chapter includes introduction of different types of artificial neural networks that are relevant to this project, description of TENNLab's projects that are being used in this work, and overview of the Whetstone algorithm.

2.1 Deep Neural Network

A Deep Neural Network (DNN) is a traditional layer structured artificial neural network that has been used for classification [4][5]. DNNs are composed of neurons, which calculate values, and synapses, which communicate values from "pre" neurons to "post" neurons. The following sections briefly describe Feed Forward Neural Network, Convolutional Neural Network, and activation function.

2.1.1 Feed forward Neural Network

A feed forward neural network (FFNN) is constructed by one dimensional layers of neurons. Every neuron in the previous layer is connected to every neuron in the next layer by a synapse. Figure 2.1 shows an example FFNN. The network performs a sequence of computations on the inputs to find a result. The computation is performed layer by layer. On each layer, except the input layer, the neuron receives charges from its incoming synapses. The charge received from each synapse is the product of the synapse weight and the pre-neuron's firing

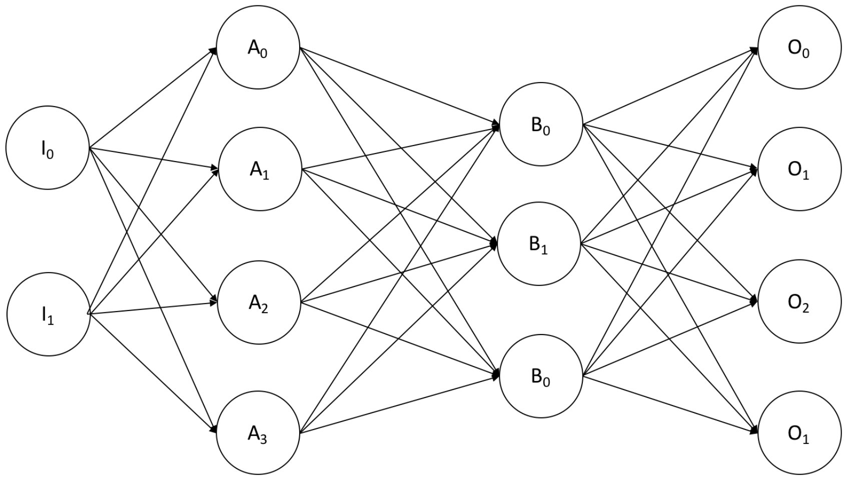

Figure 2.1: An example of FFNN with two inputs, four outputs, and two hidden layers.

charge. The neuron's firing charge is different for every problem, but the synapse weight remains the same. Figure 2.2 shows the detailed connections for B_0, and using B_0 as an example, the charge in B_0 will be

$$B_0 = \quad A_0 * w_0 + A_1 * w_1 + A_2 * w_2 + A_3 * w_3 + bias. \tag{2.1}$$

2.1.2 Convolutional Neural Network

At a high level, the "convolutional layers" of a convolutional neural network (CNN) are the layers that identify the presence or absence of different aspects of the input, which are generally called "features." [6] (Multiple features are usually checked for at the same time during a single convolutional layer.) For example, if a CNN is designed to be able to classify hand-written numerals (as in the MNIST data set), a convolutional layer might look for a straight horizontal line at the bottom of the image (as in the way many people write the number 2) and for any curve (which could indicate that the image is not a 1, a 4, or a 7). Of course, identifying whether the input includes (or doesn't include) a feature isn't enough to classify the image or eliminate possible classes, though it does raise or lower the probability

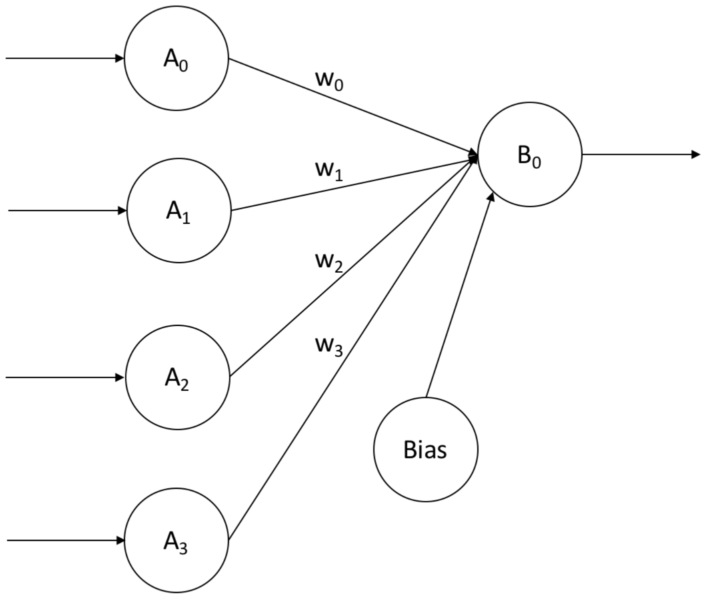

Figure 2.2: The detail connections of the neuron B_0 from figure 2.1

that the image is of a certain class; these probabilities are what the output of CNNs depends on.

Mathematical Details

Convolutional layers work by applying a set of "filters" (also called "feature maps" or "kernels" and represented here with W) to the layer's input data (represented here with I) and creating output channels (represented here with Z), one channel per filter. Each filter is a matrix of weights. The following is a description of one filter being applied to the input data. In our examples, I is composed of 0's and 1's to help keep the examples simple. In actuality, I, W, and Z can all be arbitrary values.

Assume I is a square 4x4 matrix and that W is a 3x3 filter, as in Figure 2.3. (Note that our subscripting throughout this Thesis is xy instead of rc; this is because we made the decision to use coordinates because of how often we deal with matrices in more than two dimensions in our research.) In this case, this layer is convolving from a 4x4 matrix to a 3x3

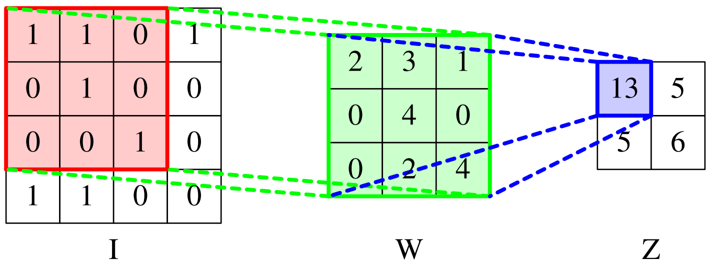

Figure 2.3: Calculating Z_{00} in a convolutional layer; thus, a pictoral representation of Equation 2.2.

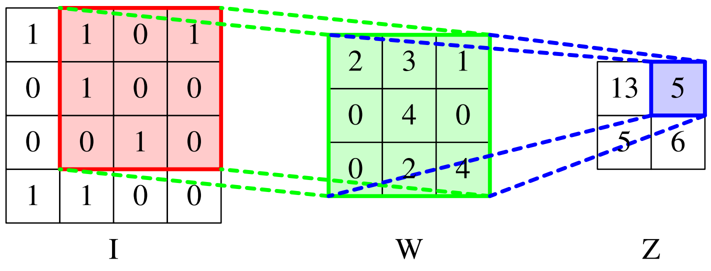

Figure 2.4: Calculating Z_{10} in a convolutional layer; thus, a pictoral representation of Equation 2.3.

matrix. Z_{00} will be

$$\begin{aligned} Z_{00} = \quad & W_{00} * I_{00} + W_{10} * I_{10} + W_{20} * I_{20} \\ & + W_{01} * I_{01} + W_{11} * I_{11} + W_{21} * I_{21} \\ & + W_{02} * I_{02} + W_{12} * I_{12} + W_{22} * I_{22} \ . \end{aligned} \tag{2.2}$$

After calculating Z_{00}, the filter moves one cell to the right, as in Figure 2.4. We now have:

$$\begin{aligned} Z_{10} = \quad & W_{00} * I_{10} + W_{10} * I_{20} + W_{20} * I_{30} \\ & + W_{01} * I_{11} + W_{11} * I_{21} + W_{21} * I_{31} \\ & + W_{02} * I_{12} + W_{12} * I_{22} + W_{22} * I_{32} \ . \end{aligned} \tag{2.3}$$

This "moving 1 cell to the right" (as from Figure 2.3 to Figure 2.4) means that there's a horizontal stride of 1. If W were to immediately move x cells to the right, it would have a horizontal stride of x. Similarly, if it were to immediately move x cells to the right and y

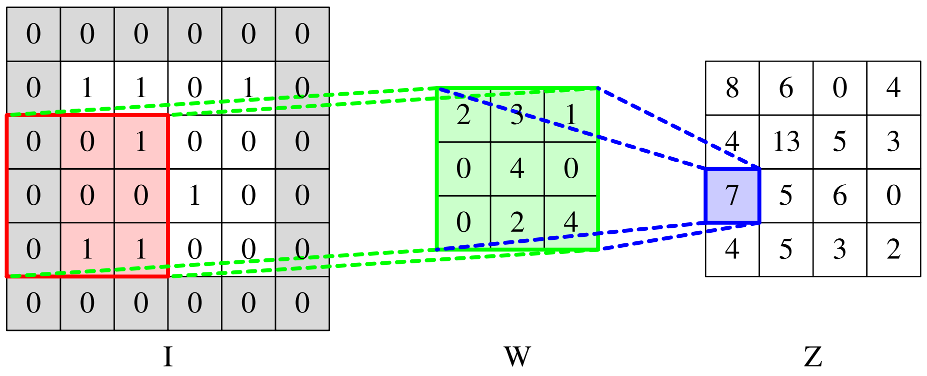

Figure 2.5: Calculating Z_{02} in a convolutional layer after adding a buffer of zeros to I.

cells down at a time, we would say that the filter has a stride of (x, y). If x and y are equal, we simply say that the filter has a stride of x. In general, for a pxp I with a qxq W and a stride of 1, we have

$$Z_{ab} = \sum_{i=0}^{q-1} \sum_{j=0}^{q-1} W_{ij} I_{(i+a)(j+b)} \quad \forall a, b \in [0, p-q] \cap \mathbb{Z} \ . \tag{2.4}$$

(Note that Equation 2.4 can be modified for non-square I, for non-square W, and for strides other than 1.)

Convolving with an Empty Buffer Around the Perimeter

Also of note is what can be done if the desired filter size doesn't match the desired dimensions of Z. In other words, for the example in Figures 2.3 and 2.4, a situation can occur where the user wants W to be 3x3 but Z to be 4x4. This can be done by adding a buffer of zeros to I, as in Figure 2.5. Aside from needing to add the buffer of zeros, this method works exactly like the general method, and, because only zeros are added, the only terms gained from this and added to the sums Z_{ij} are $W_{kl} * 0 = 0$.

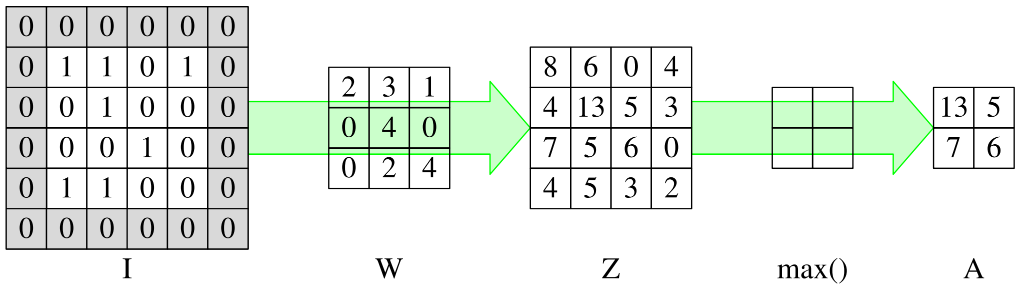

Figure 2.6: A simple example of a complete convolutional layer of a CNN.

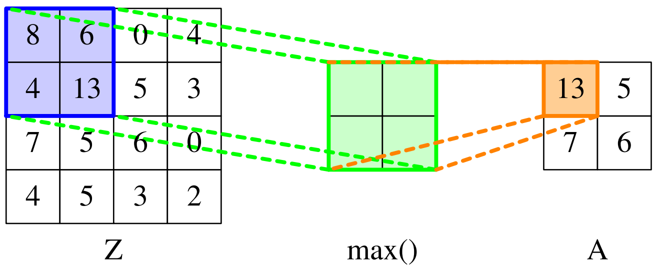

Figure 2.7: Calculating A_{00} in the pooling stage of a convolutional layer; thus, a pictoral representation of Equation 2.5 with $(a, b) = (0, 0)$.

2.1.3 Pooling

Figure 2.6 shows what a full convolutional layer might look like; notice that the left side of the Figure (the part involving I, W, and Z) shows the "convolutional" part, as has already been discussed. The rest of the Figure (the part that shows Z becoming A) shows "pooling," which is the step of a convolutional layer that shrinks the amount of data being sent to the next layer based on the idea that, by only passing along some of the information, overfitting can be prevented [7].

