The background of the slide is a detailed, grayscale image of a printed circuit board (PCB). It features a complex network of fine lines representing traces, various circular pads, and larger rectangular components, all arranged in a dense, interconnected pattern.

Single Bit **Neural Network on FPGA**

By James and Joseph Camacho

Issues with current neural networks:

- Expensive to train.
 - It takes about a month to train GPT-3 on 1,000 A100s,
 - Or about \$5 million ([source](#)).
 - It's estimated GPT-4 cost >\$100 million ([source](#)).

D Total Compute Used to Train Language Models

This appendix contains the calculations that were used to derive the approximate compute used to train the language models in Figure 2.2. As a simplifying assumption, we ignore the attention operation, as it typically uses less than 1% of the total compute for the models we are analyzing.

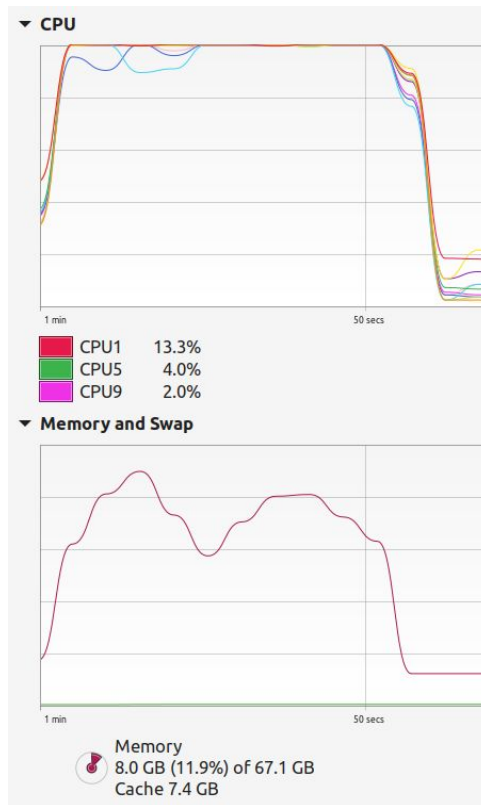
Calculations can be seen in Table D.1 and are explained within the table caption.

Model	Total train compute (PF-days)	Total train compute (flops)	Params (M)	Training tokens (billions)	Flops per param per token	Mult for bwd pass	Fwd-pass flops per active param per token	Frac of params active for each token
T5-Small	2.08E+00	1.80E+20	60	1,000	3	3	1	0.5
T5-Base	7.64E+00	6.60E+20	220	1,000	3	3	1	0.5
T5-Large	2.67E+01	2.31E+21	770	1,000	3	3	1	0.5
T5-3B	1.04E+02	9.00E+21	3,000	1,000	3	3	1	0.5
T5-11B	3.82E+02	3.30E+22	11,000	1,000	3	3	1	0.5
BERT-Base	1.89E+00	1.64E+20	109	250	6	3	2	1.0
BERT-Large	6.16E+00	5.33E+20	355	250	6	3	2	1.0
RoBERTa-Base	1.74E+01	1.50E+21	125	2,000	6	3	2	1.0
RoBERTa-Large	4.93E+01	4.26E+21	355	2,000	6	3	2	1.0
GPT-3 Small	2.60E+00	2.25E+20	125	300	6	3	2	1.0
GPT-3 Medium	7.42E+00	6.41E+20	356	300	6	3	2	1.0
GPT-3 Large	1.58E+01	1.37E+21	760	300	6	3	2	1.0
GPT-3 XL	2.75E+01	2.38E+21	1,320	300	6	3	2	1.0
GPT-3 2.7B	5.52E+01	4.77E+21	2,650	300	6	3	2	1.0
GPT-3 6.7B	1.39E+02	1.20E+22	6,660	300	6	3	2	1.0
GPT-3 13B	2.68E+02	2.31E+22	12,850	300	6	3	2	1.0
GPT-3 175B	3.64E+03	3.14E+23	174,600	300	6	3	2	1.0

