Semester Project Report

Fibonacci-code quantization for faster inference in Deep Learning

Valérian Rey

Supervised by Mr. William Simon, Dr. Marina Zapater,

Prof. David Atienza

Embedded Systems Laboratory
Switzerland
June 2020

Contents

1. Provided problem statement[1]

Over the last decade, neural networks have been applied to a wide range of applications with great success. However, they are power and performance hungry; therefore, much work has been done to reduce computational complexity through different techniques, particularly in this case weight quantization[2]. At the same time, various hardware accelerators have been proposed to improve inference, while at the same time often reducing energy consumption. The goal of this project is to accelerate neural networks by quantizing pretrained weights to Fibonacci code[3]. Once quantized, Fibonacci quantized neural networks can then be accelerated via a hardware accelerator being developed by the lab. Fibonacci coding codes base-2 integer values such that there are no consecutive 1 bits. By quantizing weights to such sequences, partial sum combination can be accelerated by adding more than two partial sums simultaneously, as demonstrated below.

170								1	0	1	0	1	0	1	0
255							x	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0
	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0
	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0
+	0	0	0	0	1	0	1	0	1	0	1	0	0	0	0
	0	0	0	1	0	1	0	1	0	1	0	0	0	0	0
	0	0	1	0	1	0	1	0	1	0	0	0	0	0	0
	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0
+	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0
				1	0	0	1	1	1	1	1	0	1	1	0
+	1	0	0	1	1	1	1	0	1	1	0	0	0	0	0
43350	1	0	1	0	1	0	0	1	0	1	0	1	0	1	1	0

Figure 1. Fibonacci code multiplication

When quantizing neural networks, care must be taken to guarantee that accuracy is maintained once the weights have been quantized. This will be the main challenge of the project. Some theories for maintaining accuracy via retraining have already been proposed in the lab based on previous literature, and will need to be tested during the project.

2. Introduction

Given a pre-trained model, the first step is to quantize it as integer. This requires a lot of changes in the code, but in many cases it induces only a very little loss of accuracy. The second step is to encode the multiplicative weights of the model as Fibonacci codes. This generally reduces the accuracy of the model by a lot, so a lot of care must be taken when doing that. In order to keep that effect as low as possible, multiple strategies to quantize, Fibonacci-encode and retrain the model have been tested and will be exposed in this report. For simplicity, the pipeline developed around Pytorch through this project only supports the use of sequential models that contain modules in the following subset:

```
torch.nn.Linear
torch.nn.Conv2d
torch.nn.BatchNorm2d
torch.nn.MaxPool2d
torch.nn.ReLU
```

While this is a very short list of possibilities, it already allows to build many existing classification neural networks in the field of computer vision. Experiments could thus be made on the MNIST[4] and CIFAR-10[5] datasets, as presented at the end of this report.

To be clearer, we'll use the terms **original model** to refer to the pretrained model without any quantization, **int-quantized model** to refer to the model after it is quantized as integer values and **x% Fibonacci-encoded model (retrained)** to refer to the model quantized as integer values, where at least x% of the weights are fixed Fibonacci codes, and possibly after it has retrained. Sometimes we simply say **encoded** instead of **Fibonacci-encoded**.

3. Integer Quantization

The work done in this section follows the ideas taken from Gemmlowp[6]. For our initial quantization we want to end up with unsigned integer values. This is mainly because Fibonacci-encoding works very bad on negative numbers, due their two's complement structure. 8-bits symmetric quantization would bring the numbers to the range {-128, -127, ..., 126, 127} while 8-bits asymmetric quantization would bring them to the range {0, 1, ..., 254, 255}, meaning that we get our desired unsigned integer numbers. Thus we use **asymmetric quantization** throughout this project.

We will present the concept with general explanations and formulas, as well as with an example.

In this section we will talk about how the weights and the activations are quantized (initially and after each layer pass). We will also detail how each quantized layer operation is done.

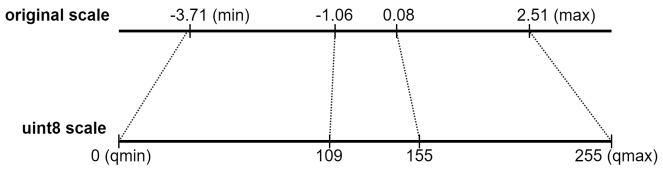

Figure 2. Asymmetric quantization

We denote as qmax the maximum quantized value (255 in 8-bits), qmin as the minimum quantized value (always 0). We also denote as low_val the value that will get quantized

to qmin and as high_val the value that will get quantized to qmax. These values can be the minimum and maximum values of the original unscaled tensors (-3.71 and 2.51 in the example of figure 2), but as explained in the section Input, this is not necessarily the case.

In the code we do often use fake types (integer stored as float) because some operations are only supported with those types on Pytorch (especially when using CUDA).

A. Initial quantization

The initial quantization operation first computes a scale and an offset (named zero_point, or sometimes zp).

$$scale = \frac{high\_val - low\_val}{qmax - qmin} \tag{1}$$

$$zero\_point = round(qmin - \frac{low\_val}{scale}) \tag{2}$$

With the same vector used in figure 2, these values are $scale = \frac{2.51+3.71}{255} = 0.02439$ and $zero\_point = round(\frac{3.71}{0.02439}) = 152$. The input is then quantized as shown in figure 3.

We measured the time taken by the initial quantization of the input (which happens during inference). It only took about 0.2% of the total inference time for the small MNIST Net model (figure 7).

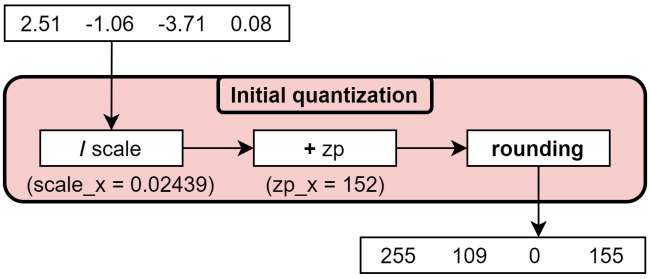

Figure 3. Initial quantization layer

1) Weight and bias

The initial quantization operation is first applied to the weights. For Linear layers, we just take for low_val and high_val the minimum and maximum weights, respectively. We then compute the corresponding scale and zero_point and quantize the whole weight matrix with those values.