The pooling step by itself is shown in Figures 2.7 and 2.8. In this step, a pooling filter is applied to Z very similarly to how the convolutional filter is applied to I in the previous step, as explained in Section 2.1.2. There are two main differences: First, while convolutional filters often have a stride of 1, generally a pxq pooling filter will have a stride of (p, q) (as can be seen in Figures 2.7 and 2.8), and second, while a convolutional filter is a matrix of weights, a pooling filter is simply the designation of a region of Z over which to apply a

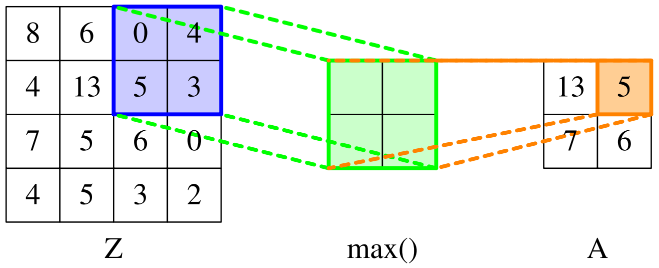

Figure 2.8: Calculating A_{10} in the pooling stage of a convolutional layer; thus, a pictoral representation of Equation 2.5 with $(a, b) = (1, 0)$.

pooling function; the output of that pooling function will then be the corresponding entry in A. A common pooling function is max(), which we use throughout this Section. Its usage is logical: To apply a pooling filter to a region of Z with a pooling function max(), take the maximum element of the region and let that be the element that is "kept" in A.

For cases such as this example that have a 2x2 filter with a stride of 2, we have

$$A_{ab} = \max(\{Z_{(2a)(2b)} \ , Z_{(2a+1)(2b)} \ , Z_{(2a)(2b+1)} \ , Z_{(2a+1)(2b+1)}\}) \ . \tag{2.5}$$

In general, for a pxq filter with a stride of (x, y) and a pooling function f(), we have

$$A_{ab} = \text{f}(\{Z_{ij} : (i, j) \in [ax, ax + q - 1] \cap \mathbb{Z} \text{ x } [by, by + p - 1] \cap \mathbb{Z}\}) \ . \tag{2.6}$$

2.1.4 Activation Function

In a DNN, synapses add charge to the neuron they are connecting. The charge added to the neuron is the product of the synapse weight and the neuron's firing charge. The neuron has to decide what charge to fire after it receives the charge from its incoming synapses. This decision is made by the activation function. The activation function is a mathematical equation that takes the total charge that the neuron receives from its synapse and outputs a firing charge. There are many kinds of activation functions exist.

2.2 Spiking Neural Network

A spiking neural network (SNN) is an artificial neural network mimics neural networks from biology [8]. In a SNN, every neuron has a threshold and a refractory period. As opposed to DNN, where neurons always transmit charge. In a SNN, the neuron has to accumulate charges that are larger than its threshold to fire. The neuron also has a rest time named ”refractory period” after each fire. The neuron cannot fire within the rest time even if it has enough total charge. Every synapse in a SNN has a weight and a delay. The weight value of the synapse is the same as in DNN. The delay of a synapse is the time required for the charge to travel from one end of the synapse to the other end. On a large scale, the SNN has a time factor which the DNN doesn't have.

2.3 TENNLab

TENNLab is a neuromorphic computing group at the University of Tennessee, Knoxville focuses on machine learning in SNN [9][10]; TENNLab has developed multiple network architectures and learning methods for SNNs, and they also have developed many applications that can be run by their method trained SNNs. This project uses TENNLab's classification application for classifying different datasets, but trains the SNNs using back propagation on a DNN, and then converts the DNN to a SNN. The following section briefly explains the application and architectures that are relevant to this project.

2.3.1 Classification Application

Class2d is an application from TENNLab for classifying different classes in a dataset [11]. The application can have a human or a neural network user. If the application is played by a human, it presents classification problems on the command line for the user to answer. If the application is played by a SNN, it connects the application with the network simulator for the simulator to answer problems. The data for each problem is converted to spikes with different amount of charge. This application is used only by a converted SNN in this project. In other words, the application is not involved in the training of the SNN.

2.3.2 Architectures

TENNLab has multiple architectures for SNNs. The architectures have different properties. For example, DANNA2 is a two-dimensional network architecture that uses integers to represent components properties such as neuron threshold and synapse weight [12]. NIDA is a three-dimensional network architecture, whose synapse delays are determined by the length of the synapses [13]. MrDANNA is an architecture whose properties are derived from its implementation using memristors [14]. The architectures that being used for this project will be detailed discussed in Chapter 5.

2.4 Whetstone

Whetstone is an algorithm that trains a conventional, non-spiking DNN to a spiking deep neural network (SDNN) by gradient descent. It first trains the network as a DNN using gradient descent for several iterations. After that, the algorithm gradually shrinks the valid range of the activation function. This shrinking process is called ”sharpening”. Eventually, the valid range of the activation function decreases to a discrete value, and this value becomes the neuron's threshold in the final SDNN. This process is described on detail in [1].

Chapter 3

Supply File

Due to the differences between the SDNN and SNN, the conversion program creates a supply file along with a converted network file from every original SDNN. The network file is straightforward: Is contains the converted SNN. The supply file includes extra information from the SDNN to make it work with the converted SNN. There are two kinds of data in every supply file. First is the input layer's synapse weights, and second is the variables for the Softmax function. The network is run by the TENNLab software framework and *class2d* application. We use the supply file to provide pro-processing of the input layer's synapse weights, which than get sent to the *class2d* application. The application's outputs are then post-processed using the variables for the Softmax function.

3.1 Softmax

Softmax is an activation function that is generally used for classification in a DNN. It is also what the SDNN use for classification. Softmax takes the spike activity as input and outputs a predicted classification. TENNLab's program can monitor the spike activity, but does not have the Softmax function built in to the program, so the conversion program from this project writes the Softmax variables to the supply file. The classification application reads the Softmax variable from the supply file and performs Softmax on the spike activity.

3.2 Input Layer Weight

In an SDNN, the input neurons don't need an activation function to determine the firing charge because the input value is their charge. Therefore, the Whetstone algorithm sharpens the activation function to a threshold for every neuron in the SDNN except the neurons in the input layer. This creates a problem when converting the SDNN to an SNN, because the SNN only has fixed thresholds for its neurons. To solve this problem, the conversion program writes the input layer neurons' synapse weights to the supply file, and the classification application uses this synapse weights to pre-process the inputs.

Chapter 4

Conversion

The conversion program is the primary focus of this project. This conversion program can convert different layer structured SDNNs to SNNs in different TENNLab architectures. It supports conversion of networks with feed forward layers, convolutional layers, and other layer structures. One network may be composed by multiple layer structures. While implementing the original conversion program, we encounter a major obstacle: the resulting SNNs were enormous. For that reason, we developed a second conversion that trades time for space, resulting in drastically smaller SNN that take longer to run.

4.1 Feed Forward Neural Networks

There is no need to rearrange neurons and synapses when converting FFNNs to SNNs, because the structure of the feed forward layer is simple. The only problem of converting a FFNN to an SNN is how to handle the bias in the FFNN, because neurons in SNNs only have a threshold and no bias. Every neuron in an SDNN produced by the Whetstone algorithm has the same threshold, due to sharpening, but each neuron has its own unique, trained bias. The charge in the bias is added to the neuron's charge along with synapse weights. Therefore, to convert a neuron in FFNN to a neuron in SNN, the conversion program assigns the neuron's threshold to be the original threshold (shared by all neurons) minus bias. In this way, each neuron has its own unique threshold that is a combination of the SDNN neurons' shared threshold and unique bias.

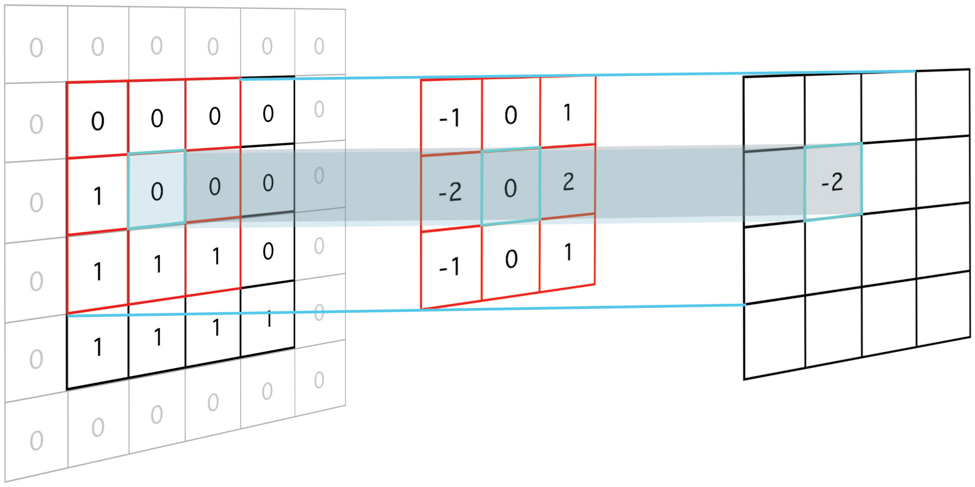

Figure 4.1: An example convolutional layer in SDNN

4.2 Convolutional Neural Networks

The CNN in SDNN is different from the normal CNN in DNN, Figure 4.1 is an example convolutional layer in SDNN. Every neuron in the SDNN has a firing charge of either 0 or 1, and this simplifies equation 2.4. This section describes the conversion of the CNN from SDNN to SNN, both the original version which results in excessively large network, and the second version which trades time for space.

4.3 Version 1

Version 1 of converting a CNN to an SNN is to connect every neuron in the output layer with every neuron in the input layer that contributes to its calculation. Figure 4.2 shows the synapse connection for a single neuron in the output layer in a convolutional layer with a 3x3 filter, and Figure 4.3 shows all of the synapses for this example. For a convolutional layer with n output neurons, and a pxq filter, the converted subnetwork has a synapse count of:

$$S = n * p * q \tag{4.1}$$

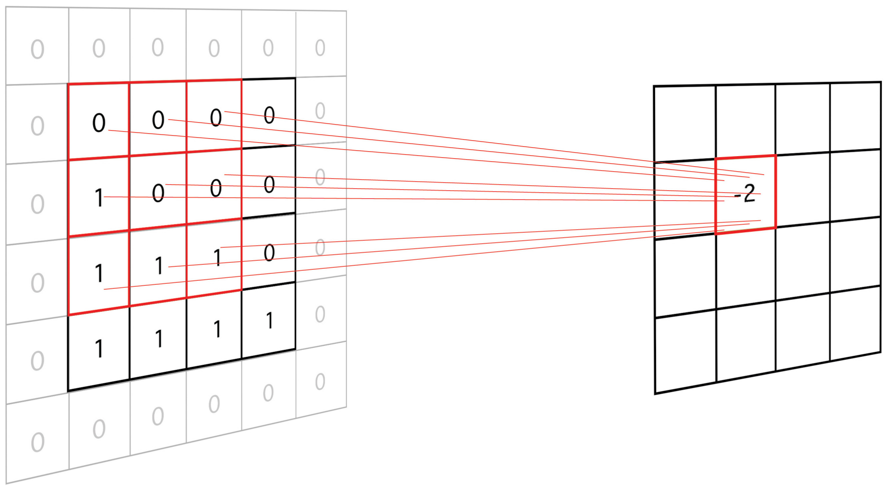

Figure 4.2: Synapse connection for a single neuron in the output channel in SNN

In the example of figure 4.2 from 4.3, this convolutional layer needs

$$4 * 4 * 3 * 3 = 144 \tag{4.2}$$

synapses. This synapse count can be overwhelming. For example, for a DNN from an example MNIST network, the total synapse count of all the converted convolutional layers was 1,143,072. As it turns out, this subnetwork is too large to store in RAM, so there needs to be another, more tractable conversion method.

4.4 Version 2

To solve this problem, we developed a method for what we're calling "trading time for space" that does just that: By using more timesteps and a more complex implementation, we reduce the synapse count to drastically raise the upper bound on network size that can be feasibly held in memory. In this implementation, for one convolutional layer with n neurons in each layer and a pxq filter, the synapse count is

$$S = O(n) + O(pq) \tag{4.3}$$

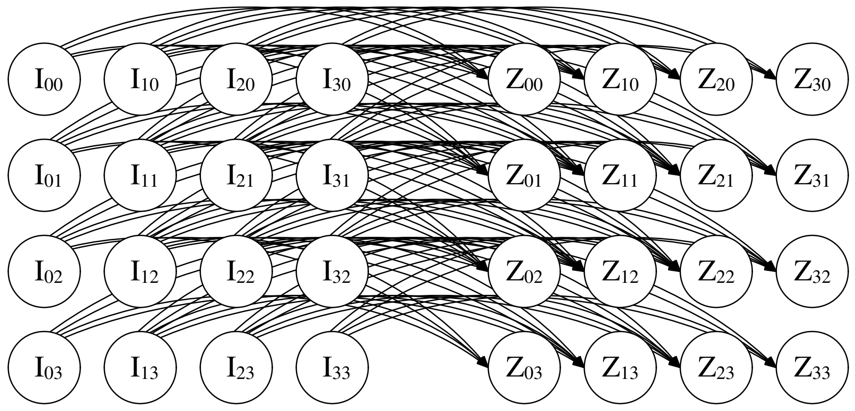

Figure 4.3: Synapse connection a 4x4 matrix with 3x3 filter in SNN

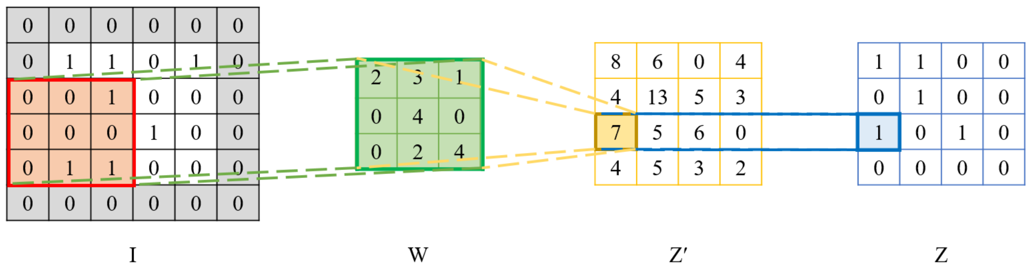

Figure 4.4: Convolution for an SNN. This is identical in form to the examples in Section 4.2 except for the fact that I and Z must be bitmatrices and for the threshold function. The threshold here is 6, so $Z'_{ij} \geq 6 \Leftrightarrow Z_{ij} = 1$.

rather than $O(npq)$ from equation 4.1. The number of additional neurons is also $O(n) + O(pq)$.

