WES 237C - Hardware for Embedded Systems, Kastner Project 5, BNN Binary Neural Networks	**Ricardo Lizárraga** rilizarraga@ucsd.edu **PID: A69028483**	***Submit to:*** *Ryan Charles Kastner* *(TA) Zhenghua Ma* *(TA) Abarajithan Gnaneswaran*

BNN Brief background

A Binomial Neural Network (BNN) refers to a type of network where weights, activations, or both might be binary or based on a binomial distribution. It's not a standard term in most deep learning frameworks, but it could refer to models such as Binary Neural Networks (BNNs) that employ binary weights and activations to improve efficiency and reduce memory usage.

The activations might be constrained to values like 0 or 1. In such a case, the network would work with binary values rather than floating-point activations.

If you're building a neural network for image classification (e.g., MNIST) and you're using binary activations or binary weights, the "784 activation input" might be:

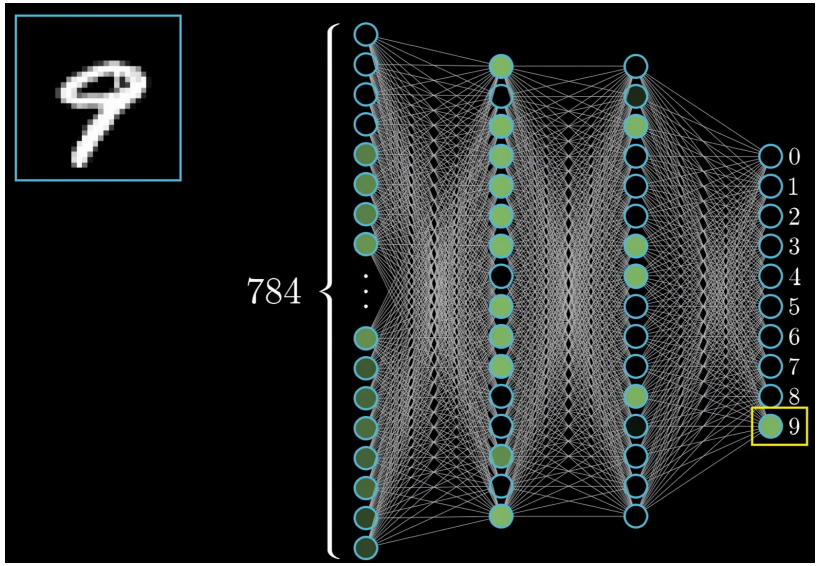
Input layer: A 784-dimensional input vector (from the MNIST dataset).

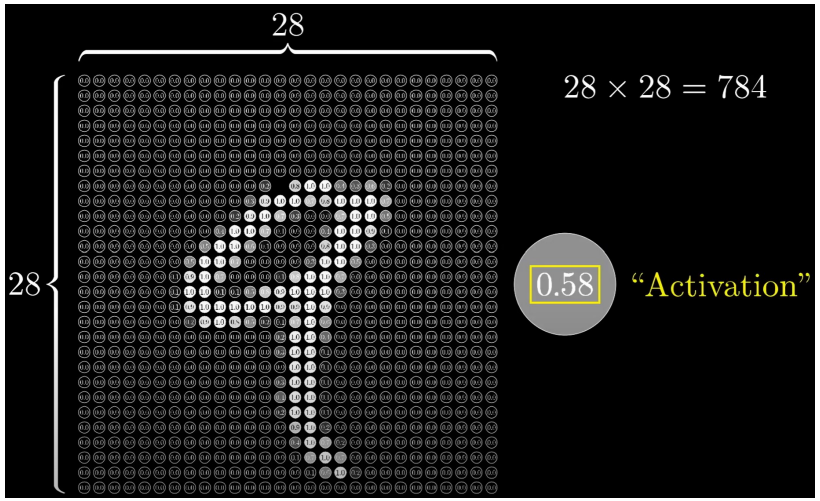
Binary Activations: The activation function for the network might binarize the output of each neuron, such as using a binomial distribution to decide whether the activation should be 0 or 1.

This could be part of a Binary Neural Network, where the weights and/or activations are binary (taking values from {0, 1} or {-1, 1}).

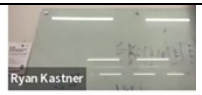
The MNIST dataset consists of 28x28 grayscale images of handwritten digits, which is flattened into a 784-dimensional vector (28 * 28 = 784). So, the 784 inputs represent the pixel values of the image, where each input corresponds to the intensity of one pixel.

