BitNetMCU

Training and Inference of Low-Bit Quantized Neural Networks on a low-end RISC-V Microcontroller

Introduction and Motivation

Recently, there has been considerable hype about large language models (LLMs) with "1 Bit" or "1.58 Bit" [^1] weight quantization. The claim is that, by using Quantization Aware Training (QAT), LLMs can be trained with almost no loss of quality when using only binary or ternary encoding of weigths.

Interestingly, low bit quantization is also advantageous for inference on microcontrollers. The CH32V003 microcontroller gained some notoriety being extremely low cost for a 32 bit MCU (less than $0.15 in low volume), but is also notable for the RV32EC ISA, which supports only 16 registers and lacks a hardware multiplier. It also only has 16kb of flash and 2kb of ram.- BitNetMCU

The use of a few bits for each weight encoding means that less memory is required to store weights, and inference can be performed using only additions. Thus, the absence of a multiplication instruction is not an impediment to running inference on this microcontroller.

The challenge I set to myself: Develop a pipeline for training and inference of low-bit quantized neural networks to run on a CH32V003 RISC-V microcontroller. As is common, I will use the MNIST dataset for this project and try to achieve as high test accuracy as possible.

This document is quite lengthy and rather serves as a personal record of experiments.

Background

Quantization of Deep Neural Networks (DNN) is not a novel concept, a review can be found in [^2]. Different approaches are usually distinguished by three characteristics:

Pre- or post-training quantization - The quantization of weights and activations can be done during training (QAT) or after training (post-quantization). QAT allows to consider the impact of quantization on the network already during training time. This comes at the expense of increased training time, complexity and loss of flexibility. QAT is most suitable for situations where the network is trained once, and inference takes place in a device with less computing power and memory ("edge device"). Post-quantization is more flexible and can be used to quantize a network to different bit-widths after training.

Quantization granularity of weights - The number of bits used to encode the weights. Networks are typically trained with floating point weights, but can be quantized to 8-bit (**W8**), 4-bit (**W4**), 2-bit (**W2**), ternary weights (**W1.58**), or even binary (**W1**) weights. Using fewer bits for weights reduces the memory footprint to store the model data (ROM or Flash in a MCU).

Quantization granularity of activations - The number of bits used to encode the activations, the data as it progresses through the neural network. Inference is usually performed with floting point activations. But to reduce memory footprint and increase speed, activations can be quantized to 16-bit integers (**A16**), 8-bit (**A8**) or 1-bit (**A1**) in the most extreme case. Reducing the size of activations helps to preserve RAM during inference.

The most extreme case is to quantize both weights and activations to one bit (**W1A1**), as in the XNOR Net[^3]. However, this approach requires increasing the network size dramatically to counteract the loss of

information from the activations. Also, handling of single bit information is more cumbersome on a standard 32-bit MCU than handling integers.

Therefore I will explore scenarios with different weight quantizations, while I keep the activations at 8 bit or more. It seems obvious that QAT is the preferred training approach when targeting inference on a very small microcontroller, but I will also explore post-quantization for comparison.

Implementation of training code

I used the code snippets given in [^4] as a starting point. The main forward pass function remained unchanged. It implements a Straight-Through-Estimator (STE)[^5] to allow backpropagation through the non-differentiable quantization functions. Essentially, the weights are quantized during forward and backward pass, but the gradients are calculated with respect to unquantized weights.

```
def forward(self, x):
    x_norm = self.Normalize(x)

    w = self.weight # a weight tensor with shape [d, k]

    if self.QuantType == 'None':
        y = F.linear(x_norm, w)
    else:
        # A trick for implementing Straight-Through-Estimator (STE) using detach()
        x_quant = x_norm + (self.activation_quant(x_norm) - x_norm).detach()
        w_quant = w + (self.weight_quant(w) - w).detach()
        y = F.linear(x_quant, w_quant)
    return y
```

I implemented custom normalization and quantization functions to explore different options. I found no benefit for using batch normalization instead of RMS normalization[^6], hence I used the latter one as in the original BitNet implementation. This choise also simplifies the on-device inference.

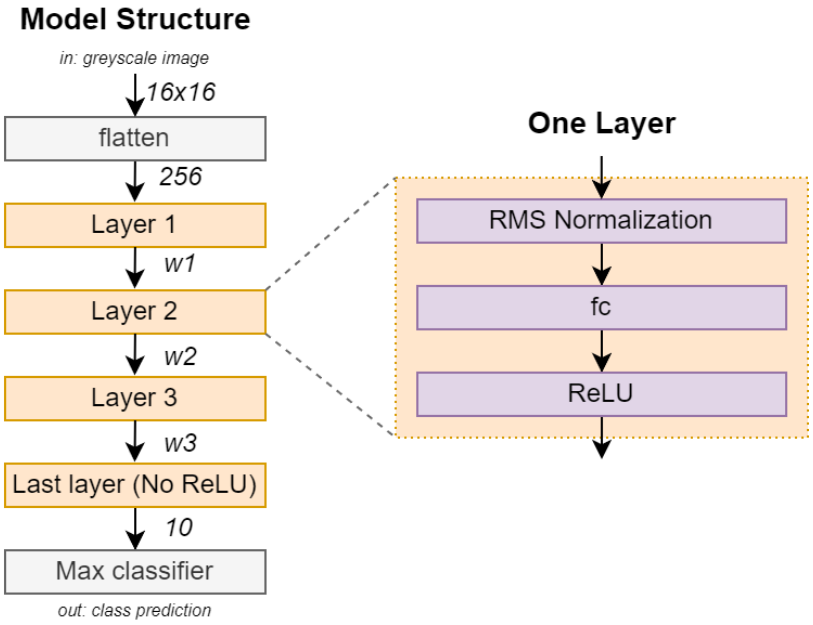
The implementation of one bit quantization is straight forward, the scale derived from the mean value of the weights in the entire layer.

```
    scale = w.abs().mean().clamp_(min=1e-5)
    e = w.mean()
    u = (w - e).sign() * scale
```

For 2 and more bit quantization I chose to use symmetric encoding without zero, e.g. **[11,10,00,01] -> [+1.5 +0.5 -0.5 -1.5]**. Assymetric encoding including zero did not show any benefits. The scale factor of 1.5 was chosen empirically. Experiments with per-channel scaling did not show any improvement over per-layer scaling in QAT, surely something to revisit later. The quantization function is shown below.

```
    mag = w.abs().mean().clamp_(min=1e-5)
    scale = 1.0 / mag # 2 worst, 1 better, 1.5 almost as bad as 2
    u = ((w * scale - 0.5).round().clamp_(-2, 1) + 0.5) / scale
```

To keep things simple, especially on side of the inference engine, I decided to use only fully connected layers and no CNN layers. To reduce memory footprint, the samples of the MNIST dataset are rescaled from 28x28 to 16x16 in size. This reduced resolution and lack of CNN layers will hamper achievable accuracy. But acceptable performance is still achievable, as shown later. The model structure is shown in the figure below.



