Resource and Instruction Development for New Binary Neural Network Class Project

Priyanshu Arora, Mason Kellogg, Morgan Lafferty

December 14, 2023

Abstract

Binary Neural Networks have emerged as compact, yet powerful, neural network architectures. Given their incredible potential in the embedded deep learning space, our final project is centered around exposing future students to the workings and benefits of BNNs. We have helped expand out a future class project in which students will implement a BNN using XNOR and popcount operations and explore tradeoffs of various design choices. To facilitate future successful completion of this project, we focused our efforts on resource and instruction development. This included writing a new chapter on BNNs for the Parallel Programming for FPGAs textbook with thorough explanations and instructive diagrams that will teach students the fundamental concepts behind BNNs and how they work. Additionally, we moved and modularized the existing python BNN code into a jupyter notebook that will be much easier for students to follow and added extensive commentary to clarify technical details. With these improvements, this notebook should serve as a good reference for students to get started with implementing their own BNN in hls. To aid hls model development and debugging, we also expanded the hls testbench to include tests for individual model layers and added helper functions to unpack the imported weights and inputs. Finally, we edited the preliminary project instructions by correcting typos, expanding upon explanations and adding clarifications where needed. In culmination, our final project serves to improve students' understanding of BNNs and provide the necessary tools for them to independently implement a design of their own.

1 Introduction

Within the past few years, artificial intelligence has grown from a topic discussed primarily in the technology world to one amassing a great amount of attention from the general public. The development and release of large-scale models, like the GPT series, has spurred AI to quickly be adopted across many domains and is undoubtedly reshaping the world we live in. However, the computational complexity and memory demands of many of these models is astronomical, not to mention the increasing carbon footprints of training them. Consider for instance, GPT-4, which has 1.76 trillion parameters and is rumored to have taken around 100 days to train using 25,000 GPUs [11]. Unfortunately, the problem is not just isolated to large language models. Even average size deep neural networks often require resources on a scale that is infeasible for small devices due to their reliance on full precision values. Thus, the sheer size and complexity of these models is greatly limiting to their applicability, especially in resource-constrained environments. This need for more efficient architectures remains a central focus in the machine learning community and has sparked many active areas of research.

Model quantization is one such technique that looks to reduce the computational and memory costs of machine learning models by reducing the precision of the network's weights and activations. Taking this idea to the extreme, Binary (or Binarized) Neural Netowrks(BNNs) emerged as a new architecture that operate solely with binary values. BNNs, by replacing floating point values with binary weights and activations and reducing all costly matrix multiplications to simple bitwise operations, yield more efficient training and inference. This fundamental shift not only simplifies computation, but also provides a suite of other benefits including accelerated inference times, real-time decision making capabilities, and hardware-friendly designs. Further, the compactness of these models widens the scope of their usability, making them strong candidates for deployment in edge devices, IoT applications, or other scenarios where resources are limited.

The motivation for our final project is threefold: (1) As expressed above, increasing the efficiency of deep learning models is crucial to enabling groundbreaking applications in the future. (2) BNNs are an excellent example of a design optimization that has the potential to transform the embedded deep learning space. (3) A central focus of this class was on optimizing various designs, but no current class projects deal with this in the context of neural networks. To address these needs, our final project focused on expanding the class curriculum to include a new project on binary neural networks. By doing do we hope to teach future students an interesting new architecture while also invoking thought as to the challenges model efficiency (or lack-thereof) poses in modern AI systems and potential solutions to address these.

Our resource and instructional development efforts yielded the following major contributions. First, is a new book chapter on BNNs for the Parallel Programming for FPGAs class textbook. This will set the stage for successful completion of the new project by providing a comprehensive overview of BNN fundamentals. With thorough explanations and clear visualizations, it is our intention that students understand how BNNs differ from typical deep neural networks, have a sense as to how the architecture and model inference/training stages work, and finally, feel confident in implementing their own design after reading the chapter. We have also reworked the exisitng BNN code (provided to us by instructors) to have a more tutorial-like structure to facilitate student learning. This involved porting the code to a jupyter notebook - an easy and familiar interface for many students - re-organizing the code so that functions are grouped by their primary purpose, and finally, adding commentary to elucidate technical details within individual functions. Next, are improvements we made to the hls environment in which students will be required to implement and optimize their own designs. We expanded the comprehensiveness of the hls testbench by adding the capability to test individual layer activations in the BNN model. These new tests will help direct students as to where their model implementation is going astray, greatly aiding the debugging process. Further, we found the existing process for packing input samples/weights for the hls code to be confusing and an unnecessary challenge on top of the assigned project task of implementing the BNN feed forward function. Hence, we created helper functions to reverse the packing of samples/weights within hls so students can focus on the design itself. Lastly, we proof-read and edited the preliminary project instructions to ensure that explanations and expectations were clear for students going forward. It is our hope that the culmination of our work will encourage student learning, introduce an interesting new topic, and enable successful completion of the new BNN class project. The following sections will dive into the details of each of the deliverables mentioned above.

