SOFTWARE IMPLEMENTATION REPORT

Background:

Initially we finalized on the Spike Response Model (SRM) for the neurons in the SNN architecture. However, there was lack of extensive prior work on the proper training of SRM based neural networks and this would make it extremely challenging to design a high accuracy SNN. Due to this reason we decided to shift to the more explored Leaky Integrate and Fire (LIF) neuron model. The extensive work that has been done on LIF based SNN networks would give us the chance to focus more on hardware optimizations and performance improvements.

We started with a Python implementation as it would be easier to get a working SNN in Python and we can use that network in future as a reference to verify the functional correctness of the hardware implementation at all times. We can port the important modules from Python to C/C++ for implementation on the FPGA.

All of the codes are available at:
https://github.com/snagiri/CSE237D-PYNQ-SNN-Accelerator

First Software Implementation:

The first software implementation was based on the LIF neuron model and was aimed at performing successful binary classification on images of 0 and 1 digits from the MNIST dataset. The main details of the implementation are as follows:

Number of Layers	2
Input layer neurons	784
Output layer neurons	3
Total timesteps	200
Type of neuron interconnection	Fully Connected
Neural Coding Method	Probabilistic encoding
Learning Method	STDP
Training Set (Images of 0 and 1 from MNIST)	100 samples

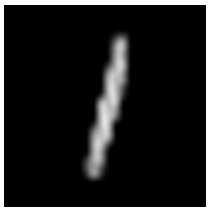
In order to achieve successful binary classification we had to mainly tune the learning rate, epoch (number of times the given set of input samples is sent to the network for training to ensure effective learning) and threshold voltage for the neurons.

We were able to achieve successful binary classification for an epoch of 12. We kept an extra neuron at the output to improve the chances of two neurons learning different patterns namely 0 and 1.

The learnt weights for each output neuron constituted an array of length 784. We reconstructed a 28x28 pixel image from these weight arrays for each of the output neurons. The reconstructed images are as follows:

Neuron 1

Neuron 2



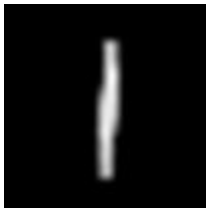
Neuron 3

Second Software Implementation:

Taking the first implementation as a base, we developed the SNN architecture further to be able to classify all 10 digits from the MNIST dataset. The number of output neurons was increased to 12 neurons and the input training set was increased to consider 5000 samples of input images corresponding to digits 0-9. Owing to the increase in the input dataset, the epoch was reduced to 5. We tried different variations in the parameters but were unable to train the network to perfectly learn all of the 10 classes. Additionally, small variations in the parameters resulted in varying outputs which made it harder for us to determine a pattern. For one of the iterations, the images learnt by each of the output neurons were as follows:

Neuron 1

Neuron 2



Neuron 3

Neuron 4

Neuron 5

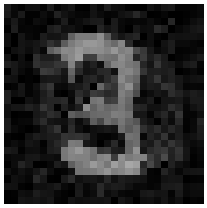
Neuron 6

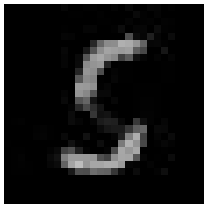
Neuron 7

Neuron 8

Neuron 9

Neuron 10


Neuron 11


Neuron 12

As can be seen from the learnt images, the network was not able to learn the digit 8 at all. The digits 2,4,5 and 9 have also not been learnt clearly and may not necessarily give good accuracy for variations in writing styles of these digits. Thus, we realized that effectively training the SNN network will be a project in itself. We decided to use existing python libraries that have provision for optimization and learning to design our final software implementation.

Third Software Implementation:

The third implementation was based on Nengo python libraries. These libraries facilitated the overall training process and optimization of the neural network. Instead of a fully connected network, the SNN architecture consisted of convolutional layers. However, the information was in the form of spike trains and the individual neurons were modeled as LIF neurons. The main parameters of the implemented architecture are as follows:

Number of Layers	5
Input layer neurons	784
Hidden layer 1	Filters - 32
Hidden layer 2	Filters - 64
Hidden layer 3	Filters - 128
Output layer neurons	10
Total timesteps	30
Type of neuron interconnection	Convolutional Layers
Training Set (Images of 0 and 1 from MNIST)	5000 images

We trained the network on the set and evaluated the accuracy of the network on the separate MNIST test dataset. Additionally we utilized a set of available pre-trained weights that had been generated through rigorous training on the given network. The accuracies obtained in both the cases and the classification time has been recorded in the table below.

Weight Matrix	Error Rate (%)	Classification Time (secs)
Pre-trained weights	0.75	89
Trained Weights	1.25	122

Final Software Implementation:
For our final software implementation, we simplified the overall network by replacing the convolutional layers by simple fully connected network. This will enable in better portability of the SNN architecture onto hardware. The key parameters of this network are as follows:

Number of Layers	5
Input layer neurons	784
Hidden layer 1 neurons	256
Hidden layer 2 neurons	64
Hidden layer 3 neurons	16
Output layer neurons	10
Total timesteps	30
Type of neuron interconnection	Fully connected
Training Set (Images of 0 and 1 from MNIST)	5000 images

We attained a 97% accuracy after training this final network and the classification time is 3 secs which implies that the network is greatly simplified with the use of the fully connected networks. We saved the weight matrix and the input spike train which will be needed for the hardware implementation.

Conclusion:

We developed a working software implementation of the SNN network and were able to finally attain an accuracy of 97% on the MNIST dataset. The learnt weight matrix can be stored in a read only memory in the programmable logic portion of the PYNQ board and used for classification. In addition, the input spike train can be generated on the processor and given as input to the programmable logic for classification.
The Python implementation will act as a reference in the process of porting of the implementation to hardware.