Single Bit Neural Nets Did Not Work

Joseph Camacho
Undergrad in Artificial Intelligence
MIT, Cambridge, MA

James Camacho
Undergrad in Artificial Intelligence
MIT, Cambridge, MA

***Abstract*—We originally planned to make and train a neural network with single bit activations, weights, and gradients, but unfortunately the neural network did not train very well. We were left with a peculiar looking CPU that we tried adapting to mine bitcoin and run Brainfuck. We successfully implemented Brainfuck but ran out of time before creating a bitcoin miner.**

Our code can be found at this HTTPS URL: https://github.com/cooljoseph1/bitnet

I. Introduction

As neural networks have increased in size, training and running them has become more expensive, both in terms of money and environmental impact. GPT-4, for example, is rumored to have cost more than $100 million and taken over 7,000 MWh of energy to train. For context, this is more energy than the entire city of New York uses in an hour.

We thought we could make training neural networks more efficient by using single bits for the parameters, activations, and gradients instead of floating point numbers. Ordinary floating point numbers take 32 bits, though large neural networks are typically trained using 16 bit floats instead. Furthermore, floating point operations are much slower than bitwise operations. Being able to train a neural network using single bits instead and binary operations would make training neural networks more efficient by *at least* two orders of magnitude.

This paper will first present our intended algorithm for the neural network, second explain how we implemented it on the FPGA, third hypothesize what went wrong, and fourth conclude with our attempts at adapting our CPU to be a bitcoin miner and Brainfuck interpreter.

II. Algorithm

Let a high bit represent -1 and a low bit represent 1. Then the "exclusive or" becomes multiplication and we can represent a majority of three bits x_0, x_1, and x_2 as

$$\operatorname{round}\left(\frac{x_0 + x_1 + x_2}{3}\right),$$

where round rounds to the closest odd integer (i.e. -1 or 1 because the range is between -1 and 1). We can assign weights w_0, w_1, and w_2 to these bits to get a function

$$f(w_0, w_1, w_2; x_0, x_1, x_2) = \operatorname{round}\left(\frac{w_0x_0 + w_1x_1 + w_2x_2}{3}\right).$$

If we fix $w_0 = w_1 = w_2 = 1$ and $x_2 = -1$, f implements a NAND gate between x_0 and x_1. So, this function can implement arbitrary circuits. This function forms the basis of our neural network.

f can be implemented using three exclusive or gates for the multiplications and a majority gate for the rounding.

We can take derivatives of this function by approximating the rounded value y as

$$\tilde{y} = \frac{w_0x_0 + w_1x_1 + w_2x_2}{3}.$$

E.g.,

$$\frac{\partial y}{\partial x_0} \approx \frac{\partial \tilde{y}}{\partial x_0} = \frac{w_0}{3}.$$

The gradients for the weights are similar.

f works fine if we just want to downsample by three at every layer, but sometimes we want to keep the number of outputs the same as the number of inputs, such as in a residual layer. We can do this by using each triple (x_0, x_1, x_2) three times with different weights, resulting in three different outputs (y_0, y_1, y_2). I.e., use the function $F : \mathbb{R}^{3\times3} \times \mathbb{R}^3$ where

$$F(W, \mathbf{x}) = \begin{pmatrix} f(W_0, x) \\ f(W_1, x) \\ f(W_2, x) \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}.$$

The gradients can be combined together by rounding. If we set $g_i = \frac{\partial \text{Loss}}{\partial y_i}$ the gradients for the backward pass become

$$\frac{\partial \text{Loss}}{\partial x_0} = \operatorname{round}\left(\frac{W_{00}g_0 + W_{10}g_1 + W_{20}g_2}{3}\right),$$
$$\frac{\partial \text{Loss}}{\partial x_1} = \operatorname{round}\left(\frac{W_{01}g_0 + W_{11}g_1 + W_{21}g_2}{3}\right), \text{ and}$$
$$\frac{\partial \text{Loss}}{\partial x_2} = \operatorname{round}\left(\frac{W_{02}g_0 + W_{12}g_1 + W_{22}g_2}{3}\right).$$

This looks exactly the same as the forward pass! Indeed, the gradients at the inputs are $F(W^T, (g_0, g_1, g_2))$. This serendipitous result lets us save space on the FPGA by combining the logic for the forward and backward pass together into a single module.