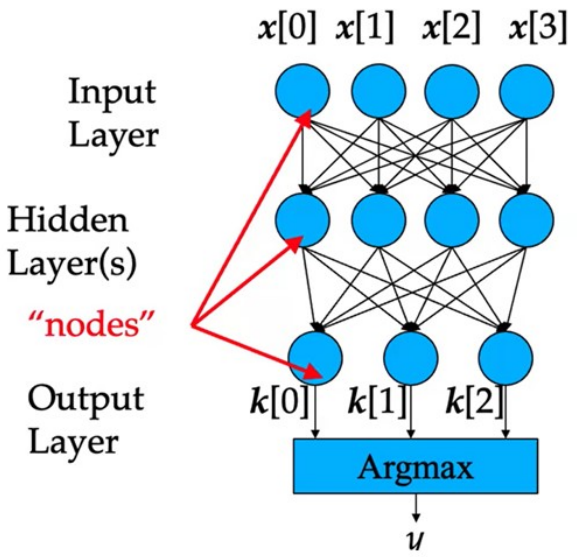
In this case, the input to the network is a 784-dimensional vector.





Multilayer Terminology



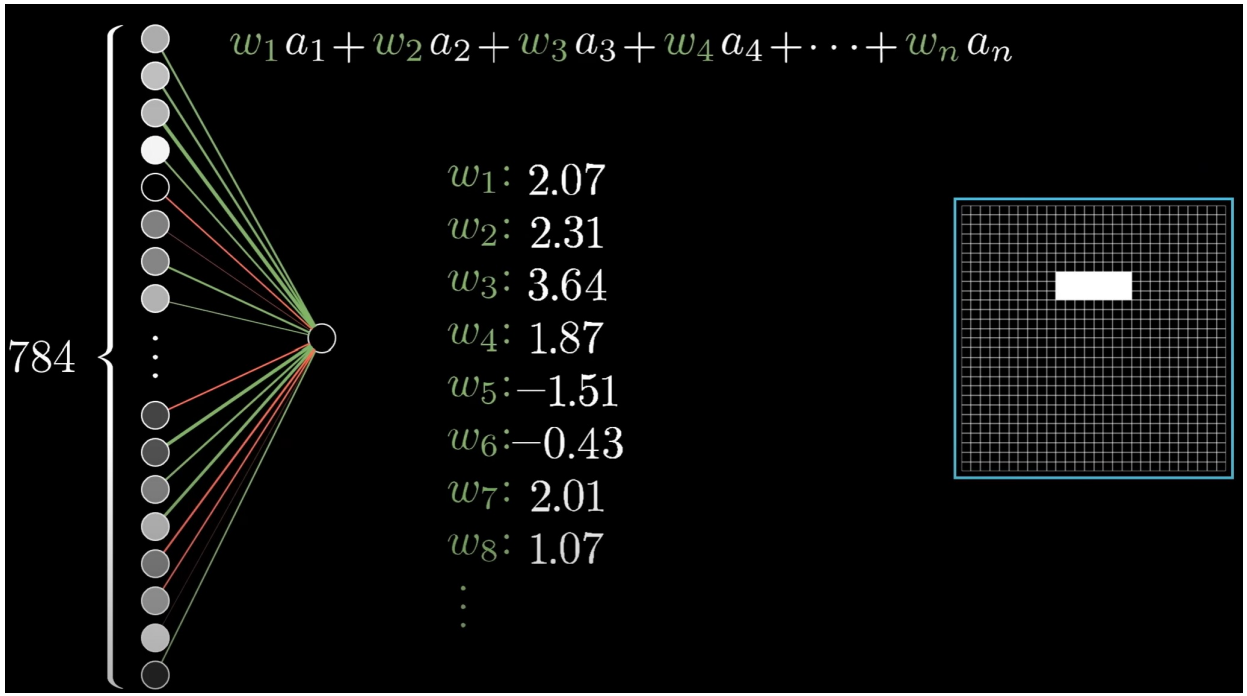


Weight matrices: Entry *i,j* is weight between i^{th} input and j^{th} output

W_1[i, j] is [4x4]
b_1[j] is [4x1]

W_2[i, j] is [4x3]
b_2[j] is [3x1]

Probability that input is of class ***k***[i]



Input samples

Sample1[25] // true label: 7	Sample2[25] // true label:2	Sample3[25] // true label:1

	Layer1	**Layer2**	**Layer3**	
Input	784 (128, 784)	128 # (64, 128)	64 # (10, 64)	10
Weight data (unsigned int)	w1[3200]	w2[256]	w3[20]	