For a Conv2d layer, with in_channels input channels and out_channels output channels, we have $in\_channels * out\_channels$ weight matrices (kernels). Thus we can quantize them separately, by in_channel, by out_channel, or all together. Only per-out-channel and per-layer (all together) quantization was tested. The first one however has the big problem of making the forward pass of the Conv2d layer much slower, because it requires many more non-Fibonacci matrix products. It is therefore very unlikely that per-out-channel quantization is actually viable in the context of Fibonacci-encoding.

For both Linear and Conv2d layers, the bias has to be an int32 number (signed) so it always has a zero_point of 0. Because it is added to the product of something scaled by scale_x (input scale) and of something scaled by scale_w (weight scale), it has to be scaled as:

$$scale\_b = scale\_x * scale\_w \tag{3}$$

2) Input

The input to the model starts by passing through an Initial-quantization operation, with a scale and zero_point computed on the whole training set. Using the maximum and minimum values of the input batch itself is not possible, since the scale and zero_point need to be precomputed and stored in the quantized model so that post-quantization can happen. The scale and zero-point thus need to be constant over the input batches. Two different approaches were tried for low_val and high_val: choose the minimum and the maximum over the whole training set, or choose the minimum and the maximum averaged over the samples. For example, with 2 2x1 input samples [-5, 5] and [-6, 4], the first method would use -6 and 5 as low_val and high_val, while the second method would use -5.5 and 4.5 as low_val and high_val. There is a trade-off between the two methods. The second one may cause some number to be quantized outside of the range they are supposed to be in, and thus to be clamped to 0 or 255. The first method, however, will be very sensitive to outliers, and thus quantize most of the data to a very small interval, meaning it will in most cases reduce the entropy of the quantized data. We will denote them later (in the Experiments and results section) as min / max stats and avg_min / avg_max stats.

Another way to do that would be to use something in-between, like the 0.01 and 0.99 quantiles, to be resilient to outliers while only having a few clamped values. Implementing that is however not easy as it is impossible to store all of the values given as input to each layer, and thus would require a streaming algorithm with strong memory constraints that would compute an approximation of the desired quantiles.

The code has a method to print the number and proportion of clamped values at each layer so that we get an idea of how often the values are quantized outisde of the range they are supposed to be in.

The values can also get clamped for other reasons:

- Rounding errors
- The fact that the stats are computed on the train set and not on the test set (that effect should be minor if the train set is large enough compared to the test set).

- The loss of precision due to separating the combined scale into a multiplier and a binary shift (explained in the Post-quantization section)

B. Post-quantization

Post-quantization consists of descaling the activation back to its original scale (by multiplying it by scale_x * scale_w) and to scale it for the next quantized layer (by dividing it by scale_x_next). The zero_point of the next layer (zero_point_next) is also added there. Theoretically, scaling the activation for the next quantized layer should be done when entering said layer, but in practice, due to the fact we work with integer numbers, going to a narrow range of numbers (something like {0, 1, ..., 10}) before going back to the full uint8 range, means that a lot of information will be lost. This was also tried and ended up either by using float numbers or by suffering a huge accuracy loss.

Therefore we combine these operations together, with something called the combined_scale in the code.

$$combined\_scale = \frac{scale\_x * scale\_w}{scale\_x\_next} \tag{4}$$

In the following, we have scale_x = 0.02439 as before, scale_w = 0.045 (arbitrarily chosen for the example) and scale_x_next = 0.36 (idem), meaning that combined_scale = 0.0305. In order to avoid doing floating point multiplications during post-quantization, we actually transform the multiplication by combined_scale into a multiplication by an integer number (called mult) and a binary shifting to the right (division by 2^{shift}). This operation is summarized in figure 4.

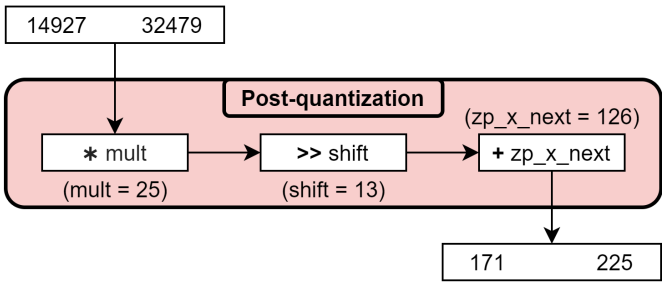

Figure 4. Post-quantization layer

As we can see, mult = 25, shift = 13, and as expected $13/2^{13} = 0.00305 = combined\_scale$. These values are found by the **get_mult_shift** function in the code, which minimizes the loss of precision when doing so.

Scaling back to the original range is not necessary for the last layer, since it will not change the decision process for classification (that just takes the maximum value), and it is therefore not included. For other tasks, this could create problems and it could be necessary to scale back at the end. Also, since we do not use any post-quantization after the last layer, the output is scaled differently than before quantization. The loss value therefore does not make any more sense (it goes to huge numbers even when the accuracy does not drop so much), and the value '-1' is displayed instead when testing the quantized network.

C. Quantized layer operations

1) Linear

Now that we have quantized our input values and weight matrix into unsigned integer numbers, we can smartly separate the operations in order to have only multiplications of positive numbers (this is the only way to make Fibonacci-coding work afterwards). You can see in the formulas below that part 1, part 2, part 3 and part 4 are all computed with products of only positive numbers. Note that in these formulas, zp_w and zp_x are actually matrices, of the same size as W and x respectively, filled with the zp_w value and the zp_x value respectively.

$$x \bullet W + b = scale\_x * (q\_x - zp\_x) \bullet scale\_w * (q\_w - zp\_w) + q\_b * scale\_b$$

$$\frac{x \bullet W + b}{scale\_b} = (q\_x - zp\_x) \bullet (q\_w - zp\_w) + q\_b$$

$$\frac{x \bullet W + b}{scale\_b} = q\_x \bullet q\_w - q\_x \bullet zp\_w - zp\_x \bullet q\_w + zp\_x \bullet zp\_w + q\_b$$

$$\frac{x \bullet W + b}{scale\_b} = part1 - part2 - part3 + part4 + q\_b$$

Even though we have 4 vector-matrix products instead of 1, the whole operation can now be made faster. Parts 3 and 4 do not really count in the computation time, since they are constant with respect to the input, and can thus be precomputed. We do that with:

```
def precompute_constants(qmodel, dummy_datapoint):
```

This method still needs a datapoint as parameter, but only uses it's shape (its value does not change the result of the precomputation). It makes the quantized model compute the constant part and store its value.