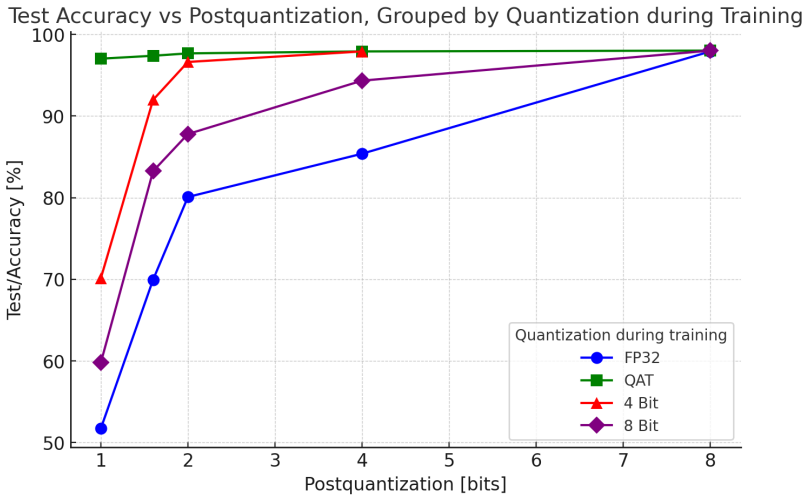
The sizes of the hidden layers are parametrizable.

Model Optimization

The MNIST dataset with standard train/test split was used. Unless noted otherwise, the model was trained with a batch size of 128 and intial learning rate of 0.01 for 30 Epochs. The learning rate was reduced by a factor of 0.1 after each 10 epochs. Adam optimizer was used and cross entropy loss function.

Quantization Aware Training vs Post-Quantization

To investigate the efficacy of QAT, I trained the model with different bit-widths for weights and quantizated it to the same or smaller bitwidth before testing. I kept the network size at *w1=w2=w3=64* for all runs.

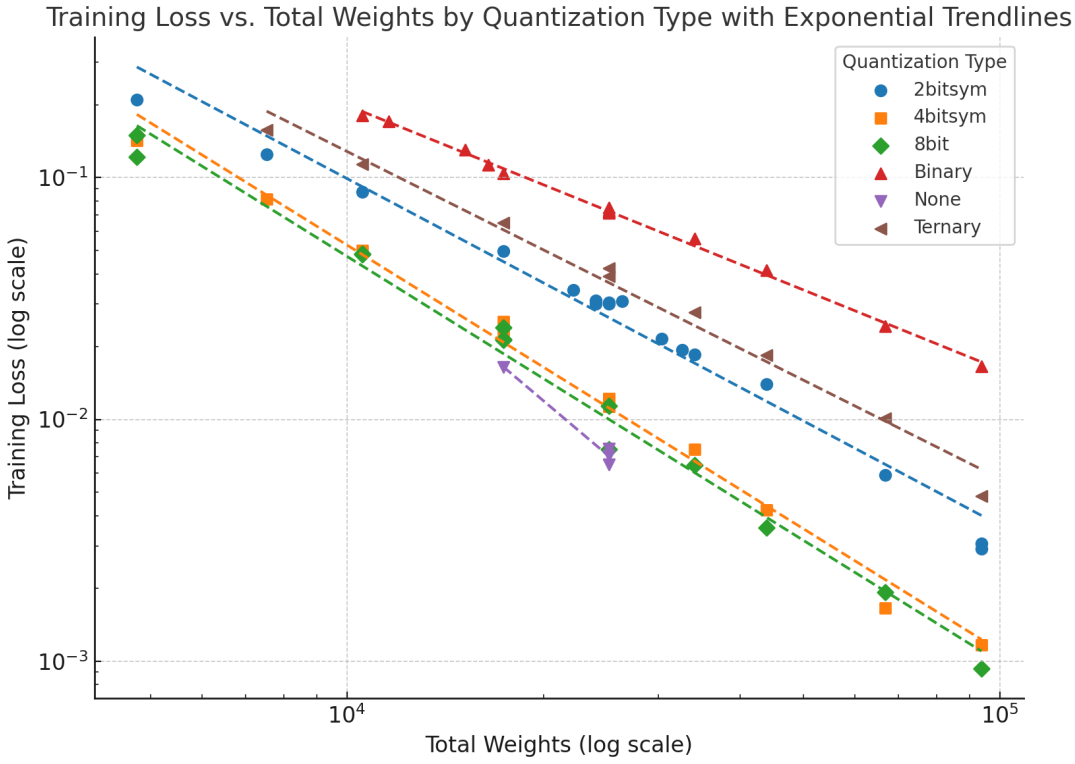


Around 98% accuracy is achieved with 8-bit quantized weights, whether trained with QAT or post-quantization. When a model is quantized to 4-bits or less after training, a significant drop in accuracy is observed. Quantization aware training can distribute the quantization error to other weights and achieves good accuracy even for one bit weights.

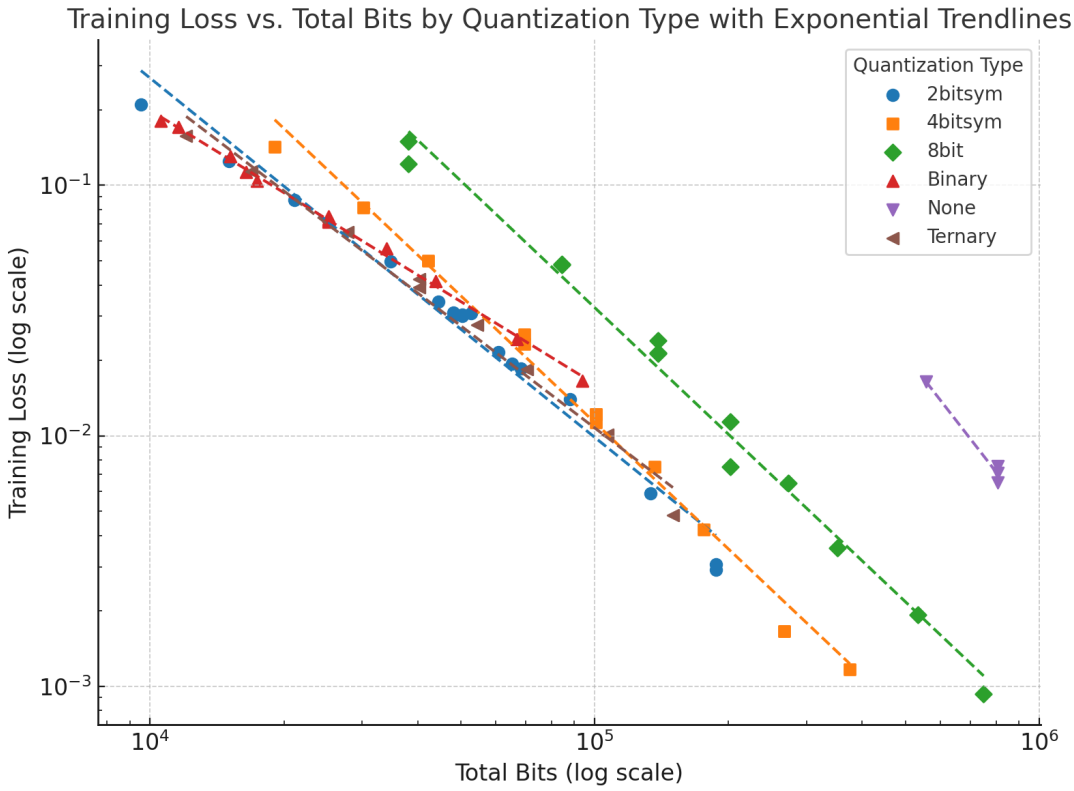
Advanced post-quantization schemes as used in LLMs could improve post-quantization accuracy, but QAT is cleary the preferred approach for this use case.

