EE 271: Final Project Inference Engine on FPGA

Jay Pankaj Patel

April 9, 2025

Contents

1 Building Blocks

For our inference implementation, I am considering using Convolutional Neural Networks (CNNs). They are an algorithm used to recognize patterns in data. I believe the target data should be handwritten digits from 0–9. Neural networks are built using a collection of neurons organized into layers, each with their own weights and biases. The building blocks of CNNs are the following:

- **Tensor** – can be thought of as an n-dimensional matrix. We need to decide how big to make our tensor. I was considering making it smaller, like three dimensions, because our goal is to perform simple classification.
- **Neuron** – is a mathematical function that takes multiple inputs and gives a single output.
- **Layer** – is simply defined as a collection of neurons.
- **Kernel Weights and Biases** – unique to each neuron and are determined during the training phase of our network. We will train the model on a computer and extract the weights and biases to the FPGA for inference.
- **Differentiable Score Function** – represented as class scores on the output layer.

These are the high-level components needed. Using this knowledge, let's go over the proposed design.

2 Design

Our neural network is going to be as simple as possible because the emphasis of this project will be optimizing the operations required for the model to run. Hence, we are using the "Hello World" equivalent of Machine Learning. Our architecture is as follows.

2.1 Architecture

Our design will take a **28×28 grayscale image** as input.

After:

- A **convolution layer** will be used to apply 8 kernels to the input, generating **8 feature maps of size 26×26**, which are then passed to a max-pooling layer.
- A **max-pooling layer** will reduce the **spatial dimensions** of each feature map from **26×26 to 13×13**, maintaining 8 feature maps, and will pass the results to the softmax layer.
- For training, a **softmax layer** will process the **flattened feature maps** and produce **10 probability values**, each representing the likelihood of the image belonging to one of the 10 classes (digits 0–9).

- For the inference and implementation on hardware, a **hardmax layer** will be used as implementing softmax on hardware is resource intensive and will increase development time.

Here is a visual representation of these processes.

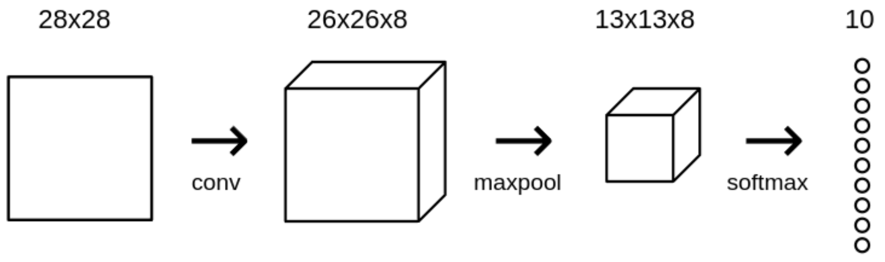

Figure 1: Visual representation of the CNN layers that will implemented as training in software

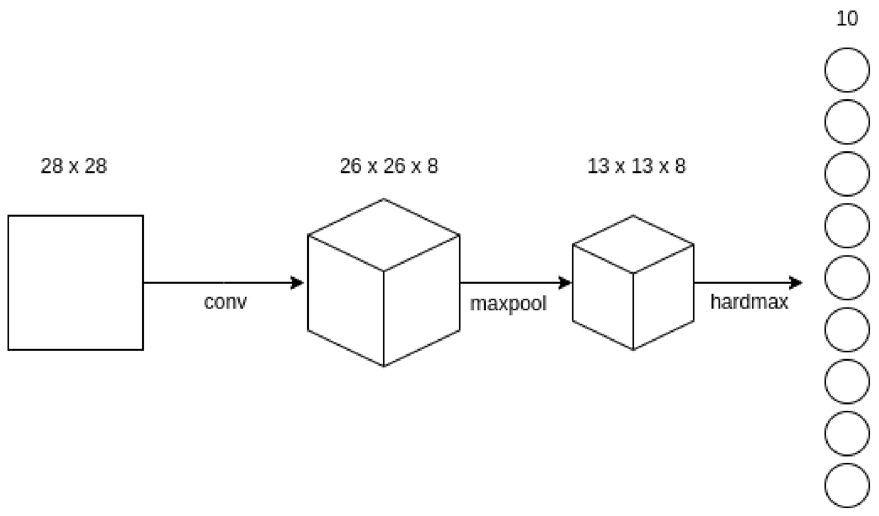

Figure 2: Visual representation of the CNN layers that will implemented in hardware

2.2 Model

A Python golden model has been created and should be used to validate the FPGA implementation. The Python model is not optimized for hardware, and operations should be adjusted for efficient computation on the FPGA. The model has been pre-trained in Python, and the trained weights and biases will be ported to the FPGA for inference. FPGA-specific design considerations will be discussed after detailing the theory behind each layer.