Graph from “Language Models Are Few-Shot Learners” by OpenAI

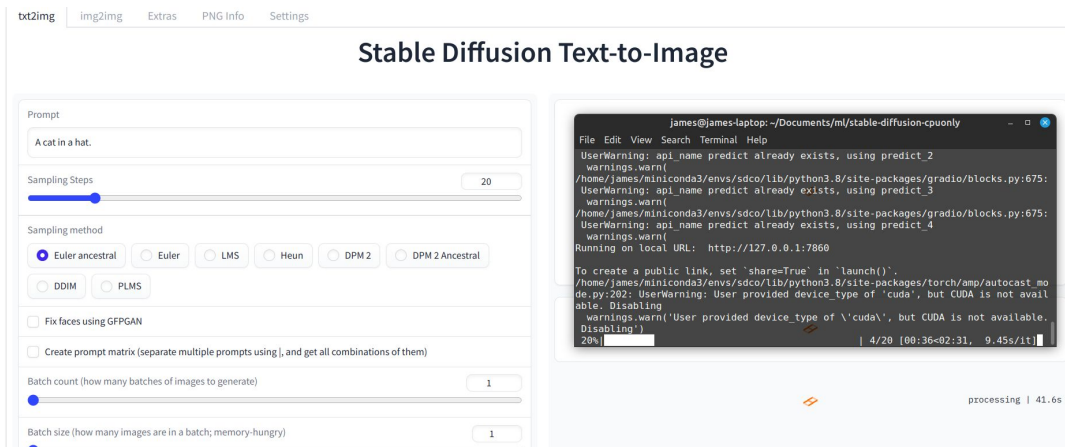
Issues with current neural networks:

- Expensive to train.
- Large memory footprint.
 - Small stable denoiser models consume 8-16GB.
 - Small LLMs (such as LLaMA) are *at least* 10GB.



Issues with current neural networks:

- Expensive to train.
- Large memory footprint.



- Slow at inference.
 - Takes several minutes on CPU for image generation, several seconds for next-token prediction.
 - GPUs are several orders of magnitude faster, but memory access is still too slow.

The Solution

Put the neural network on a chip!

- More efficient architectures.
- Higher throughput.
- Less energy consumption.