Part 1 will be made with Fibonacci-code integer values, making it fast on a specialized hardware. Part 2 is very fast as well for Linear layers, since each entry of part 2 is actually just the sum of the entries of the input, multiplied by the zero-point of the weight matrix. So there is only one multiplication to do. Fibonacci-encoding the zero-point of the weight matrix has been tried, but since it induces a big bias in the computation, it made the accuracy drop too much to be a viable option. Another idea which was not tested was change the quantization of the weight matrix so that it always chooses a Fibonacci code as the zero-point, even if that forces us to choose a different scale that makes the quantized data not cover the whole uint8 range.

As we can see on the figure, the part 3 is computed with the zero-point-filled input and the Fibonacci-encoded weight matrix. Since the computation speed of this operation does not matter (it is precomputed), we could have used the non-encoded weight matrix instead, hoping for a smaller loss of accuracy. This would require to always keep two versions of the weight matrix (both quantized, one Fibonacci-encoded and another one non-encoded).

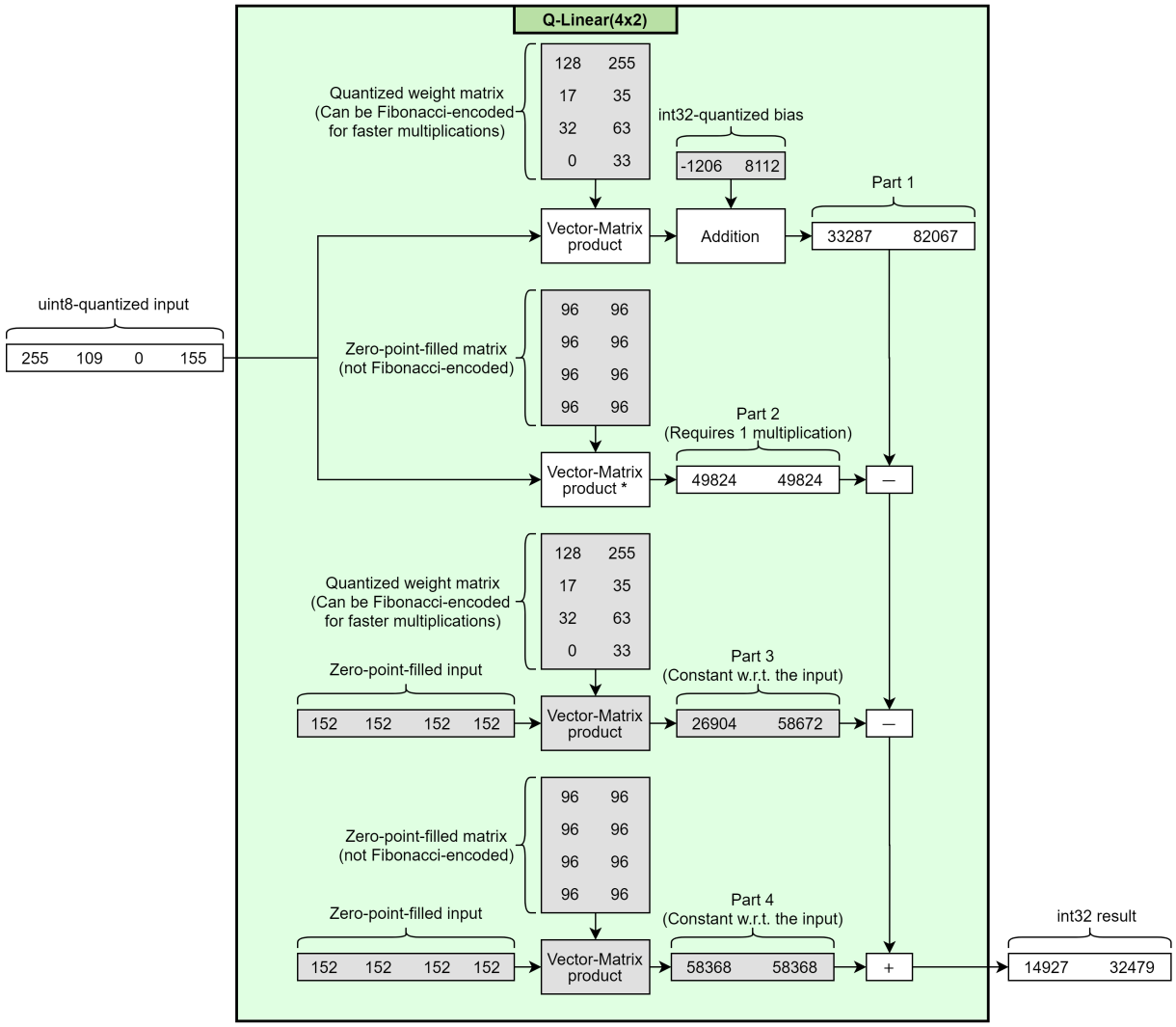

Figure 5. Quantized Linear layer

2) Conv2d

Conv2d layers work very similarly to Linear layers: the computation is also split between part 1, part 2, part 3 and part 4. However now the part 2 computation cannot be simplified as just a sum and a single multiplication. We have to multiply each input channel by the same zero_point-filled kernel. In order to do that with Pytorch, we have to duplicate these kernels, so we end up with in_channels kernels used for part 2. When we tried per_out_channel quantization, we even had $in\_channels * out\_channels$ kernels used for part 2. Since those were not Fibonacci-encoded, it meant that the computation would take even longer than originally, and the idea was abandoned.

Since all weights in the kernels used for part 2 are the same, we tried to emulate the computation by a sum pooling, and a multiplication of the result by the single value present in the kernels (the zero_point_w). This did not lead to any speed improvement (it even had a small opposite effect) in Pytorch, but it should definitely be possible to take advantage of the special structure of the computation of part 2 in Conv2d layers to make it even faster.

3) Activation functions

Since we scale the activation already since the output of the previous (Conv2d or Linear) layer, any activation function that lies in between will be given as input a value that is not scaled as usual. This means that we can only use activation functions f such that the following holds:

$$\forall scale \geq 0, f(x * scale) = scale * f(x) \tag{5}$$

This is the case with for example ReLU, leaky ReLU and parametric ReLU, but it is not the case with for example Tanh and Sigmoid. Note that leaky ReLU and parametric ReLU were not tested and are only supposed to work. ReLU, Sigmoid and Tanh were tested though, giving, as expected, the following results on the MNIST classification task. The model used is the one of figure 7, where ReLU is replaced with the appropriate activation function (ReLU, Sigmoid or Tanh).