F can be implemented using three copies of f.

III. Architecture

A. Neural Network–Forward and Backward

The smallest building block on the FPGA is the 3-Majority block which implements f, as shown in Figure 1.

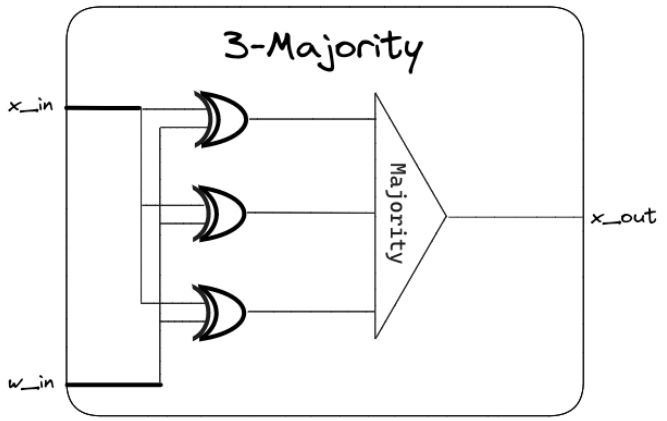

Fig. 1. The 3-majority is the building block of our network, a majority between three inputs XOR'ed with three weights. There are three symmetries: the inputs can be permuted, the inputs and weights can be swapped, and the forward and backward passes are very similar. It takes exactly one logic unit on the Spartan 7 FPGA.

These can be combined together in two ways to form two kinds of layers. If we simply apply f to every group of three inputs, we can get a downsample layer as shown in Figure 2. We can downsample in multiple ways by changing how we group together the inputs. One simple way of doing this is by grouping together inputs whose indices are all the same in ternary except for a single trit.

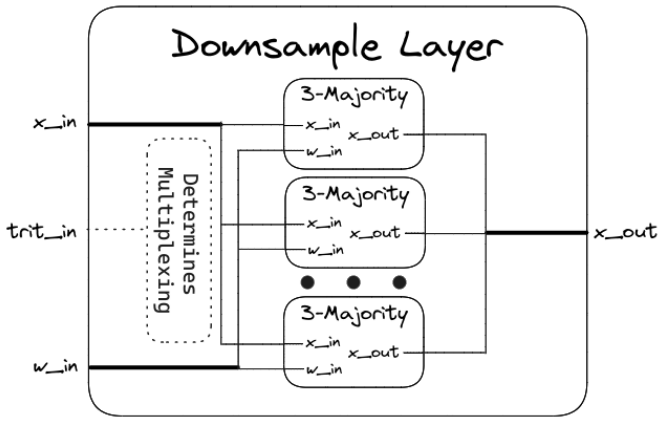

Fig. 2. Performing a 3-majority between inputs whose indices only differ on a single trit gives one-third as many outputs. For example, a 27×27 image can be downsampled to 9×9 by selecting trits 0 and 3.

On the other hand, if we apply F to every group of three inputs we get an interweave layer as shown in Figure 3. Like the downsample layer, we can group together the inputs with multiplexing based on a single trit.

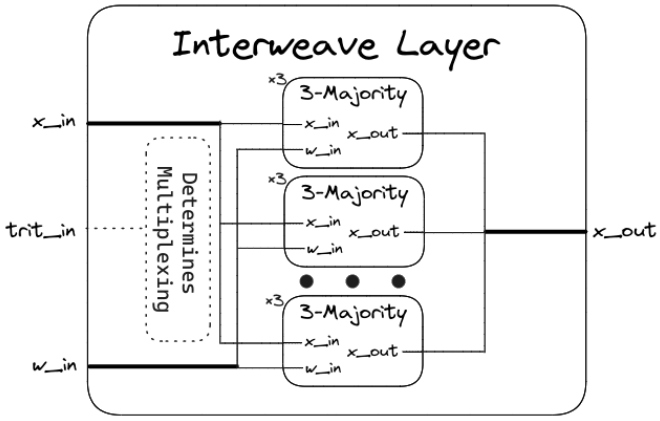

Fig. 3. Using each triple of inputs three times lets us keep the number of outputs the same as the number of inputs.