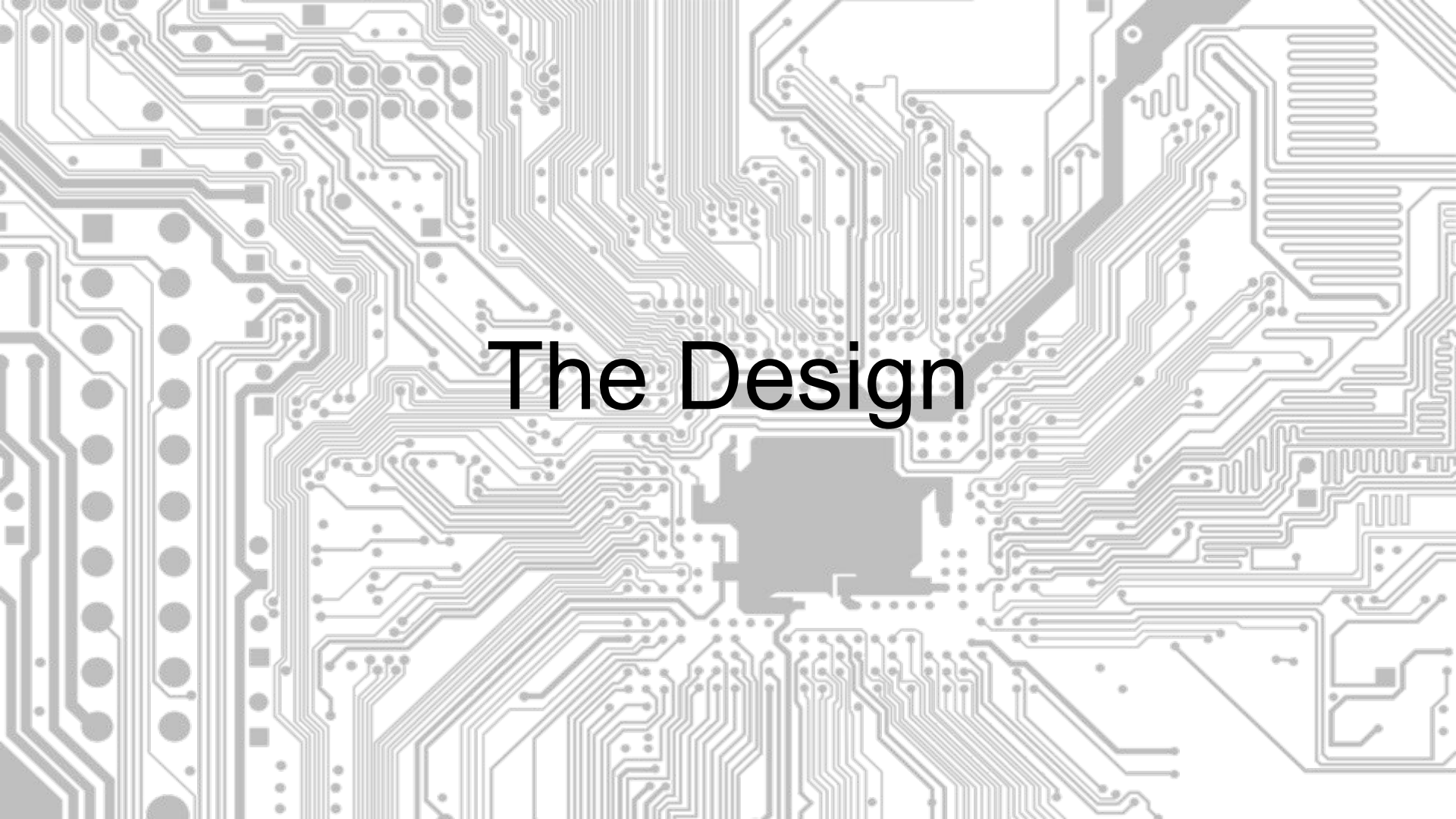
Our Project

Design a neural network to run on an FPGA (specifically, the Spartan-7 XC7S50).

- Allows testing multiple architectures before committing to silicon.
- Can chunk into layers, not needing to build out the entire network at once.

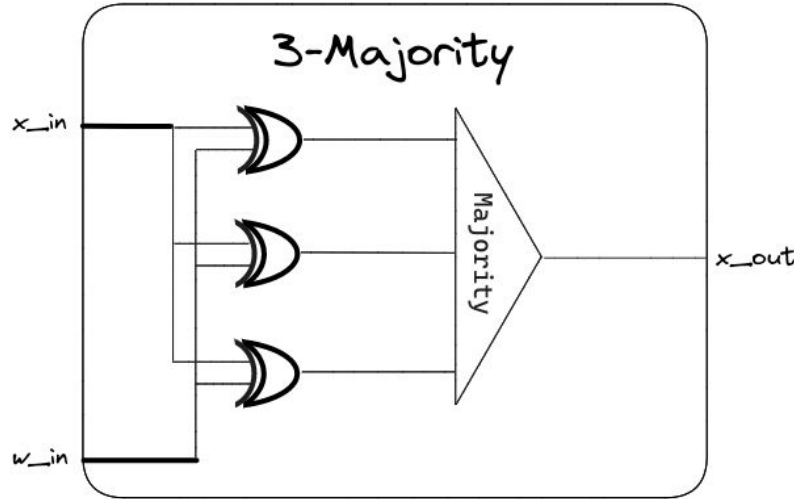
Goals:

- Minimum: 20% accuracy on MNIST.
- Target: 90% accuracy on MNIST.
- Stretch: Train a chess engine.



The Design

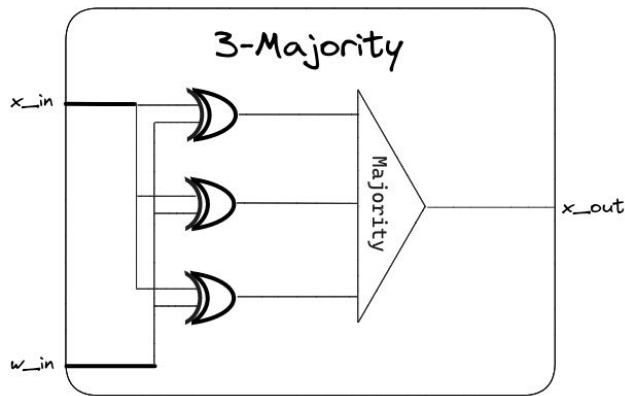
The Basic Unit: “3-Maj”



Features:

- View each wire as +1 (1'b0) or -1 (1'b1).
- Best approximation of $(w1 * x1 + w2 * x2 + w3 * x3) / 3$
- 6-in-1-out happens to be the exact size of the Spartan-7 logic cells.

