L46: An Investigation into Stochastic Quantization

Jonah Miller
jm2399@cam.ac.uk

Abstract

In this project, I explore the roulette algorithm behind stochastic quantization (SQ) [1]. This is done by constructing the algorithm which is then applied to multiple low-bit DNN models, including those not tested in the original paper. To increase accuracy, further changes are made to the roulette algorithm so it works at a filter/element rather than layer level. This includes defining and rigorously testing my own probability mechanisms. After, I examine a claim comparing SQ during training to a low-bit DNN approach on a pretrained model. The latter method is then improved with SQ.

1 Introduction

Quantizing a deep neural network (DNN) involves reducing the bit-width a parameter (ie. weight/gradient/activation) in the network can take. Typical DNNs start with 32 bits per parameter - called full-precision parameters. There are multiple approaches to reduce this. FP16 halves the total model size by only allowing 16 bits per parameter. Mixed precision [2] by Nvidia is a combination of 16 and 32 bit parameters with the gradients being modified to prevent underflow/overflow error. There are also more drastic approaches called low-bit DNNs. These aim to reduce parameters (mainly weights) to either 1 bit [3, 4] or 2 bits [5, 6, 7] with either layer-wise or filter/element-wise scaling factors. However, these often suffer large accuracy drops due to the significant loss in data.

Stochastic quantization [1] introduced by Dong et al. proposes an algorithmic improvement for training low-bit DNNs. Their paper builds on the premise that as you train a DNN, some weights will have much larger quantization error than others. When training, it is optimal to quantize the weights with the least quantization error to start, and then progressively quantize the remaining weights. Whilst doing so, it adds in stochasticity. At each iteration, a roulette algorithm assigns each layer a probability that is inversely proportional to its quantization error. The layers are then sampled via this probability, and those selected are fully quantized whilst the others are left at full-precision. A visual demonstration is in Figure 1.

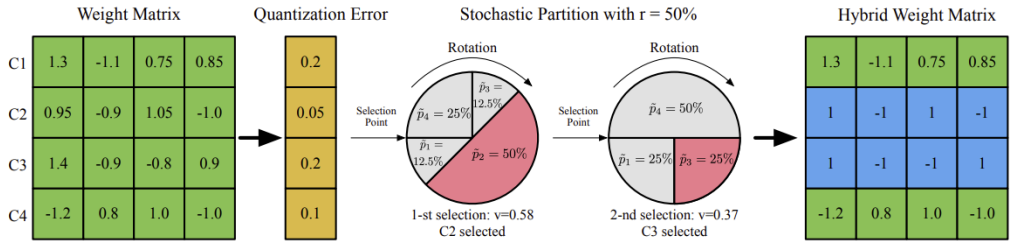

Figure 1: A demonstration of the SQ algorithm. (Figure from [1].)

The advantage of a stochastic algorithm can be seen through the understanding of the randomness in training and developing of a model. Examples of these include: the different images in each mini-batch for training, the random weight initialisation for the initial DNN and the path taken by

the stochastic gradient descent optimizer. The DNN can be thought of as existing in an environment filled with uncertainty. This inhomogeneity is taken into account by running a stochastic algorithm on the quantization error.

2 Low-bit DNN approaches

This section details the low-bit DNN quantization approaches used in this project. Below, n is used to denote the number of filters/elements in layer i. W^t and γ^t denote the full weight set and learning rate respectively at the t-th iteration. Additional variables are described when necessary.

2.1 Binary Weighted Networks (BWN)

BWN [4] quantizes each weight to 1 bit sizes, with filter/element-wise scaling factors. The j-th filter/element in the i-th layer $\mathbf{W}_i^j$ is quantized to $\alpha_i \tilde{\mathbf{W}}_i^j$ where

$$\tilde{\mathbf{W}}_i^j = sign\mathbf{W}_i^j \quad \text{and} \quad \alpha_i = \frac{1}{n}\sum_{j=1}^{n} |\mathbf{W}_i^j|$$