Model Capacity vs Quantization scaling

To deploy the model on a microcontroller, the model size should be kept as small as possible. The required memory footprint is proportional to the number of weights and the quantization level of the weights. To understand all tradeoffs I trained a large number of models with different widths and quantization levels, all using QAT. Typically, the width of all layers was kept the same.



The plot above shows training loss vs. total number of weights. We can see that there is a polynomial relationship between the number of weights and the training loss. Reducing the number of bits per weigths increases the loss proportionally. Interestingly, there are diminishing returns when increasing the number of bits beyond 4 and loss is not reduced further. It appears that beyond 4 bit, no more information per weight can be stored in the model.



To understand which quantization level is the most memory efficient, I plotted the training loss vs. the total number of bits in the model. The number of bits is the product of the number of weights and the number of bits per weight. As can be seen in the plot above, the loss is proportional to the total number of bits, almost regardless of the quantization level between 1 and 4 bits.

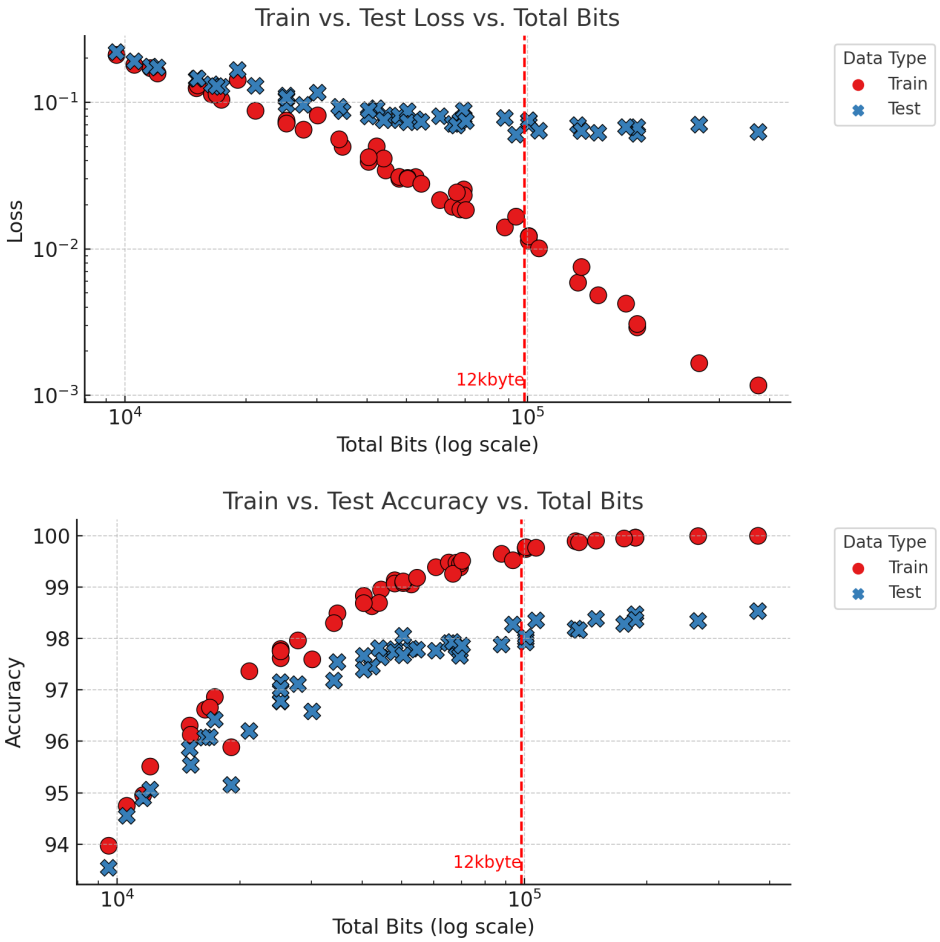
This trend seems to indicate that it is possible to trade bits per weight for an increased number of weights. Assuming that the training loss is indicative of the density of information stored in the model, this indicates that each bit carries the same amount of information, regardless of whether it is spent on increasing quantization levels or introducing more weights.

<rant> I am really intrigued by this result. Although this scaling relationship intuitively makes sense, I was not able to find much information about it. It raises the question of how to maximize information storage in the neural network. A cursory analysis of the weights according to Shannon information entropy suggests that it is near capacity for QAT, indicating that all weight encodings are used equally often. However, I also found that post-quantization of 4 bits can achieve the same loss with a weight encoding that follows a more normal distribution and therefore has lower entropy. If the entropy of the weights is not maximized, it means there could be on-the-fly compression to improve memory footprint. There are plenty of interesting things to come back to for later investigation.</rant>

Practically, this scaling relationship means that the number of bits per weight between 1-4 is a free variable that can be chosen depending on other requirements, such as compute requirements for inference.

Test Accuracy and Loss

The scaling relationship above allows prediciting train loss from model size. The plots below show the relationship between train and test loss and accuracy vs. model size.



The tests above reveal a clear monotonic relationship between the model size and both test loss and accuracy, given the training setup used. However, there's a point of saturation where the test loss does not decrease any further. This could be indicative of overfitting, or it may suggest that the training data lacks sufficient variability to accurately represent the test data.

Considering the memory constraints of the target platform for inference, I've assumed that a maximum of 12kb of memory is available for the model. This is indicated by the red line in the plots above.

To improve model performance further, I fixed the model size to 12kb and explored various training parameters.

Optimizing training parameters

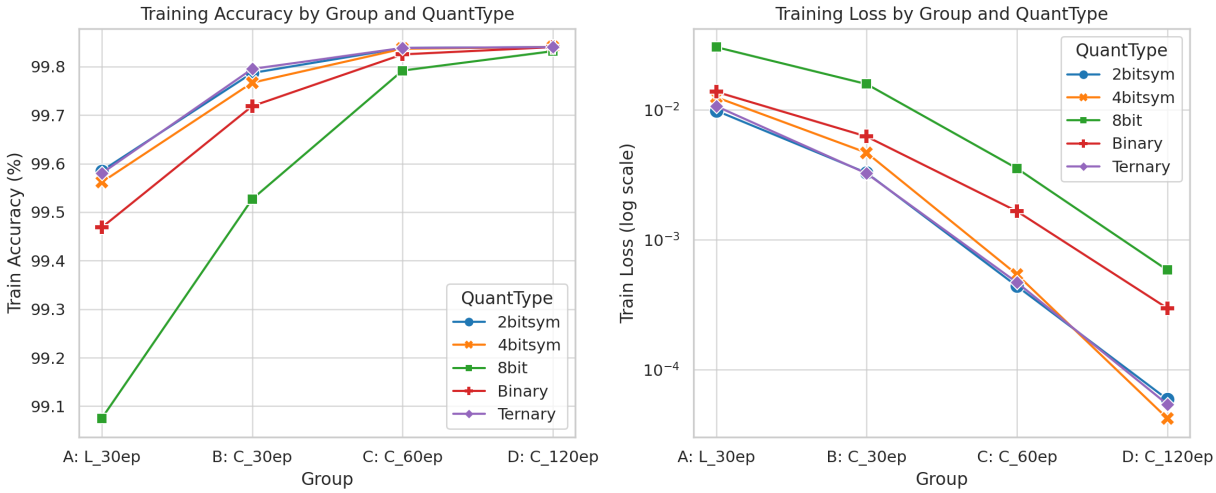
The table below shows a set of network parameters that result in a model size close to 12kbyte for various quantization levels. The number of weights in each layer was chosen to align parameter storage with int32. I tried to maintain the same width across all layers.