Activation function	ReLU	Sigmoid	Tanh
Original accuracy (%)	99.10	96.58	98.89
Int-quantized accuracy (%)	99.10	19.65	10.28

Table I
ACTIVATION FUNCTIONS

4) Pooling layers

For some reason, many output values of the last post-quantization of MPARN-18 were negative and thus clamped to 0. Because of that, the average pooling layer which was supposed to get a majority of negative values and therefore to output a majority of negative values, could only compute its averages with positive and 0 values, making its output very far from what it was supposed to be. Changing it to maximum pooling completely solved this issue, because with a kernel of size 4x4, the maxiumum value is very likely to be positive, even if the majority of the input is negative. This means that in most cases, the maximum pooling does not create any problem with our quantization scheme. On the other hand, if for some reason the input has many negative values, average pooling and possibly other pooling layers can lead to a complete failure of the quantization. We thus only used maximum pooling when making experiments later.

5) BatchNorm2d

After each Conv2d layer, the quantizer automatically searches for any following BatchNorm2d module. If one is found, its operations are included in the combined scale of post quantization, and in a bias, called **bn_bias** in the code. The module itself is then disabled during inference, because it is already handled by post-quantization. Note that because BatchNorm2d works per-channel, the combined scale is now different for each output channel of the Conv2d layer.

4. Fibonacci Coding

A. Encoding

The methods implemented to compute Fibonacci codes are the following:

```
def fib_code_int(num, bits=8):
def fib_code_int_down(num, bits=8):
def fib_code_int_up(num, bits=8):
```

fib_code_int starts by clamping the 'num' value to the range allowed by the 'bits' value. For example {0, 1, ..., 255} for 8-bits. It then computes the closest upper and lower Fibonacci codes (using the 2nd and 3rd methods), and returns the one closest to 'num'.
fib_code_int_down and **fib_code_int_up** loop over the bits of 'num' to make it a Fibonacci code. This is probably far from being the most efficient algorithm to do that, but it works, and if the encoding speed starts to be a real problem (even though it does not affect the inference speed), one could precompute a table associating the closest Fibonacci code to any number between in $\{0, 1, ..., 255\}$.

It has to be noted that only the multiplicative weights (and not the input values) are Fibonacci encoded. The bias values are only additive so Fibonacci-encoding them would not make the operation any faster.

B. Statistics

Methods were implemented to compute statistics about the Fibonacci-encoding of the quantized model:

```
def is_fib(num, bits=8):
def is_fib_tensor(x, bits=8):
def proportion_fib(x, bits=8):
def fib_distances(x, bits=8):
def average_proportion_fib(qmodel, weighted=False):
def average_distance_fib(qmodel, weighted=False):
```

While the first 4 are straight-forward, the 5th and the 6th deserve special attention. **average_proportion_fib** is used to compute the proportion of Fibonacci-code weights in a given quantized model, averaged over the layers. If the parameter 'weighted' is set to False, it considers that each layer has an equal contribution to the average, as in a typical non-weighted average. This means that a Conv2d layer containing for example 9 weights will contribute as much as a Linear layer containing millions of weights, and thus it must be used with care. If 'weighted' is set to True, we compute the true proportion of Fibonacci-code weights in the quantized model. The same reasoning is used for the **average_distance_fib** method which computes the average distance to Fibonacci codes, averaged over the layers. To have the true average in the whole quantized model, we must set 'weighted' to True.

C. Fine-tuning of qmax

During 8-bits integer quantization, qmax is set to 255. This number has only bits of value 1, and is thus just 1 unit smaller than the Fibonacci code 256 (which is outside of the allowed range of numbers). That means that any number ≥ 170 (the largest Fibonacci code in the allowed range of numbers) will be quantized to 170. An interesting idea was to set qmax to $(255 - 170)/2$ for the weight matrices, so that the largest int-quantized weights are as close to 170 as they are to 256, thus making their Fibonacci-encoding to 170 more natural. Doing that improved the accuracy by a lot (from 25% to 38% accuracy for one-shot Fibonacci encoding of CIFAR-10 LeNet model, before retraining). Note that this result of 38% was obtained early in the project, and is not as good as the result of the same experience presented in the Result section (41.79%). Other values of qmax were tried, seemingly giving slightly better results for values even smaller, but it would require more testing to have confidence in these results.

D. Strategies

1) Layer encoding

The main idea used here is the incremental network quantization[2] (INQ). It means that, after quantization, we iteratively Fibonacci-encode a fraction of the weights, then retrain the neural network, and then repeat until we have 100% Fibonacci weights. During retraining, the weights that are already Fibonacci codes are frozen, and therefore cannot be changed by retraining. This is implemented with a redefinition of Pytorch's SGD, originally used on the Pytorch's version of INQ[7]. It associates to each weight matrix another matrix made of zeros and ones, that is element-wise multiplied by the gradient of the loss w.r.t. the weights at each optimizer step. Freezing a weight sets the corresponding value in the matrix to 0, meaning the gradient with respect to this weight will always be multiplied by 0, and the weight will not be able to move anymore. Originally, all of these values are ones (no weight is frozen). In order to be able to retrain the neural network after it has been quantized and partly Fibonacci-encoded, we need to turn it back to its original unscaled version, doing the inverse operation of the initial weight quantization (subtracting to the weight matrix its the zero-point value and multiplying it by its scale). In the code we always keep the unscaled model and the quantized model, so that we can easily go back and forth between both, while updating them with the modifications they made (the unscaled model retrains and thus modifies the non-frozen weights, while the quantized model Fibonacci-encodes some weights, modifying them and freezing them). Obviously the scale and the zero-point associated with each weight tensor has to be constant throughout this operation, and thus cannot be recomputed. Otherwise, numbers that originally were Fibonacci codes would become regular integers by being scaled differently.

At first thought, the idea was to Fibonacci-encode layers one by one. However, we tried Fibonacci-encoding only certain layers to see how it would impact the loss of accuracy. The results are shown in the following figure.

Layer	Fib encoded										
Conv2d #1	No	Yes	No	Yes	No	No	No	No	Yes	Yes	Yes
Conv2d #2	No	No	Yes	Yes	No	No	No	No	Yes	Yes	Yes
Linear #1	No	No	No	No	Yes	No	No	Yes	Yes	Yes	Yes
Linear #2	No	No	No	No	No	Yes	No	Yes	No	Yes	Yes
Linear #3	No	No	No	No	No	No	Yes	Yes	No	No	Yes
Test accuracy	**62.480**	**58.58**	**51.29**	**49.34**	**57.42**	**61.08**	**61.16**	**54.74**	**43.44**	**42.48**	**41.5**
Remark	Int-quantized			all conv Fib.				all linear Fib.			all Fib.