The weights are updated by:

$$W^{t+1} = W^t - \gamma^t \frac{\partial \mathcal{L}}{\partial \tilde{W}^t}$$

2.2 Ternary Weighted Networks (TWN)

TWN [6] quantizes each weight to 2 bit sizes, with filter/element-wise scaling factors. The j-th filter/element in the i-th layer $\mathbf{W}_i^j$ is quantized to $\alpha_i \tilde{\mathbf{W}}_i^j$ where

$$\tilde{\mathbf{W}}_i^j = \begin{cases} +1 & \text{if} \quad \mathbf{W}_i^j > \Delta \\ 0 & \text{if} \quad |\mathbf{W}_i^j| \leq \Delta \\ -1 & \text{if} \quad \mathbf{W}_i^j < -\Delta \end{cases} \qquad \text{and} \quad \alpha_i = \frac{1}{|\mathbf{I}_\Delta|}\sum_{j \in \mathbf{I}_\Delta} |\mathbf{W}_i^j|$$

with $\Delta = \frac{0.7}{n}\sum_{j=1}^{n} |\mathbf{W}_i^j|$ and $\mathbf{I}_\Delta = \{j \text{ st. } |\mathbf{W}_i^j| > \Delta\}$. $|\mathbf{I}_\Delta|$ is the cardinality of $\mathbf{I}_\Delta$. The weights are updated by:

$$W^{t+1} = W^t - \gamma^t \frac{\partial \mathcal{L}}{\partial \tilde{W}^t}$$

2.3 Trained Ternary Quantization (TTQ)

TTQ [5] also quantizes weights to 2 bits. However, instead it uses layer-wise scaling factors, that are learnable parameters. Weights for the i-th layer $\mathbf{W}_i^j$ are quantized to $\tilde{\mathbf{W}}_i^j$ where

$$\tilde{\mathbf{W}}_i^j = \begin{cases} \beta_p & \text{if} \quad \mathbf{W}_i^j > \Delta \\ 0 & \text{if} \quad |\mathbf{W}_i^j| \leq \Delta \\ -\beta_q & \text{if} \quad \mathbf{W}_i^j < -\Delta \end{cases}$$

with $\Delta = \frac{0.7}{n}\sum_{j=1}^{n} |\mathbf{W}_i^j|$ and $\beta_p,\ \beta_q$ are the additional parameters that are learnt at each step and can be thought of as additional weights. The weights are updated by:

$$W^{t+1} = W^t - \gamma^t \frac{\partial \mathcal{L}}{\partial \tilde{W}_\star^t} \quad \text{with} \quad \frac{\partial \mathcal{L}}{\partial \tilde{W}_\star^t} = \begin{cases} \beta_p \dfrac{\partial \mathcal{L}}{\partial \tilde{W}^t} & \text{if} \quad \mathbf{W}_i^j > \Delta \\ \dfrac{\partial \mathcal{L}}{\partial \tilde{W}^t} & \text{if} \quad |\mathbf{W}_i^j| \leq \Delta \\ \beta_q \dfrac{\partial \mathcal{L}}{\partial \tilde{W}^t} & \text{if} \quad \mathbf{W}_i^j < -\Delta \end{cases}$$

2.4 Explicit Loss-Aware Quantization (ELQ)

ELQ [8] turns a pretrained, full-precision model into a 2 bit network, with layer-wise scaling factors. Weights for the j-th filter in the i-th layer $\mathbf{W}_i^j$ are quantized to $\alpha_i \tilde{\mathbf{W}}_i^j$ where

$$\tilde{\mathbf{W}}_i^j = \begin{cases} +1 & \text{if} \quad \mathbf{W}_i^j > 0.5\alpha_i \\ 0 & \text{if} \quad |\mathbf{W}_i^j| \leq 0.5\alpha_i \\ -1 & \text{if} \quad \mathbf{W}_i^j < -0.5_i \end{cases} \qquad \text{and} \quad \alpha_i = mean|\mathbf{W}_i| + 0.05max|\mathbf{W}_i|.$$

The $mean$ and max functions are taken over the layer. The weights are updated by:

$$W^{t+1} = W^t - \gamma^t \frac{\partial \mathcal{L}}{\partial W^t} - 10^{-5} sign(W^t - \tilde{W}^t)$$

Unlike the previous methods, after a weight is quantized it is no longer trainable.

3 Methodology

3.1 Baseline algorithm

The work is built on the baseline algorithm for SQ which can be broken down into two steps, the low-bit DNN quantization and the roulette algorithm.

3.1.1 Low-bit DNN quantization

BWN, TWN and TTQ follow the same algorithmic pattern to train shown in Procedure 1.