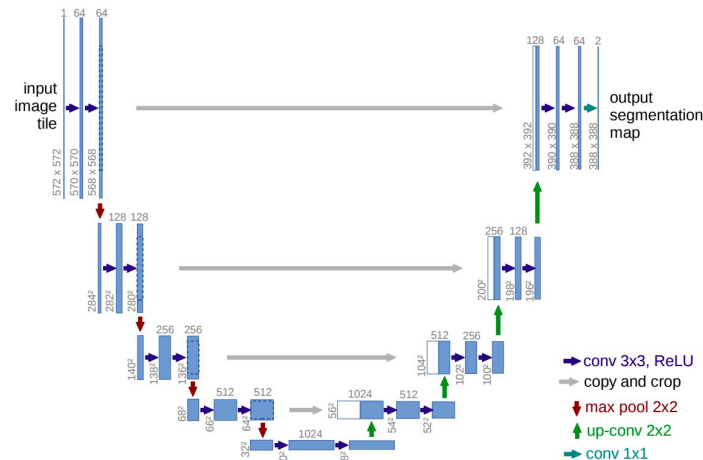
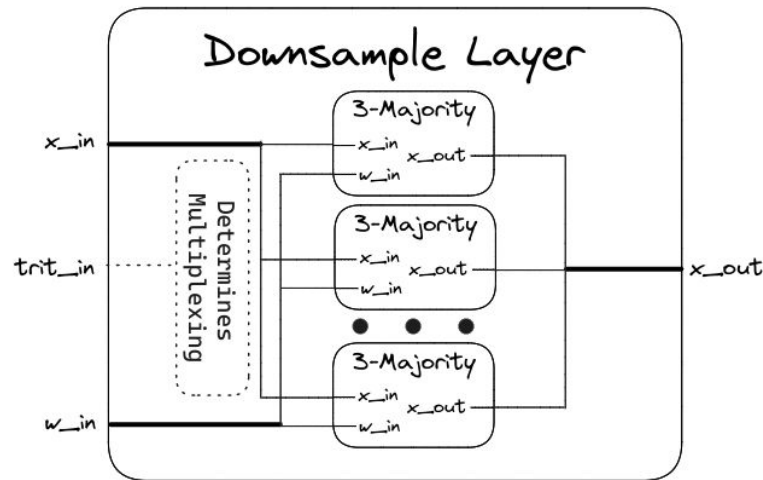
Back Propagation



Features:

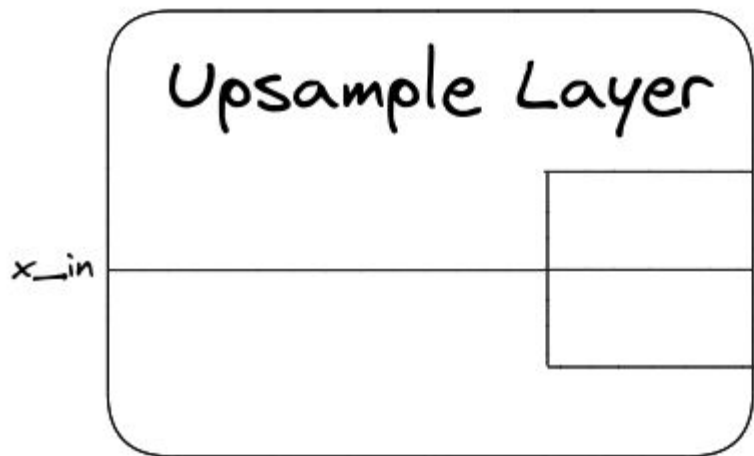
- View each wire as “value should stay the same” (1'b0) or “value should change” (1'b1)
- Back propagation is $b_{out} = \{3\{b_{in} \wedge x_{out}\}\} \wedge x_{in} \wedge w_{in}$
- Weight gradients are $b_w = x_{in} \wedge \{3\{b_{in}\}\}$

