Hardware Architectures for Artificial Intelligence
Lab 2 - BNN

Group 10

Alexandra Dobrița (5373301, adobrita)
Nishad Mandlik (5242541, nmandlik)
Mathijs Heidekamp (4485866, MathijsHeideka)
Prithvish Nembhani (5352576, pnembhani)
Hendrik Jan Kruijsse (4313798, hkruijsse)
Rohan Kaushik (5280036, rkaushik)

June 29, 2024

1 Introduction

Over the past decade AI has advanced by leaps and bounds, and this advancement has been lead by the advent of deep neural networks. More specifically, it was the convolutional architecture based AlexNET that spurred on this revolution by blasting its competition out of the water at the 2012 ImageNet Challenge. Since then, these networks have become ever larger in size, demanding more compute power and more memory. This becomes a big problem for edge computing and embedded systems devices since these are limited by their power consumption. To reduce the power consumption of DNNs, Binary Neural Networks (BNN) have been devised. These store their weights in only single bit values - $+1$ or -1 - rather than full precision floating point numbers. These networks reduce inference time, power consumption and the total area requirement since simple bit-wise operations can be used to execute the required computations.

1.1 Binary Neural Networks

[1] provides a good overview of BNNs, and introduces the key concepts that allow us to train such networks using traditional methods such as back-propagation. The training is further discussed in section 2. In this project we have implemented a binarized convolutional network in software using TensorFlow and its hardware analogue in VHDL. The network has been trained in software on the MNIST database, and the final trained weights and biases transferred to the VHDL model.

As mentioned previously, BNNs are not only helpful in reducing the total memory required to store a network, but also speed up computations due to these being reduced down to simple bitwise operations. Additionally there is another unexpected benefit - robustness. As we have seen in the last couple of years, it is possible to launch adverserial attacks on networks, i.e. fool the network into misclassifying certain images/inputs by simply adding a very calculated amount of 'noise' to it. This 'noise' seems completely ordinary to the human eye, but can cause the network to fail miserably at its tasks. This attack vector is almost neutralized in the case of BNNs since the binarized nature of the weights and activations makes it near impossible to alter the output with 'small' changes to the input. While this is not something we were explicitly focusing on, it is still good to know that our network possesses such an advantage.

2 Training

We have built a convolutional network in TensorFlow and trained it on the MNIST dataset. However, in an effort to further cut down on computational effort we have also binarized the training/testing datasets. This binarization is quite straightforward - since the MNIST datasets consist of 28×28 pixels each ranging from $0 - 255$, those with values ≥ 127 have been set to $+1$ and the those with values < 127 have been set to -1.

2.1 Architecture

The input layer of the network is a convolutional layer with 8 neurons, each having a kernel size of 9×9. This is followed by a max-pool layer with a window size of 2×2 and finally a full-connected dense layer having 10 outputs. Since TensorFlow does not offer native support for quantized models, we used another package called `Larq` which can produce TensorFlow compatible quantized layers. Thus the convolutional layer is actually a `larq.layers.QuantConv2D` layer, and we have used the straight-through-estimator (STE) as described in [1]. The final dense layer is also a `larq` quantized layer using STE. The model has been trained using the categorical cross entropy loss.

2.2 Back-Propagation for BNNs

As mentioned previously, the quantized layers use something called a straight-through-esitimator to train their weights. This is just a fancy term for a relatively straightforward process. The layers maintain a set of real full-precision weights during training. These are just quantized to $+1$ or -1 according to their sign during the forward pass of the network. However, in the backward pass, the full-precision values of the weights are updated using full-precision updates. This process is shown in Figure 2. This is done so that the weights can be updated incrementally, since if the weight were constrained to be quantized in the backward pass we would see little to no change in their values for long training periods.