Pseudo code (my understanding)

```
//Start here:
//Layer 1
       For(a->128)
              For(b->SIZE=25)
                     xnor(IN[b],w1[index])
                     Popcount()
              End
       Activation_1 = (accum1-16)*2-784
       End

       Sign()
       quantize()
       Pack()

//Layer 2
       Same as above with corresponding bounds

//Layer 3 {last layer}
       … no sign/quantize/pack
       swap endianness
```

```
//Packing
       unsigned int accum=0;
       for(a->32)
              for(b->pack_length) //packlen__1=4,packlen_2=2
                     output[b] |= (quantized[j * 32+ a]&1) << (31-a));
              end
       end

//popcount
       accum=0;
       for(a->32)
              accum+=input&1//if least significant  bit is 1
              bitshift here
       end
       XNOR by ~(a^b);
```

The input is 25 values of 32-bit each = 800 bits, since we have 28*28=784 binary inputs to NIST, then we ignore the last 16-bits if IN[24]

"**Popcount operator**" in the context of MNIST, it is used for feature extraction, model optimization, or creating a custom model. Here are a few potential ways that popcount might be relevant when dealing with MNIST:

1. Feature Extraction with Popcount

   The MNIST dataset consists of 28x28 grayscale images of handwritten digits. One idea might be to represent these images in a binary format (e.g., thresholding pixel values), and then use the popcount operator to extract features from each image.

   For example, a binary version of the image could be created by thresholding the pixel values (e.g., pixels greater than a certain value could be set to 1, and others to 0). The popcount could then be applied to count the number of 1's in each row or region of the image. This could provide insight into the distribution of active pixels in the image.

2. Popcount in Neural Networks

   Popcount operations could potentially be used to optimize neural network computations or serve as part of a binary neural network (BNN) approach. In binary neural networks, weights and activations are constrained to binary values, and the popcount operation can efficiently compute sums of binary activations during forward passes or during backpropagation.

   For example, rather than computing a traditional dot product of weights and activations, the popcount could be used to efficiently count the number of matching bits between the binary weight and activation vectors.

3. Optimization or Approximation

   The use of popcount may also be explored in certain approximation techniques for speeding up the model inference, particularly in cases where a network or model uses binary operations or quantized weights.

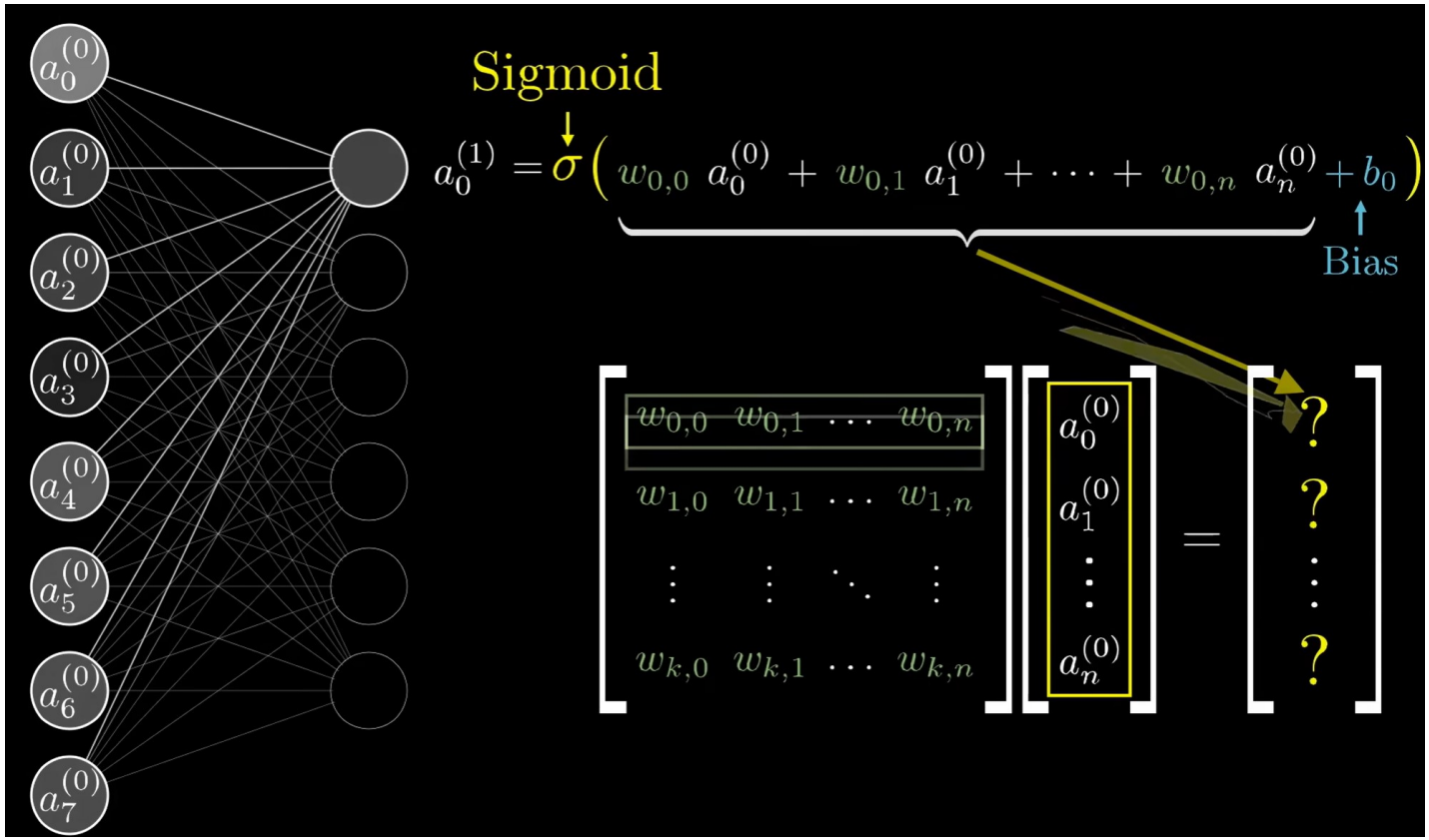
   The popcount operator might help in implementing hardware-efficient versions of neural networks that are optimized for speed and memory usage (like FPGA or ASIC designs).