Making Layers: Downsampling



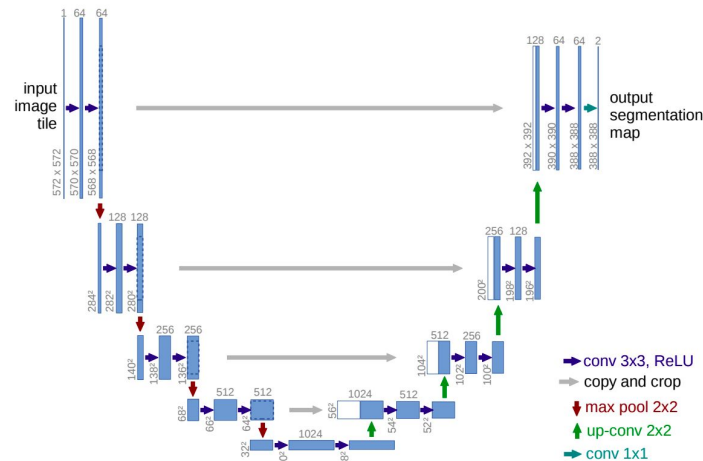
- Choosing successive trit positions 0 and $\text{clog}_3(N)/2$ will turn each 3×3 square into one output.
- Such downsampling is traditional in image tasks (see U-Net to the right).

Making Layers: Upsampling



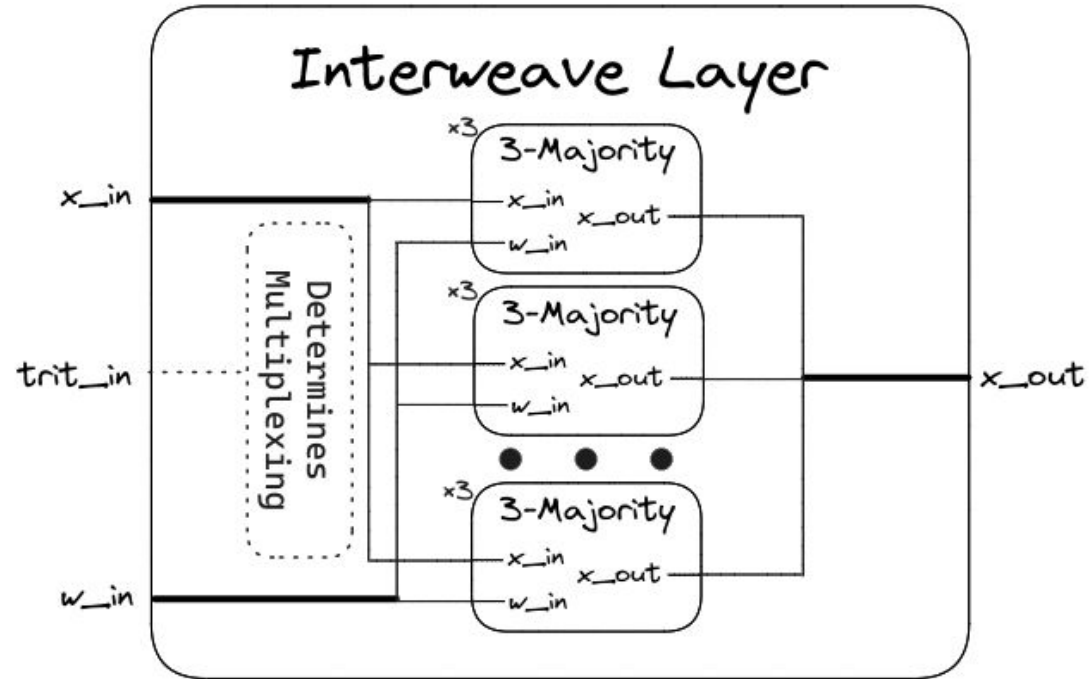
x_{out}

- Just copy each bit three times
- Back propagate via majority function
- Simple enough to not need its own module.