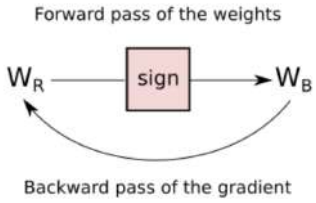

Figure 1: Straight-through-estimator [1]

2.3 Other Considerations

We used a batch-size of 32 images and 100 epochs to train our network. Additionally, we also implemented early stopping to make sure we get the network with the highest possible accuracy and to prevent overfitting. In the end, we get a trained network with an accuracy of 94.24% (on the test set). The variation in validation accuracy over multiple epochs during the training phase can be seen in **??**

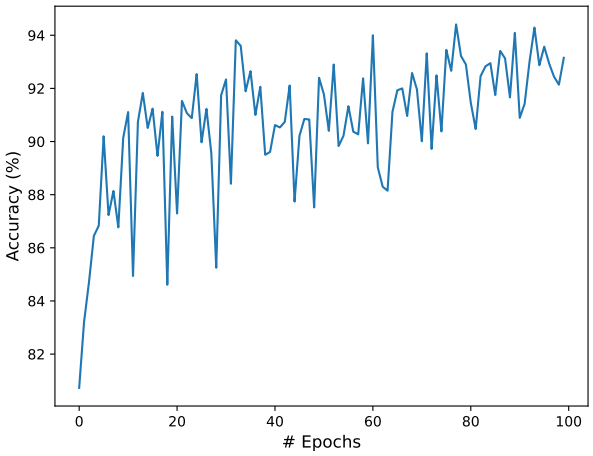

Figure 2: Variation of validation accuracy during the training phase.

3 Inference

In the previous section, the chosen Neural Network and binarization method have been described. In this section, the hardware implementation of the BNN inference phase is detailed.

A top-view of the hardware architecture is depicted in Figure 6. There architecture consists in three main modules: a convolutional layer module, a max pool layer module and a fully connected layer module. A fourth module is an internal clock counter used to syncronize the other three modules.

Figure 4 provides a visual representation of the inputs and outputs to of the three layers. The convolutional layer receives as input the test image and the weights computed in the software training part. The output will consist in $8 \times 20 \times 20$ values, which will be further reduced to $8 \times 10 \times 10$ in the max pool layer. Finally, the output of the max pool layer is flattened into a 1D array and convolved with each of the 10 arrays of weights for the fully connected layer. The result will be 10 neurons, one for each digit.

In the next subsections, the implementation of these modules is detailed.