The interweave layer is the largest block we actually implemented on the FPGA. We chose to make this the largest layer because the parallel block for gradient descent takes 9 times as many logic units (so 9K logic units, if the interweave layer takes 1K logic units as we estimated), and we would not be able to fit more than a few interweave blocks on the FPGA. The rest of the network we implemented by designing a central processing unit to cycle through appropriate instructions, as we explain more about in Subsection III-D.

Combining together a series of interweave layers with different groupings allows each input to interact with every other input, giving a fully connected layer as shown in Figure 4.

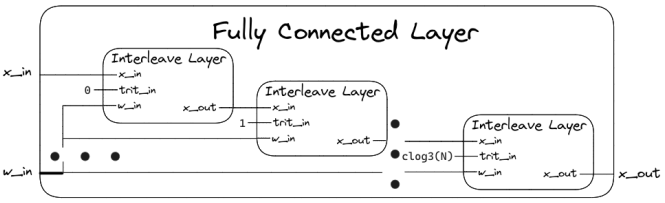

Fig. 4. Looping through each trit connects every input to every output.

Finally, we can make a residual layer by combining multiple fully connected layers in parallel and inverting the inputs if all the fully connected layers agree that the inputs should be inverted. This is shown in Figure 5.

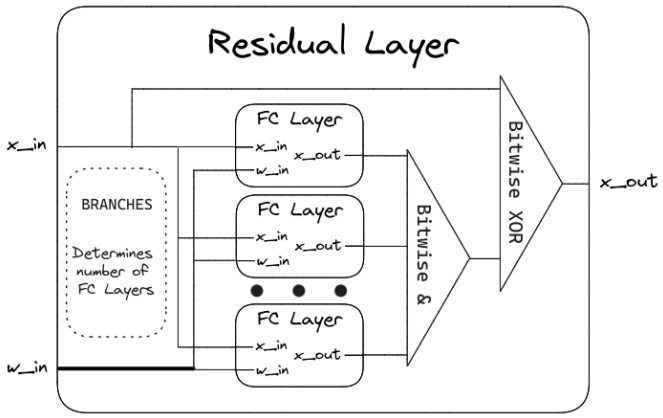

Fig. 5. Often the outputs will be close to correct, only needing slight tweaking. The residual layer creates several fully connected layers, and only if all agree that a bit needs changing does it flip the bit. The direct connection backward also solves the vanishing gradient problem deep networks are susceptible to.

B. Neural Network—Gradient Descent

We decided to use a variant of stochastic gradient descent to update the weights. Subsequent layers propagate backwards whether their inputs should change, and weights randomly switch with probability $1/256$ to the correct value to make this happen. We chose to use "change" values instead of gradients because true gradients would be useless half the time—if the weight is already 1, what update should come about if the gradient is also positive?

This backprop layer looks similar to the interweave layer, except it outputs two values: Gradients at the beginning of the interweave layer and gradients at the weights. This is pictured in 6.

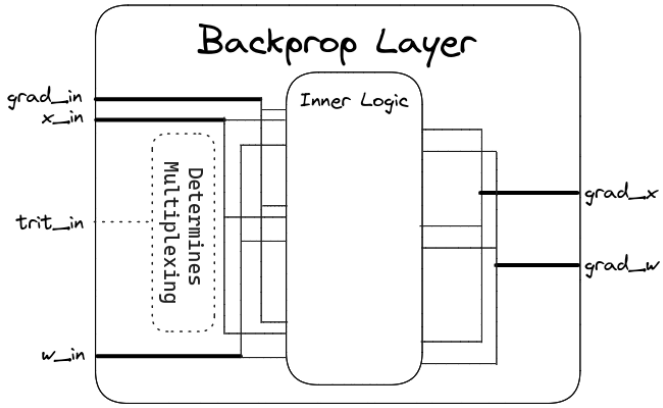

Fig. 6. Given input "gradients" denoting whether the outputs of an interweave layer should change, the backprop layer computes gradients for the inputs and weights of the interweave layer.

To avoid needless clutter, we have left the inner logic of the backprop layer out of Figure 6. The ith weight gradient is the same as the the $\lfloor i/3 \rfloor$th gradient in if $w_i \wedge x_{\lfloor i/3 \rfloor}$ is in the majority of the output bits. Otherwise, the ith weight gradient is the inverse of this value. This makes a weight update always flip their inputs to what the gradient asks for.