Quantization [bit]	**1 bit**	**Ternary**	**2 bit**	**4 bit**	**8 bit**
input	256	256	256	256	256
w1	176	128	112	64	40
w2	160	128	96	64	32

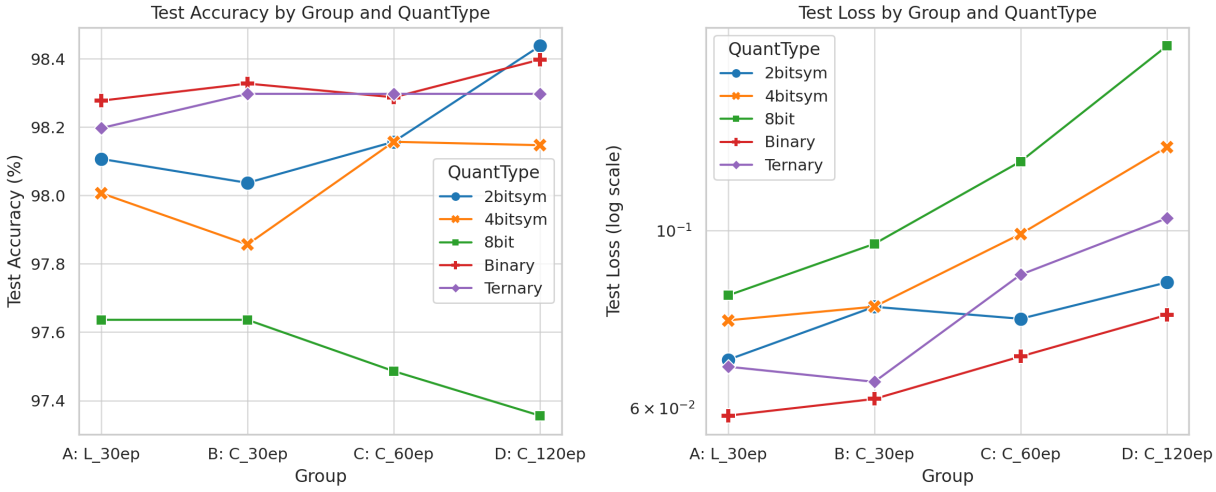
Quantization [bit]	1 bit	Ternary	2 bit	4 bit	8 bit
w3	160	112	96	64	32
output	10	10	10	10	10
Number of weights	100416	64608	49600	25216	12864
total bits [kbit]	100416	103372.8	99200	100864	102912

Learning rate and number of epochs

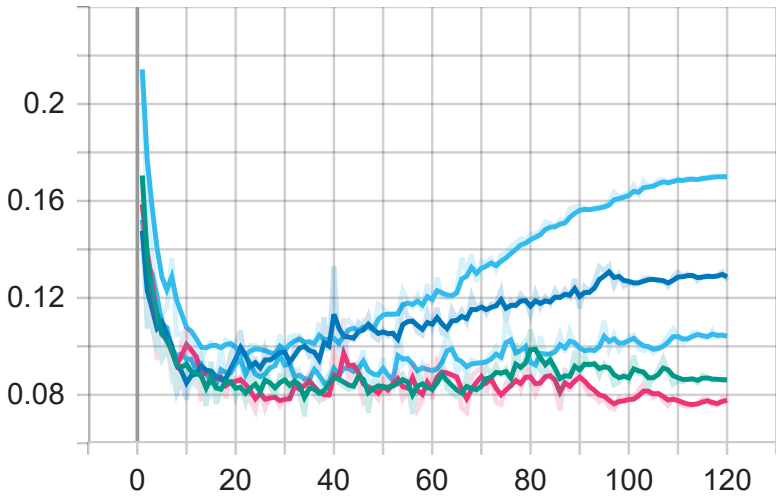
I trained the models with different learning schedules and for a varying number of epochs. Switching from a linear to a cosine schedule resulted in a modest improvement, hence I kept this throughout the rest of the experiments.



The training loss and accuracy is shown above. As expected from the experiments above, all models perform similarly. The 8bit quantized model has the higher loss, again. We can see that longer training reduces the loss and increases the accuracy.



The test loss and accuracy, in contrast, does not show a significant improvement with longer training. The test loss increases with longer training time, suggesting overfit of the training data.



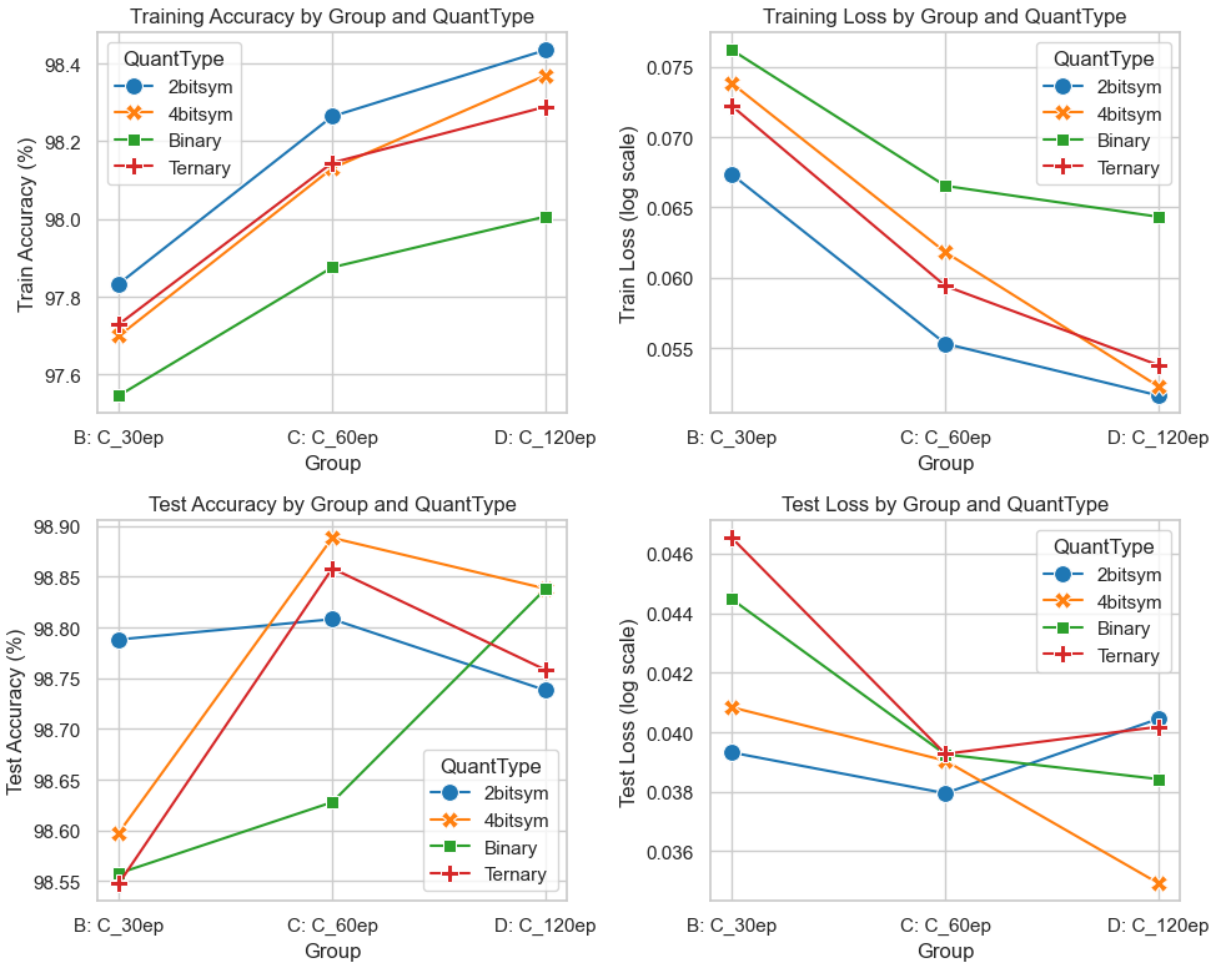
The test loss plot above for the 120 epoch runs clearly shows that the higher the number of bits per weight, the greater the increase in test loss. This dependence is somewhat surprising, as one might assume from the previous results that all models have the same capacity and therefore should exhibit similar overfitting behavior. However, it has been previously suggested that low bit quantization can have a regularizing effect on the network[^7]. This could explain the observed behavior.