4. Popcount in Binary Classification

   If you're interested in binary classification tasks related to MNIST, the popcount operator could potentially be used as a custom feature in a binary classification model. For example, comparing the number of active pixels (after thresholding) for different digits could help distinguish them.

Code EXPLANATION

The program starts by doing a nested for loop to do an XNOR and popCount on the input layer 1.
The inner for loop iterates through the 25 values, then the out for loop iterates through each separate input value.
The XNOR works by using a bitwise XOR("^") and then a bitwise not ("~") to make it an XNOR.
The popCount function works by counting the number of 1'S in the 32-bit unsigned integer using a for loop and accumulator; then get the output by setting the activation variable: activation[i]=2*accum—size.



Layer1:

We have to subtract 16 (800-784), since the input is limited to 784, and data IN is 25*32-bits=800 bits

Next is to use sign() function (bitwise), this makes every variable a -1 or +1, by checking the sign bit on the integers,

The quantize() function then converts all of the -1 values to 0.

Use pack() function to a compact vector so we can input into the next Layer.

Layer2:

Did the same process described in Layer 1, except subtracting 16 bits, and used corresponding w2.

Layer3:

This is simplified since the output of the activation is the desired output {no need for sign(), quantize() or pack()

```c
void bnn(DTYPE IN[SIZE], ITYPE ys[10])
{
    ///unsigned int a = 0xAAAAAAAA;
    ///int n;   n = popCount(a);printf("n=%d\n",n);///
    ///unsigned int a = 0xAC000000;unsigned int b = 0xCB000000;unsigned int res = XNOR(a,b);printf("res=%u\n",res);
    static const unsigned int w1[3200] = {1369938329,338570864,3449824829,3240895963,1476599616,1328019041,2965306460,2146265097,3691678023,3226747582,174245824,504823303,3356
    static const unsigned int w2[256] = {1913607555,675556588,3902637308,1303398415,2804841594,62821375,3752619650,1301598150,1163584328,4265134891,3022287108,3131978726,15167
    static const unsigned int w3[20] = {3898795476,3841654544,1999377659,478627577,2610578576,437068610,3468252685,2057174436,897088322,3798044925,1579394953,2235493397,354026
    int layer1_out[128] = {0};
    int layer2_out[64] = {0};
    int accum[128] = {0};
    int bias;
    unsigned int xnor_result;
    // Layer1
    memset(accum, 0, sizeof(accum));
    int thisInputSz = 25, weight_words = 128;
    for(int in = 0; in < thisInputSz; in++) {       /// Lets unpack the 25 weights from w1[3200], {resulting on packets 128 words size each (unsinged int)
       for(int out_n = 0; out_n < weight_words; out_n++) {  /// packet size = 128
           unsigned int input_word = IN[in];    //here is the Sample1[25]
           unsigned int weight_word = w1[out_n * thisInputSz + in]; /// w1[(0-127)*25+(0-25)
           xnor_result = ~(input_word ^ weight_word);
           accum[out_n] += __builtin_popcount(xnor_result); /// count the number of set bits in an unsigned integer
           if (in == 24) {// since 25*32=800 bits, we have to clear the last 16-bits, since input is 784 bits,
               input_word &= 0xFFFF0000;
               weight_word &= 0xFFFF0000;
           }
           xnor_result = ~(input_word ^ weight_word);
           accum[out_n] += __builtin_popcount(xnor_result); /// counting the number of set bits
       }
    }
