Final Project

**Neural Network Inference on FPGA**

ECE 6213 – Design of VLSI Circuits

The George Washington University – Fall 2022

Osama Yousuf, Joseph Riem

1. **Project Description**

In this project, we implemented inference of a feed-forward neural network on FPGA for an image classification problem. Mini-ITX boards from the ADAM Lab @ GW, as shown in Figure XX below, were used. These boards contain a hard processor or a processing system (PS), as well as programmable logic (PL). The implementation was done in two ways – firstly through a high-level synthesis workflow, and secondly through custom RTL modules (written in SystemVerilog). Accuracy results were obtained from both neural networks (HLS and RTL) and it was found that the custom RTL network occupied less resources compared to the HLS network and achieved a similar accuracy performance. Additionally, a web camera was integrated into the inference workflow, and we successfully demonstrated how our network could classify an image taken from the camera in real-time using a server-client approach.

**Graphical user interface

Description automatically generated**

1. **Specification**
   1. **Problem Overview**

The problem investigated is a reduced version of the handwritten MNIST dataset. This is a dataset consisting of handwritten images of digits labelled 0 through 9 and is publicly available. The neural network is fed an image from this dataset as an input, and it attempts to classify the digit as its output. Figure XX below captures this input-output function, with the neural network treated as a black box.

Graphical user interface, application

Description automatically generated

The MNIST dataset has 60,000 training images, and 10,000 testing images. Originally, all of these images contain a total of 28 x 28 pixels. For our project, we reduced the dataset size to 18 x 18 by center-cropping these images for purposes of achieving fast inference latency and supporting a future hardware prototype where an array of memristive devices will be used to represent network weights. Additionally, the images were binarized for simplicity i.e., input images can either have pixels that are fully black or fully white.

* 1. **Neural Network Operation**

A feed-forward network has two modes of operation: **training** and **inference**.

In the **training** phase, the network has a forward pass as well as a backwards pass. In the forward pass, the network is presented an image in the form of a simple vector. This vector gets multiplied to the internal parameters of the network (weights and biases – explained in Section XX below), and produces an output vector, which is then used to draw a classification prediction. In the backwards pass (referring to the backpropagation algorithm), the internal parameters are adjusted based on the prediction which was made. In brief, the idea is as follows: Based on the quality of the prediction, a gradient matrix is computed using a loss function for each network layer or weight matrix. Once all gradients are calculated, they are scaled by a scalar hyperparameter known as the learning rate. Finally, the network weights are updated by simply subtracting from the scaled gradient. This process (of a forward pass and a backwards pass) is repeated with different images from the training dataset until either the network has been trained for a set number of iterations, or the network meets some accuracy benchmark.

In the **inference** phase, the network only goes through a forward pass, meaning that internal parameters of the network are not adjusted.

For our project, we implemented the training phase in software and only the inference phase on FPGA. Once the software weights have been trained to desired accuracy performance, they are converted to text files and are loaded in RTL code in custom-width fixed-point precision as specified.

* 1. **Network Details**

We implemented a simple 2-layer perceptron network (or a multi-layer perceptron/MLP), also known as a feed-forward network. The first layer had dimensions 324 x 10, and the second layer had dimensions 10 x 10, meaning that the architecture was 324 x 10 x 10. The HLS network was trained with biases since the HLS library used (hls4ml, more details in section XX) supported biases, whereas the custom RTL was trained without biases for simplicity. A ReLU function was used as a non-linear activation function, primarily because it was easy to implement on the FPGA. For the loss function, we opted to use CrossEntropy since we train the network in software and are not concerned about the ease to train the network in hardware.

* 1. **Inference Details**

Figure XX below summarizes the inference operation in our chosen network architecture. First, the 18 x 18 input image is vectorized and multiplied with the weights of the first layer of the neural network. The output is passed to the ReLU non-linear activation function, which is then multiplied with the weights of the second layer of the neural network. The output vector is a 1 x 10 vector, and the predicted class is identified by doing a simple argmax (or softmax) operation.

Diagram

Description automatically generated

1. **Task Breakdown**

We broke down the project into the following phases, so as to manage the workload in easy-to-accomplish chunks. Implementation details of each tasks are presented in Section 4.

* **Phase 0:** Implement neural network training in software
* **Phase 1:** RTL coding and simulations for neural network inference
* **Phase 2:** HLS coding and simulations for neural network inference
* **Phase 2:** Synthesis and deployment of the RTL and HLS neural networks on the Mini-ITX Zynq 045/100 FPGA boards
* **Phase 3:** Study how reducing network bit-precision impacts network accuracy in simulations (%)
* **Phase 4:** Implement inference based on live-camera feed instead of preset testing dataset

1. **Implementation**
   1. **Phase 0: Network Training in Software**

Related files are provided in the accompanying “Phase 0 – Network Training in Software” folder.

We implemented a neural network in PyTorch matching our proposed specification. Source files are provided in the directory “pretraining” directory. Appropriate Python dependencies can be installed using the provided “requirements.txt” file via running the command “pip install -r requirements.txt”. The file “main.py” can be executed with appropriate command line arguments to train the network for either the HLS case or the RTL case, with the difference being that the HLS network has biases as well. For the custom RTL network, layers from the trained weights are exported as text files and are provided in the “data\_rtl” directory, whereas for the HLS network, the weights are dumped using PyTorch’s state dictionary functionality in a “.pt” format, provided in the “data\_hls” directory. Moreover, the center-cropped and binarized MNIST dataset is provided in the “mnist” directory.

* 1. **Phase 1: Custom RTL Network**

Related files are provided in the accompanying “Phase 1 - Custom RTL Network” folder. File “inference.v” contains our synthesizable top-level module for network inference, and “inference\_tb.v” is the accompanying testbench. The testing image dataset is provided in the “testimages.rar” archive, and this needs to be extracted and loaded in the testbench for replicating our simulation results. The flow for network inference is as follows:

* Module MM1 performs matrix multiplication for layer 1.
* Module fullReLU performs the non-linear activation function for inference on the output from MM1.
* Module MM2 performs matrix multiplication for layer 2.
* Module label produces the final classification.

We use fixed-point representation for the network weights and input images. Our testbench outputs a “label\_v{i}.txt” file, where {i} denotes the number of fractional bits in the chosen fixed-point representation (with a total of 16 bits), which has all the predictions for the input testing dataset (10,000 predictions, one on each testing image in the dataset). These output files are provided in the “outputs” folder. We wrote an additional verification script in Python “comparelabelfiles.py” to compare the predictions of the network in the “labels\_v.txt” file with the actual classes of each image in the testing dataset, and this is provided in the “XX.XX” file. Using 16-bit fixed point precision with 11 fractional bits, our network achieved an inference accuracy of 90.47 % on the 10,000 images in the testing dataset.

* 1. **Phase 2: High-level Synthesis (HLS) Network**

Related files are provided in the accompanying “Phase 2 - High-level Synthesis (HLS) Network” folder. TODO.

* 1. **Phase 3: Studying Network Bit-Precision**
  2. **Phase 4: Web Camera Integration**

1. Results
2. Future Work
3. Appendix