However, despite the regularizing effect, the test accuracy does not exceed 98.4%, suggesting that the model is unable to generalize to all of the test data.

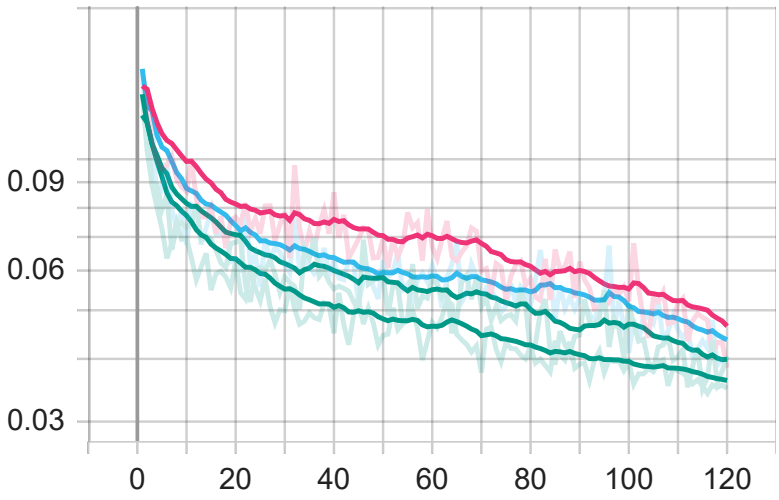
Data Augmentation

To improve the generalization of the model, and counter the overfitting, I I applied data augmentation to the training data. Randomized affine transformations were used to add a second set of training images to the unaltered MNIST dataset. In each epoch, the standard set of 60000 training images plus 60000 images with randomized transformations were used for training.

```
    augmented_transform = transforms.Compose([
        transforms.RandomRotation(degrees=10),
        transforms.RandomAffine(degrees=10, translate=(0.1, 0.1), scale=(0.9,
1.1)),
        transforms.Resize((16, 16)),  # Resize images to 16x16
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])
```



The loss and accuracy for different learning run epochs are displayed above. The training loss is higher than that observed without data augmentation, but the test accuracy also increases by more than 0.5%, approaches 98.9%. The test loss decreases with a higher number of epochs, suggesting that the model is not yet overfitting the data. This is further confirmed by the test loss plot below. Inserting dropout layers was also able to reduce overfitting, but I found that data augmentation was more effective in improving the test accuracy.



Interestingly, the test accuracy trend is now reversed, with higher bit quantization showing a slight advantage, despite identical total model sizes. The reason for this is not entirely clear.

I was able to achieve almost 99.0% accuracy with 4-bit quantization after tweaking the data augmentation parameters slightly. Further improvements might require a different model architecture, such as a CNN.

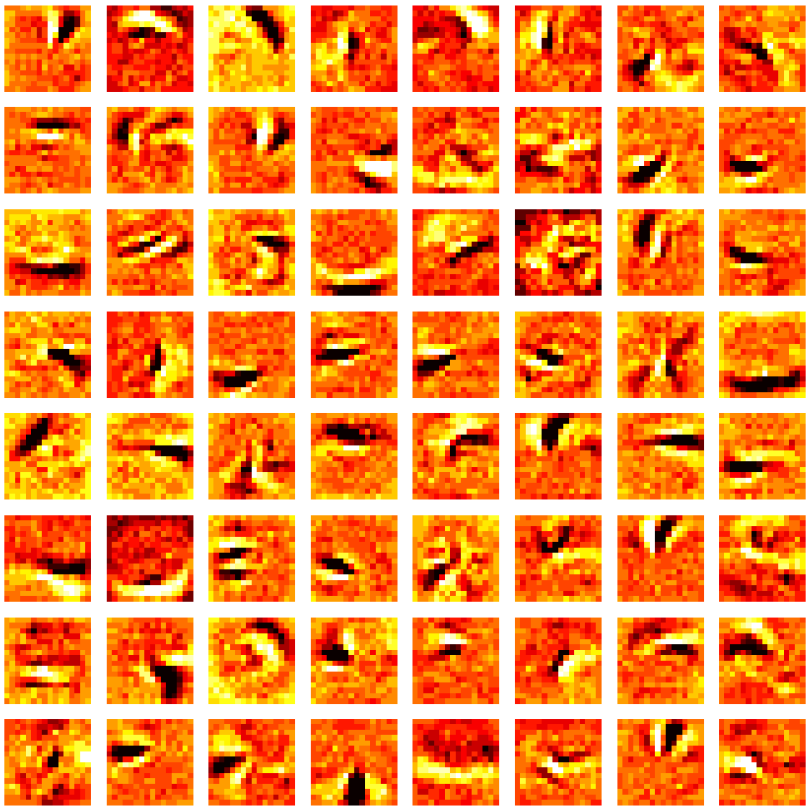
However, to keep things simple, I will stop here. The 99.0% accuracy already surpasses most (all?) other MNIST inference implementations I have seen on low-end MCUs such as AVR.