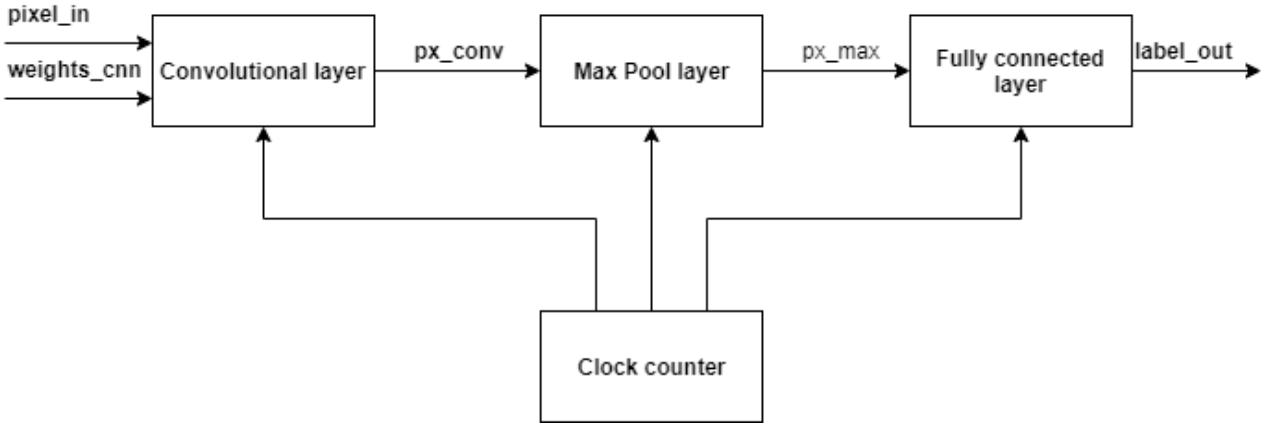

Figure 3: Top-view of the inference hardware architecture

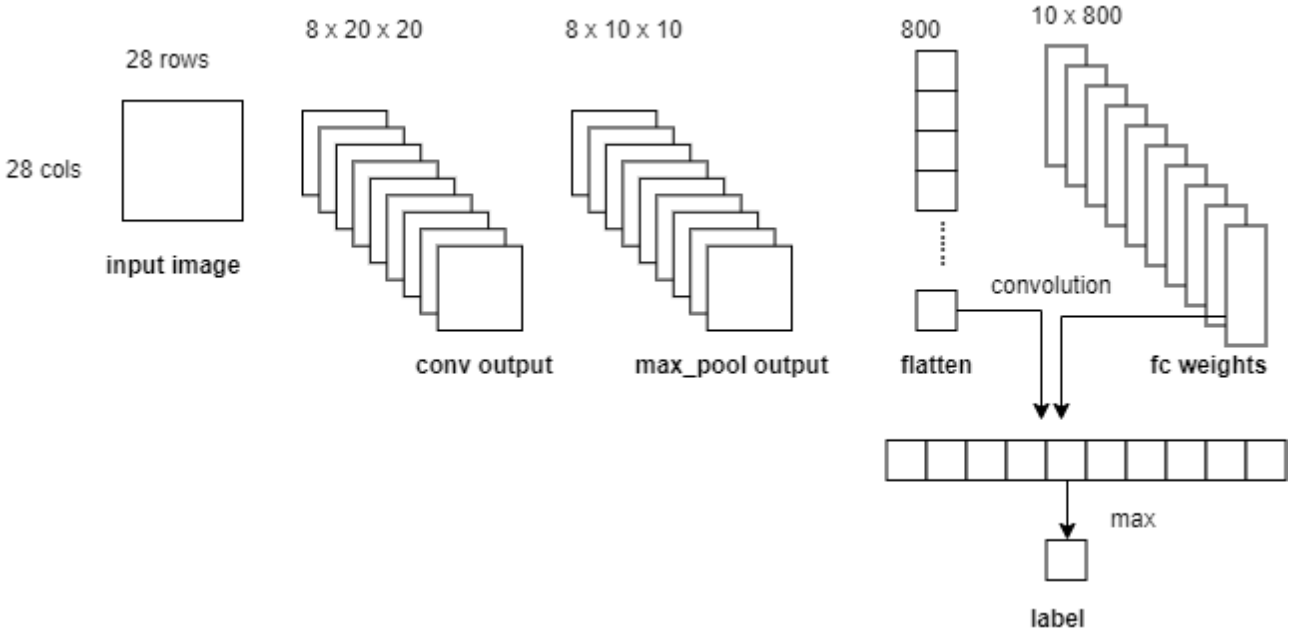

Figure 4: Top-view of the inference hardware architecture

3.1 Convolutional layer

Firstly, for encoding the {-1,1} values to binary values, and for the corresponding multiplication, the representation in Figure 5 is used.

The input to the convolutional layer is given by one sample of the test images available in the MNIST database. The image is streamed to the convolution module one pixel per clock cycle. The streamed pixels are already binarized, thus one bit is needed for its representation. This allows for future optimizations, such that 8 or 16 pixels could be streamed at a time.

Encoding (Value)		XNOR (Multiply)
0 (−1)	0 (−1)	1 (+1)
0 (−1)	1 (+1)	0 (−1)
1 (+1)	0 (−1)	0 (−1)
1 (+1)	1 (+1)	1 (+1)

Figure 5: Binary encoding of {-1,1} and multiplication using a XNOR gate [1]

The convolution kernels have 9×9 elements. In order to convolve the image with a kernel in hardware, a number of rows from the image equal to the number of rows from the kernel need to be buffered. Thus, we buffer 9 image lines by using a shift register of 28×9 elements.

Once the shift registers are fully loaded, the corresponding registers indexes are XNOR-ed with the corresponding weights in each of the 8 kernels. Thus, at each of the next 20 clock cycles, 8 output pixels are computed, one for each neuron. The advantage of using the shift register is that, with each shifting, when computing an output line, the corresponding input pixels are aligned to the corresponding weights indexes. For this reason, after each output line is computed, for the next 8 clock cycles the output is not valid, since the values in the shift registers are not aligned to the same weights indexes anymore.

The 8×81 XNOR operations are all computed in a clock cycle. The 81 value corresponds to the multiplications needed to compute one output pixel, while the 8 value refers to the number of neurons. In a BNN, the convolution output needs to be binarized to ± 1, using the signum function. Since $\{-1, 1\}$ is encoded as $\{0, 1\}$, the number of '1' bits is counted, after the XNOR operations are performed. The final accumulation sum is computed as $2 \times \texttt{count\_ones} - 81$ [1], and thresholded around '0'.