2.3 Convolutional Layer Design

2.3.1 Background

- **What is a feature?**
  A feature is an individual measurable property that a model uses to make a prediction or classification.
- **What is a feature map?**
  A feature map is the output produced when a convolutional kernel extracts relevant features from an input. In this field, the 'filter' is referred to as a *kernel*. Hence, I will be using "kernel" from this point forward.
- **What is convolution?**
  Convolution, in a mathematical sense, is an operation performed on two functions that produces a third function. The mathematical operation is defined as the integral of the two functions after one of them is reflected about the y-axis and shifted. Mathematically:

2.4 Equations

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)\, g(t - \tau)\, d\tau$$

Graphically:

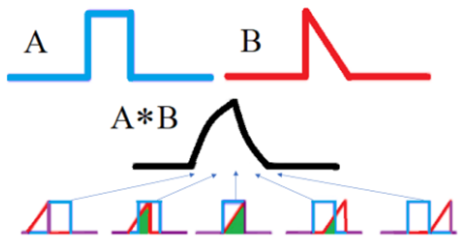

Figure 3: Convolution of two functions A (red) and B (blue) producing a third function describing the overlap (green)

Since we are dealing with discrete systems, convolution is defined using summation instead of an integral:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]\, g[n-m]$$

In CNNs, the most computationally expensive operation is the convolution layer. The convolution of the input feature map with the kernel results in an output feature map:

$$\boxed{y_j' = f\left(\sum_{i \in M_j} x_i^{l-1} * k_{ij}^l + b_j^l\right)}$$

Where:

- y_j^l : Feature map of the jth convolution kernel result of the lth layer.
- M_j : Represents the selection of the previous input feature map by the current convolution.
- x_i^{l-1}: Represents the previous input feature map.
- k_{ij}^l: Represents the ith weighting coefficient of the jth convolution of the lth layer.
- b_j^l : Represents the bias parameter of the jth convolution kernel of the lth layer.
- f : The nonlinear activation function.

2.5 Pooling Layer Design

2.5.1 Background

- **What is pooling?**
  Pooling is a technique used to downsample feature maps by aggregating adjacent values.
- **Why pooling?**
  Neighboring pixels in images tend to have similar values, leading to redundant information. Pooling reduces this redundancy while retaining important features.
- **How is pooling done?**
  Pooling is implemented using a max, min, or average value comparison over a small region.

2.6 Pooling Equation

A 3×3 convolution computes each output pixel using a weighted sum of a 3×3 region from the input:

$$\boxed{Y(i,j) = \sum_{m=0}^{2}\sum_{n=0}^{2} X(i+m, j+n) \cdot W(m,n)}$$

Where:

- $X(i,j)$ is the input feature map (28×28).
- $W(m,n)$ is the 3×3 convolution kernel (filter).
- $Y(i,j)$ is the output feature map after convolution.

Valid padding means no padding is added, so the output size is:

$$(28-3+1) \times (28-3+1) = 26 \times 26$$

And we repeat this 8 times because we have 8 kernels. Hence our final output will be 26 x 26 x 8.

2.7 Softmax Layer Design

2.7.1 Background

- **What is an activation function?** An activation function allows our model to achieve non-linearity as most things in real life are not a linear equation. By doing so, it allows our model to achieve higher accuracies.
- **What is softmax?** Softmax is an activation function that converts an arbitrary set of numbers into a probability distribution.
- **Why Softmax?** Softmax is useful when you have a multi-class model as we do here, (0-9 are different classes).

2.8 Softmax Equation

Mathematically, given an input vector $x = [x_1, x_2, ..., x_n]$, the Softmax function is defined as:

$$\boxed{s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}}$$

where:

- x_i represents the raw input (logit) for class i.

- e^{x_i} represents exponentiation of the input values.
- The denominator is the **sum of exponentials** over all input values.
- The function ensures that all outputs sum to 1.

2.8.1 Sample Calculation

1. Compute exponentials:

$$e^{-1} = 0.368, \quad e^0 = 1, \quad e^3 = 20.09 \quad e^5 = 148.41$$

2. Compute denominator (sum of exponentials):

$$0.368 + 1 + 20.09 + 148.41 = 169.87$$

3. Compute Softmax probabilities:

$$\begin{aligned} s(-1) &= \frac{0.368}{169.87} = 0.002 \\ s(0) &= \frac{1}{169.87} = 0.006 \\ s(3) &= \frac{20.09}{169.87} = 0.118 \\ s(5) &= \frac{148.41}{169.87} = 0.874 \end{aligned}$$

Thus, the final probability distribution is:

$$[0.002, 0.006, 0.118, 0.874]$$

This means the model is **87.4% confident in class 5** and assigns lower probabilities to other classes.