4.4.1 Method Overview

For a general idea of what version 2 looks like, compare Figure 4.4 with Figures 4.5-4.15.

Figure 4.4 shows a CNN layer's calculations when converted to a SDNN. The inputs I and output Z are constrained to be 0's and 1's, as they are implemented by spikes. The Z' matrix represents the original CNN calculations. It is converted to Z by applying a threshold, which is 6 in this example. In Figures 4.5-4.15, I is the bitmatrix (matrix composed only of zeros and ones) to the left, W is the matrix to the bottom left, and Z is the matrix that has yet to

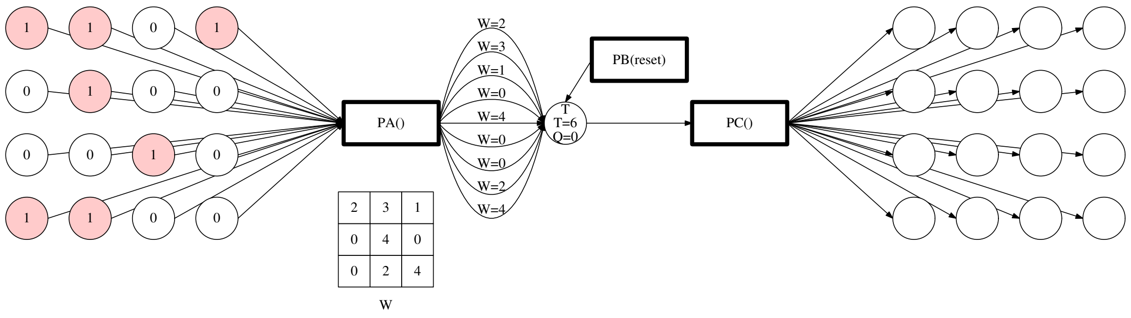

Figure 4.5: Time 0: All of the nonzero input neurons spike.

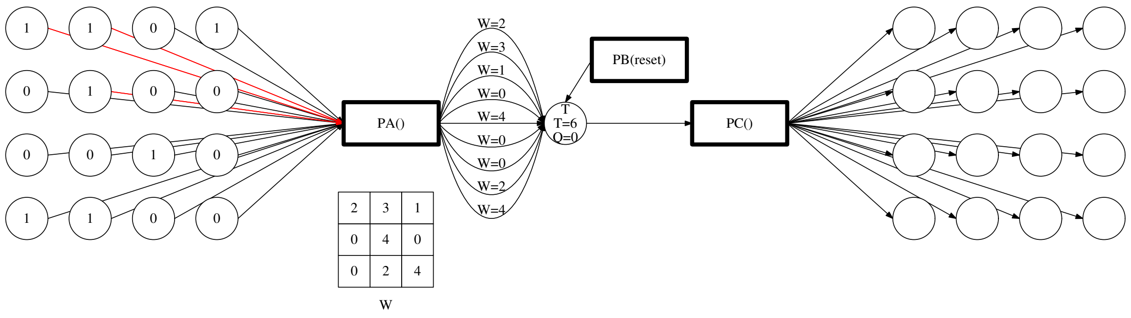

Figure 4.6: Time 1: Synapses from nonzero input neurons to PA() with delay 1 spike.

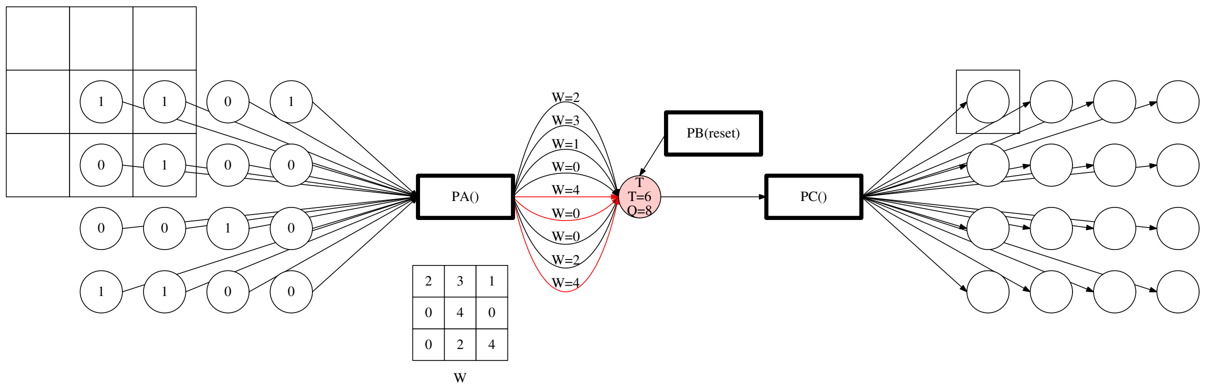

Figure 4.7: Time 2: Synapses from PA() to the threshold neuron corresponding to the weights that will sum to Z'_{00} spike. The charge of the threshold neuron now equals $Z'_{00} = 8$, which is greater than or equal to the threshold 6, so the threshold neuron spikes.

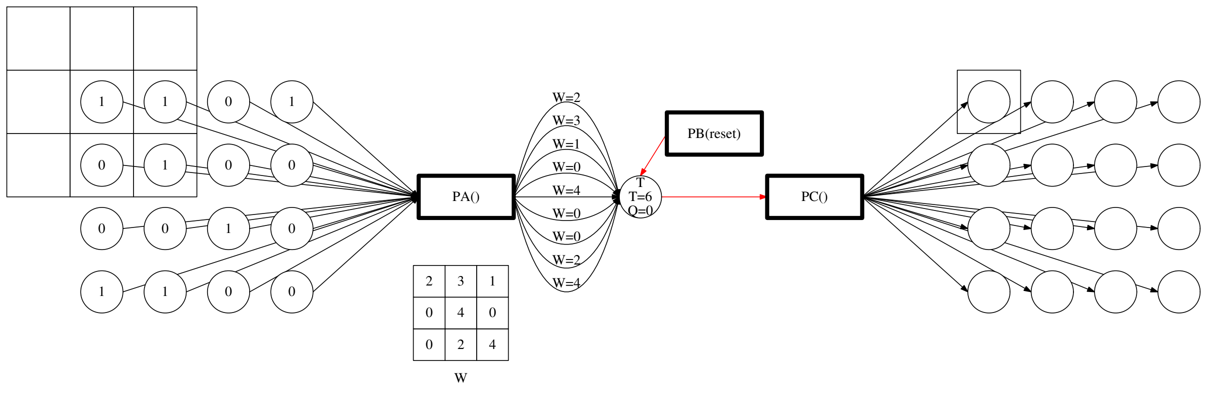

Figure 4.8: Time 3: Synapses from nonzero input neurons to PA() with delay 3 spike. The synapse from the threshold neuron to PC() fires because the threshold neuron fired in the last timestep, indicating that Z_{00} will be 1. The threshold neuron receives a spike of weight reset.

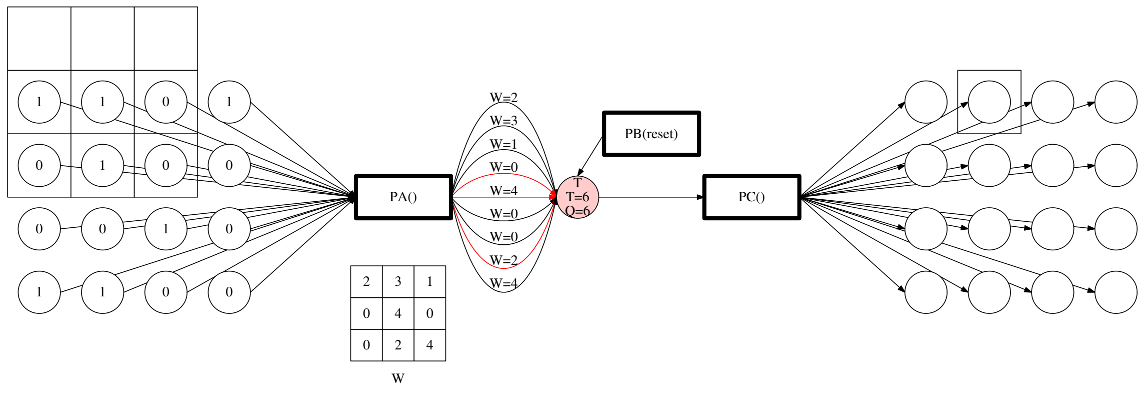

Figure 4.9: Time 4: Synapses from PA() to the threshold neuron corresponding to the weights that will sum to Z'_{10} spike. The charge of the threshold neuron now equals $Z'_{10} = 6$, which is greater than or equal to the threshold 6, so the threshold neuron spikes.

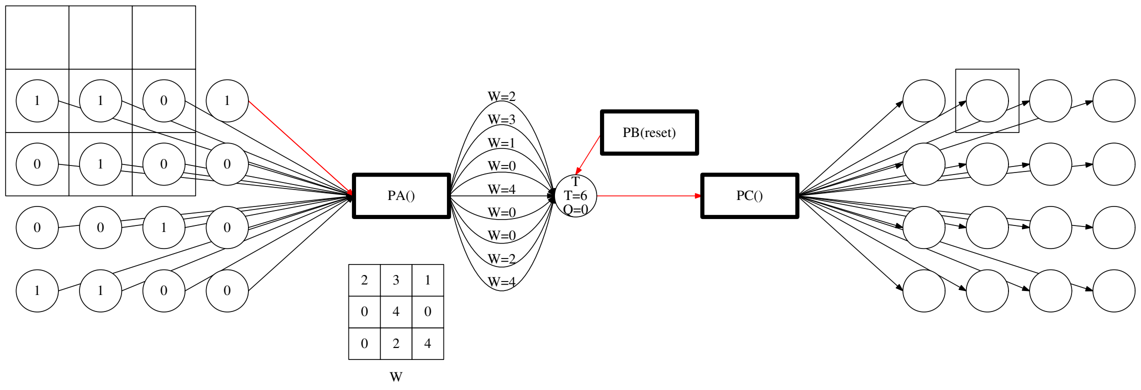

Figure 4.10: Time 5: Synapses from nonzero input neurons to PA() with delay 5 spike. The synapse from the threshold neuron to PC() fires because the threshold neuron fired in the last timestep, indicating that Z_{10} will be 1. The threshold neuron receives a spike of weight reset.

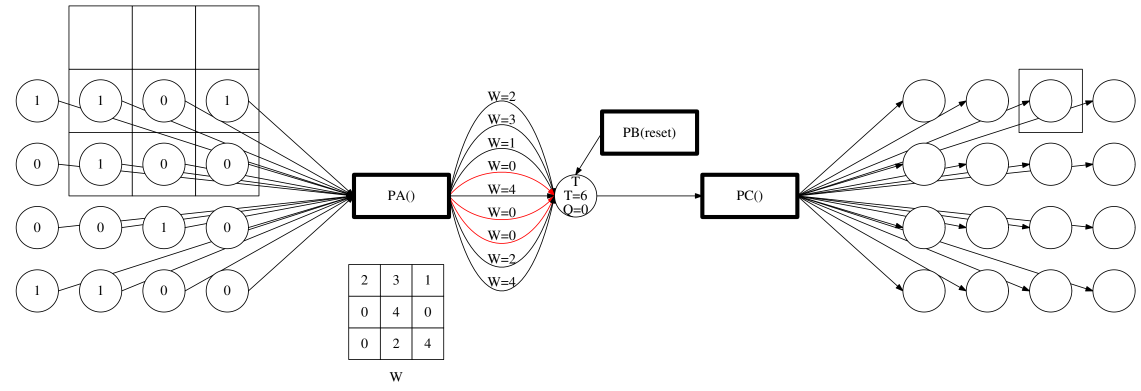

Figure 4.11: Time 6: Synapses from PA() to the threshold neuron corresponding to the weights that will sum to Z'_{20} spike. The charge of the threshold neuron now equals $Z'_{20} = 0$, which is not greater than or equal to the threshold 6, so the threshold neuron doesn't spike.

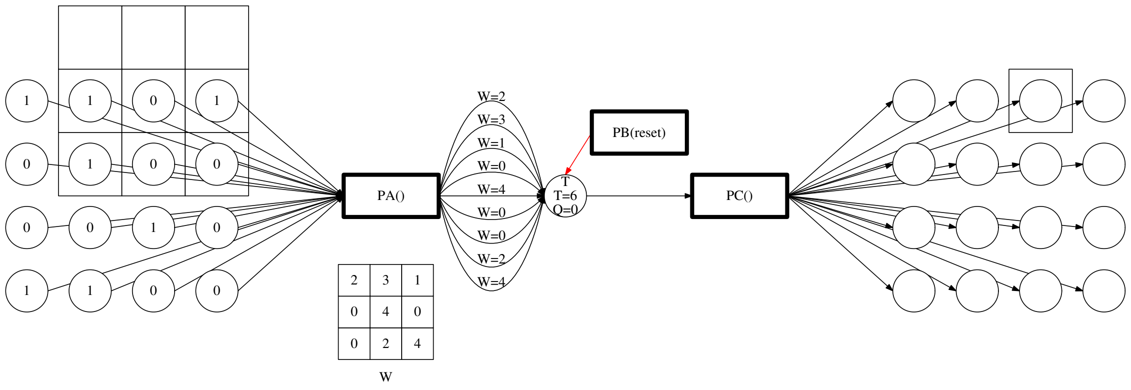

Figure 4.12: Time 7: Synapses from nonzero input neurons to PA() with delay 7 spike. The synapse from the threshold neuron to PC() doesn't fire because the threshold neuron didn't fire in the last timestep, indicating that Z_{20} will be 0. The threshold neuron receives a spike of weight reset.