Backpropagating through multiple layers is done using the CPU, which we explain more in Subsection III-D.

C. Buffers, a.k.a. "Mediums"

The BRAMs on the Spartan 7 FPGA only has a maximum width of 72 bits. This is not nearly enough to fit in a single binarized MNIST image, let alone the weights for a layer! One method to store these into memory is to use multiple BRAMs in parallel, but *this would use too many BRAMs*. There are only 75 BRAMs on the device. If the MNIST data takes $729/72 = 11$ BRAMs to save, the weights take an additional $2187/72 = 31$ BRAMs to save, and our heap takes another $729/72 = 11$ BRAMs to save, we would use at least $11 + 31 + 11 = 53$ BRAMs to save. This is not much slack—and we ran into some problems initially with running out of BRAMs.

Instead, we chose to split each datum or weight vector across multiple BRAMs and read them into a buffer when needed. We made modules called "mediums" that did this. This used fewer overall BRAMs because we did not need as much storage as 11 BRAMs would give for the heap or data. 7 shows the schematic for our medium.

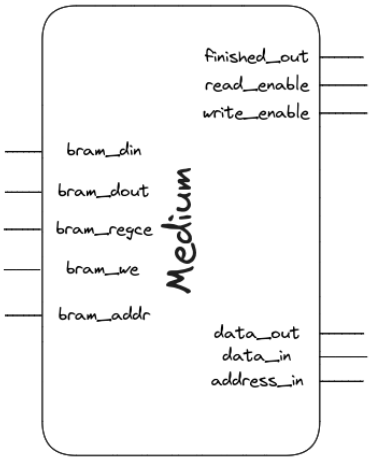

Fig. 7. A schematic of the "medium', a module that stores and retrieves values across multiple addresses in a single BRAM.

The reading and writing from BRAM was the slowest operation our CPU performed by a large margin (taking ~ 60 clock cycles because we were a bit inefficient in our implementation)—but our real limiting factor was communications with the external computer so we were not overly concerned about this.

D. Central Processing Unit

Our CPU had two data registers and two weight registers. The data registers, named `X` and `Y`, stored data and activations. The weight registers, named `W` and `G`, stored weights and gradients.

We also connected our CPU to various mediums. The data medium, weight medium, and instruction mediums provided

data, weights, and instructions, respectively (albeit the instruction medium was technically just a loose interface to the BRAM). The data medium saved an x and y value together; it had twice the width of a data register. Furthermore, we created a medium to what we called a *heap* (named for the heap in C) to store intermediate activations for later use in backpropagation.

Next, our CPU had four pointers. The instruction, data, weight, and heap pointers pointed to addresses in their respective mediums. We denoted these pointers as `I`, `D`, `A`, `H`.

Finally, our CPU stored two more values: a six bit value `trit` which stored the trit to use for muxing in the interweave and backprop layers, and a value `inference` the size of the data registers. This final value was shared with our communications to be sent back to the computer. (The original idea was to have the FPGA be used as a digit recognizer—you would send it an image over UART and it would return what digit it is.)

The complete instruction set for our CPU can be found in Table I. All instructions except the first three increment the instruction pointer by 1 after they are complete. The second and third instructions increment the instruction pointer by 2 while the first sets the instruction pointer to 0.

TABLE I
INSTRUCTION SET FOR THE CENTRAL PROCESSING UNIT.

Opcode	Description
0	Set the instruction pointer to 0.
1	Set `trit` to the value of the next instruction.
2	Set `H` to the value of the next instruction.
3	Increment `D` by one.
4	Decrement `D` by one.
5	Increment `A` by one.
6	Decrement `A` by one.
7	Load the first half of the value at `D` from the data medium to `X`.
8	Load the second half of the value at `D` from the data medium to `Y`.
9	Load the value at `D` into the pair of registers {`X`, `Y`}.
10	Set the value of `inference` to `Y`.
11	Load from the weight medium at `A` into `W`.
12	Save `W` to the weight medium at `A`.
13	Load from the heap at `H` into `X`.
14	Save `X` to the heap at `H`.
15	Swap `X` and `Y`
16	Set `X := Y`
17	Set `Y := X`
18	Set `X := X ^ Y` (bitwise XOR)
19	Set `Y := X ^ Y` (bitwise XOR)
20	Set `X := X & Y` (bitwise AND)
21	Set `Y := X & Y` (bitwise AND)
22	Set `X := X | Y` (bitwise OR)
23	Set `Y := X | Y` (bitwise OR)
24	Run Interweave(x=`X`, W=`W`) and save the result in `X`.
25	Run Backprop(x=`X`, W=`W`, g=`Y`), saving dx in `Y` and dW in `G`.
26	Save StochGrad(g=`G`, W=`W`) in `W`