Procedure 1 DNN algorithm

Input: Minibatch of inputs and targets $\{\mathbf{X}, \mathbf{Y}\}$; loss function $\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}})$; weights for current iteration W^t; learning rate for current iteration γ^t
Output: Weights for next iteration W^{t+1}

```
for i = 1 to m do
    Calculate quantized weights W_i^j based on DNN
end for
Ŷ = Forward(X, W^t)                                   ▷ Standard forward step
∂L/∂W̃^{t-1} = Backward(∂L/∂Ŷ, W̃)                      ▷ Standard backward step using W̃^t instead of W^t
W^{t+1} = UpdateGradients(W^t, ∂L/∂W̃^t, γ^t)          ▷ Update gradients based on formulas given in 2
```

The ELQ algorithm (shown in 2) instead works on a pretrained model. The weights that are quantized are based on the selection algorithm in Figure 1. The remaining weights are retrained and gradients updated with the rules from section 2.4.

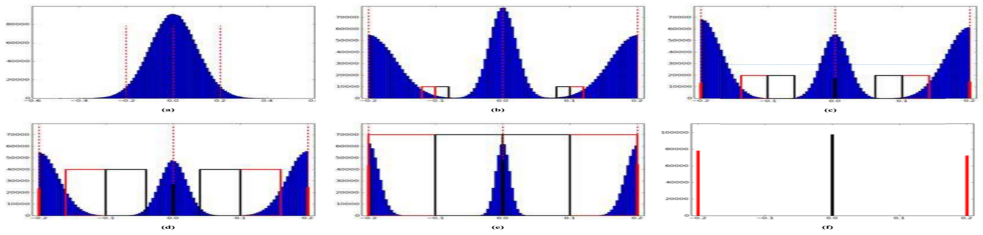

Figure 2: The ELQ algorithm. The dashed red lines are the the $-\alpha_i, 0, \alpha_i$ for that layer. The weights that fall into the boxes are quantized on the iteration (then fixed) whilst the other weights are retrained using a forward-backward step. The boxes increase until all the weights are quantized. (Image from Figure. 1 in [8].)

3.1.2 Roulette algorithm

To add stochasticity, I use the roulette algorithm defined by [1] to choose which weights are quantized in the step after the full quantized weight set for each layer has been calculated.

Procedure 2 Roulette algorithm for stochastic quantization defined in [1]

Input: Weights for current iteration W^t; SQ ratio r^t for current iteration
Output: Quantized weight matrix $\tilde{W}^t$ for current iteration

```
if r^t < 100% then
    Calculate quantization error e and quantization probability p for each layer
    Initialise G_q = ∅, G_r = ∅, N_q = r^t × m        ▷ m is the number of layers in the model
    for i = 1 to N_q do
        Normalise p with p̃ = p / ||p||_1
        Sample a random value v_i uniformly in (0, 1]
        Set s_i = 0 and j = 0                          ▷ s_i accumulates the normalised probability
        while s_i < v_i do
            j = j + 1
            s_i = s_i + p̃_j                            ▷ p̃_j is the j-th element in p̃
        end while
        G_q = G_q ∪ W_j                                ▷ W_j is the j-th weight layer
        p_j = 0                                        ▷ Avoids j-th channel being selected again
    end for
    G_r = W \ G_q
end if
```

3.2 Algorithm improvements

I make an initial alteration to the algorithm to quantize with ratio r over filter/elements in a layer rather than layers in the entire model. I make a following alteration to improve speed - this requires additional testing to maximise accuracy.

3.2.1 Quantization error by filter/element

The motivation is that doing it by layer removes a considerable amount of data the filters have, and thus if the stochasticity is brought down to the filter/element level it will do a better job removing excess quantization error whilst also preserving the randomness in the algorithm. Shown in Figure 3.

Figure 3: A possible occurrence of the input and output matrix using both methods with a 1-bit approach quantizing to either -1 or 1 [3]. My filter/element approach has error $e = \frac{||\mathbf{W}-\tilde{\mathbf{W}}||_1}{||\mathbf{W}||_1}$ of 0.041, whilst the initial approach is nearly double at $e = 0.075$. (Values from Figure. 2 in [1].)