2 Background

This background will focus on two main areas:

- A description of the state of the BNN project before our edits.
- A high level description of BNNs.

2.1 Background - BNN Project Before Edits

Before our edits, the BNN project consisted of the following:

- Bare bones project instructions including a few sentence introduction, a one-sentence description of the feed forward function python implementation, and a few tasks.
- Python files for the dataset, network weights, and a BNN class implemented in object-oriented programming. Note that the BNN class was well structured, but lacked documentation and a clear order of the functions. It was also difficult to understand the preprocessing that occurred on the 28x28 MNIST images before including them in the test bench.
- HLS test bench with a few MNIST examples; testing only for correct output label, not weights in each layer in the BNN.
- No chapter in the FPGA textbook about BNNs.

2.2 Background - BNN Theory and Implementation

The chapter we wrote for the Parallel Programming for FPGAs textbook titled "Binary Neural Networks (BNNs)" provides the reader with sufficient background to understand BNNs, assuming they have a basic understanding of neural network layers and activation functions. Here, we provide a very brief review of the main points of the chapter.

2.2.1 BNN Chapter - Overview, Motivation

Binary (or Binarized) Neural Networks have emerged as compact, yet powerful, neural network architectures for embedded deep learning. At their core, BNNs operate with binary weights and activations rather than full precision values as in typical deep neural networks. The development of BNNs has largely been driven by the need for more efficient architectures, particularly in resource-constrained environments. BNNs are able to drastically reduce the computational complexity and memory footprint of deep learinng by employing binary values and bitwise operations rather than floating point operations in full precision neural networks.

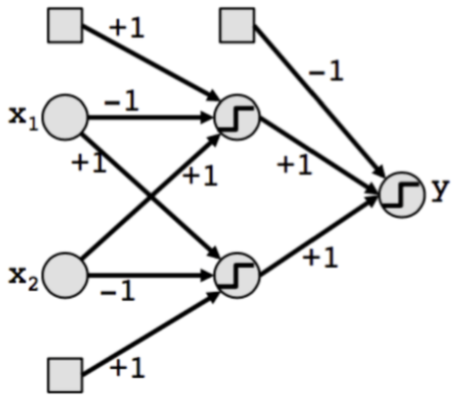

Figure 1: High-level depiction of BNN

2.2.2 BNN Chapter - XNOR, Popcout for Multiplication

In place of floating point matrix multiplication, BNNs use XNOR logic gates and popcount operations. XNOR logic gates (with 0's replaced with -1's) simulate a binary (with +1, -1) multiplication. The population count (popcount) operation counts the number of bits set to +1 in a binary vector. In combination, XNOR and popcount operations can replace normal MAC (multiply-accumulate) operations while using significantly less computational resources.

2.2.3 BNN Chapter - Binarizing a Network

Typically, the Sign function is used to "binarize" a network - that is, convert the neural network's floating point weights and bias into binary values of -1 or +1. The Sign function is defined as:

$$x^b = Sign(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

where x^b is the binarized weight or activation and x is the original floating point value.

2.2.4 BNN Chapter - Forward Propagation

Recall that in deep learning, forward propagation refers to model inference. That is, given an input, use the model's learned parameters to predict the output. For BNNs, the forward pass is similar to that of any typical deep neural networks architecture with the exception that all weights and activations

are binarized to either -1 or +1. For the BNN lab, there is code for a forward pass in a BNN using binarization, XNOR, and popcount operations ("quantized"). There is also code for a forward pass in a BNN using multiply and accumulate loop for comparison. Below we have included the quantized forward propagation function from the lab for a 3 layer BNN.

```
def feed_forward_quantized(self, input):
    #param input: MNIST sample input

    # layer 1
    X0_input = self.quantize(self.sign(self.adj(input)))
    layer1_output = self.matmul_xnor(X0_input, self.fc1w_qntz.T)
    layer1_activations = (layer1_output * 2 - 784)

    # layer 2
    layer2_input = self.sign(layer1_activations)
    layer2_quantized = self.quantize(layer2_input)
    layer2_output = self.matmul_xnor(layer2_quantized, self.fc2w_qntz.T)
    layer2_activations = (layer2_output * 2 - 128)

    # layer 3
    layer3_input = self.sign(layer2_activations)
    layer3_quantized = self.quantize(layer3_input)
    layer3_output = self.matmul_xnor(layer3_quantized, self.fc3w_qntz.T)

    final_output = (layer3_output * 2 - 64)
    A = np.array([final_output], np.int32)

    return A
```