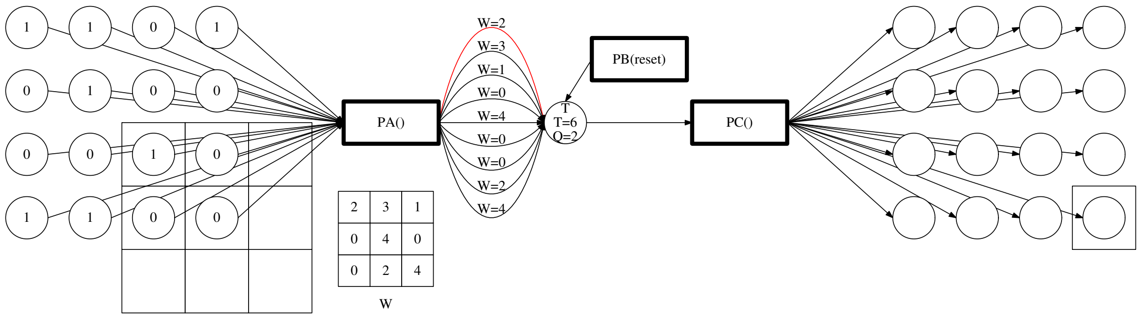

Figure 4.13: Time 32: Synapses from PA() to the threshold neuron corresponding to the weights that will sum to Z'_{33} spike. The charge of the threshold neuron now equals $Z'_{33} = 2$, which is not greater than or equal to the threshold 6, so the threshold neuron doesn't spike.

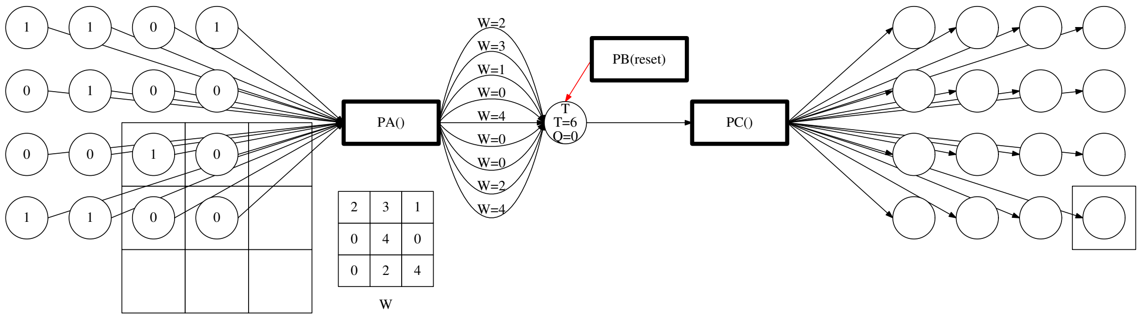

Figure 4.14: Time 33: There are no synapses from nonzero input neurons to PA() that haven't spiked yet. The synapse from the threshold neuron to PC() doesn't fire because the threshold neuron didn't fire in the last timestep, indicating that Z_{33} will be 0. The threshold neuron receives a spike of weight reset.

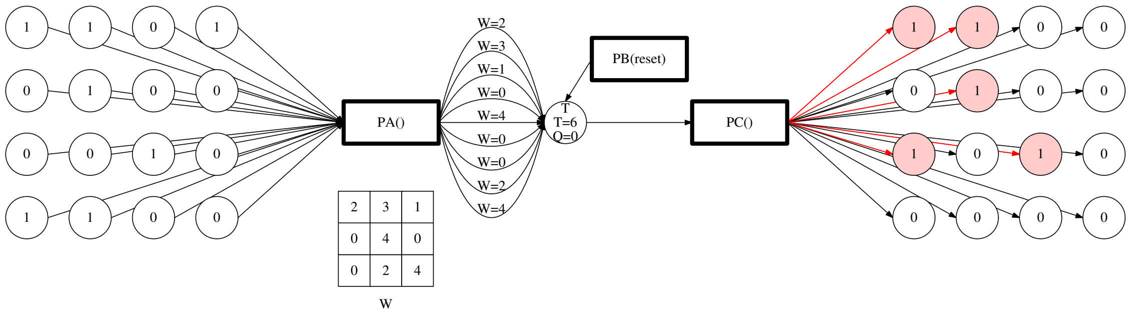

Figure 4.15: Time 37: Synapses from PC() to an output neuron that should be a 1 spike, causing the appropriate output neurons to fire. This completes this convolutional layer.

be filled in to the very right of the Figures. (See the top of Section 2.1.2 for an explanation of the variables used here. Also, note that everything in Section 4.4 involves only the actual convolution part of a convolutional layer and leaves out pooling.) This Section uses 4x4 I, 3x3 W, and 4x4 Z; some of the parameters to the subnetworks depend on these dimensions.

We'll hold off on explaining the subnetworks PA(), PB(), and PC() for now; just think of them as "black boxes" until Section 4.4.2. (They should actually be styled with their parameter lists as PA(n, w, i, d), PB(i, d, w, c, o), and PC(n, i, d), but, again, we're leaving that until Section 4.4.2.)

Notice that all of the (non-buffer) entries of I are neurons, each with a synapse connected to PA(); each of these synapses has weight 1 and delay 1. There's also a synapse from PA() to the threshold neuron $T = t$ corresponding to each element of W, each with weight equal to the value of the W_{ij} that it corresponds to and with delay 1. There's a single synapse from PB(reset) to the threshold neuron with weight reset and delay 1. (The meaning of weight reset will be discussed in 4.4.2; for now, just know that a synapse firing with a weight of reset will zero out the charge of the neuron that receives it.) There's a single synapse from the threshold neuron to PC() with weight 1 and delay 1, and, lastly, for all Z_{ij}, there's a synapse from PC() to Z_{ij} with weight 1 and delay 1.

At time 0, the neurons in I fire. This is the point where the method described in this Section starts to take more time: To reduce the amount of data needed to be held in memory, instead of finding all elements of Z' at once (as in Figure 4.4), we take one timestep for each Z'_{ij} (the even timesteps where the synapses corresponding to W fire) and a second timestep for each corresponding Z_{ij} (the odd timesteps where the synapse from the threshold neuron to PC() fires).

Example: Z'_{00} and Z_{00}

Time 2 corresponds to Z'_{00} and time 3 corresponds to Z_{00} (see Figures 4.7-4.8). At time 2, all synapses corresponding to elements of W that are multiplied by a 1 from I to calculate Z'_{00} fire. (Note that the original zeros in this equation come from the "buffer" of zeros around

I.)

$$\begin{aligned} Z'_{00} = & \quad W_{00} * 0 + W_{10} * 0 + W_{20} * 0 \\ & + W_{01} * 0 + W_{11} * I_{00} + W_{21} * I_{10} \\ & + W_{02} * 0 + W_{12} * I_{01} + W_{22} * I_{11} \\ = & \quad 0 + 0 + 0 \\ & + 0 + W_{11} * 1 + W_{21} * 1 \\ & + 0 + W_{12} * 0 + W_{22} * 1 \\ = & \quad W_{11} + W_{21} + W_{22} \\ = & \quad 4 + 0 + 4 \\ = & \quad 8 \ . \end{aligned} \tag{4.4}$$

Thus, the synapses corresponding to W_{11}, W_{21}, and W_{22} are the only weight synapses that fire, as can be seen in Figure 4.7. At this point, the Z'_{00} is known to be the charge of the threshold neuron, with is 8.

At time 3, if $Z'_{00} \geq T$, the synapse from the threshold neuron to PC() will fire. In the example we've been using, the threshold is 6. Since $8 \geq 6$, the synapse to PC() does fire; thus, at time 37 (Figure 4.15), the synapse from PC() to Z_{00} will fire, indicating that $Z_{00} = 1$.

The synapse from PB(reset) to the the threshold neuron also fires at time 3 to clear out the threshold neuron in case it didn't fire. (If it did fire, this wouldn't be necessary because firing will zero out a neuron).

Note the agreement between Z' and Z here and Z' and Z in Figure 4.4.

General Timestep Expressions

Let Z be a pxq matrix. For our system of counting timesteps where time 0 is the timestep when the neurons of I fire, we have the following:

- The synapses from PA() to the threshold neuron corresponding to the weights that will be summed to calculate Z'_{ij} fire at time $2(jq + i) + 2$.

- If Z_{ij} should be 1, the synapse from the threshold neuron to PC() fires at time $2(jq + i) + 3$.

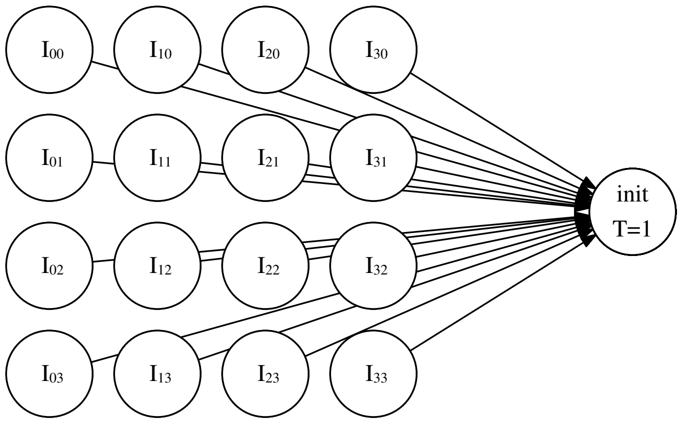

Figure 4.16: The init neuron. Each neuron of I has a synapse to init with weight 1 and delay 1.

- For all Z_{ij} that should be 1, the synapses from PC() to Z_{ij} fire at time $2pq + 5$.
- The synapse from PB(reset) to the threshold neuron fires at time $1+2k \ \ \forall k \in \mathbb{Z} : k > 0$.

4.4.2 Subnetwork Details

Init Neuron

The init neuron (Figure 4.16) is a neuron local to each layer that has synapses to all PB() and to a few other neurons (discussed later), causing them to fire at certain times. To ensure that init fires in each layer, there is a synapse from every neuron of I to init. This way, if any input neuron fires, init will fire; if no input neuron spikes, there is no reason for init to fire. Assuming init fires, it will fire at time 1.

PB(i, d, w, c, o)

The PB(i, d, w, c, o) subnetwork is the simplest of the subnetworks presented here. It's composed of three neurons, as in Figure 4.17, and produces a spike of weight w at time $ij + d \ \ \forall j \in [1, c] \cap \mathbb{Z}$. Each PB$(i, d, w, c, o)$ has a synapse of weight 1 from the init neuron to its own B_0 and a synapse of weight -1 from the init neuron to both B_0 and B_1. The

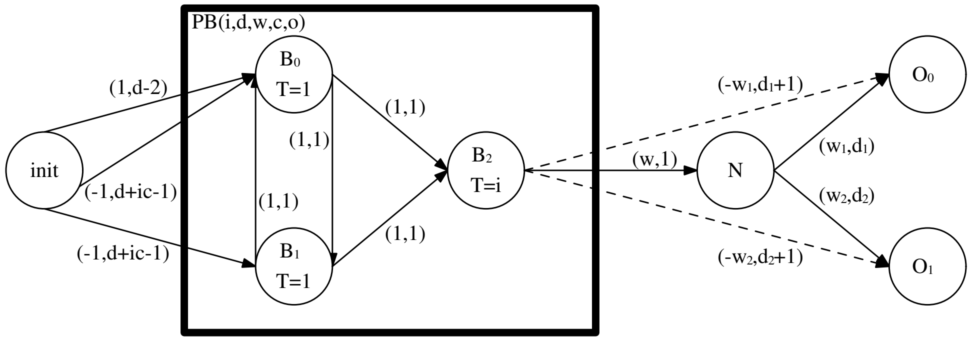

Figure 4.17: PB(i, d, w, c, o) produces a spike of weight w at time $ij + d \;\; \forall j \in [1, c] \cap \mathbb{Z}$. Synapse weights and delays are written as (weight,delay).