Summary of Learnings from Training

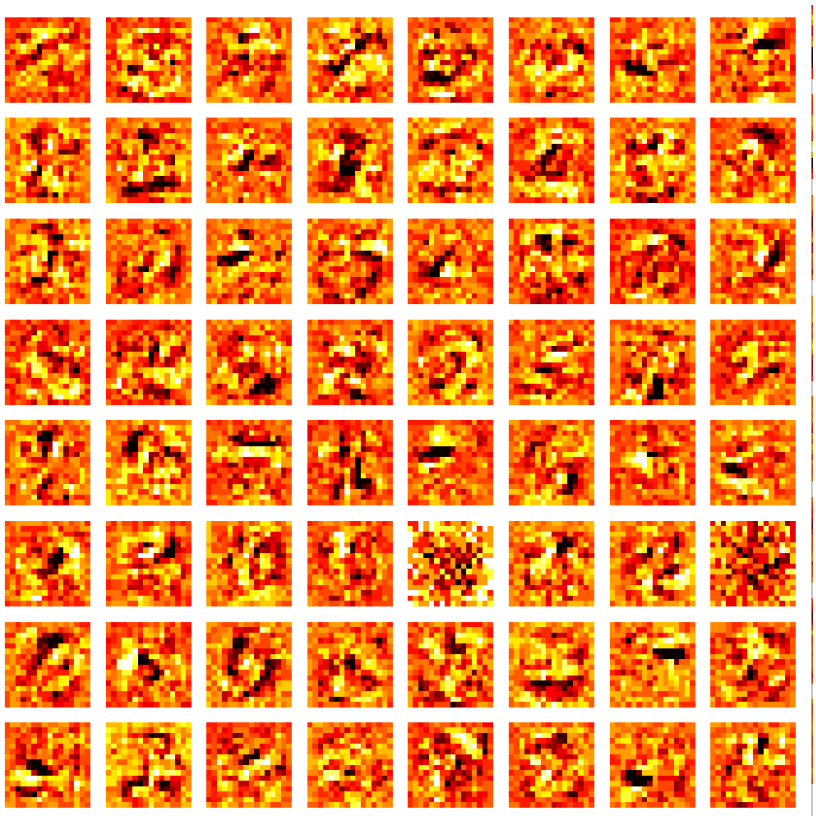
- Quantization Aware Training (QAT) enables high-accuracy models with low-bit quantization.
- Between 1 and 4-bit quantization, it was possible to trade the number of bits per weight for more weights without a significant impact on accuracy and loss.
- 8-bit quantization was less efficient in terms of memory efficiency.
- Lower bit quantization can have a regularizing effect on the network, leading to less overfitting.
- When overfitting was addressed with data augmentation, higher bit quantization showed increasingly improved accuracy.
- No specific benefit of ternary encoding over symmetric 2 or 4-bit encoding was observed.
- Considering all the trade-offs above, it appears that 4-bit quantization is the most preferable option for the problem at hand, as it offered the best performance and required the least number of weights, reducing computational effort.
- 99.0% accuracy was achieved with 4-bit quantization and data augmentation.

Addendum: Visualization of First Layer Weights

I visualized the weights of the first layer of the model trained with data augmentation. There are 16x16 input channels and 64 output channels. We can see that each channel detects certain structured features of the full input image.



In contrast, the visualization below represents the first layer weights of the model trained without data augmentation. The weights seem less structured and appear more random. Instead of learning general features, the network seems to tend to fit directly to the images, as suggested by the discernible shapes of numbers.



Since the weights of a CNN are exposed to many different features within the same image, it can learn generalized features much more effectively and less brute-force is required to prevent the model from overfitting.

Addendum: Potential of Additional CNN layers

To explore the potential of adding CNN layers, I added two 3x3 Conv2D layers to the model. The Conv2d layers were trained in float, while the fc layers were trained with 4-bit QAT. The model was trained with the same parameters as before (w1=w2=w3=64, augmentation, 60epochs).

```
    self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1, bias=False)
    self.conv2 = nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1, bias=False)
...
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x, kernel_size=2, stride=2)
    x = F.relu(self.conv2(x))
    x = F.max_pool2d(x, kernel_size=2, stride=2)
```

This modification increased the test accuracy to above 99%. Memory-efficient depthwise convolution with similar parameters yielded similar results. More to follow up opon later...

Architecture of the Inference Engine

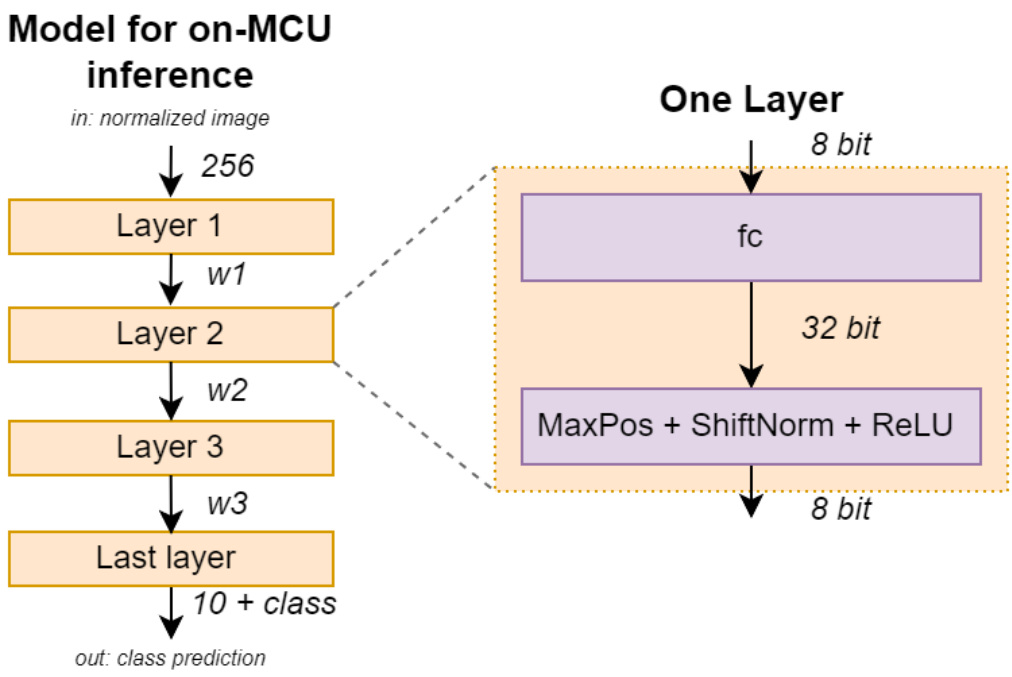
Since the training is performed on the model in float format (the weights are only quantized during forward and backward pass), the model needs to be converted to a quantized format and then exported to a format that can be included into the inference code on the CH32V003.

Let's first start with the target architecture. To reduce computational effort on the microcontroller as much as possible, I slighty modified the network architecture implementation.

Key observations:

- No normalization is needed for the first layer as the input data (test sample) is already normalized.
- The fully connected layer accepts 8 bit signed integers, but as a result of the summation of partial products, the output is 32 bit.
- The normalization operation rescales the output back to an 8-bit value, but can only do this once all activations from the previous layer are available.
- Classification is performed by selecting the maximum value from the output layer and returning the index, which needs to be done before the ReLU operation.

These requirements can be met by the network architecture below. Each layer consists of a fully connected layer, followed by a fused block where normalization and ReLU are combined. Classification (maxpos) is performed before the ReLU operation in the same block, but the result is only used for the output layer.



One crucial observation for further optimization of the inference engine:

The model is tolerant to scaling errors as it doesn't employ any operations, other than ReLU, that alter the mean or linearity.

This is true because RMS norm is used as opposed to batch or layer norm, which prevents altering the mean of the activations. Additionally, no bias was used. While ReLU sets negative values to zero, it maintains the linearity of positive values.

This allows for a simpler normalization scheme to replace the RMS norm. I discovered that a simple shift operation, **ShiftNorm**, could be used instead. This operation shifts all output values in the layer until the maximum is less than 127 (int8_t). While this introduces some quantization noise, they I found it inconsequential in testing.

Ultimately, this means that no multiplication operation is required outside of the fully connected layer!