    for(int out_n = 0; out_n < 128; out_n++) {
       accum[out_n] = (accum[out_n]-16) * 2 - 784;  ///Activation
       layer1_out[out_n] = accum[out_n] > 0 ? 1 : -1;
    }

   // Layer2
   memset(accum, 0, sizeof(accum));
   for(int out_n = 0; out_n < 64; out_n++) {
       accum[out_n] = 0;
       for(int word = 0; word < 4; word++) {
           unsigned int input_bits = 0;
           for(int bit = 0; bit < 32; bit++) {
               int input_idx = (word * 32) + bit;
               if(input_idx < 128) {
                   if(layer1_out[input_idx] == 1)
                       input_bits |= (1u << bit);
               }
           }
           xnor_result = ~(input_bits ^ w2[out_n * 4 + word]); // XNOR
           accum[out_n] += __builtin_popcount(xnor_result);
       }
       accum[out_n] = accum[out_n] * 2 - 128;///Activation
       layer2_out[out_n] = accum[out_n] > 0 ? 1 : -1;
   }

   // Final Layer
   memset(accum, 0, sizeof(accum));
   for(int out_n = 0; out_n < 10; out_n++) {
       accum[out_n] = 0;
       for(int word = 0; word < 2; word++) {
           unsigned int input_bits = 0;
           for(int bit = 0; bit < 32; bit++) {
               int input_idx = word * 32 + bit;
               if(input_idx < 64) {
                   if(layer2_out[input_idx] == 1)
                       input_bits |= (1u << bit);
               }
           }
           xnor_result = ~(input_bits ^ w3[out_n * 2 + word]);
           accum[out_n] += __builtin_popcount(xnor_result);
       }
       accum[out_n] = accum[out_n] * 2 - 64;
   }
   for(int i = 0; i < 10; i++) {
       ys[i] = accum[i];
   }
}
```

Test code

bnn.cpp | bnn_test.cpp | bnn_csim.log

```
INFO: [SIM 2] *************** CSIM start **************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../bnn.cpp in debug mode
   Generating csim.exe
-----------------------------------------------------------
Verifying the sample: Sample 1
Wrong output: Expected: -2 Obtained: 4
Wrong output: Expected: 4 Obtained: 2
Wrong output: Expected: -8 Obtained: -6
Wrong output: Expected: 14 Obtained: 8
Wrong output: Expected: -4 Obtained: 2
Wrong output: Expected: 0 Obtained: 2
Wrong output: Expected: -42 Obtained: -8
Wrong output: Expected: 48 Obtained: -6
Wrong output: Expected: -4 Obtained: 2
Wrong output: Expected: 2 Obtained: 4
Sample: Sample 1 FAILED
-----------------------------------------------------------
Verifying the sample: Sample 2
Wrong output: Expected: 2 Obtained: 4
Wrong output: Expected: 0 Obtained: 2
Wrong output: Expected: 48 Obtained: -6
Wrong output: Expected: 10 Obtained: 8
Wrong output: Expected: -20 Obtained: 2
Wrong output: Expected: 4 Obtained: 2
Wrong output: Expected: 2 Obtained: -8
Wrong output: Expected: -16 Obtained: -6
Wrong output: Expected: 4 Obtained: 2
Wrong output: Expected: -14 Obtained: 4
Sample: Sample 2 FAILED
-----------------------------------------------------------
Verifying the sample: Sample 3
Wrong output: Expected: -16 Obtained: 4
Wrong output: Expected: 42 Obtained: 2
Wrong output: Expected: 2 Obtained: -6
Wrong output: Expected: 4 Obtained: 8
Wrong output: Expected: -2 Obtained: 2
Wrong output: Expected: -14 Obtained: 2
Wrong output: Expected: 8 Obtained: -8
Wrong output: Expected: 2 Obtained: -6
Wrong output: Expected: 14 Obtained: 2
Wrong output: Expected: -8 Obtained: 4
Sample: Sample 3 FAILED
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish **************

```

DEMO implementation to PYNQ is not required