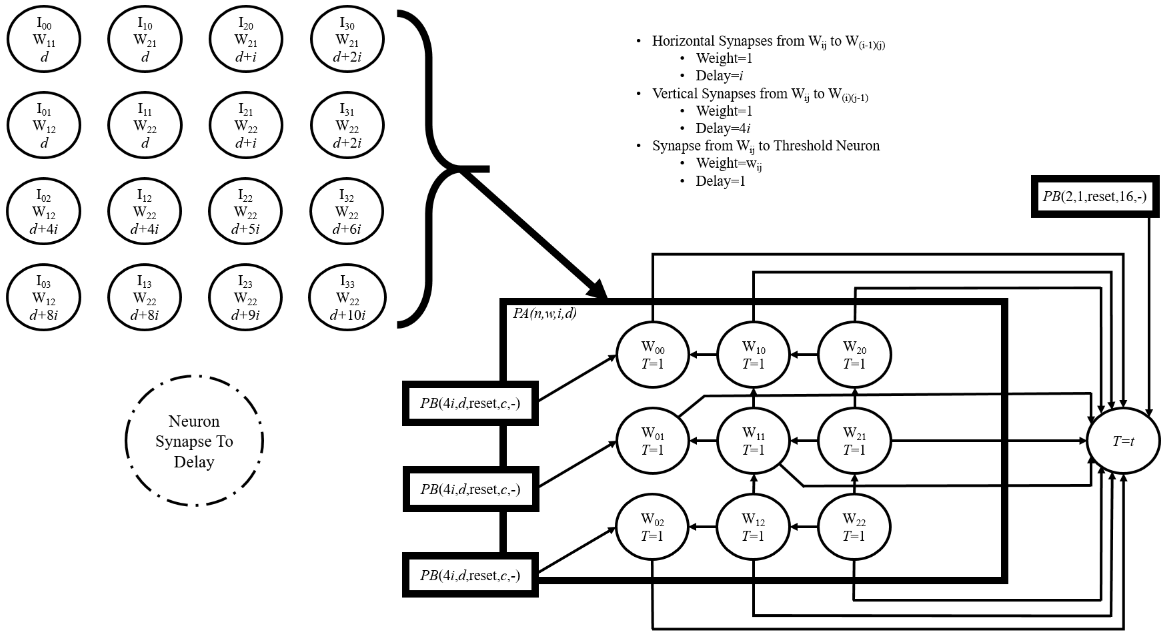

Figure 4.18: PA(n, w, i, d) has n input synapses of weight 1 and w output synapses whose weights correspond to the weights of W. The parameters i and d are used for timing. In this case, $n = (4, 4)$, $w = (3, 3)$, $i = 2$, and $d = 1$.

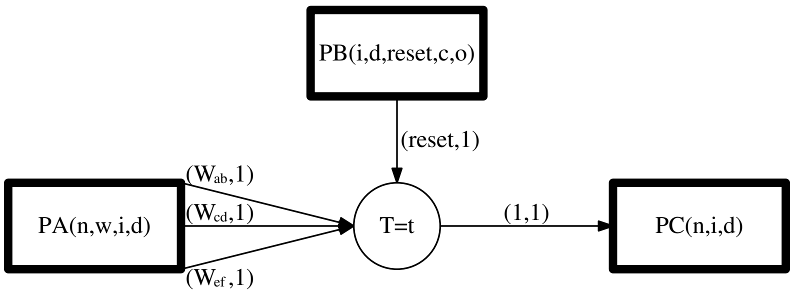

Figure 4.19: The threshold neuron. Each W_{ij} of PA(n, w, i, d) has a synapse to the threshold neuron, which then has one synapse to PC(n, i, d). There is also a PB(reset) with a synapse to the threshold neuron to zero it out after every spike from PA(n, w, i, d). Synapse weights and delays are written as (weight,delay).

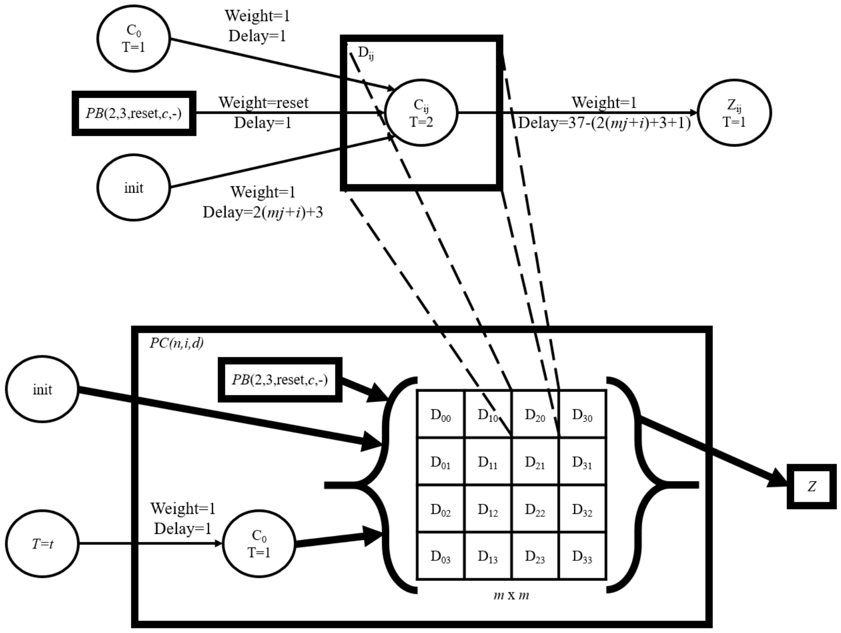

Figure 4.20: PC(n, i, d) has a synapse from the threshold neuron to C_0 and has n output synapses to Z. The parameters i and d are used for timing. In this case, $n = (4, 4)$, $i = 2$, and $d = 1$.

synapses of weight -1 are there to ensure that PB(i, d, w, c, o) stops producing spikes when they're no longer needed, which makes each PB(i, d, w, c, o) significantly more efficient.

What has been referred to in this Thesis as PB(reset) is really PB(i, d, w, c, o) with $w = \textbf{reset}$, which, in our implementation, is simply a "very large number." If $w = \textbf{reset}$, then that will cause the neuron N to fire, thus "resetting" it. To cancel out the consequences of N having to fire to be reset, the parameter o contains a list of neurons that N has a synapse to and the weights and delays of those synapses. PB(i, d, w, c, o) will then contain a synapse to every neuron that N has a synapse to, but with a delay of one more and with negative weight. Thus, when N fires to be reset, it triggers synapses that shouldn't be triggered, but they won't actually affect anything since at the timestep that those synapses fire with weight w_1 to a neuron, the neuron also receives a pulse of weight $-w_1$. Note that, outside of this subsection, this Thesis doesn't mention the synapses from PB(reset) to the neurons in o.

PA(n, w, i, d)

The first subnetwork that the input neurons are connected to is PA(n, w, i, d) (see Figure 4.18). It's in this subnetwork that the "trading time for space" begins to happen: Whereas, without this method, we would need a lot of synapses at once, with our implementation, we simply take two timesteps apiece for each Z_{ij}.

At time 0, all I_{ij} fire. Each I_{ij} is connected to a synapse whose weight and delay are shown inside I_{ij}'s neuron, in the top left of Figure 4.18. In this case, our parameters to PA(n, w, i, d) are PA$((4, 4), (3, 3), 2, 1)$, so at time 1, neurons I_{00}, I_{10}, I_{01}, and I_{11} have their spikes reach PA(n, w, i, d). This corresponds to the top-left 3x3 submatrix of I surrounded by a buffer of zeros. When a W_{ij} spikes, it sends a spike of weight W_{ij} to the threshold neuron. If the sum of the spiking W_{ij} is greater than or equal to the threshold, then the threshold neuron will send a spike to PC(n, i, d); otherwise, the threshold neuron is zeroed out by PB(reset). Then, at time 3, the spikes from I_{20} and I_{21} reach PA(n, w, i, d) as the spikes which were already there move to the left one. This corresponds to the next 3x3 submatrix of I surrounded by a buffer of zeros. This continues; see Figures 4.21-4.30 for a more detailed walkthrough.

Threshold Neuron

The threshold neuron (as shown in Figure 4.19) is the equivalent in this method of going from Z'_{ij} to Z_{ij} (as in Figure 4.4). If Z'_{ij} is greater than or equal to the threshold t, then Z_{ij} should be 1; this happen by the threshold neuron sending a spike to PC(n, i, d). However, if Z'_{ij} is less than the threshold, then Z_{ij} should be 0, which will be forced by the *lack* of a spike from the threshold neuron to PC(n, i, d).

To ensure that the threshold neuron doesn't spike because of a build-up of charge that doesn't on its own cause it to spike, PB(reset) will zero out the threshold neuron after every timestep where it might spike.

PC(n, i, d)

The final subnetwork of our convolutional layer is PC(n, i, d), as shown in Figure 4.20. This subnetwork is responsible for catching when Z_{ij} should be 0 or 1. The first timestep when C_0 can spike is time 3, which corresponds to Z_{00}. (In other words, if Z_{00} should be a 1 if and only if C_0 spikes at time 3.) If C_0 does spike at time 3, then all C_{ij} will receive a charge of 1 at time 4. The way that we ensure that only the appropriate neuron fires is by making each C_{ij} have a threshold of 2 and giving each C_{ij} a synapse from init with delay $2(mj + i) + 3$. For C_{00}, this synapse will fire at time $4 = 2(0) + 3$ plus another 1 to include the delay from I_{ij} to init. This will ensure that, if C_0 fires at time 3, C_{00} will have a charge of 2 at time 4, causing it to fire. At time 5, a host of PB(reset)s zero out all C_{ij}.

The delay from C_{ij} to Z_{ij} is to ensure that all Z_{ij} spike at the same time. The "37" comes from

$$\begin{aligned}
37 &= (\text{size}(Z')) + (\text{size}(Z)) + (I \text{ spike time}) + (C_{ij} \text{ spike time}) + (C_0 \text{ to } C_{33}) + (\text{final delay}) \\
&= 16 + 16 + 1 + 1 + 1 + 2 \ .
\end{aligned} \tag{4.5}$$

Thus, for 2-dimensional I, 2-dimensional W, and 4x4 Z, this method will require 37 timesteps to complete the layer; at time 37, all relevant Z_{ij} will fire.

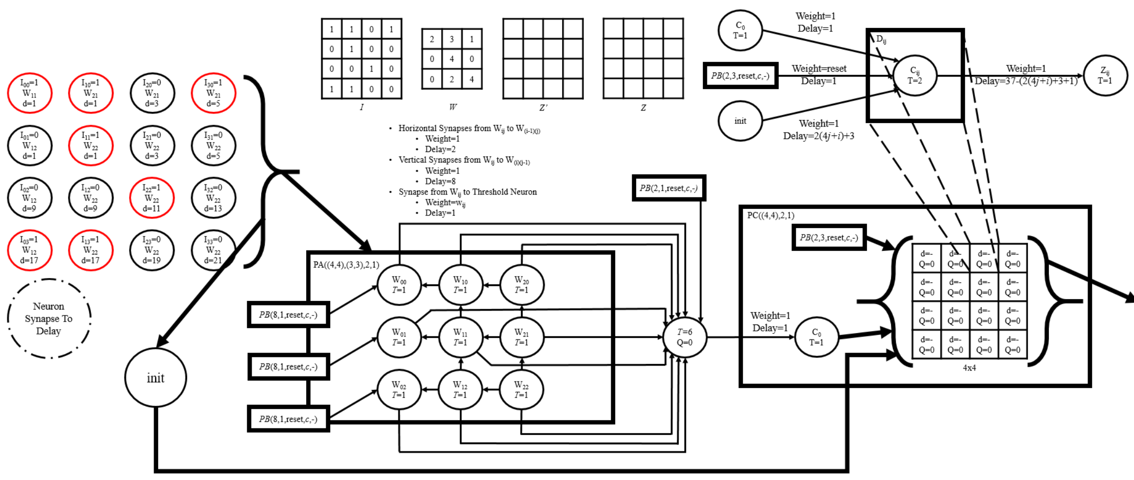

Figure 4.21: Time 0: All of the nonzero input neurons spike.

4.4.3 The Complete Picture

A step-by-step walkthrough of the first ten timesteps of the example from Figure 4.4 is shown in Figures 4.21-4.30. (This walkthrough is significantly more detailed than that given in Figures 4.5-4.15.)

Notice that we don't show the inner parts of any PB(i, d, w, c, o), nor do we show the synapses to them from init. Since PB(i, d, w, c, o) is so comparatively simple, we felt showing the spikes moving around in it to be unnecessary. Refer back to Section 4.4.2 if it doesn't quite make sense.

4.5 Max-pooling

Conversion of the max-pooling layer to SNN is really straightforward. In SDNNs, max-pooling sends 1 to the next layer if and only if there is at lease one active neuron in the selected region. The conversion program assigns a threshold of 1 to each neuron in the output layer. It connects each of these with a synapse to a pre-neuron in the input layer that is part of its output calculation. Each of these synapse have weights and delays of one. In this way, if any neuron in the region fires, the synapse is going to add 1 to the output neuron and cause the neuron to fire.

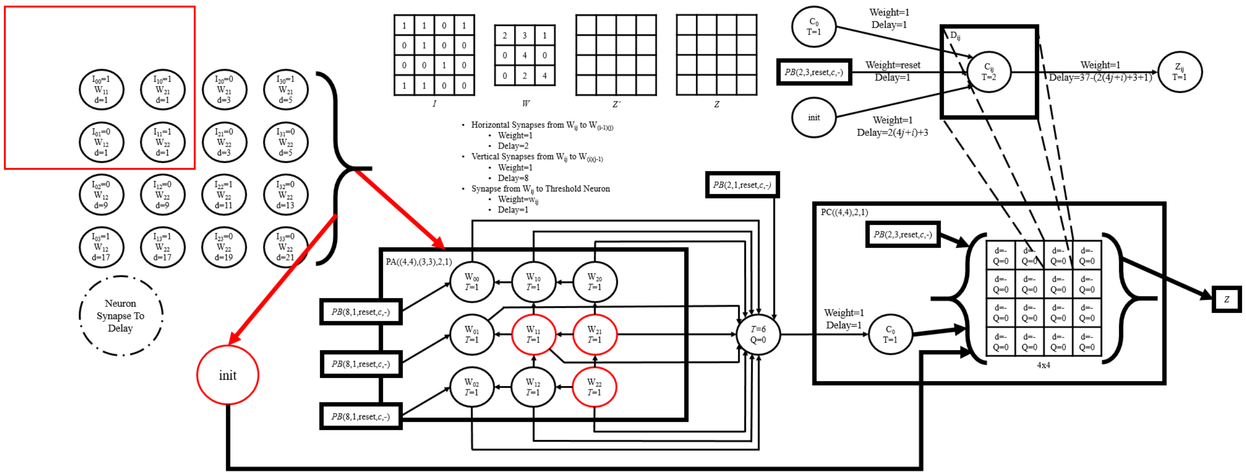

Figure 4.22: Time 1: Synapses from nonzero input neurons to PA() with delay 1 spike.