Implementation in Ansi-C

fc-layer

The code for a convolution with binary weights is straightforward:

```
    int32_t sum = 0;
    for (uint32_t k = 0; k < n_input; k+=32) {
        uint32_t weightChunk = *weightidx++;

        for (uint32_t j = 0; j < 32; j++) {
            int32_t in=*activations_idx++;
            sum += (weightChunk & 0x80000000) ? in : -in;
            weightChunk <<= 1;
        }
    }
    output[i] = sum;
```

This is how the inner loop looks in RV32EC assembly (compiled with -O3)

```
<processfclayer+0x28>
     1d8:   0785                addi    a5,a5,1
     1da:   fff78303            lb      t1,-1(a5) # ffff <_data_lma+0xc6db>
     1de:   02074563            bltz    a4,208    <processfclayer+0x58>
     1e2:   0706                slli    a4,a4,0x1
     1e4:   40660633            sub     a2,a2,t1
     1e8:   fe5798e3            bne     a5,t0,1d8 <processfclayer+0x28>
    ...
<processfclayer+0x58>
     208:   0706                slli    a4,a4,0x1
     20a:   961a                add     a2,a2,t1
     20c:   fc5796e3            bne     a5,t0,1d8 <processfclayer+0x28>
```

The full loop is 6 instructions while the actual computation is just 3 instructions (lb, bltz,neg/add). The compiler did quite a good job to split the conditional into two code paths to avoid an addition "neg" instruction.

It would be possible to unroll the loop to remove loop overhead. In that case 4 instructions are required per weight, since the trick with two codes pathes would not work easily anymore.

Convolution with 4 bit weight is shown below. The multiplication is implemented by individual bit test and shift, as the MCU does not support a native multiplication instruction. The encoding as one-complement number without zero helps with code efficiency.

```
    int32_t sum = 0;
    for (uint32_t k = 0; k < n_input; k+=8) {
        uint32_t weightChunk = *weightidx++;
```

```
        for (uint32_t j = 0; j < 8; j++) {
            int32_t in=*activations_idx++;
            int32_t tmpsum = (weightChunk & 0x80000000) ? -in : in;
            sum += tmpsum;                                  // sign*in*1
            if (weightChunk & 0x40000000) sum += tmpsum<<3; // sign*in*8
            if (weightChunk & 0x20000000) sum += tmpsum<<2; // sign*in*4
            if (weightChunk & 0x10000000) sum += tmpsum<<1; // sign*in*2
            weightChunk <<= 4;
        }
    }
    output[i] = sum;
```

Again, the compiled code of the inner loop below. The compiler decided to unroll the loop (8x), which removed the loop overhead.

```
     1d2:   00060383            lb      t2,0(a2)
     1d6:   00075463            bgez    a4,1de <processfclayer+0x2e>
     1da:   407003b3            neg     t2,t2
<processfclayer+0x2e>
     1de:   00371413            slli    s0,a4,0x3
     1e2:   979e                add     a5,a5,t2
     1e4:   00045563            bgez    s0,1ee <processfclayer+0x3e>
     1e8:   00139413            slli    s0,t2,0x1
     1ec:   97a2                add     a5,a5,s0
<processfclayer+0x3e>
     1ee:   00271413            slli    s0,a4,0x2
     1f2:   00045563            bgez    s0,1fc <processfclayer+0x4c>
     1f6:   00239413            slli    s0,t2,0x2
     1fa:   97a2                add     a5,a5,s0
<processfclayer+0x4c>
     1fc:   00171413            slli    s0,a4,0x1
     200:   00045463            bgez    s0,208 <processfclayer+0x58>
     204:   038e                slli    t2,t2,0x3
     206:   979e                add     a5,a5,t2
<processfclayer+0x58>
     208:   00471413            slli    s0,a4,0x4
```

In total 17 instructions are required per weight, with no additional loop overhead.

Considering the observations during model optimization, binary weights require approximately four times as many weights as 4-bit quantization to achieve the same performance. The execution time for binary inference is `4 cycles * 4 * number of weights`, while for 4-bit quantization, it's `17 cycles * number of weights`.

Consequently, the pure computation time is comparable for both quantization levels, offering no inference time advantage for binary weights for the given problem setting. In fact, due to the additional overhead from the increased number of activations required with binary weights, the total execution time is likely higher for binary weights.

The implementation for 2 bit quantization is not shown here, but it is similar to the 4 bit quantization. I did not implement Ternary weights due to complexity of encoding the weights in a compact way.

It should be noted, that the execution time can be improved by skipping zero activations. Typically, more than half of the activations are zero.

ShiftNorm / ReLU block

The fused MaxPos / ShiftNorm / ReLU block is straightforward to implement.

```
    // Find the maximum value in the input array
    for (uint32_t i = 0; i < n_input; i++) {
        if (input[i] > max_val) {
            max_val = input[i];
            max_pos = i;
        }
    }

    // Find shift value to normalize the maximum value to <127
    tmp=max_val>>7;
    scale=0;

    while (tmp>0) {
        scale++;
        tmp>>=1;
    }

    // Apply ReLU activation and normalize to 8-bit
    for (uint32_t i = 0; i < n_input; i++) {
        if (input[i] < 0) {
            output[i] = 0;
        } else {
            output[i]=input[i] >> scale;
        }
    }
    return max_pos;
```

And that's all - **not a single multiplication operation was required**.

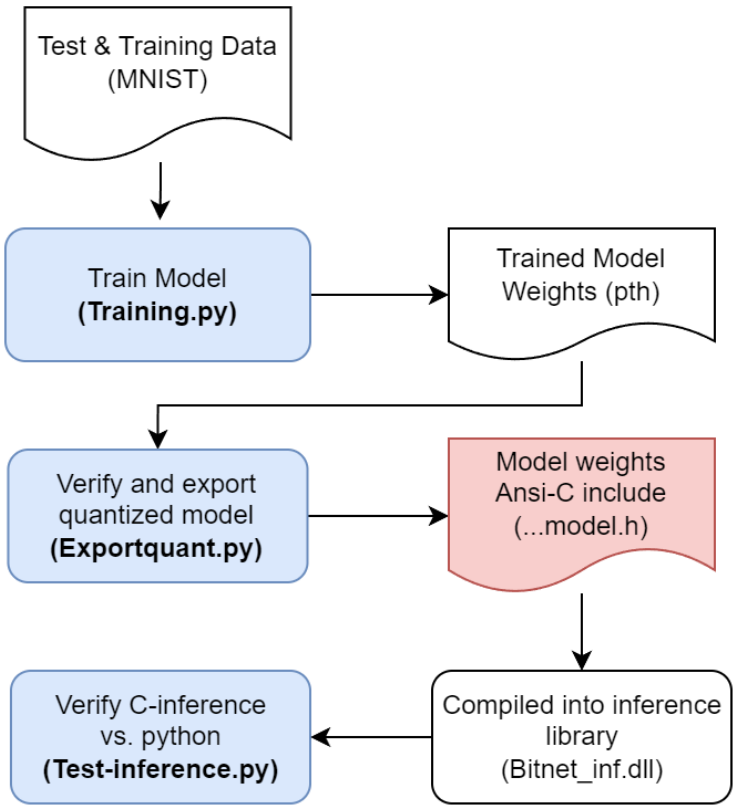
Putting it all together

To keep things flexible, I split up the data pipeline into several python scripts. **training.py** is used to train the model and store it as *.pth* file. The model weights are still in float format at that time, since they are quantized on-the-fly during training. **exportquant.py** converts the model into a quantized format, a custom python class, that is only used as an intermediate representation for export and testing. The quantized model data is then merged into 32 bit integers and exported to a C header file.