Figure 6. Layer-specific Fibonacci-coding

From this table we can see that the loss of accuracy from the fully Fibonacci-encoded model (62.48 - 41.5 = 20.98) seems very close to the sum of the losses of accuracy from the models where only 1 layer is quantized ((62.48 - 58.58) + (62.48 - 51.29) + (62.48 - 57.42) + (62.48 - 61.08) + (62.48 - 61.16) = 22.51). This result goes against the idea of Fibonacci-encoding layers one by one, because the main point of this idea was that we thought the loss of accuracy due to quantizing each layer would be multiplicative, which is clearly not the case.

2) One-shot

This is the baseline strategy: it Fibonacci-encodes every weight at once, and then retrains the model (with all weights fixed since they are already Fibonacci codes, so only the bias part of the Conv2d and Linear layers and the BatchNorms are retrained). Note that with this strategy we do not use the 'incremental' idea from INQ. In order to use the one-shot strategy, you need to use the following parameters in the code:

```
' iterative_steps ' :  [1.0],
```

You also need to fill the 'strategy' parameter with any of the 3 following strategies ('random', 'quantile' or 'reverse_quantile'). It does not matter which one you choose since all of the weights will be quantized at once anyway.

3) Random

This strategy is the most straight-forward way to implement INQ: it randomly chooses a proportion of the weights to be Fibonacci-encoded, fixes them, retrains everything (that is not fixed yet). This is repeated until every weight is a Fibonacci code. In order to use the random strategy, you need to use the following parameters in the code:

```
' strategy ' :  ' random',
' iterative_steps ' :  [0.2,  0.4,  0.6,  0.8,  1.0],
```

The iterative_steps list given here is just an example that evenly divides the job to do into 5 steps. Reducing the step size towards the end could have better results, but it was not tested.

4) Quantile

In this strategy we use the fact that some weights are already Fibonacci codes by chance (about 21% of the numbers between 0 and 255 are Fibonacci codes, so we can expect a similar percentage of the weights to be Fibonacci codes before any Fibonacci-encoding has been done). For the other weights, some are closer than others to Fibonacci codes. We thus tried the intuitive idea of first fixing the weights that are Fibonacci codes, or that are the closest to Fibonacci codes, then retraining, then repeating until all of the weights are Fibonacci codes. To encode, say, 40% of the weights as Fibonacci codes, we compute the distance matrix of all weights with the Fibonacci code the closest to them. We then compute the 0.4-quantile of these distance, and we encode all weights whose distance to a Fibonacci code is less than or equal to that quantile. That way we have at least 40% of the weights that are fixed to Fibonacci codes. Since we use the $\leq$ operation, and since many weights can share the exact distance value of the quantile, it is frequent that this strategy encodes more numbers than required (by a little bit). In order to use the quantile strategy, you need to use the following parameters in the code:

```
' strategy ' :  ' quantile ' ,
' iterative_steps ' :  [0.5,  0.8,  0.9,  0.95,  1.0],
```

One again, the iterative_steps list given here is just an example, however for this strategy, many different step sizes have been tried, and reducing it towards the end seems to be the best way. Even though this strategy is intuitive, it actually often gives the worst results (see Experiments and result section below). This can be explained in the following way: the first steps Fibonacci-encode the weights that are already very close to Fibonacci codes. The model and the accuracy thus do not change a lot at the beginning. It is also at this moment that most of the weights are not fixed yet and are free to change during retraining. However since the accuracy does not drop by a lot, the retraining becomes almost useless. In the last few steps though, the remaining weights to quantize are very far away from Fibonacci codes (retraining does not bring them closer to Fibonacci codes, or at least not enough). The accuracy thus drops by a lot.

At these steps, most of the weights are fixed, and retraining, while now necessary, becomes much harder. This is why we use reducing step sizes, and also what motivated the next strategy.

5) Reverse-quantile

In this strategy, we do the opposite as in the quantile strategy. This time to encode 40% of the weights as Fibonacci codes, we compute the distance matrix as before, but the quantile to be computed is the 0.6-quantile (so that 40% of the weights have a distance to a Fibonacci code more than or equal to that quantile). We then encode and fix the weights whose distance to a Fibonacci code is more than or equal to the quantile. In order to use the reverse-quantile strategy, you need to use the following parameters in the code:

```
'strategy': 'reverse-quantile',
'iterative_steps': [0.1, 0.3, 0.5, 0.7, 1.0],
```

Once again, the iterative_steps list can be changed, but a generally good idea is to have the proportion low at the beginning and increasing towards the end. As before, a little bit more weights than required can be encoded and fixed at a given step (due to the $\geq$ operation), but this time, the numbers that were originally Fibonacci codes by chance will stay the same, but will not be fixed, and may thus change during retraining and become non-Fibonacci codes. By computing the proportion of Fibonacci codes on the model, we can thus have a lot more of them than we would expect (for example: 10% as required + 1% because multiple weights have the distance value of the quantile + 21% by chance = 32% after step 1). While this strategy seems completely counter-intuitive, it makes most of the changes to the model at the first few steps, while retraining is still possible (because only a few weights are fixed) and when almost all weights are fixed, only small changes are to be made to weights. In practice, it also seems to give better results than the other strategies most of the time.

5. A few quantized models

This project has been implemented in Pytorch and tested on computer vision tasks: image classification of the MNIST (black and white handwritten digit images) and CIFAR-10 (RGB images of cats, dogs, airplanes, etc...) datasets. The original model used for MNIST is a convolutional neural network (CNN), provided as a Pytorch example[8]. It is made of 2 convolutional layers and 2 linear layers. It reached 99.21% accuracy. For CIFAR-10, we first tried a very simple **LeNet**[9], made of 2 convolutional layers and 3 linear layers. Given that its accuracy (before any quantization) was pretty bad (63.5%), we also tried a deeper model, **Pre-Activation ResNet 18**[10]. For simplicity, we transformed it into a sequential model (removing its residual connections), and we turned the average pooling layer into a max pooling layer (easier to quantize). We will call it **MPARN-18** (for Modified Pre-Activation ResNet 18). With all these modifications, it reached 87.46% of accuracy (before quantization). This is still a lot below what the state-of-the-art achieves, but it remains a decent baseline to compare with the quantized versions of these models. All of the original networks' structures, as well as their quantized counterparts, are shown in figures 7, 8 and 9. The optimizers used for this project are all **torch.optim.SGD**, because some of the code taken from the Pytorch-version of INQ worked only with SGD.