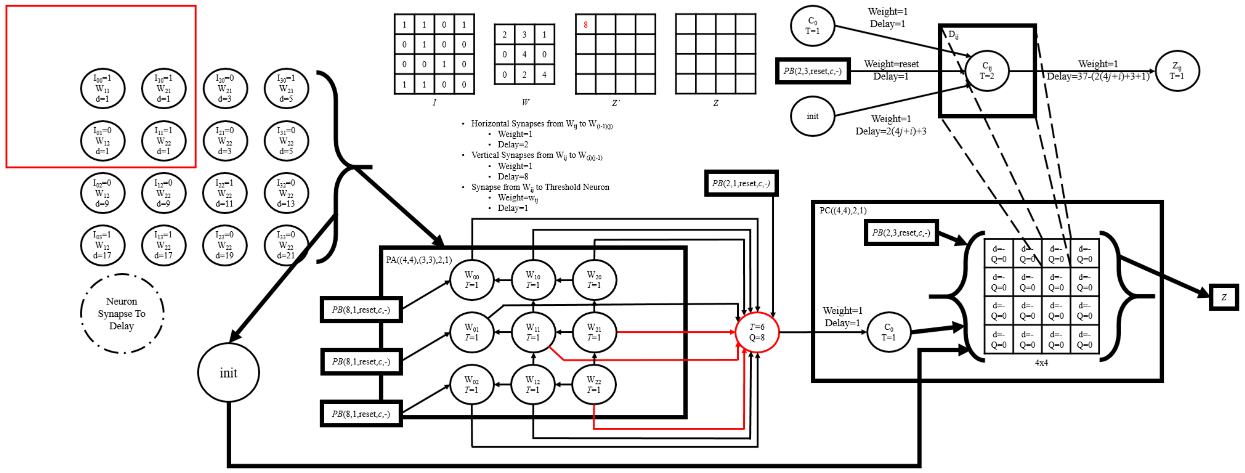

Figure 4.23: Time 2: Synapses from PA() to the threshold neuron corresponding to the weights that will sum to Z'_{00} spike. The charge of the threshold neuron now equals $Z'_{00} = 8$, which is greater than or equal to the threshold 6, so the threshold neuron spikes.

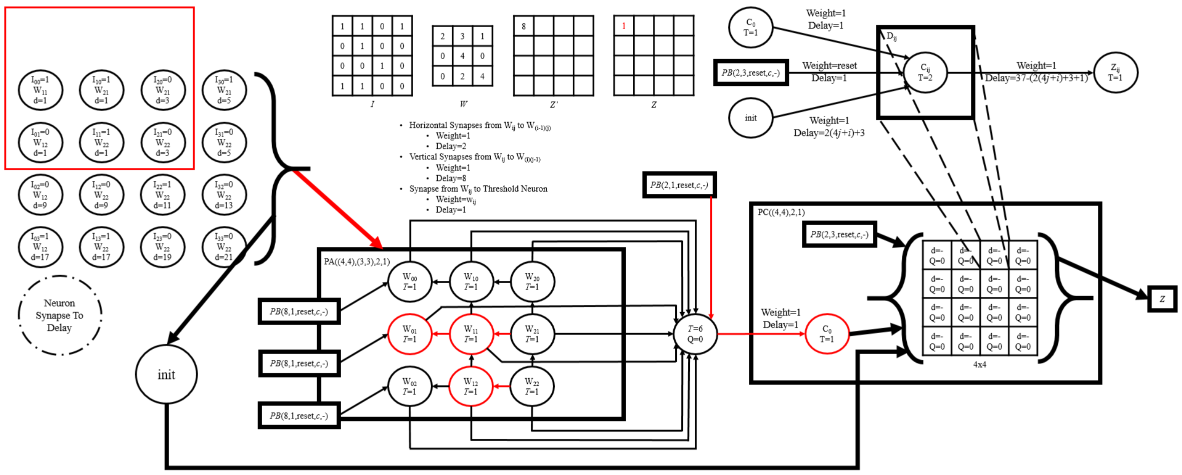
Figure 4.24: Time 3: Synapses from nonzero input neurons to PA() with delay 3 spike. The synapse from the threshold neuron to PC() fires because the threshold neuron fired in the last timestep, indicating that Z_{00} will be 1. The threshold neuron receives a spike of weight reset.

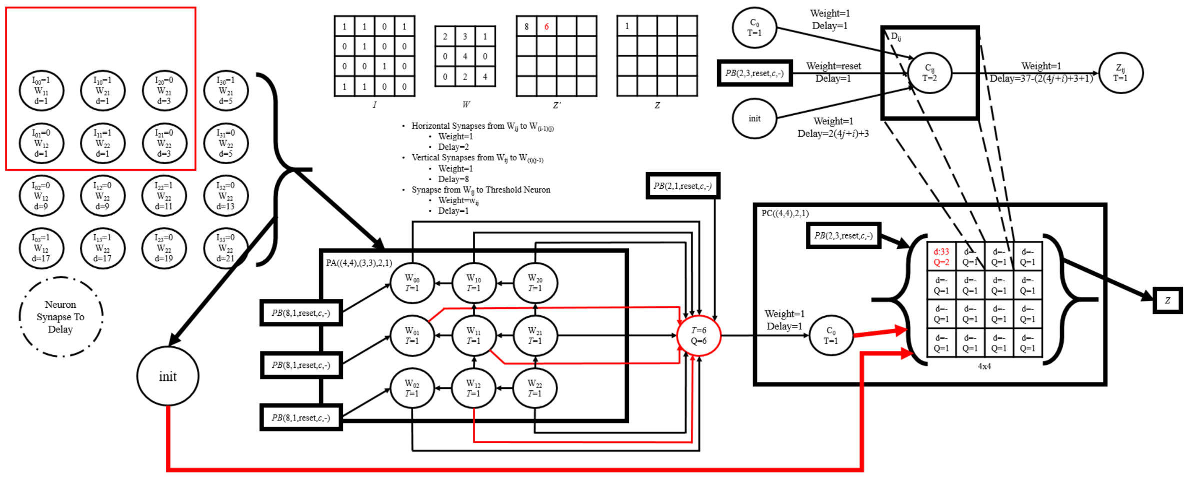
Figure 4.25: Time 4: Synapses from PA() to the threshold neuron corresponding to the weights that will sum to Z'_{10} spike. The charge of the threshold neuron now equals $Z'_{10} = 6$, which is greater than or equal to the threshold 6, so the threshold neuron spikes.

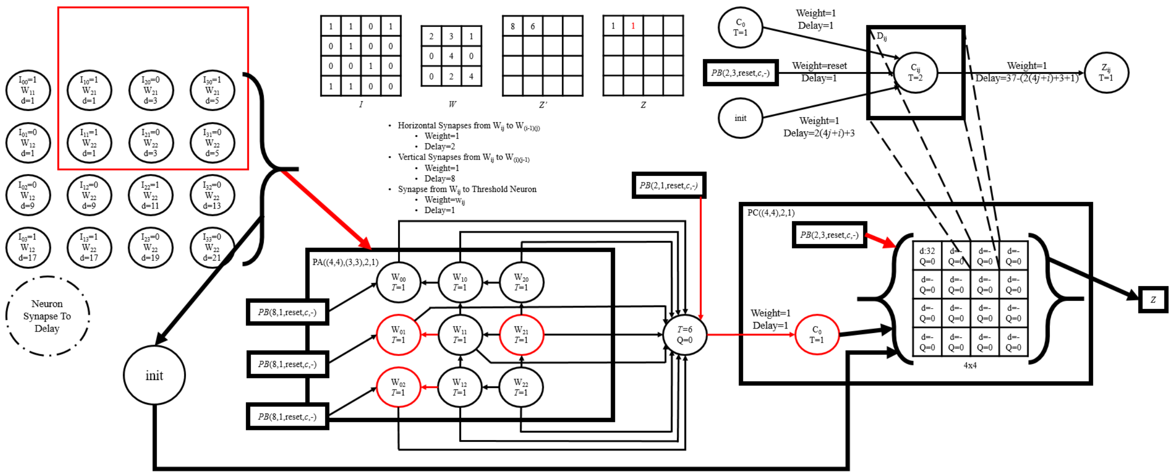

Figure 4.26: Time 5: Synapses from nonzero input neurons to PA() with delay 5 spike. The synapse from the threshold neuron to PC() fires because the threshold neuron fired in the last timestep, indicating that Z_{10} will be 1. The threshold neuron receives a spike of weight reset.

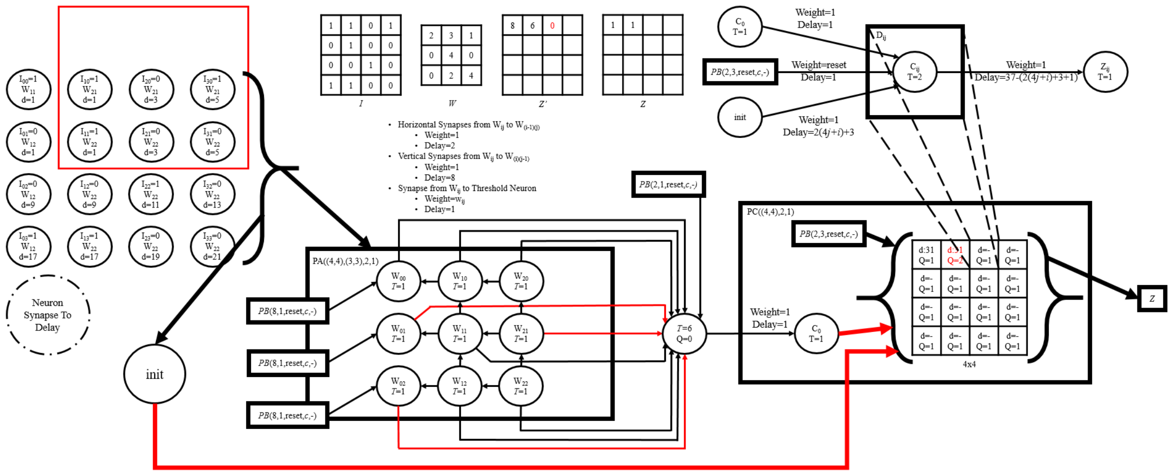

Figure 4.27: Time 6: Synapses from PA() to the threshold neuron corresponding to the weights that will sum to Z'_{20} spike. The charge of the threshold neuron now equals $Z'_{20} = 0$, which is not greater than or equal to the threshold 6, so the threshold neuron doesn't spike.

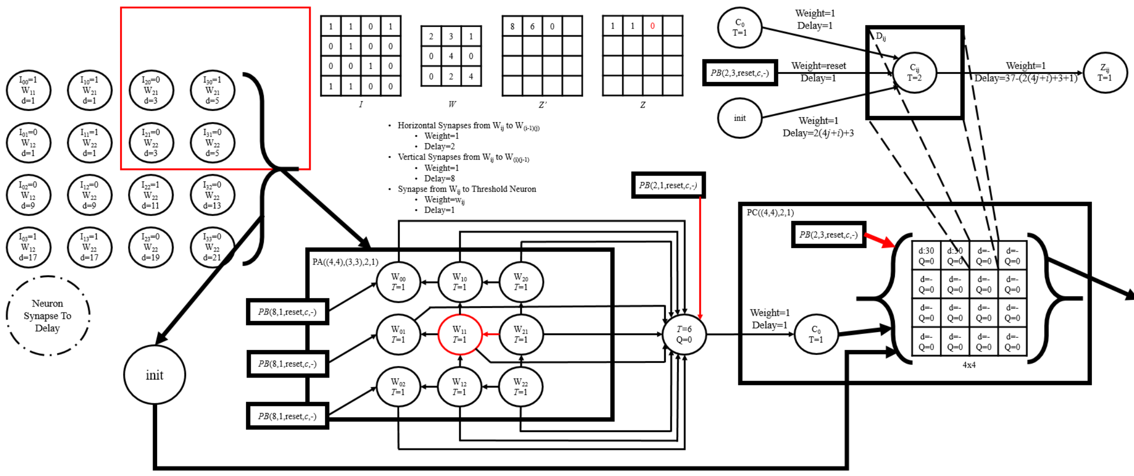

Figure 4.28: Time 7: Synapses from nonzero input neurons to PA() with delay 7 spike. The synapse from the threshold neuron to PC() doesn't fire because the threshold neuron didn't fire in the last timestep, indicating that Z_{20} will be 0. The threshold neuron receives a spike of weight reset.