Note that there is no instruction to write to the data medium. This is because we did not want the FPGA accidentally overwriting correct data. Inferences can instead be read using the `inference` register that the communications module has access to.

E. Communications

To run a neural network, we need to get data, weights, and instructions to the neural network, and its inference back. We adopted the UART protocol, sending three bytes as a request header, followed up by any information we wish to communicate. An example header in least-significant bit order looks like:

$$\underbrace{1}_{\text{io}}\ \underbrace{10}_{\text{loc}}\ \underbrace{00000}_{\text{unused}}\ ||\ \underbrace{1011000\ ||\ 00000000}_{\text{address}}$$

Let's break it up:

- Sending information (input) is a zero, while receiving (output) is a one.
- The location is either the data (00), weight (01), or instruction (10) BRAMS, or additionally a special inference (11) register that can only be read out.
- Without error checking, UART needs eight bits per frame, so we pad out the byte.
- Addressing up to 1,024 instructions in the cpu takes two bytes.
- Altogether, this example asks the FPGA to transmit the instruction at address thirteen.

If instead we wished to set the instruction at address thirteen, we could send

$$\underbrace{01000000||10110000||00000000}_{\text{write instruction to address 13}}\ ||\ \underbrace{00011000}_{\text{Interweave (24)}}$$

During testing *in silico* we misattributed some communication errors to dropped bits or bytes, so we implemented two error checks:

1) The FPGA will drop a packet if it takes 10% longer than it should. For example, sending 800 bits of data at a 3Mbaud rate should take 250μs after taking into account the start and halt bits. It will drop the packet if it takes more than 275μs to receive all the data.
2) The laptop immediately reads out after every input, and resends if the stored value is incorrect.

A comparison between Fischer's `rx` module (in Manta) and ours revealed the error was actually in our code. Like Fischer (who adapted from Gisselquist), we adapted his `rx` into our communication system.

This project very much emphasized why data transfer is currently a bottleneck in machine learning: it takes around 1ms to transfer a datum. Training something like GPT-4 requires 10^{10} datums, or around 100 days.

However, once on the FPGA each datum could be run through the neural network in around 1μs, three orders of magnitude faster. If the neural network architecture was embedded on a chip, and the data continuously streamed, the slowest part on the FPGA—loading the data and weights into the registers—could be eliminated, gaining another order of magnitude speedup, and taking around fifteen minutes for the entire training run.

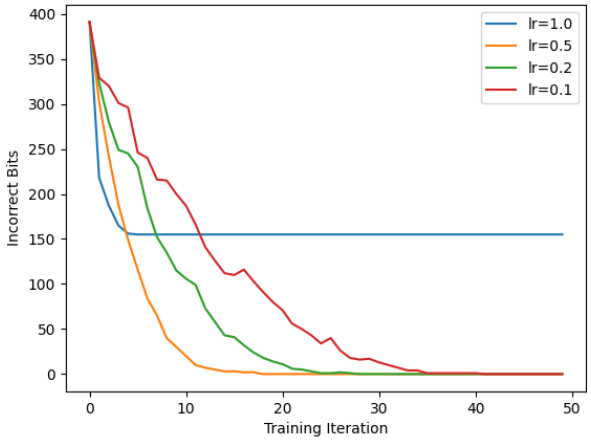

Fig. 9. Training from a constant input to output. When the learning rate is one, it flips weights that counter each other, getting stuck in a local minimum.

F. Putting Everything Together

Figure 8 shows the result of everything put together on the FPGA.