3.2.2 Change of the roulette algorithm

The roulette algorithm is initially performed N_q amount of times. As r grows, N_q gets closer to m which increases the runtime every iteration. Conversely, iterating to $N_r = (1 - r) \times m$ times, the runtime is significantly reduced as the program progresses. This is especially prevalent when run on the number of filter/element in a layer rather than just the number of layers in a model. This method then entails selecting the layers that are *not* quantized. I devise two calculation methods for this. One of my approaches selects proportionally with the error, the other tries to maintain an inverted relationship between selection and error which is done in the initial SQ paper:

1. For each layer calculate the quantization error $e_i = \frac{||\mathbf{W}_i-\tilde{\mathbf{W}}_i||_1}{||\mathbf{W}_i||_1}$ such that $\mathbf{e} = [e_1, ..., e_m]$.

2. Define intermediate value $f_i = e_i$ or $f_i = \frac{1}{e_i} + 10^{-7}$, with 10^{-7} to prevent overflow error.
3. Calculate p_i via one of four possible functions from f_i. Either constant $\left(p_i = \frac{1}{m}\right)$, linear $\left(p_i = \frac{f_i}{\sum_{i=1}^{m} f_i}\right)$, sigmoid $\left(p_i = \frac{1}{1+\exp(-f_i)}\right)$ normalised or softmax $\left(p_i = \frac{\exp(f_i)}{\sum_{i=1}^{m} \exp(f_i)}\right)$.
4. If $f_i = \frac{1}{e_i} + 10^{-7}$ is chosen in step 2 then return $1 - p_i$ instead of p_i.

To change to a filter/element based approach, simply do this process but instead iterate over all the filter/elements in a layer, rather than the layers themself. After, repeat the entire process for each layer. An illustration of the outcome of this is shown in Figure 3.

3.3 Implementation details

I make an implementation of **SQ** in PyTorch [9]. Currently the only implementation is in Caffe[1] [10], however I do not use this as the other projects I integrate with are in PyTorch. My implementation can be located at `https://github.com/JonahMiller/Stochastic-Quantization`. All models whose results are recorded can be found in that repository.

I implement **BWN** and **TWN** based loosely on a TWN PyTorch implementation[2]. I make large-scale changes to improve speed and add functionality for my stochastic quantization mechanisms and their associated parameters. Similarly, I take inspiration from the official **TTQ** implementation[3] with significant modifications to fit into my framework as well as to add additional logic.

Whilst the **ELQ** paper says it will make it's code open-source, I have found no evidence of that. I have written my own code for the project based on the Incremental Network Quantization (INQ) [11] code[4] the ELQ paper claims improves upon. My implementation allows their initial pruning approach, as well as modifications for SQ based methods.

4 Experiments and evaluations

4.1 Models

The tests are run on two model types with different sizes to see how SQ performs in multiple environments.

VGG-9 from VGGNet is a model [12] with many convolutional layers for image recognition. I use an open-source implementation of the code[5]. I use it as it's one of the two main network architectures from the SQ paper. It contains 2,821,322 trainable parameters. There are 10 layers, 1546 filter/elements and the convolutional layers contain either 64, 128 or 256 filters.

ResNet-20 is from the ResNet [13] family introduced in 2015 by He et al. ResNet-56 is used in the original SQ paper, however, ResNet-20 is used here as it's considerably smaller and thus more time efficient, whilst maintaining the same form with Basic Blocks. The code is similarly from an open-source project[6]. It contains 269,722 trainable parameters making it about a tenth of the size of VGG-9. There are 14 layers, 698 filter/elements and the convolutional layers contain either 16, 32 or 64 filters.

Both are trained for 90 epochs, with a SGD solver that has weight decay 10^{-4} and momentum 0.9. The learning rate starts at 0.1 and is divided by 10 every 30 epochs. All of this is similar to what is introduced in the original SQ paper [1], however small changes (such as using less epochs) have been implemented due to time constraints. From the SQ paper I also use the SQ ratio of $[50, 75, 87.5, 100]$ and a linear probability function when using the default SQ method. Both are chosen as they are concluded as the best options in the SQ paper.

[1] `https://github.com/dongyp13/Stochastic-Quantization/`
[2] `https://github.com/buaabai/Ternary-Weights-Network/`
[3] `https://github.com/TropComplique/trained-ternary-quantization/`
[4] `https://github.com/Mxbonn/INQ-pytorch/`
[5] `https://github.com/seongkyun/pytorch-classifications/`
[6] `https://github.com/akamaster/pytorch_resnet_cifar10/`

4.2 Dataset

The training set is CIFAR-10 [14] which consists of 50,000 training images and 10,000 testing images. The training and testing image batch sizes are 100, with 10 image classes. The images are 32×32 and are initially augmented to means $[0.485, 0.456, 0.406]$ and standard deviation $[0.229, 0.224, 0.225]$.