2.2.5 BNN Chapter - Backward Propagation

Backpropagation (short for "backwards propagation of errors") is a fundamental algorithm used to train supervised learning algorithms - BNNs being one of them. At the end of the forward pass (forward propagation), the network computes the loss (error) by comparing its outputs to the ground truth values using a loss function such as mean squared error for regression tasks or cross-entropy for classification tasks. The algorithm then calculates the gradient of the loss function with respect to each weight in the network by applying the chain rule. Backpropagation gets its name from the fact that this chain rule process starts at the output layer and works backward through the network. The weights are then updated to minimize the loss (error) function. This update process is called gradient decent.

Since the derivative is 0 nearly everywhere for the Sign activation function, for backpropagation in BNNs, we use the hard tanh function to calculate the gradient of the loss with respect to the weights. This is equivalent to a concept called a straight-through estimator (STE).

The lab does not include code for backpropagation in BNNs. The focus of the lab is not in training a BNN, rather the focus of the lab is in inference - that is, predicting labels of MNIST images. In a future lab (maybe a follow-up to this lab), students could train a BNN in hardware.

2.2.6 BNN Chapter - Inference, Advantage, Limitations

At this stage, the BNN has been trained using multiple iterations of forward and backward propagation. An input to a BNN is a list (x vector). In the lab, the input to the network is a 28x28 grayscale pixel value image of a hand-written digit. The network, having taken in a vector x, will perform the following operations to predict the output, y (referred to as inference in neural networks). In the lab, the output y is a prediction of the hand-written digit (0-9). During the forward pass, the input data is processed through the network's layers, using the trained binary weights and activations. In BNNs, the standard dot product in the neurons is replaced by XNOR and popcount operations. These operations are much more efficient than floating-point multiplications and additions. At each layer, after the XNOR and bitcount operations, an activation function is applied (Sign). BNNs often use batch normalization to stabilize and speed up training. Finally, at the output layer, we receive our value for y (regression) or pass the final vector through a soft max (classification, as we see with digit classification of MNIST images in lab).

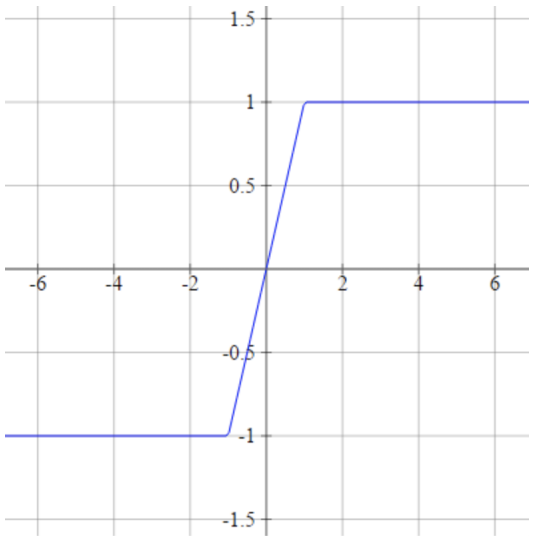

Figure 2: STE estimation of Sign function, as hard tanh function.

The key advantage of BNNs is that they require significantly less computational power and memory than traditional neural networks, making them well-suited for resource-constrained environments like mobile devices or embedded systems. However, the trade-off is that the binarization of weights and activations can lead to a loss in model accuracy, especially for complex tasks. In simple tasks such as number recognition, the BNN performs well, although requiring more layers than a full-precision NN. For complex tasks such as high-resolution image classificaiton and natural language processing, the lower expressiveness of the BNN may struggle to recognize complex patterns.

3 Implementation and Results

3.1 Project Objective:

The primary aim of this project was to enhance the Binary Neural Networks (BNNs) project, focusing on specific deliverables while exploring additional avenues beyond initial project scope.

3.2 Achieved Deliverables:

3.2.1 Chapter Development for "Parallel Programming for FPGAs":

Our team successfully drafted a comprehensive chapter dedicated to Binary Neural Networks (BNNs) for Prof Kastner's book. This chapter encompassed an extensive overview of BNNs, elucidating their motivation and practical utility. Detailed explanations of key BNN operations, such as the XNOR operation, popcount, and their application in vector dot products, were provided. To enhance comprehension, we incorporated diagrams and illustrations. The chapter elaborated on forward and back-propagation algorithms, supported by pseudo code and actual C++ implementations. These efforts resulted in a comprehensive resource contributing to the understanding of BNN concepts and their practical implications. We have created a pull request into Prof Kastner's GitHub repository for the book.