The valid pixels to be outputed are triggered by an internal clock counter. When the value of this counter is in the valid range, the thresholded pixel value is written to the output.

The convolution module implemented in hardware has an overhead influenced by the number of clock cycles needed to load the shift register and the number of clock cycles in which the output pixels are not valid. Thus, the overhead is given as $28 \times 9 + 8 \times 19 = 404$ clock cycles.

The total latency of the module is $overhead + 400 = 804$ clock cycles. A diagram of the hardware implementation of the convolution layer is depicted in Figure 6.

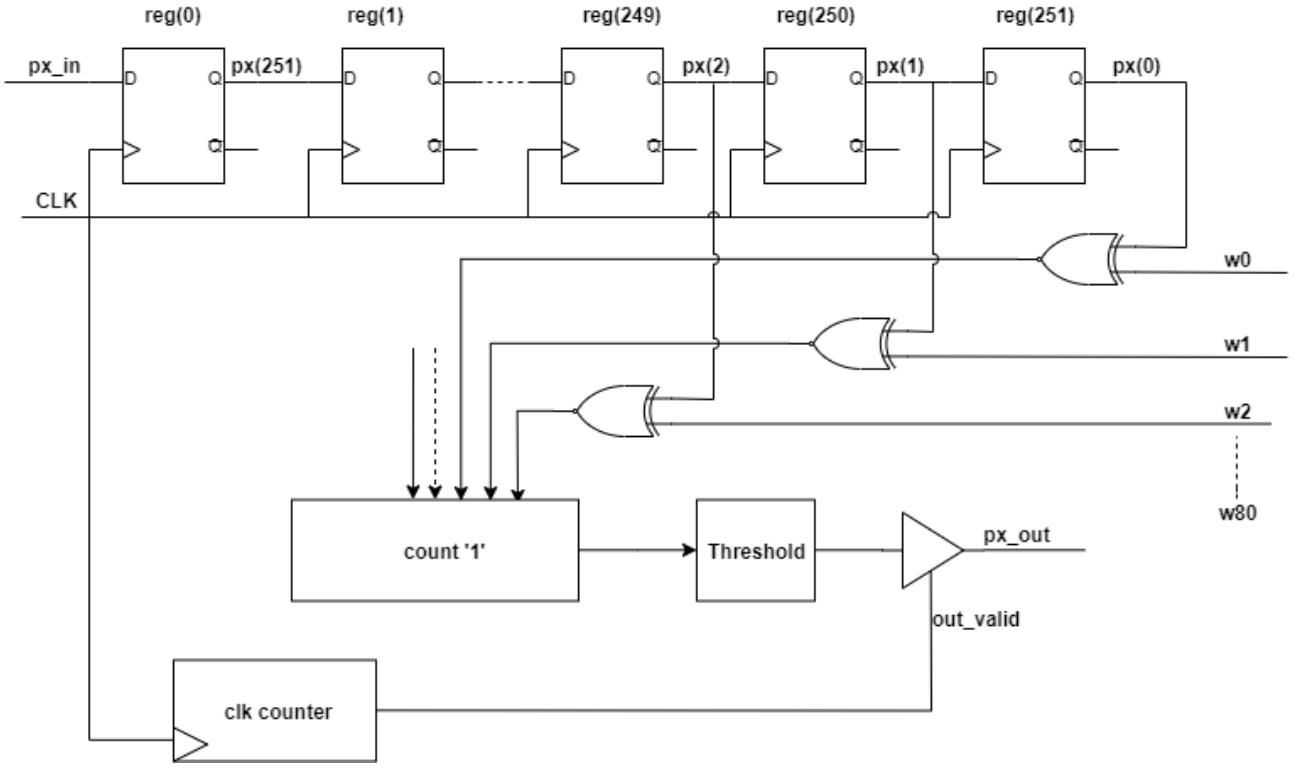

Figure 6: Convolution layer hardware diagram

3.2 Max pool layer

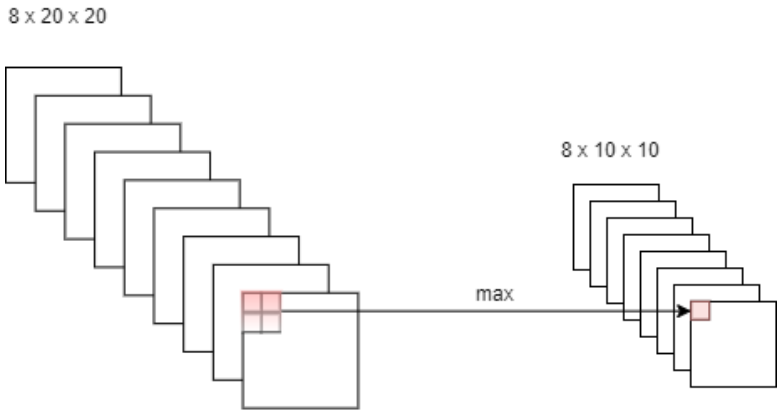

Figure 7: Visual representation of the max pool layer

The max pool layer finds the maximum value in a window of 2×2. This operations is repeated with a stride of 2 horizontally and vertically, with zero-overlap. In hardware, this can be implemented by applying an OR operation on 4 pixel values. The input to the max pool layer comes in from the output of the convolution layer. Similarly to the convolutional module, the inputs to the max pool layer are buffered using a shift register that shifts 8 value at a time. Since a max value is chosen from a window of 2×2, 2 rows for each convolutional output need to be buffered, and 10 outputs are computed per clock cycle, for each of the 8 output channels.

Since the convolutional layer produces 20 valid outputs, after which 8 invalid outputs, a *valid* condition is used for checking when a valid output should be added in the shift register, namely, the clock counter modulo 28 should be less than 10 (*Note.* Here, the clock counter has a bias of (-252 - 1), which accounts for the overhead of the convolutional layer and a one clock cycle delay for reading the first output).

Similarly, a *valid* condition is used for writing each valid output, namely the clock counter modulo $56 = 48$ (20 valid inputs + 8 garbage inputs + 20 valid inputs).

The latency of this module is $56 \times 9 + 48 = 552$ clock cycles.