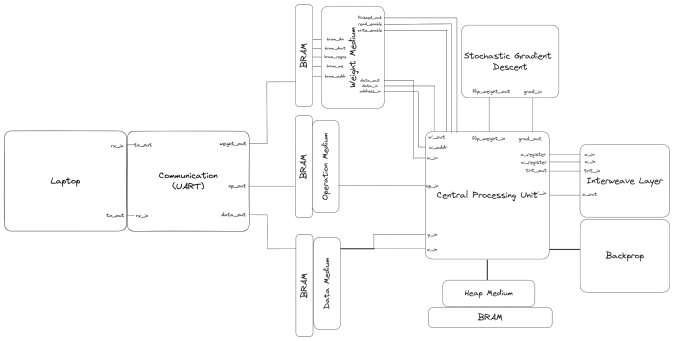

Fig. 8. The central processing unit. An external computer can transfer data, weights, and program operations to the cpu, which runs the neural network described by the program. To perform backpropagation, the cpu is connected to a stack and SGD unit. Some wires and ports have been left out for brevity.

On the computer side, we have a Python program that preprocesses the MNIST data and communicates with the FPGA.

IV. Results

A. Constant Input to Constant Output

Our optimism in this architecture came from a simulation in the beginning of October: feeding in a constant nine bits, and training it to output another constant nine bits. It worked! We scaled it up in Python to 729 bits, added residual connections and several layers. Figure 9 shows the results.

Placing it on the FPGA similarly worked flawlessly. It runs too fast to capture a training curve, but the final bit error is less than 10%. Here are the gradients of the output layer for a 128-bit to 128-bit run:

```
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000001000000
0110101111110111
```

As 128 is not a power of three, the last few bits jump around more.

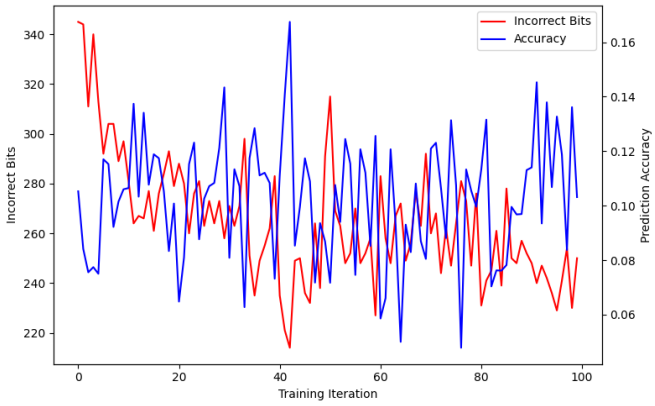

Fig. 10. Training on the modified MNIST dataset at a learning rate of 0.1. The accuracy is as bad as random guessing.

B. Image Classification

Classifying images is much more interesting than outputting a constant value. We chose to start with the MNIST dataset. As we're rather limited in space, we truncated the inputs to 27×27 binary values, as opposed to 28×28 pixels with 8-bit quantization. We set the correct output to

$$y_i = \begin{cases} 1 & y_i/72 = \text{label} \\ 0 & \text{otherwise,} \end{cases}$$

essentially a "many-hot" encoding of the label.

A big problem showed up in our Python simulation, as seen in 10: The error would not decrease. We tried several modifications to fix this:

1) Replacing every third bit with a bias term to destroy symmetry.
2) Using a five-majority instead of the weighted 3-majority.
3) Changing the backpropagation to check majority.
4) And many more.

It turns out that the original architecture from October was not universal, as inverting all of the inputs leads to an inverted output, so only odd functions can be created. To counteract this, every third bit can be fixed to a *bias* value. A majority between two possibly inverted inputs and a bias can implement

$2^3 = 8$ logic gates, including NAND and NOR, so this addition allows universality.

That was not enough, so we tried using *two* bias terms. Because we still wanted to fit on a single logic unit, we removed the weights, giving three inputs and two biases to a five-majority. Unfortunately, this similarly failed.

The third modification we actually included in our final architecture, as mentioned in III-B. Without checking for majority, each backpropagation will be wrong around one-fourth of the time, so after four interweave layers you could expect pure noise. Surprisingly, it would still train, albeit noisily.