4.3 Results for algorithm reconstruction and TTQ abstraction

The results for the initial algorithm reconstruction are in Figure 4.

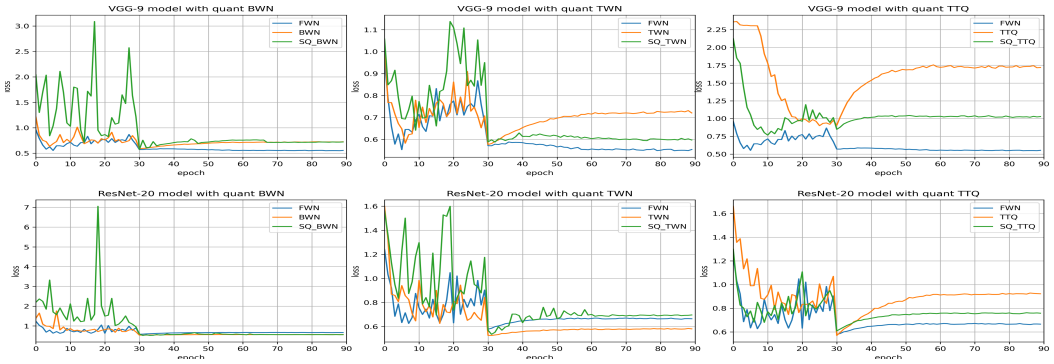

Figure 4: Test losses against epochs for each stochastic low-bit DNN compared with the non-stochastic version, and the full-weighted network (FWN) for each model.

All the models have finished converging seen by the constant loss, thus, it's very unlikely any model would give a better found accuracy if continued for more epochs. For TTQ approaches, β_p, β_q both initialise at 1 and need a moderate amount of learning iterations before they become convergent - hence the higher loss to start. The best accuracy reported over all the epochs was recorded:

	FWN	BWN	SQ BWN	TWN	SQ TWN	TTQ	SQ TTQ
VGG-9	87.76	83.72	85.42	84.72	86.38	76.88	81.21
ResNet-20	85.31	82.70	83.48	83.69	84.24	81.99	83.47

Table 1: Best accuracy using the original SQ algorithm

There is clear improvement from each instance of the SQ algorithm which displays its effectiveness. Moreover, this shows significant increases on TTQ which demonstrates the algorithm can be abstracted to a layered based quantization approach which contains fewer parameters. However, when comparing to the VGG-9 results from their paper [1]:

	FWN	BWN	SQ BWN	TWN	SQ TWN
Their SQ	91.00	89.33	90.87	90.13	91.63
My SQ	87.76	83.72	85.42	84.72	86.38

Table 2: VGG-9 comparison with original paper

There is an accuracy decrease in each column including on the FWN. This indicates that a combination of the VGG model, SGD parameters or initial data augmentation are slightly different. However, the SQ algorithm is still working as intended, shown by the gains seen in each case. Furthermore, on my implementation I get a BWN and TWN improvement of $\sim 2\%$ each time whilst theirs gives $\sim 1.7\%$ improvement. As they have slight better initial accuracy this is expected as their mechanism will have less to improve. I continue with my models.

4.4 Results for algorithm changes

I compare the results using my non-selection algorithm both on a layered and filter/element approach. Initially I planned to only do the latter, however the results I found were fascinating in both cases and I'd like to present them. The table below has the best accuracy found when running each algorithm with the selected settings.

	Prob. fn. p_i	VGG-9				ResNet-20			
		SQ BWN		SQ TWN		SQ BWN		SQ TWN	
		layer	fil./el.	layer	fil./el.	layer	fil./el.	layer	fil./el.
Init. SQ Table 1	Linear	85.42	N/A	86.38	N/A	83.48	N/A	84.24	N/A
My SQ w/ $p_i(e_i)$	Constant	84.81	**85.16**	**86.47**	85.67	**83.26**	83.08	**84.47**	83.86
	Linear	85.15	**85.85**	**86.84**	85.93	**83.18**	82.79	84.35	**84.58**
	Sigmoid	**85.38**	85.24	**86.80**	85.89	83.42	**83.95**	83.29	**84.97**
	Softmax	84.72	**85.81**	**85.56**	85.46	**83.56**	83.12	**85.14**	84.44
My SQ w/ $1 - p_i\left(\frac{1}{e_i}\right)$	Constant	85.08	**85.46**	**86.56**	85.76	**83.31**	83.14	**84.47**	83.97
	Linear	84.78	**85.62**	**86.70**	86.26	**83.09**	82.16	83.92	**84.81**
	Sigmoid	**85.80**	85.59	**87.09**	86.18	**84.38**	83.82	**84.23**	83.96
	Softmax	85.41	**85.51**	**87.16**	86.04	82.93	**83.73**	84.27	**84.41**