3.3 Fully connected layer

The fully connected layer gets its data from the Max Pool layer. It has 10 output neurons, one for each digit. It performs a weighted sum on the outputs of the Max Pool layer. Then the neuron that is activated is the one which has the highest weighted sum.

The implementation of this module follows the same approach as the previous ones: firstly, the outputs from the Max Pool layer are buffered in a shift register. When the shift register is fully loaded, a flatten operation is performed, which actually represents a remapping of the values in the shift register to a 1D array.

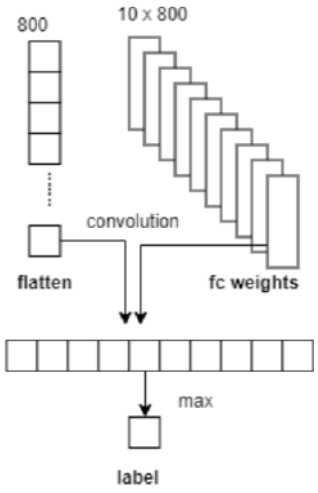

Figure 8: Visual representation of the Fully Connected layer

The flattened array is then convolved with the set of weights corresponding to each neuron in the Fully Connected layer, which have been computed in the software training phase. The result consists in 10 neurons,

and the index of the highest value from these neurons represents the output label.

4 Results and concluding remarks

The ASIC implementation simulated in GENUS gives the results depicted in Table 1. The software inference computed on a GPU has a latency of 0.3 seconds, thus the ASIC implementation achieves a speedup of 37k times.

The minimum achievable clock period is 2.79 ns, thus the chip can operate at a maximum frequency of 358 Mhz.

Metric	**Value**
Area (μm^2)	60880.750
Power (mW)	10.34
Leakage power (mW)	1.14
Clock Period (CP) Achieved (ns)	2.79
Clock Frequency achieved (MHz)	358
Latency (ns)	8080
Energy (nJ)	83.61

The latency of this implementation (808 clock cycles x 10ns) can be easily improved if the overhead given by buffering the first 9 lines of the input image is improved (this overhead accounts for 30% of the total latency). This could be achieved by streaming 8 or 16 pixels at a time.

Overall, the BNN inference implementation in hardware shows significant performance improvement, therefore further exploration of architectural improvements are motivated.

References

[1] T. Simons and D.-J. Lee, "A review of binarized neural networks," *Electronics*, vol. 8, no. 6, p. 661, 2019.