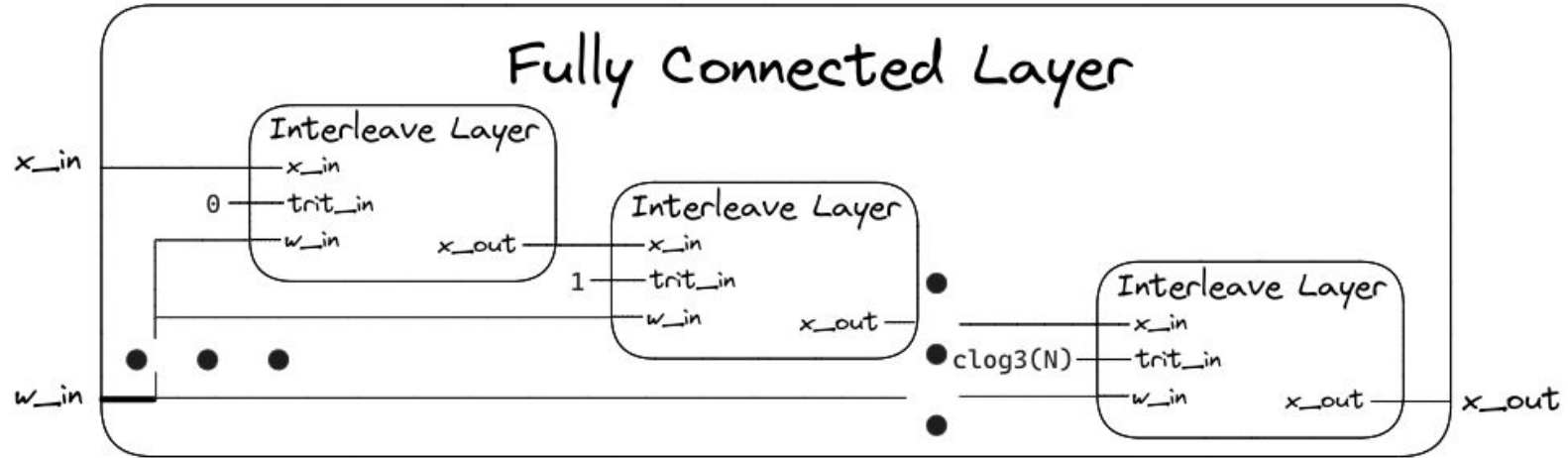


Making Layers: Interweaving

- Upsample layer and then downsample layer
- Output shape = input shape
- Interesting quirk: Back propagation looks the same as forward propagation



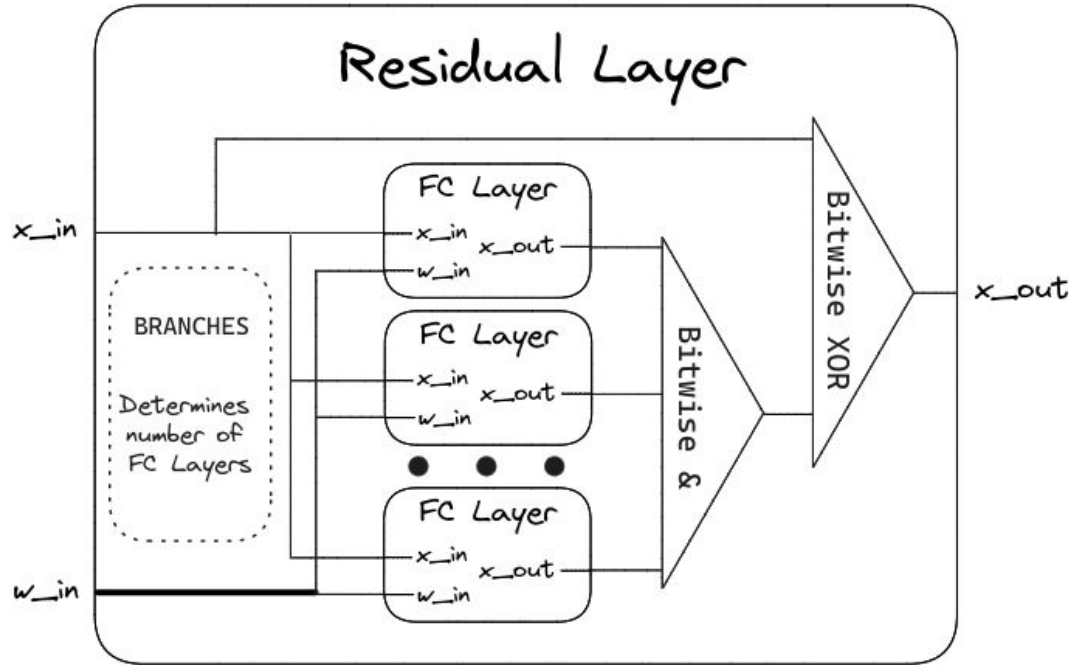
Making Layers: Fully Connected



Features:

- Creates a tree structure between the inputs and every output.
- Ternary is the most efficient encoding scheme.
- Running time is $O(\log N)$ due to parallelization, memory requirement is $O(N \log N)$. Note that matrix-vector multiplication would take $O(N^2)$ space.

Making Layers: Residual Layer



Features:

- Only changes input if all fully connected layers agree.
- Each module learns "in parallel".