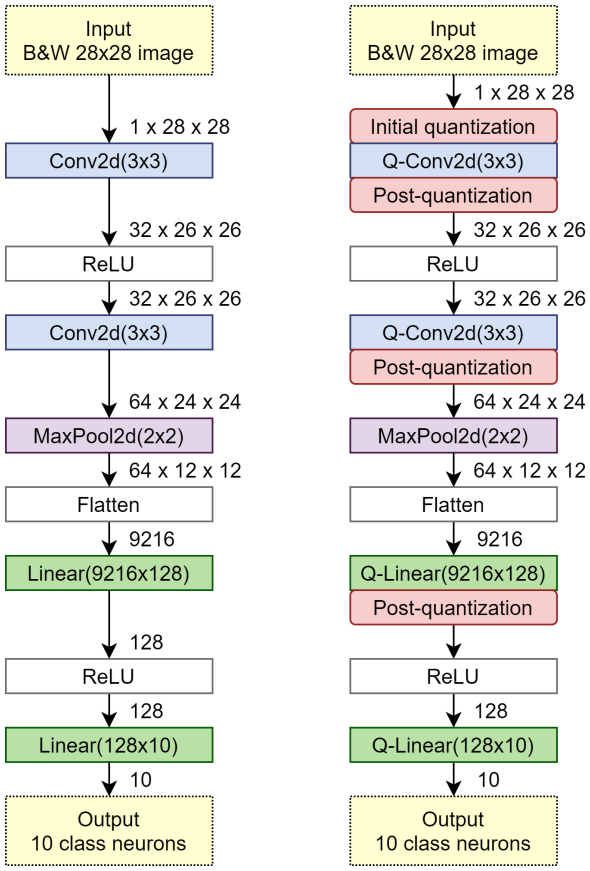

Figure 7. MNIST Net and its quantized version

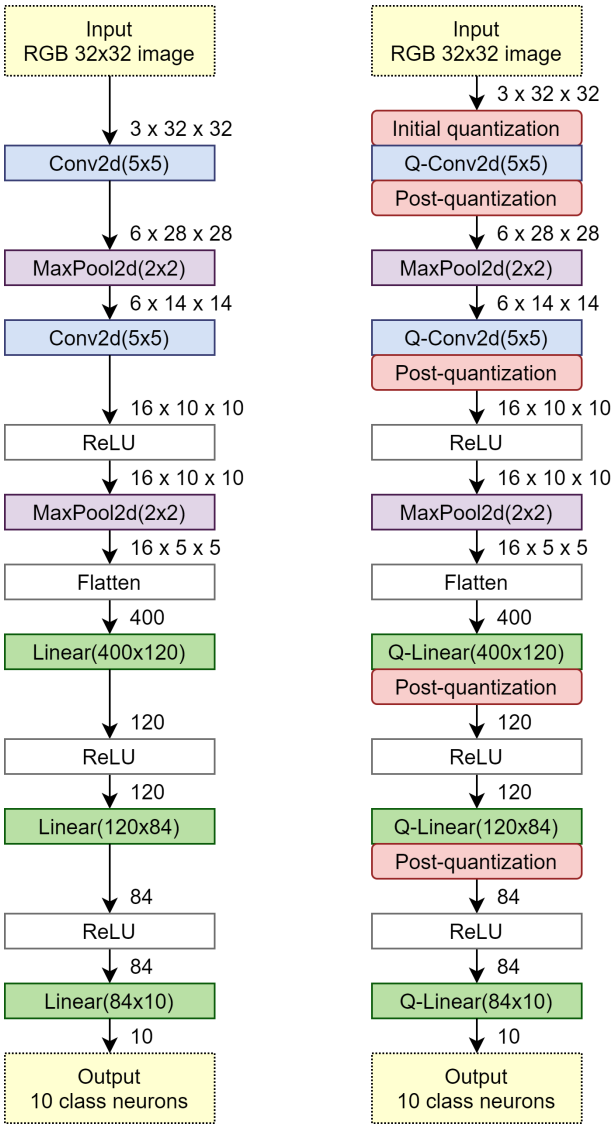

Figure 8. LeNet and its quantized version

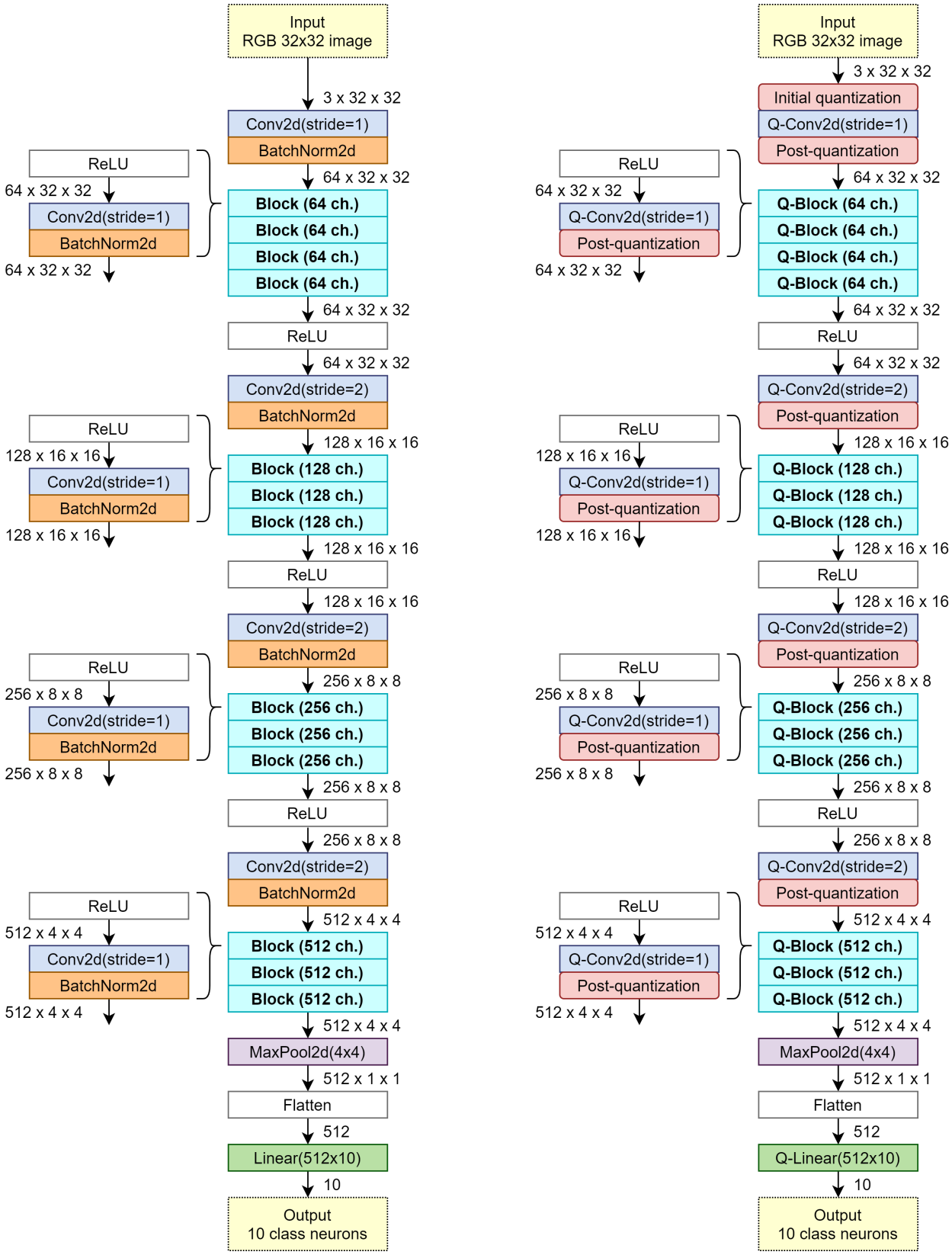

Figure 9. Modified Pre-Activation ResNet 18 and its quantized version

6. Experiments and results

The batch size used for pre-training and re-training was 64 for the MNIST Net and MPARN-18, and it was 128 for the CIFAR-10 LeNet.

The original models were trained for 100 epochs with SGD (lr=0.01, momentum=0.9, weight_decay=0.0005) and a learning rate scheduler that multiplies the learning rate by gamma=0.97 after each epoch.

In this project we are mostly interested in preserving the original accuracy while quantizing the weights as Fibonacci codes. Therefore we didn't focus on reaching state-of-the-art accuracy on the classification tasks. However it has to be noted that before any quantization or Fibonacci-encoding operation, the models already more or less converged, meaning that the retraining is very unlikely to improve the accuracy beyond what the original model could achieve.

Even though the number of bits to store the weights is a parameter of the quantizer, we only made experiments with 8-bits quantization.

The following retraining parameters were used:
1-Retrain: SGD (1 epoch, lr=0.0025, momentum=0.5, weight_decay=0.0005)
4-Retrain: SGD (4 epochs, lr=0.01, momentum=0.5, weight_decay=0.0005), learning rate multiplied by gamma=0.6 after each epoch
16-Retrain: SGD(16 epochs, lr=0.01, momentum=0.5, weight_decay=0.0005), learning rate multiplied by gamma=0.85 after each epoch

The following iterative steps were used for the different strategies:

One-shot: [1.0]
Random (short): [0.2, 0.4, 0.6, 0.8, 1.0]
Random (long): [0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.60, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0]
Quantile (short): [0.5, 0.8, 0.9, 0.95, 1.0]
Quantile (long): [0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.85, 0.9, 0.95, 0.98, 0.99, 0.995, 0.998, 0.999, 0.9995, 0.9998, 0.9999, 1.0]
Reverse-quantile (short): [0.1, 0.3, 0.5, 0.7, 1.0]
Reverse-quantile (long): [0.001, 0.0025, 0.005, 0.01, 0.025, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

For the MNIST Net and the CIFAR-10 MPARN-18, we did not experiment with all the possible settings. For MNIST Net this is because we already reached good results with the fastest methods (short strategies, 4-Retraining). For the MPARN-18 model, this is because we wanted to keep the quantization time acceptable (around 2 hours per experiment with a GeForce GTX 1080 GPU).