We originally thought the benefits outweighed the extra noise; without the check, the backpropagation does not require the `x` input, making a heap is unnecessary. In fact, the neural network would be entirely local, which means forward and backpropagation can be run in parallel, allowing extreme throughput.

However, as we could not get MNIST training with our original architecture, we needed to remove as much noise as possible. This was a known source, so it had to be modified. Unfortunately, it too was not enough.

V. Failed Attempt to Make a Bitcoin Miner

We already had most everything needed to make a bitcoin miner—communication and a processor capable of doing operations quickly. We thought we could replace the interleave and backprop layers with the extend and compress functions of SHA-256 and attempt to make a bitcoin miner. Unfortunately, we did not have time to debug our SHA-256 implementation before the deadline (it would output hashes, but those hashes didn't match up with the *correct* hashes).

VI. Success at Turning It Into a Brainfuck Interpreter

Our design practically required the creation of a general purpose computer. We purposely made our central processing unit *not* Turing complete (no AIs escaping into the world on our watch!) but once we had everything set up it was rather simple to turn into a Turing complete computer (excepting memory constraints—our FPGA does not have infinite memory). One simple programming language that is well known to be Turing complete is Brainfuck. As a last ditch effort to have *something* working for our project (read: we started this less than 5 hours before the deadline), we attempted to modify it to run Brainfuck code (writing a compiler in Python to send the appropriate opcodes). This required adding five instructions to the instruction, as shown in Table II.

TABLE II
Additional operations added to the instruction set to make the CPU Turing complete.

Opcode	Description
27	Jump to the address given in the next instruction if `W` == 0.
28	Jump to the address given in the next instruction if `W` != 0.
29	Shift the bits of `inference` left 8 bits and insert in the last 8 bits of `W`
30	Add one to `W`
31	Subtract one from `W`

Of especial importance are the two jump instructions, 27 and 28. These allow the the program to change its instruction pointer and are crucial to making it (basically) Turing complete.

We successfully implemented a Brainfuck interpreter. As a test, we copied the following "Hello world!" program from Wikipedia and sent it to our FPGA:

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+
>+>->>+[<]<-]>>.>---.+++++++..+++.>>
.< -.<.+++.------.--------.>>+.>++.
```

Our FPGA sucessfully sent back the message "Hello World!".

VII. Postmortem

We learned a few important lessons from this project:

1) Get communication working first. We wasted a lot of time trying to debug our CPU because of a buggy communication module (`recv_rx`) that we *knew* was buggy but thought we could work around. We should have spent more time making sure our communication module was absolutely correct before trying to debug the CPU.
2) Try more simulations. We had promising early results when training for a constant value in simulation and rushed into writing Verilog code for it. We should have done more tests in simulation—such as to see if it could learn MNIST—earlier.

Overall, we each spent about 70 hours on this project, but are still a bit disappointed in the results. It turns out that quantizing neural networks to single bits is a hard problem that is an active area of research. There are a number of recent papers that focus on quantizing large language models (e.g. "PB-LLM: Partially Binarized Large Language Models" [1]), but most of these focus on quantizing a pretrained neural network—which is as much a compression problem as a quantization problem. Other papers (e.g. "Accelerating Deep Convolutional Networks using low-precision and sparsity" [2]) actually train a neural network with quantized weights, but they don't also quantize the gradients.

No one has yet managed to quantize the entire training process of a neural network—including activations, weights, and gradients—so it is not too much of a surprise that we failed.

A. *Specific Contributions*

Overall, this was a team effort, and everybody contributed more than adequately. Joseph Camacho took point on the CPU—he's the one who implemented the 30+ operations and mediums between the BRAMS, while James acted as his rubber duck and tested half the instruction set. James Camacho took responsibility for the communications—he wrote our UART protocol and debugged most of it, but Joseph found the critical fix in Manta. Both spent dozens of hours working on simulations and implementing the neural network in Python and Verilog. James implemented the architecture we showed in our final video and the assembler for it. Joseph implemented Brainfuck and its compiler. Joseph also started (but did not finish) work on the bitcoin miner.

REFERENCES

[1] Yuzhang Shang, Zhihang Yuan, Qiang Wu, and Zhen Dong. Pb-llm: Partially binarized large language models, 2023.

[2] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. Accelerating deep convolutional networks using low-precision and sparsity, 2016.