3.2.2 Conversion to Jupyter Notebook Format:

The Python BNN code underwent a meticulous restructuring process, transforming it into a step-by-step Jupyter notebook. This interactive resource was designed to enhance comprehension of BNN concepts by employing practical demonstrations with the MNIST dataset. Initially, the notebook loads the MNIST data and presents visualizations detailing its dimensions. Subsequently, it guides through a demonstration comparing standard matrix multiplication with XNOR-based multiplication. Moreover, it elucidates the BNN's forward pass by using dummy inputs and weights to demonstrate

its functionality. The notebook then defines the BNN model, replicating the original Python script. Utilizing this model, inferences are made on the MNIST dataset, and the resulting scores are displayed. Additionally, certain bug fixes, such as resolving issues with the packing weights function, were incorporated to ensure smooth functionality. Notably, the notebook's structure is designed to engage and educate students while navigating through these intricate concepts.

3.2.3 Improvement of Project Instructions:

We improved upon the preliminary project instructions which included both typos and unclear language in an effort to make it easier to follow. This involved restructuring the materials sections to clearly outline the files that are available in the GitHub, what each of their purpose is, and which ones should be modified. We also corrected the way in which the example code was displayed (i.e. making sure it was displayed in a code block) and edit the "task" section to have more direct and actionable tasks for the students to complete. Additionally, we corrected any grammar, spelling errors, or typos we could find. We have created a pull request into Prof Kastner's GitHub for the updated instructions.

3.3 Supplementary Contributions:

3.3.1 Enhancements to Test Bench Functionality:

Beyond initial project goals, we extended the functionality of the existing HLS test bench. This modification enabled individual layer output testing, significantly aiding debugging processes.

3.3.2 C++ Function for Unpacking Inputs:

Recognizing the necessity to unpack packed BNN weights and input data for HLS code usage, we developed a helper function. This function efficiently converts packed inputs/weights into a binary bit array, intending to simplify the learning process for students engaging with the material.

3.3.3 C++ Implementation of feed_forward_quantize():

The Python-based forward pass function was translated into C++ to maintain consistency with the book's context and offer students a reference point for their own C++ implementation. This additional implementation aims to support students in writing their own forward pass functions in C++. The functions is a forward pass for a 2 layer BNN. The input to the function is the input data (X), the flattened weights matrices as arrays ($W1$, $W2$), pointers to the activations of each layer ($layer1\_activations, layer2\_activations$) and the dimensions of the input and weights. There is no output, but the $layer1\_activations, layer2\_activations$ are filled with the proper forward pass layer activations from the BNN. This implementation is consistent with the lab and the testbench ensuring simplicity for the students.

4 Challenges

4.1 HLS4ML Environmental Setup and Tool Compatibility:

Our initial objective centered around creating a lab for HLS4ML tutorials, which required an Ubuntu environment—a platform unfamiliar to our team members. Subsequently, we embarked on setting up a virtual machine. However we encountered complications when downloading Vitis within the virtual environment as well as when building the docker image required for hls4ml, prolonging our troubleshooting efforts. After weeks of persistent attempts to resolve these issues, we determined it was prudent to pivot towards a different project, leading us to transition to the BNN project.

4.2 Resource Scarcity for BNN Understanding:

A significant hurdle emerged in sourcing comprehensive learning material for Binary Neural Networks (BNNs). Despite extensive searches across platforms like Medium Articles and Towards Data Science, we struggled to find adequate explanations, particularly regarding backpropagation in BNNs. Fortunately, a pivotal discovery of an instructive YouTube resource immensely clarified these concepts.

Additionally, delving into the original paper titled "Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or 1" significantly enriched our understanding. The chapter we've crafted aims to address this scarcity, offering a comprehensive resource for fellow students.

4.3 Understanding Input/Weight Packing Procedures:

Another obstacle we faced involved grasping the input/weights packing process. Initially, we struggled to comprehend this procedure, but through debugging and stepping through the code, we managed to grasp its essence. We realized that packing essentially involves converting an array of bits into an array of 32-bit integers. However, despite this understanding, we remained puzzled about the necessity of this packing. Even now, we're not entirely clear on its purpose. Moving forward, it's imperative to seek clarification and delve deeper into why packing is essential. Our current assumption is that since we're embedding the weights and inputs directly into the C++ program, packing serves as a means to streamline the array for hardcoding purposes.