In the tables below, we only show the final validation accuracy at the end of each experiment, but the console outputs much more information about when the accuracy drops, and about the importance of retraining. Note that on the console screenshot (figure 10), the loss is manually set at -1 as previously explained, and the inference time for the quantized model is greatly increased by the fact we use fake types (integer values stored as float32) and that we do not use a specialized hardware that takes advantage of the quantization and of the Fibonacci code structure. Therefore this measure is not at all indicative of the real inference time.

```
Quantization epoch 5/18 - Will quantize 70% of the weights as fibonacci
                                Batch         Time   Avg loss       Avg acc     LR
Test 70% fib                    [40/40]       30s    -1.0000        86.370
Test unscaled                   [40/40]       8s     0.5118         87.160
Training [1/4]                  [782/782]     48s    0.0011         100.000     0.01
Training [2/4]                  [782/782]     48s    0.0010         100.000     0.006
Training [3/4]                  [782/782]     48s    0.0010         100.000     0.0036
Training [4/4]                  [782/782]     48s    0.0010         100.000     0.00216
Test 70% fib retrained          [40/40]       29s    -1.0000        86.750

Quantization epoch 6/18 - Will quantize 80% of the weights as fibonacci
                                Batch         Time   Avg loss       Avg acc     LR
Test 80% fib                    [40/40]       29s    -1.0000        84.070
Test unscaled                   [40/40]       8s     0.5372         85.630
Training [1/4]                  [782/782]     48s    0.0015         100.000     0.01
Training [2/4]                  [782/782]     48s    0.0013         100.000     0.006
Training [3/4]                  [782/782]     48s    0.0013         100.000     0.0036
Training [4/4]                  [782/782]     48s    0.0012         100.000     0.00216
Test 80% fib retrained          [40/40]       31s    -1.0000        86.550

Quantization epoch 7/18 - Will quantize 85% of the weights as fibonacci
                                Batch         Time   Avg loss       Avg acc     LR
Test 85% fib                    [40/40]       32s    -1.0000        85.950
Test unscaled                   [40/40]       8s     0.5316         85.870
Training [1/4]                  [782/782]     47s    0.0016         100.000     0.01
Training [2/4]                  [782/782]     48s    0.0013         100.000     0.006
```

Figure 10. Console output

MNIST Net		
Strategy	Validation accuracy (%)	
Original model (no quantization)	99.21	
Using min / max stats		
Int quantization (no retraining)	99.20	
One-shot (no retraining)	96.31	
	4-Retrain	16-Retrain
One-shot		99.04
Random (short)	99.08	
Quantile (short)	99.10	
Reverse-quantile (short)	99.16	
Using avg_min / avg_max stats		
Int quantization (no retraining)	99.20	
One-shot (no retraining)	96.31	
	4-Retrain	16-Retrain
One-shot		99.06
Random (short)	99.18	
Quantile (short)	99.18	
Reverse-quantile (short)	99.18	