Putting it on the FPGA

Need five parts:

1. Neural network (forward & back propagation)
2. Gradient accumulator
3. Central processing unit (CPU)
4. Memory (data, weights, & accumulations)
 - a. We'll probably hard code in the NN architecture
5. Communications (data & weight transfer)

Putting it on the FPGA: Estimated Resource Usage

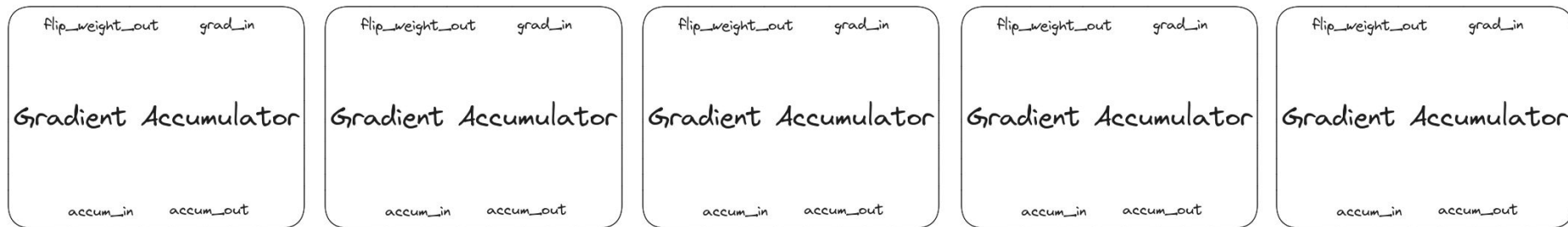
1. Neural network (forward & back propagation)
 - a. For MNIST, we want an fc layer that can handle 768 inputs→ closest power of 3 is 3^{**7}
 - b. Each interweave layer takes 1 logic cell per input, so 3^{**7} logic cells per interweave layer
 - c. 7 interweave layers * 3^{**7} logic cells is about **15,000 logic units**
 - d. Back propagation can use the same logic as forward propagation
 - e. Down sample logic negligible in comparison
2. Gradient accumulator
 - a. Requires ~10 logic units per weight. We'll make as many of these as we can.
3. Central processing unit (CPU)
 - a. Small compared to network
4. Memory (data, weights, & accumulations)
 - a. Data: Can stream in. Negligible.
 - b. Weights: 3 bits per 3-maj. Per fc layer on MNIST, this is about 45 kilobits.
At inference, we can fit ~60 layers in memory, which is plenty.
 - c. Accumulations: ~9 bits per weight. During training, we can only fit ~6 layers at a time.
We'll have to stream the weights or find an FPGA with more memory for larger networks.
5. Communications (data & weight transfer)
 - a. Negligible.

On the FPGA: Neural Network

- Uses combinational logic
- Limited size of FPGA → only use logic for a downsampling layer and a fully connected layer
- Takes inputs from CPU and sends outputs back to CPU
- Discussed more in detail earlier

On the FPGA: Gradient Accumulator

- Accumulate gradients (+1 if should change, -1 if should not change) over many forward/backward runs
- Only flip if threshold is reached
- Takes gradient input and old accumulation from CPU
- Sends back to the CPU whether the weight should flip and the new accumulation
- Many copies of this module that the CPU can use



On the FPGA: Central Processing Unit

- State machine
- Takes a sequence of instructions and executes them
 - E.g. 1. Take the data at address 0x0000 and put it in our x register.
2. Run the data in the x register through the fc_layer and put the output in the y register
3. Move the data from the y register to the x register
4. ...
 - For this project, we plan to hard code in instructions.
- Full list of registers:
 - x ←Used for inputs and gradient outputs
 - w ←Used for weights and weight gradients
 - y ←Used for outputs and gradient inputs
- Lots of communication inputs/outputs with periphery modules
 - fc_fin, fc_fout, fc_win, fc_bin, fc_bout, fc_gradout, etc.
 - valid, ready, etc.
 - control for the modules that talk with the external computer (e.g. request new data)

On the FPGA: Memory

- Need data, weights, and accumulations
- Use the FPGA's BRAM

On the FPGA: Communication

- We'll use UART at 3Mbaud.
- Timing-wise, this should be fast enough:
 - An MNIST image is $28 \times 28 \times 8$ bits = 2ms transfer rate.
 - Stockfish uses ~45,000 input bits = 15ms transfer rate if we use a similar architecture.
 - Total number of parameters are $< 3M$ = 1s for weight transfer.