5 Future Work

Next steps involve revising the chapter, aiming to enhance clarity by adding necessary explanations while trimming redundant details. Additionally, we aim to introduce elucidations regarding optimizations within the code. Specifically, we'll delve into low-level optimizations like array partitioning and loop unrolling, highlighting their potential to accelerate the forward pass functions.

Expanding the project questions stands as another area for future development. By crafting more engaging and stimulating queries, we aim to encourage deeper exploration by the students.

Moreover, integrating C++ code functions for backpropagation remains on the agenda. This will involve not only implementing these functions but also designing a separate lab module that tasks students with mastering backpropagation, illuminating the intricate process through which Binary Neural Networks learn from data.

An intriguing avenue for future exploration involves constructing a Binary Neural Network using HLS4ML, aiming to discern and assess performance disparities compared to other methodologies. This exploration could unveil novel insights into the network's behavior and efficiency under different frameworks.

6 Conclusion

Binary Neural Networks require significantly less computational power and memory than traditional neural networks, making them well-suited for resource-constrained environments like mobile devices or embedded systems. In this project, we improved a future class project in which students will implement BNN inference using XNOR and popcount operations and explore tradeoffs of various design choices. We wrote a new chapter on BNNs for the Parallel Programming for FPGAs textbook with thorough explanations and instructive diagrams that will teach students the fundamental concepts behind BNNs, as well as provide questions and advice on implementation in code. Additionally, we modularized and reordered the existing python BNN code into a jupyter notebook and added thorough documentation. With these improvements, this notebook should serve as a good reference for students to get started with implementing their own BNN in hls. To aid hls model development and debugging, we also expanded the hls testbench to include tests for individual model layers and added helper functions to unpack the imported weights and inputs. Finally, we edited the preliminary project instructions by correcting typos, expanding upon explanations and adding clarifications where needed. Our final project serves to improve students' understanding of hardware benefits and performance limitations of BNNs. We provide the necessary tools for students to independently implement and test their own BNN inference design in HLS.

References

[1] Binary neural networks. https://docs.google.com/presentation/d/10lVe51Nh7w_qmhlYheaYP1b8q6iOsutZ/edit#slide=id.p1. Lecture, n.d.

[2] Ee545 (week 10) "binary neural networks" (part i). https://www.youtube.com/watch?v=5K6ko3H_ePg&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=23&t=338s. YouTube video, 2020.

[3] Ee545 (week 10) "binary neural networks" (part ii). https://www.youtube.com/watch?v=LcECmeFqbrI&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=24. YouTube video, 2020.

[4] Ee545 (week 10) "binary neural networks" (part iii). https://www.youtube.com/watch?v=RC_8bPv0uhM&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=25. YouTube video, 2020.

[5] Ee545 (week 10) "binary neural networks" (part iv). https://www.youtube.com/watch?v=P6sD8lI61Uk&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=26. YouTube video, 2020.

[6] Ee545 (week 10) "binary neural networks" (part v). https://www.youtube.com/watch?v=rHa0-mG5SuM&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=27. YouTube video, 2020.

[7] Ee545 (week 10) "binary neural networks" (part vi). https://www.youtube.com/watch?v=z8CclU5wKuY&list=PLC89UNusI0eSBZhwHlGauwNqVQWTquWqp&index=28. YouTube video, 2020.

[8] Neural network. https://docs.google.com/presentation/d/1oC1Z_LzzlGMdDCFpC9rp6U2udwSpCaj-/edit#slide=id.p1. Lecture, n.d.

[9] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. arXiv preprint arXiv:1602.02830, 2016.

[10] F. Lin. Xnor neural networks on fpga. http://cs231n.stanford.edu/reports/2017/pdfs/118.pdf. CS231n: Deep Learning for Computer Vision report, n.d.

[11] K. G. A. Ludvigsen. The carbon footprint of gpt-4. https://towardsdatascience.com/the-carbon-footprint-of-gpt-4-d6c676eb21ae. Medium article, July 18, 2023.

[12] Natsu. Paper explanation: Binarized neural networks: Training neural networks with weights and activations constrained to +1 or 1. https://mohitjain.me/2018/07/14/bnn/. Blog post by Mohit Jain, August 16, 2018.

[13] V. K. Ojha. Binary neural networks: A game changer in machine learning. https://medium.com/geekculture/binary-neural-networks-a-game-changer-in-machine-learning-6ae0013d3dcb. Medium blog post, February 19, 2023.

[14] C. Yuan and S. S. Agaian. A comprehensive review of binary neural network. Artificial Intelligence Review, 2023. Pages 1-65.