2.9 Hardmax Layer Design

2.9.1 Background

- **What is hardmax?** Hardmax is a function that converts a set of values into a one-hot encoded vector, where only the index with the highest value is set to 1 and all others are set to 0.
- **Why Hardmax?** Hardmax is useful during inference when we want to make a final decision and select the most likely class. It is deterministic and discrete, which makes it appropriate for classification output.

2.10 Hardmax Equation

Given an input vector $x = [x_1, x_2, ..., x_n]$, the Hardmax function outputs a vector $h = [h_1, h_2, ..., h_n]$, where:

$$h_i = \begin{cases} 1 & \text{if } i = \arg\max_j x_j \\ 0 & \text{otherwise} \end{cases}$$

where:

- x_i represents the raw input (logit) for class i.
- $\arg\max_j x_j$ returns the index of the maximum value in the input vector.
- The result is a one-hot vector with a 1 at the index of the largest value.

2.10.1 Sample Calculation

1. Input vector:
$$x = [-1,\ 0,\ 3,\ 5]$$
2. Find index of maximum value:
$$\arg\max(x) = 3 \quad \text{(since } x_4 = 5 \text{ is largest)}$$
3. Convert to one-hot encoding:
$$h = [0,\ 0,\ 0,\ 1]$$

Thus, the final hardmax output is:

$$[0,\ 0,\ 0,\ 1]$$

This means the model predicts class 3 with full confidence and disregards the other classes completely.

3 Hardware Design

3.1 Number Representation

3.1.1 Qm.n format

Q format is a standardized way to represent fractional numbers in hardware. We cannot use floating point as the FPGA we are targeting does not have enough resources to implement a floating point model. In Qm.n format, m is the number of bits that will be used to represent the whole number portion of the number. n is the number of bits that will be used to represent the fractional portion of the number. Based on the hardware used, it was determined that 16 bits was enough to fit our design on a FPGA.

3.2 Fixed-Point Format Selection

After running the software model, it was determined that the number range that is present in our system is between **-3.1 to 3.1**. To represent this in hardware we need to use To represent numbers in the range -3.1 to $+3.1$ using fixed-point arithmetic with a 16-bit word, we select an appropriate Q-format that maximizes precision while avoiding overflow.

Bit Allocation

The 16-bit word must include:

- 1 sign bit (for two's complement representation)
- m integer bits
- n fractional bits

Thus, we must satisfy:

$$1 \text{ (sign)} + m \text{ (integer)} + n \text{ (fraction)} = 16$$

Determining Integer Bits

To safely represent the maximum magnitude value 3.1, we need:

$$2^m > 3.1$$

Solving this, we find:

$$m = 2 \quad \text{since} \quad 2^2 = 4 > 3.1$$

Fractional Bits

Given $m = 2$ and 1 sign bit, the remaining bits for the fractional part are:

$$n = 16 - 1 - 2 = 13$$

Selected Q Format

We choose the Q-format:

Q2.13

This format uses:

- 1 sign bit
- 2 integer bits
- 13 fractional bits

Representable Range and Precision

The Q2.13 format can represent values in the approximate range:

$$[-2^2, 2^2 - 2^{-13}] = [-4, 3.99987793]$$

The smallest positive value (resolution) is:

$$2^{-13} \approx 0.00012207$$

This ensures that values within $[-3.1, +3.1]$ can be accurately represented without overflow, with high fractional precision.

3.3 Multiply Accumulate (MAC) Unit

In Convolutional Neural Networks (CNNs), the majority of computations come from convolution and fully connected layers. These layers apply a large number of dot products between learned filters (weights) and input data. Each dot product is computed through a series of **multiply-accumulate (MAC)** operations, which follow the form:

$$\text{MAC: } a \times b + c$$

where a is a weight, b is an input value, and c is the accumulation result (running sum).

Fixed-Point Constraints in MAC Design

When using fixed-point numbers—such as the Q2.13 format, where 16-bit signed integers represent values in the range $[-4.0, 3.9998]$—the MAC unit must handle several constraints:

3.3.1 Saturation Handling

Without saturation, any arithmetic overflow may wrap around due to two's complement behavior. For instance, a small positive overflow can produce a large negative result, leading to incorrect predictions. Saturation logic ensures that results exceeding the representable range are clamped to the maximum or minimum:

```
if (result > MAX_Q213)
    output = MAX_Q213;
else if (result < MIN_Q213)
    output = MIN_Q213;
else
    output = result;
```

Truncation and Rescaling Multiplying two Q2.13 values produces a Q4.26 intermediate result:

$$Q2.13 \times Q2.13 = Q4.26$$

To convert this back to Q2.13, the result must be truncated by shifting 13 bits to the right:

```
wire signed [31:0] full_product = a * b;
wire signed [15:0] scaled_product = full_product >>> 13;
```

This truncation maintains numerical scale and fits the result into 16 bits.