Table 3: Best accuracy attained with my SQ algorithm. The better method for each probability/element for a particular quantization is in bold.

For the VGG-9 model, a filter/element approach work better than a layered approach on SQ BWN and vice versa on SQ TWN. It's unclear what the best approach is for ResNet-20. Furthermore, there's no consistent mechanism between the probability types that garners the greatest accuracy.

The initial SQ paper tells us [1] each element in a filter interacts with each other to form an inherent structure. Then, in a given filter, each element is quantized to the same value set to preserve this structure. I hypothesise that there is an inherent layer structure for our model, and that the filter/element quantization approach breaks this structure and introduces a large distortion in the SQ TWN case. The SQ BWN has significantly more error in each filter (demonstrated by the variability of the loss in figure 4 compared to TWN), and so the layered structure is already broken and thus improved by the stochastic algorithm. However as the SQ TWN error will be smaller - as each has more bits which preserves accuracy, the structure is only broken when quantized by the filter/element approach. Hence, the layer approach gives larger accuracy.

For ResNet-20 - as the number of filters per layer is closer to the total layer count, combined with the fact there are considerably less parameters - the two approaches are similar. This means there's more variability about whether the layer structure is broken by large quantization errors or my filter/element approach and hence why in neither SQ BWN or SQ TWN there's a clearly better approach.

Each test was only run once due to time constraints. It's valid to question how much of the results found are accurate compared to randomness in the model. However, when the probability function is constant, each e-type should give the same results. Taking the average difference for each quantization type and model as an approximate error bound, I find best accuracies:

- VGG-9 with SQ BWN has best accuracy $85.85 \pm 0.29\%$ - gain of 0.43%.
- VGG-9 with SQ TWN has best accuracy $87.16 \pm 0.09\%$ - gain of 0.78%.
- ResNet-20 with SQ BWN has best accuracy $84.38 \pm 0.06\%$ - gain of 0.90%.
- ResNet-20 with SQ TWN has best accuracy $85.14 \pm 0.06\%$ - gain of 0.90%.

This is an improvement in each model taking into account the error. This demonstrates there is value in both the alternative selection algorithm I've used, and also a potential for the filter/element approach to be viable if further investigation is done into the inherent layer structure. Moreover, the lack of clear pattern on which probability method worked best, justified testing each mechanism.

4.5 Results for ELQ

The SQ paper [1] states: "It is relatively difficult for post-quantization techniques to transfer the local minimum from a full precision weight space to a low bitwdith weight space losslessly". As such, it implies the post-training methods shouldn't be able to attain the same accuracy. At the time of writing this, the main mechanism for quantizing pretrained models was INQ. Since then, the ELQ paper [8] was published which showed significant improvements. I now compare to the results for these with my best SQ TWN that occurred in Table 4.4.

	FWN	Best SQ TWN	INQ	ELQ
VGG-9	87.76	87.16	78.04	83.80
ResNet-20	85.31	85.14	74.90	67.07

Table 4: Best accuracy using different ELQ mechanisms

First note that for ResNet-20 the results are considerably worse. I haven't been able to diagnose why, and continue this section using just VGG-9. The SQ claim seems correct regarding INQ, however for ELQ is is only $\sim 3\%$ lower and so the correct algorithmic/hyperparameter changes may raise it enough to become competitive. Replacing the ELQ quantization selection algorithm from figure 2 with SQ, and comparing with both the layer and filter/element approach over each probability mechanism:

	Constant		Linear		Sigmoid		Softmax	
	layer	fil./el.	layer	fil./el.	layer	fil./el.	layer	fil./el.
SQ with $p_i(e_i)$	85.88	86.47	**86.66**	86.23	86.30	86.19	86.37	86.38
SQ with $1 - p_i\left(\frac{1}{e_i}\right)$	86.29	86.46	86.35	86.06	86.05	86.39	86.01	86.14

Table 5: SQ ELQ on VGG-9. The greatest accuracy in the table is in bold.