To test inference of the actual model as a C-implementation, the inference code along with the model data is compiled into a DLL. **test-inference.py** calls the DLL and compares the results with the original python model

test case by test case. This allows accurate comparison to the entire MNIST test data set of 10000 images.

The flow is shown in the figure below.



Model Exporting

Output of the exporting tool is shown below. Some statistics on parameter usage are calculcated. We can see that the model is using all available codes, but they are not evenly distributed. This means that the model could be compressed further and that the entropy is not maximized.

```
Inference using the original model...
Accuracy/Test of trained model: 98.98 %
Quantizing model...

Layer: 1, Max: 7.5, Min: -7.5, Mean: -0.00360107421875, Std: 2.544043042288096
Values: [-7.5 -6.5 -5.5 -4.5 -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5  4.5  5.5 6.5
7.5]
Percent: [ 2.47192383  0.73242188  0.8972168   1.59301758  3.08227539  6.68334961
12.83569336 20.71533203 20.86181641 13.9831543   7.36083984  3.47900391 2.12402344
1.21459961  0.81176758  1.15356445]
Entropy: 3.25 bits. Code capacity used: 81.16177579491874 %

Layer: 2, Max: 7.5, Min: -7.5, Mean: -0.12158203125, Std: 2.5687543790088463
Values: [-7.5 -6.5 -5.5 -4.5 -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5  4.5  5.5 6.5
7.5]
Percent: [ 1.171875    0.95214844  1.53808594  3.56445312  5.54199219  8.71582031
12.25585938 16.38183594 16.50390625 13.50097656 10.05859375  5.56640625 2.41699219
1.09863281  0.46386719  0.26855469]
```

```
Entropy: 3.38 bits. Code capacity used: 84.61113322636102 %

Layer: 3, Max: 6.5, Min: -7.5, Mean: -0.23291015625, Std: 2.508764116126823
Values: [-7.5 -6.5 -5.5 -4.5 -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5  4.5  5.5
6.5]
Percent: [ 0.78125     0.9765625   1.92871094  4.05273438  5.71289062 10.08300781
12.93945312 14.23339844 16.33300781 14.23339844  9.91210938  5.59082031 2.34375
0.63476562  0.24414062]
Entropy: 3.35 bits. Code capacity used: 83.84599479239081 %

Layer: 4, Max: 4.5, Min: -7.5, Mean: -0.73125, Std: 2.269283683786582
Values: [-7.5 -5.5 -4.5 -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5  4.5]
Percent: [ 0.15625  0.78125  4.0625  12.96875 15.3125  15.625   14.6875  11.25
9.53125 10.46875  4.21875  0.9375 ]
Entropy: 3.15 bits. Code capacity used: 78.82800031261552 %
Total number of bits: 100864 (12.3125 kbytes)
inference of quantized model
Accuracy/Test of quantized model: 99.00999999999999 %
```

The total size of the model is 12.3 kilobytes. The tool also performs inference on both the original PyTorch model and a quantized version that emulates the ShiftNorm operation. Interestingly, the accuracy of the quantized model is 99.01%, which is slightly better than the original model. The model data is written to a header file, which is subsequently included in the C code.

```
const uint32_t L2_weights[] = {0x1231aa2, 0x29c09a90, 0x20a16b50, 0x8c938109,
0x320a2301, 0x2810008, 0x89114a9a, 0x9fb1c101, 0x899c90, 0x2889329, 0xab0b9bcc,
0x9a419319, 0x8209091a, 0x2b8da0b9, 0x282144a0, 0x3fb8881, ...
```

Verification of the Ansi-C Inference Engine vs. Python

The exported data and the inference engine are compiled to DLL, which is then called from the python test script and compares the predictions image- by image.

The output is shown below. Curiously, the C inference engine is yet again slightly better than the Python implementation. There are still three (out of 10000) test images where both engine disagree. I believe this is due to different rounding behavior of the two engines. I was already able to reduce this from a larger number by adding additional rounding to the ShiftNorm operation.

```
Loading model...
Inference using the original model...
Accuracy/Test of trained model: 98.98 %
Quantizing model...
Total number of bits: 100864 (12.3125 kbytes)
Verifying inference of quantized model in Python and C
Mismatch between inference engines found. Preduction C: 6 Prediction Python: 4
True: 4
Mismatch between inference engines found. Preduction C: 5 Prediction Python: 0
True: 5
```

```
Mismatch between inference engines found. Preduction C: 3 Prediction Python: 5
True: 3
size of test data: 10000
Mispredictions C: 98 Py: 99
Overall accuracy C: 99.02 %
Overall accuracy Python: 99.00999999999999 %
Mismatches between engines: 3 (0.03%)
```

Implementation on the CH32V003

The implementation on the CH32V003 is straightforward and can be found here. The model data is included in the C code, and the inference engine is called from the main loop. I used the excellent CH32V003fun environment to minimize overhead code as much as possible. This allowed me to include up to 12KB of model data into the 16KB of flash memory.

Four test cases are evaluated, and the execution timing is measured. The execution timing was optimized by moving the fc-layer code to the SRAM, which avoids flash wait states.

Example output for inference with a 25126 4-bit parameter model is shown below.

```
Starting MNIST inference...
Inference of Sample 1  Prediction: 7  Label: 7  Timing: 655433 clock cycles
Inference of Sample 2  Prediction: 1  Label: 1  Timing: 647093 clock cycles
Inference of Sample 3  Prediction: 9  Label: 9  Timing: 639987 clock cycles
Inference of Sample 4  Prediction: 4  Label: 4  Timing: 650411 clock cycles
```

The execution time is approximately 650,000 cycles, which corresponds to 13.66ms at a 48MHz main clock. This is equivalent to 3.69 million operations per second ("MOPS"). The model achieves a test accuracy of 99.02%, which is quite impressive for such a small microcontroller and surpasses all other MCU-based MNIST implementations I have encountered.

I also tested a smaller model with 4,512 2-bit parameters. Despite its size, it still achieves a 94.22% test accuracy. Due to its lower computational requirements, it executes in only 1.88ms.

Summary and Conclusions

This concludes my journey to implement an MNIST inference engine with an impressive 99.02% test accuracy on a very limited $0.15 RISC-V microcontroller, which lacks a multiplication instruction and has only 16KB of flash memory and 2KB of RAM.

This was possible by employing Quantization Aware Training (QAT) in combination with low-bit quantization and accurate model capacity tuning. By simplifying the model architecture and a full-custom implementation I side-stepped the usual complexities and overhead associated with Edge-ML inference engines.

While this project focused on MNIST inference as a test case, I hope to use this approach for other applications in the future.

References

[^1]: S. Ma et al *The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits* (arXiv:2402.17764) and discussion here

[^2]: A. Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference* (arXiv:2103.13630)

[^3]: M. Rastegari et al. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks* (arXiv:1603.05279) and *BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1* (arXiv:1602.02830)

[^4]: S. Ma et al. *The-Era-of-1-bit-LLMs__Training_Tips_Code_FAQ* (Github)

[^5]: Y. Bengio et al. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation* arXiv:1308.3432

[^6]: B. Zhang et al. *Root Mean Square Layer Normalization* arXiv:1910.07467

[^7] M. Courbariaux et al. *BinaryConnect: Training Deep Neural Networks with binary weights during propagations* arXiv:1511.00363

regex: ^(?!.*\).*2bit