Figure 11. Results for the MNIST Net

CIFAR-10 LeNet			
Strategy	Validation accuracy (%)		
Original model (no quantization)	63.50		
Using min / max stats			
Int quantization (no retraining)	63.32		
One-shot (no retraining)	42.56		
	1-Retrain	4-Retrain	16-Retrain
One-shot	45.91	46.90	46.74
Random (short)	55.82	55.13	57.32
Quantile (short)	44.43	51.85	51.85
Reverse-quantile (short)	58.58	61.24	61.75
Random (long)	58.62	55.71	56.01
Quantile (long)	49.85	52.26	52.77
Reverse-quantile (long)	60.22	61.47	61.25
Using avg_min / avg_max stats			
Int quantization (no retraining)	63.11		
One-shot (no retraining)	41.79		
	1-Retrain	4-Retrain	16-Retrain
One-shot	45.25	46.17	46.74
Random (short)	55.54	57.41	53.33
Quantile (short)	44.42	51.44	50.64
Reverse-quantile (short)	58.42	61.01	60.89
Random (long)	57.11	59.53	58.29
Quantile (long)	50.00	52.82	51.13
Reverse-quantile (long)	60.05	60.88	60.40

Figure 12. Results for the CIFAR-10 LeNet

CIFAR-10 Modified Pre-activation ResNet 18		
Strategy	Validation accuracy (%)	
Original model (no quantization)	87.46	
Using min / max stats		
Int quantization (no retraining)	87.21	
One-shot (no retraining)	66.05	
	4-Retrain	16-Retrain
One-shot		85.90
Random (short)		86.19
Quantile (short)		86.03
Reverse-quantile (short)		86.21
Random (long)	86.18	
Quantile (long)	85.84	
Reverse-quantile (long)	86.11	
Using avg_min / avg_max stats		
Int quantization (no retraining)	87.22	
One-shot (no retraining)	69.34	
	4-Retrain	16-Retrain
One-shot		86.32
Random (short)		86.26
Quantile (short)		86.10
Reverse-quantile (short)		86.50
Random (long)	86.04	
Quantile (long)	86.54	
Reverse-quantile (long)	86.71	

Figure 13. Results for the CIFAR-10 MPARN-18

7. Discussion

We can see that for all models, int quantization gives almost no loss of accuracy, while also being very simple. The naive idea of one-shot Fibonacci-encoding without any retraining proves to be pretty bad in terms of accuracy loss (from 99.2 to 96.31 for the MNIST Net for example). The simple addition of retraining the biases and BatchNorm2d modules of the model after doing so already helps recovering most of the accuracy loss (going back from 96.31 to 99.04 for the MNIST Net for example). The more advandced strategies, using the idea of INQ, almost always perform better than one-shot combined with retraining. It is still very hard to come close to the int-quantized accuracy however. The best accuracy that we could achieve on LeNet was 61.75% when using the short reverse-quantile strategy combined with 16 epochs retraining. The loss compared to int-quantized accuracy (63.32%) is still too big to justify the small improvement in inference time due to Fibonacci encoding. On the MNIST Net, all 3 strategies worked well when using the average min / max stats (which prove to be generally a bit better than the min / max stats), making the quantized model go from 99.20% accuracy to 99.18%. In this case, the difference is so small that the use of Fibonacci encoding would be possible. The last model, MPARN-18,

even though much deeper, is able to only lose about 0.5% accuracy between the int-quantized version and the best Fibonacci-encoded version (going from 87.22 to 86.71 with the long reverse-quantile strategy combined with 4 epochs of retraining).

The quantization time clearly is an issue for most real-world applications, and even for more advanced researches on the topic. It obviously varies a lot with the network to encode, with the number of iterative steps and with the number of retraining epochs. On a laptop with a GeForce GTX 1080 GPU / Intel Code i7-8750H CPU, it took from a few minutes for the MNIST Net with a short strategy (5 iterative steps) and 4 retraining epochs, to around 6 hours for the MPARN-18 model with a long strategy (20 iterative steps) and 16 retraining epochs. The strategy we use seems to have a lot of impact on the final accuracy, with reverse-quantile generally performing the best, followed by random, and then quantile. It seems that the number of iterative steps, when it is high enough, has a low impact on the final validation accuracy.

Note that the long version of the Random strategy has 20 steps, while the Quantile and Reverse-quantile strategies only have 18 in their long version. Random thus takes a bit more time.

8. Conclusion and future work

The idea of Fibonacci-encoding a quantized network proved to be feasible with little, yet often non-negligible, loss of accuracy. This means that there is a trade-off between having a lesser precision and having a potentially faster inference. Unfortunately, for the latter, benchmarking the inference times is very complicated without access to the hardware that the quantized models would run onto. Counting the exact number of additions and multiplications to be made could be a step towards this, but it still would not account for many things, such as parallelization and other optimizations. While the results presented above are sometimes not good enough to allow the use of Fibonacci-encoding in a real-world application, it is very likely that they could get improved. Through more researching in this topic, we can probably get even closer to lossless Fibonacci-coding quantization of deep neural networks.

Many improvement are still possible on this topic, and future works could include the following ideas:

- Supporting any model in the form of a **DAG** (Directed Acyclic Graph) instead of just sequential models. This would allow for example to quantize and Fibonacci-encode ResNets (as explained earlier, we had to remove the 'ResNet' property of the Pre-Activation ResNet tested in this project, to make it fully sequential).
- Supporting **other optimizers**, instead of only SGD. Also allowing these optimizers to use multiple parameter groups (some non-essential parts of the code make this currently not possible). It should be pretty straightforward, and is just an implementation issue. It would prove to be necessary for any real-world application of this project.
- Supporting **more modules** (even though some modules, like some activation functions, are much harder than others to quantize, we could work on handling more of them).
- Implementing **Fibonacci-aware retraining**: one could try adding a regularization term bringing each weight closer to a Fibonacci code, or even considering the distance to Fibonacci codes in a custom loss function.
- Increasing **Post-quantization speed**, for example by restricting 'mult' values to Fibonacci codes, and taking advantage of that.
- Trying to recompute statistics of the model, and to change the activation scales (scale_x in the code) and zero-points (zp_x in the code) after retraining.

References

[1] **Project description**

[2] **Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights**

[3] **Fibonacci coding**

[4] **MNIST dataset**

[5] **CIFAR-10 dataset**

[6] **The low-precision paradigm in gemmlowp, and how it's implemented**

[7] **Pytorch implementation of Incremental Network Quantization**

[8] **Pytorch example Net for MNIST**

[9] **Pytorch LeNet model for CIFAR-10**

[10] **Pytorch Pre-Activation ResNet for CIFAR-10**