The greatest accuracy is 86.66%, 0.5% off the best accuracy found in Table 4.4. It's likely with optimised hyperparameters (ie. SGD settings or epochs), it could match or improve on SQ TWN. Whilst not directly disproving the claim, it displays that it's likely untrue, pending further investigation.

5 Discussion and conclusion

5.1 Model size and training time

Each model needed the same size memory to train as a FWN is used at each step. However once trained there are significant differences. In the 1-bit quantization case the new model is approximately $\frac{1}{32}$ of the size, and for 2-bit the new models are approximately $\frac{1}{16}$ of the size. There's not a significant difference in size from the layered or filter/element base methods for the same bit quantization (difference of 7KB for VGG-9 and 3KB for ResNet-20). In terms of training times, a FWN in either model took about 4 minutes, however, when applying the stochastic BWN/TWN the times ranged between 45 minutes to 3 hours for VGG-9, and 20 to 45 minutes for ResNet-20. Applying a stochastic ELQ the times were closer to 1 minute for VGG-9. Combining with the accuracy results, SQ ELQ is the optimal low-bit DNN method to continue testing on.

5.2 Limitations

There are definitive differences between my models and the original paper such that some of the hyperparameters I used on their recommendation may not be optimal. Further empirical testing should have been done on these including looking at alternative SQ ratios and probability mechanisms for the default SQ case. ResNet-20 also displayed inconsistent results throughout, and additional investigation is needed as to why. Lastly, whilst I touch on reproducibility earlier, each test was ultimately run once and so considerably more testing is needed to draw stronger conclusions.

5.3 Conclusion

This project recreated and adapted the SQ algorithm. It displayed versatility by giving significant accuracy improvements to two layered-based low-bit DNNs - TTQ which is applied whilst training, and ELQ which is applied to the pretrained model. This project also involved modifying the algorithm to work on selecting what not to be quantized - and explored the consequences of this when scaling it to a filter/element based approach. Future improvements may contain combining the filter/element and layered based approach (ie. both approaches at different epochs) to try and preserve (and test existence of) the inherent layer structure whilst reducing excess quantization errors. This could further involve exploring other optimizers and associated hyperparameters to improve overall accuracy. There are also alternative low-bit DNNs I didn't have time to implement. For example, DoReFa-Net [7] has a quantization approach specialising on convolutional networks which may yield fascinating results.

References

[1] Yingpeng Dong et al. "Learning Accurate Low-Bit Deep Neural Networks with Stochastic Quantization". In: *arXiv:1708.01001* (2017).

[2] Paulius Micikevicius et al. "Mixed Precision Training". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=r1gs9JgRZ.

[3] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "BinaryConnect: Training Deep Neural Networks with binary weights during propagations". In: *NIPS* (2015).

[4] Mohammad Rastegari et al. "Xnor-net: Imagenet classification using binary convolutional neural networks." In: *ECCV* (2016).

[5] Chenzhuo Zhu et al. "Trained Ternary Quantization". In: *ICLR* (2017).

[6] Fengfu Li et al. "Ternary Weight Networks". In: *arXiv:1605.04711* (2016).

[7] Shuchang Zhou et al. "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients". In: *CoRR* abs/1606.06160 (2016). arXiv: 1606.06160. URL: http://arxiv.org/abs/1606.06160.

[8] Aojun Zhou et al. "Explicit Loss-Error-Aware Quantization for Low-Bit Deep Neural Networks". In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018).

[9] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[10] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM '14. Orlando, Florida, USA: ACM, 2014, pp. 675–678. ISBN: 978-1-4503-3063-3. DOI: 10.1145/2647868.2654889. URL: http://doi.acm.org/10.1145/2647868.2654889.

[11] Anbang Yao Aojun Zhou et al. "Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights". In: *ICLR* (2017).

[12] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *arxiv:1409.1556* (2014).

[13] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *arXiv:1512.03385* (2015).

[14] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. "CIFAR-10 (Canadian Institute for Advanced Research)". In: (2009). URL: http://www.cs.toronto.edu/~kriz/cifar.html.