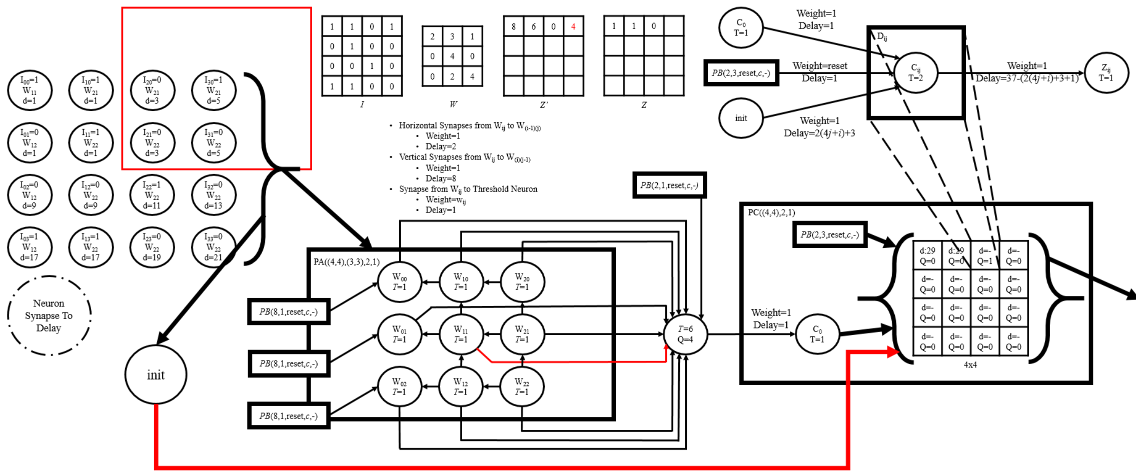

Figure 4.29: Time 8: Synapses from PA() to the threshold neuron corresponding to the weights that will sum to Z'_{30} spike. The charge of the threshold neuron now equals $Z'_{30} = 4$, which is not greater than or equal to the threshold 6, so the threshold neuron doesn't spike.

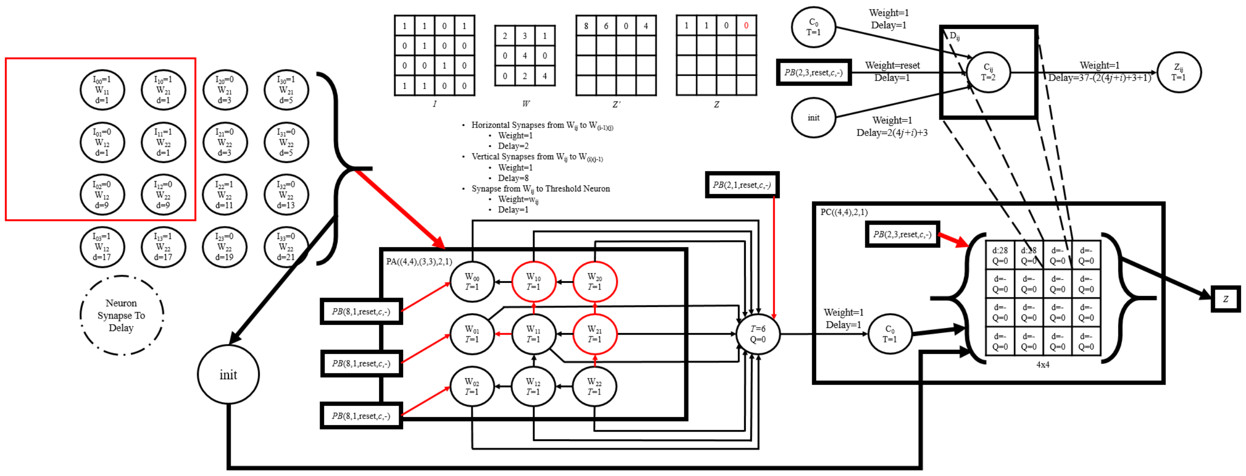

Figure 4.30: Time 9: Synapses from nonzero input neurons to PA() with delay 9 spike. The synapse from the threshold neuron to PC() doesn't fire because the threshold neuron didn't fire in the last timestep, indicating that Z_{30} will be 0. The threshold neuron receives a spike of weight reset.

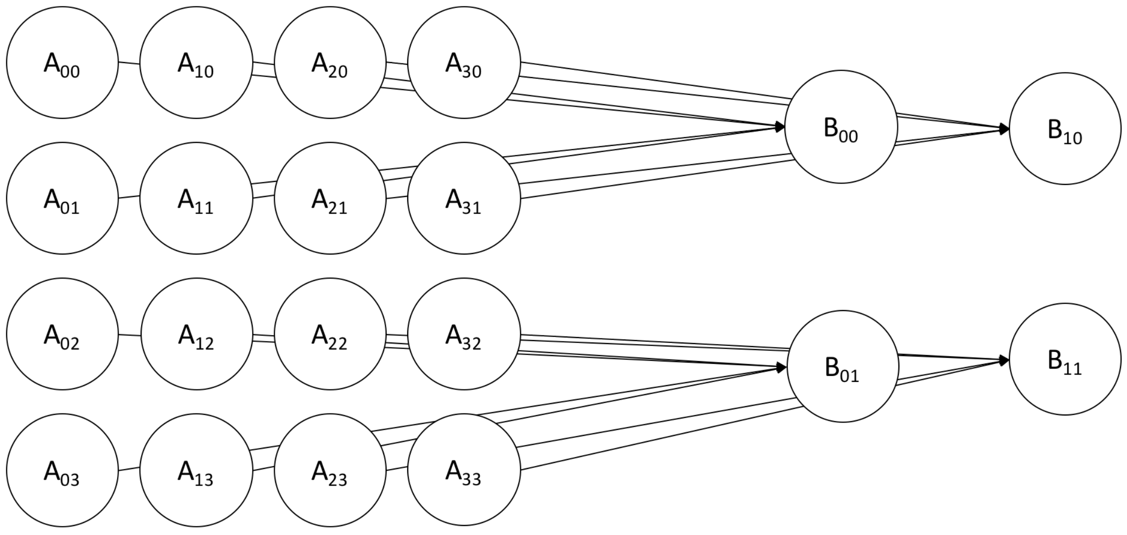

Figure 4.31: A max-pooling layer with a stride of 2 on a 4x4 matrix. All synapses have weights and delays of one. The B_{ij} neurons all have thresholds of one.

Chapter 5

Spiking Architecture

The conversion program converts the SDNN to an SNN in the TENNLab framework. Every spiking architecture has a simulator that can be run by a computer, but some architectures also have a hardware implementation that allows the network to run by a piece of hardware. TENNLab has many network architectures for SNN, and this conversion program supports two architectures: WHETSTONE and DANNA2. Details of those two architectures is explained in the following sections.

5.1 WHETSTONE

WHETSTONE is a three-dimensional spiking architecture that we created in TENNLab, which aims to match the original SDNN behavior spike by spike. TENNLab has many network architectures, but none of them matches the original SDNN 100%. Since one goal of this project is to prove that the converted networks match the original networks, WHETSTONE was created within the TENNLab framework. WHETSTONE stores network elements in floating point, and the WHETSTONE simulator processes one layer every one timestep to implement the time factor in SNN. As the result, the converted WHETSTONE network shows 100% match on spike activity with the original SDNN, and this architecture proves that TENNLab's spiking architecture can mimic the SDNN behavior.

5.2 DANNA2

DANNA2 is a two-dimensional spiking architecture in TENNLab with software, FPGA, and VLSI implementations. Although it is not identical to the WHETSTONE architecture, we converted Whetstone SNNs to DANNA2s, to see how well they would perform. DANNA2 has a hardware constraint that forces network to be represented as integers instead of floating-point. Therefore, the original network has to be reduced to two-dimensions and converted it to a integer network. We performed these conversions, and performed experiments on both DANNA2 and WHETSTONE. The results are presented in Chapter 6.

Chapter 6

Benchmark and Result

The benchmark tests are split into three parts. The first test is on the effectiveness of version 2 for converting CNNs to SNNs. The second test is on the accuracy of the original SDNNs and converted SNNs in DANNA2. The last test is on the spike rate and event rate. All three tests are done on twelve different CNN or FFNN topologies. The twelve topologies are the cross combination of four network depths and three filter sizes. We name these depths "deep" (6 convolutional layers and 2 feed forward layers), "medium" (4 convolutional layers and 2 feed forward layers), "shallow" (2 convolutional layers and 2 feed forward layers), and "dense" (2 feed forward layers). The filter sizes are 3x3, 5x5, and 7x7.

6.1 Network Size

We first compare the neuron and synapse counts for version 1 and version 2 in the twelve MNIST networks. This provides a direct way of showing how version 2 improves the network size compare with version 1. Figure 6.1 shows the total neuron and synapse count. The dense network is not shown because it doesn't have convolutional layer. It is very obvious that the difference on synapse count is much larger than the difference on neuron count, and this is especially true for larger networks. This graph demonstrates that version 2 does decrease the network size, more precisely the synapse count, to a reasonable number, allow the networks to fit in memory. Version 2 does require more neurons than version 1, but compared to the improvement on synapse count, the effect is negligible.

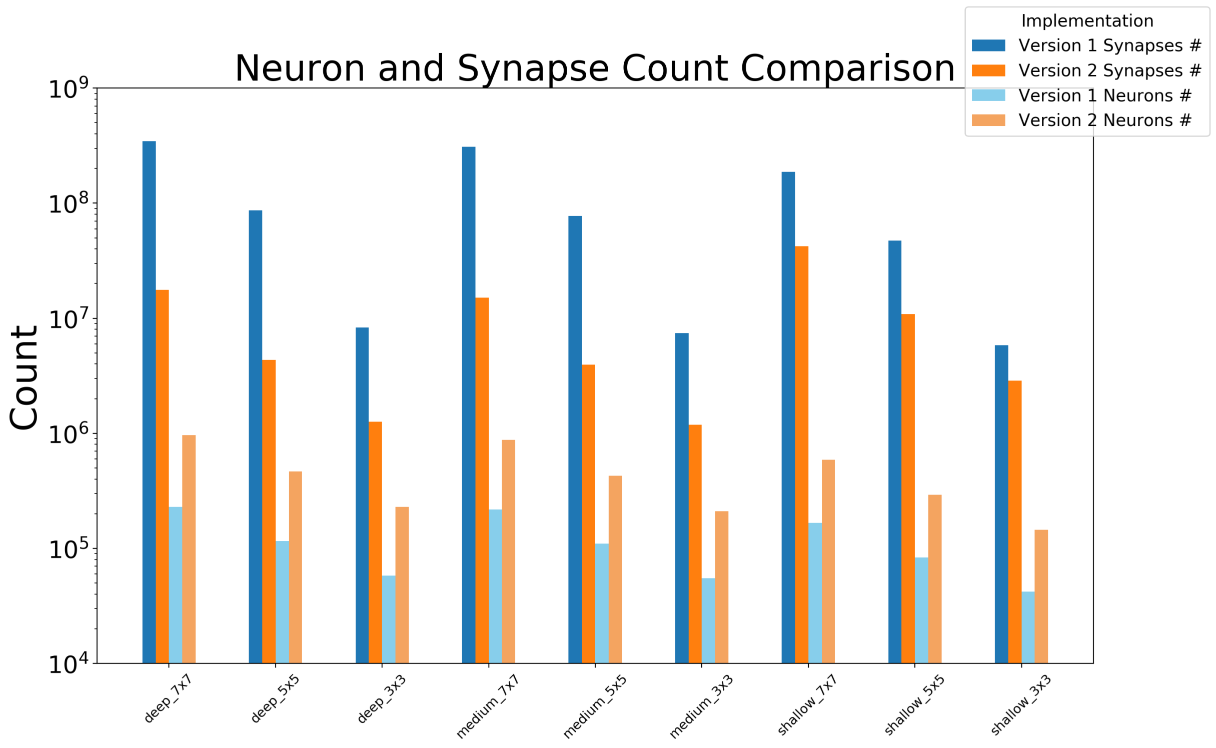

Figure 6.1: Comparison of neuron and synapse counts by the two CNN conversion methods on different network topologies. The x-axis displays nine different network sizes, and the y-axis represent the neuron and synapse counts.

6.2 Accuracy of DANNA2

This test is necessary because DANNA2 networks use integers to represent their network elements, so the converted network is different from the original network. The WHETSTONE architecture matches the SDNN spike by spike, so the accuracy of the converted WHETSTONE network is the same as the accuracy of the original SDNN. This test includes accuracy testing on both the converted DANNA2 networks and original SDNNs, which is the same as the WHETSTONE networks.

Figure 6.2 is a bubble graph of DANNA2 and Whetstone accuracy on four different classification datasets:

1. MNIST - handwritten digits.
2. FASHION MNIST - dataset of 10 classes of fashion images [15].
3. CIFAR10 - dataset of 10 classes of color images [16].
4. CIFAR100 - extrapolate of CIFAR10 with 100 classes.

Due to time, we only tested 100 test cases on DANNA2. On Whetstone, we ran all of the test cases.

Most datasets have better accuracy on Whetstone than on DANNA2. The reason very likely is that the DANNA2 architecture loses precision by using integers for its network properties. The loss of precision hurts accuracy of the FASHION MNIST, CIFAR10, and CIFAR100 dataset. It helps the MNIST dataset, most likely because the loss of precision reduces overfitting.

6.3 Other

The converted SNN can run either on a simulator or potentially a hardware device. Therefore, it is interesting to test the event rate and spike rate of classification. The event rate is important, because the simulators for DANNA2 and WHETSTONE architectures are all event based simulators. Similarly, when implemented on neuromorphic hardware, event and spike counts are aid in estimating power consumption.