Example 1: Convolution Filter on MNIST Input

For an 8-filter 3×3 convolution on a 28×28 grayscale MNIST image, each output pixel requires:

$$\sum_{i=1}^{9} w_i \cdot x_i$$

This results in $9 \times 26 \times 26 \times 8 = 48,672$ MAC operations per image in the first layer. Fixed-point MAC units allow this to be computed efficiently using parallelism and pipelining.

Example 2: Fully Connected Output Layer

In a final dense layer mapping to 10 output classes, with 1352 inputs (e.g., $13 \times 13 \times 8$), the number of MACs is:

$$1352 \times 10 = 13,520$$

All these operations must respect Q2.13 truncation and saturation constraints to ensure valid inference results.

Summary

CNN inference relies heavily on MAC operations. Fixed-point formats like Q2.13 are ideal for resource-constrained hardware platforms. Proper MAC unit design must include:

- **Saturation logic** to prevent overflow wraparound
- **Truncation logic** to maintain fixed-point scale

3.4 Bus Design

3.4.1 Component Interface: AXI-Stream

AXI-Stream is a lightweight and high-throughput unidirectional data transfer protocol derived from the Advanced eXtensible Interface (AXI) family. AXI-Stream is designed purely for streaming data between components, making it ideal for our application.

Core Principles

AXI-Stream operates on a simple handshake mechanism using a few key signals:

- `TVALID` – Indicates that the source has valid data on the `TDATA` bus.
- `TREADY` – Indicates that the destination is ready to receive data.
- `TDATA` – Carries the actual data payload (width is configurable).
- `TLAST` (optional) – Marks the last word of a frame (used in packetized streams).
- `TKEEP` (optional) – Marks which bytes in `TDATA` are valid (useful in wide transfers).

A data transfer occurs only when both `TVALID` and `TREADY` are asserted in the same cycle. This makes AXI-Stream inherently backpressure-aware — downstream modules can control data flow by deasserting `TREADY`.

Basic Example

Our CNN inference has a MAC unit connected to a maxpool unit, followed by a hardmax unit. Using AXI-Stream, each block only needs to manage data flow via the handshake:

```
MAC ---> maxpool ---> hardmax
 AXI-Stream     AXI-Stream     AXI-Stream
```

Each unit asserts `TVALID` when data is ready and waits for the next unit to assert `TREADY` before proceeding. This decouples timing between stages and allows for easy optimization and debugging.

3.5 Memory Interface: AXI

DDR Memory Access with AXI and MIG

The limited on-chip memory (BRAM) is not sufficient to store all weights, intermediate feature maps, or input datasets. To overcome this, off-chip memory such as DDR SDRAM is used as it offers larger storage.

Interfacing with DDR via AXI

In Xilinx FPGA designs, DDR memory is typically interfaced through an AXI-based interface using a memory controller. The AXI interface allows the logic on the FPGA fabric to read and write to DDR memory in a standardized way. Most processing cores, DMA engines, and data movers in a system are designed to communicate over AXI.

AXI4 (Memory-Mapped) is the specific AXI protocol used for this purpose. It supports:

- **Burst transfers** for high-throughput access
- **Read/write address channels** for flexible access patterns
- **Byte-level addressing** and full 32- or 64-bit data widths

Using the Memory Interface Generator (MIG) IP

To physically access the external DDR memory on the FPGA board (e.g., Nexys A7), Xilinx provides the **Memory Interface Generator (MIG)** IP core. MIG handles the low-level DDR protocol and provides a high-speed interface to the FPGA logic.

- MIG is customized for the specific DDR chip and board constraints.
- It generates a user interface that is typically AXI4 or a native interface.
- It abstracts the timing, calibration, and protocol of DDR SDRAM.

In Vivado, MIG can be configured through a GUI where the developer selects:

- DDR type (e.g., DDR3 for Nexys A7)
- Clocking and timing parameters
- Interface type (AXI or native)

Connecting MIG to AXI-Based Systems

When configured to use the AXI interface, the MIG module exposes AXI master/slave ports that can be connected to AXI interconnects or directly to AXI master peripherals. For example, a DMA engine or custom logic (CNN accelerator) can read weights from DDR using AXI reads and write results using AXI writes.

Data Movement

To efficiently move data between DDR and processing blocks, a **DMA (Direct Memory Access)** engine is often used. AXI DMA connects:

- One end to AXI4-MM for DDR
- The other end to AXI-Stream for internal processing

This decouples processing from memory latency and enables streaming computation without stalls.