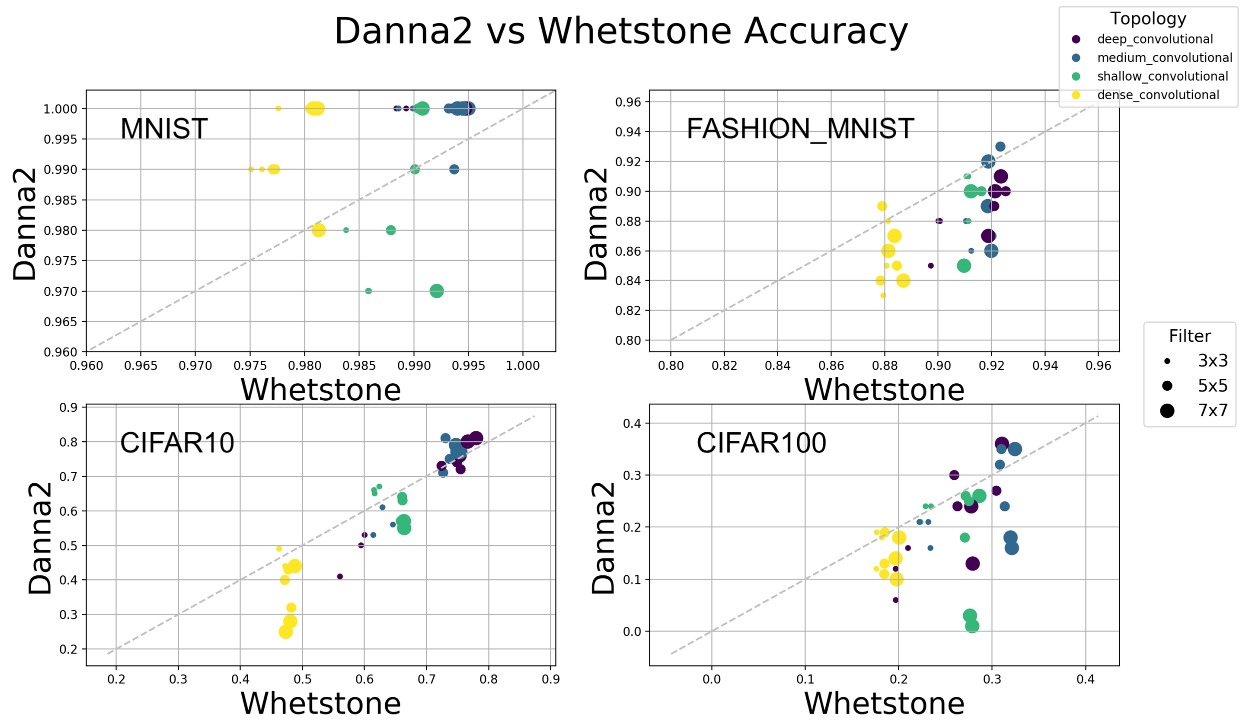

Figure 6.2: DANNA2 and Whetstone accuracy comparison on MNIST, FASHION MNIST, CIFAR10, and CIFAR100 dataset. The bubble color represent the network topologies, and the bubble size represent the filter size of the networks. Bubble below the diagonal line indicate the Whetstone network has a better accuracy than the converted DANNA2 networks, and vice versa.

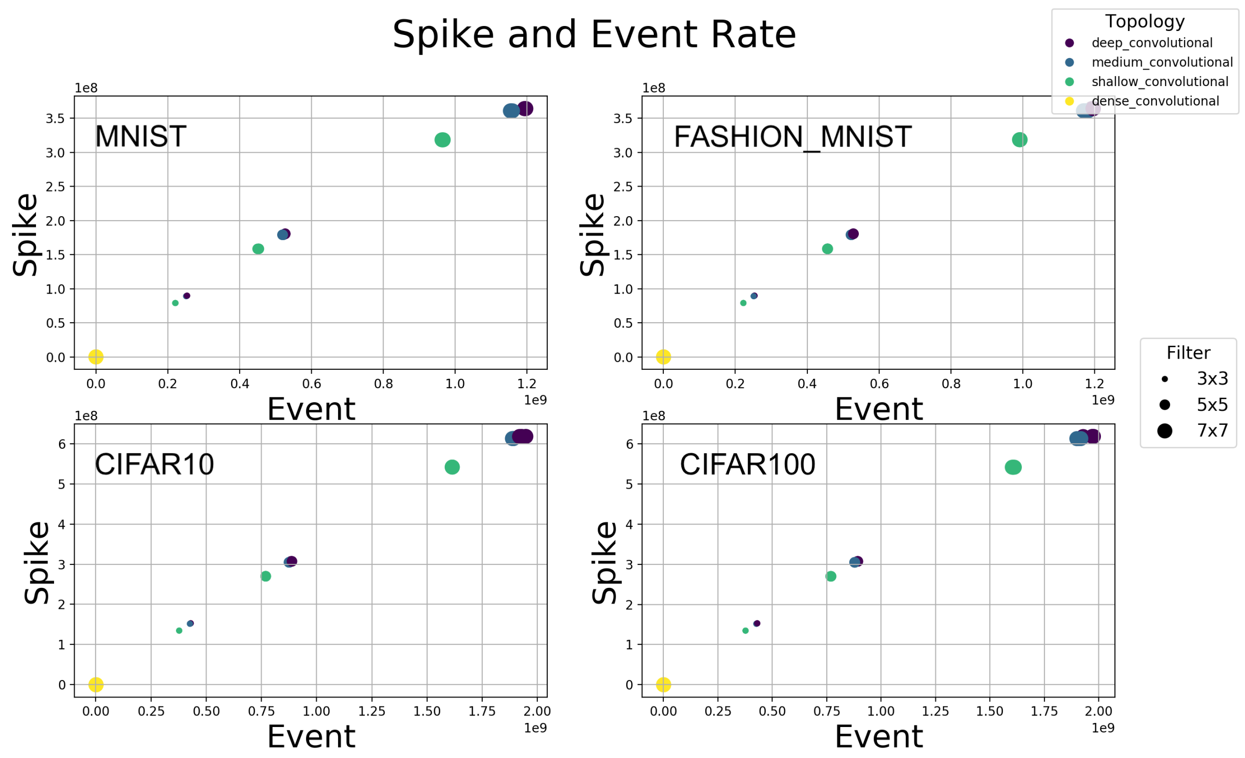

Figure 6.3: Spike and event rate for different networks on MNIST, FASHION MNIST, CIFAR10, and CIFAR100 dataset. The graph variables are the same as Figure 6.2.

Figure 6.3 displays a bubble graph of spikes vs. events in the four applications. Events compose a superset of spikes as not all events lend to spikes. This is borne out by the graphs, where the event-to-spike ratio is roughly 3.3 to 1.

Chapter 7

Future Work

7.1 PB()

In a converted CNN, PB() spikes at least one synapse every timestep. As future work, a new implementation of PB() utilizes on interval i to be the delay of the synapses which connect B_0 and B_1. Figure 7.1 shows the new implementation, This decreases the spike rate of PB() from once every timestep to once every i timesteps. Since PB() has been used in multiple places in the converted convolutional layer, the new implementation should increase simulation speed, and decrease power consumption on neuromorphic hardware.

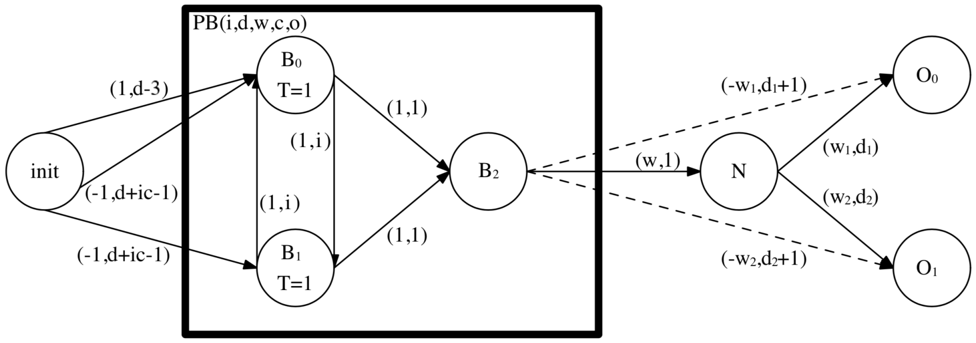

Figure 7.1: A new implementation of PB() that should speed up the network.

7.2 PC()

One other way to make the network run faster is to improve PC(). Every C_{ij} neuron is connected by a PB(), and every PB() generates at lease one spike every timestep (or one spike every i timesteps in the future version). The spikes take up most of the network activity. To improve PC(), the conversion program can create just one PB() that connects to every C_{ij} instead of having one PB() per C_{ij}. The new PC() would still have the same activity because all the original PB() units have the same parameters, and this implementation could decrease the network size and increase the network speed because it has fewer PB() unit which generate fewer spikes.

7.3 Pipelining

Each C_{ij} neuron has a synapse connects that to Z_{ij}, and those synapses all have different delay values so the spikes will arrive to Z at the same time. One way to improve the network speed is, instead of using the delays to synchronize the spikes, to have the spikes arrive to Z_{ij} directly after C_{ij} fires. The next convolutional layer will start after Z_{ij} accumulates enough spikes.

Chapter 8

Conclusion

The goal of this project is to convert a Whetstone trained SDNN to a SNN in TENNLab, enabling the SNN to be trained by back propagation. The conversion method can convert a network with feed forward layers, convolutional layers, and other types of layers to a SNN of either DANNA2 or WHETSTONE architectures. The first approach of converting CNN to SNN failed, because the converted network was too large to store in memory. Another conversion method was designed to solve this problem. The converted subnetwork trades time for space, so the network fits in memory. The experiment shows that version 2 significantly decreases the network size and successfully converts the SDNN to a SNN. The experiment also shows that the WHETSTONE network has the same accuracy as the original Whetstone network, and DANNA2 network can be better or worse depending on the dataset complexity.

Bibliography

[1] W. Severa, C. M. Vineyard, R. Dellana, S. J. Verzi, and J. B. Aimone, “Whetstone: A method for training deep artificial neural networks for binary communication,” *CoRR*, vol. abs/1810.11521, 2018. 1, 11

[2] J. S. Plank, C. D. Schuman, G. Bruer, M. E. Dean, and G. S. Rose, “The TENNLab exploratory neuromorphic computing framework,” *IEEE Letters of the Computer Society*, vol. 1, pp. 17–20, July 2018. 1

[3] J. P. Mitchell, M. E. Dean, G. Bruer, J. S. Plank, and G. S. Rose, “DANNA 2: Dynamic adaptive neural network arrays,” in *International Conference on Neuromorphic Computing Systems*, (Knoxville, TN), ACM, July 2018. 2

[4] J. Schmidhuber, “Deep learning in neural networks: An overview,” *CoRR*, vol. abs/1404.7828, 2014. 3

[5] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. 3

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012. 4

[7] G. Tolias, R. Sicre, and H. Jégou, “Particular object retrieval with integral max-pooling of cnn activations,” *arXiv preprint arXiv:1511.05879*, 2015. 8

[8] W. Maass, “Networks of spiking neurons: the third generation of neural network models,” *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997. 10

[9] J. S. Plank, C. Rizzo, K. Shahat, G. Bruer, T. Dixon, M. Goin, G. Zhao, J. Anantharaj, C. D. Schuman, M. E. Dean, G. S. Rose, N. C. Cady, and J. Van Nostrand, “The TENNLab suite of LIDAR-based control applications for recurrent, spiking, neuromorphic systems,” in *44th Annual GOMACTech Conference*, (Albuquerque), Mar. 2019. 10

[10] M. E. Dean, J. Chan, C. Daffron, A. Disney, J. Reynolds, G. S. Rose, J. S. Plank, J. Birdwell, and C. D. Schuman, "An application development platform for neuromorphic computing," in *International Joint Conference on Neural Networks*, (Vancouver), July 2016. 10

[11] J. J. M. Reynolds, J. S. Plank, C. D. Schuman, G. R. Bruer, A. W. Disney, M. E. Dean, and G. S. Rose, "A comparison of neuromorphic classification tasks," in *Proceedings of the International Conference on Neuromorphic Systems*, ICONS '18, (New York, NY, USA), pp. 12:1–12:8, ACM, 2018. 10

[12] M. E. Dean, C. D. Schuman, and J. D. Birdwell, "Dynamic adaptive neural network array," in *13th International Conference on Unconventional Computation and Natural Computation (UCNC)*, (London, ON), pp. 129–141, Springer, July 2014. 11

[13] C. D. Schuman, *Neuroscience-Inspired Dynamic Architectures*. PhD thesis, University of Tennessee, May 2015. 11

[14] G. Chakma, M. E. Dean, G. S. Rose, K. Beckmann, H. Manem, and N. Cady, "A hafnium-oxide memristive dynamic adaptive neural network array," in *International Workshop on Post-Moore's Era Supercomputing (PMES)*, (Salt Lake City, UT), Nov. 2016. 11

[15] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *CoRR*, vol. abs/1708.07747, 2017. 40

[16] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," tech. rep., Citeseer, 2009. 40

Vita

Jiajia Zhao is originally from Zhuzhou, Hunan, China. She attended Tianxin No.1 Middle School. After graduating, she studied English at Austin Peay State University in 2013 and started as an undergraduate student in 2014. She transferred to the University of Tennessee, Knoxville in 2015, and in the spring of 2018, she received her Bachelor of Science degree in Computer Science. Soon after, she continued pursuing a Master of Science degree in Computer Science in the University of Tennessee, Knoxville. During college, she interned at TEAMHealth Inc. as a System Engineer in the summer of 2016. As a student, she assisted low level computer science classes as a teaching assistant. In the summer of 2017, she worked as an research assistant for TENN Lab until she received her Master of Science degree in August 2019.