Finding Non-Uniform Quantization Schemes using Multi-Task Gaussian Processes

Marcelo Gennari do Nascimento[1 [0000−0003−2025−5601]], Theo W. Costain[1 [0000−0002−7803−6965]], and Victor Adrian Prisacariu[1 [0000−0002−0630−6129]]

Active Vision Lab, University of Oxford, UK
{marcelo,costain,victor}@robots.ox.ac.uk
https://code.active.vision

Abstract. We propose a novel method for neural network quantization that casts the neural architecture search problem as one of hyperparameter search to find non-uniform bit distributions throughout the layers of a CNN. We perform the search assuming a Multi-Task Gaussian Processes prior, which splits the problem to multiple tasks, each corresponding to different number of training epochs, and explore the space by sampling those configurations that yield maximum information. We then show that with significantly lower precision in the last layers we achieve a minimal loss of accuracy with appreciable memory savings. We test our findings on the CIFAR10 and ImageNet datasets using the VGG, ResNet and GoogLeNet architectures.

Keywords: Quantization; Bayesian Optimization; Gaussian Process

1 Introduction

The strategy of quantizing neural networks to achieve fast inference has been a popular method of deploying neural networks in compute constrained environments. Its benefits include significant memory savings, improved computational speed, and a decreased cost in the energy needed per inference. Many methods have used this family of strategies, quantizing down to anywhere between 8-bits and 2-bits, with little loss in accuracy[10,30]. It also bears noting that in most of these methods, after quantizing to very low precisions (1 to 5 bits), retraining is necessary to recover accuracy.

Recently, even though the quantization algorithms have significantly improved, they have almost exclusively *implicitly* assumed that the best strategy is to quantize all the layers uniformly with the same precision. However, there are two main reasons to believe otherwise: i) we argue that as it has been interpreted [17] that different layers extract different levels of features, it follows that different layers might require different levels of precision; ii) the idea of quantization as an approximation to the floating point (FP) version of the network suggests that lower error in the early layers reduces the propagation of errors down the whole network, minimizing any drop in accuracy. We believe that as

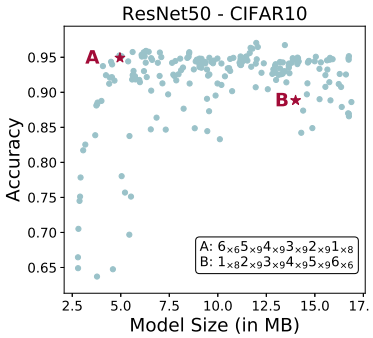

Fig. 1. Gaussian Process prediction for bit distribution in memory vs accuracy plot

important as having a good quantization strategy, is to also have a good strategy for the distribution of bits through the network, thereby eliminating any redundant bits. The goal is then to find a configuration in a search space that uses the least amount of bits and achieves the highest accuracy per bit used.

We cast this Neural Architecture Search (NAS) problem into the framework of hyperparameter search, since the bit-width of each layer should ideally be found automatically. As with many NAS approaches, measuring the accuracy of a single configuration can take a considerable amount of time. To mitigate this issue, we propose a two stage approach. First, we map the full search space into a lower dimensional counterpart through a parameterised constraint function, and second, we use a Multi-task Gaussian Process to predict the accuracy at a higher epoch number from lower epoch numbers. This approach allows us to reduce both the complexity of the search space as well as the time required to determine the accuracy of a given configuration. Finally, as our Gaussian Process based approach is suitable for probabalistic inference, we use Bayesian Optimisation (BO) to explore and search the hyperparameter space of variable bit-size configurations.

For the quantization of the network, we use the DSConv method [16]. It achieves high accuracy without significant retraining, meaning the number of epochs needed for full training, and implicitly, the requirement for prediction power, is minimised.

To summarise, our main contributions are as follows:

1. we cast NAS as hyperparameter search, which we apply to the problem of variable bit-size quantization;
2. we reduce the time needed to measure the accuracy of a proposed bit configuration considerably by using multi-task GPs to infer future accuracy from current estimates;
3. we demonstrate performance across a broad range of configurations, described by Bezier curves and Chebyshev series.

The next sections are as follows: Section 2 shows previous work on quantization and hyperparameter search. Section 3 elaborates on the methodology used

for search, including the constraint, exploration, and sampling procedures. Section 4 shows the results achieved on the CIFAR10 and ImageNet datasets using the networks listed above. Section 5 draws a conclusion and considers insights from the paper.

2 Related Work

Neural Architecture / Hyperparameter Search One can consider finding bit distributions as a form of model selection [18], given its complexity and the limit on the parameters that it accepts as a solution. Previous methods have predominantly used Reinforcement Learning (RL) and Evolutionary Algorithms (EA) to model search, which is referred to in the literature as Neural Architecture Search. Examples include NASNet [34], MNasNet [26], ReLeq-Net [6], HAQ [29], among others [1,31] for RL and [33,13,15,24] for EA. Our work overlaps with these papers only on the goal of finding an optimal strategy given a search space.

ReLeQ-Net and HAQ, to the best of our knowledge, are the only methods whose aims are to find the optimal bit distribution through different layers of a network, and are therefore the papers that overlap the most with our work. It is notable that both of them use an RL based approach to search for optimal bit distributions. However, HAQ is more focused on hardware specific optimization, whereas both ours and ReLeQ-Net's methods attempt to be Hardware-Agnostic. Recently some work involving Bayesian Optimization (BO) for model architecture selection has been carried out, with systems such as NASBOT [11]. One of the reasons why BO has not been used for model selection has to do with how unclear it is to find a measure of "distance" between two models, which is the main problem that was addressed by NASBOT.

Alternatively, one can see determining bit distribution as finding hyperparameters to be tuned given a model, *i.e.* not different from finding the optimal learning rates or weight decays. Historically, this has been tackled by BO techniques. In neural networks specifically, this was popularized after the work of [21], and followed by others [27,2,7,22]. As a result BO can be considered a natural method for searching for optimum bit distribution configurations.

Quantization Quantization strategies can be either trained from scratch or derived from a pretrained network. The methods of [30,10,4,32] initialize their networks from scratch. This ensures that there is no initial bias on the values of the parameters, and they can achieve the minimum difference in accuracy when extremely low bit values are used (1-bit / 2-bits) - a notable exception being DoReFa-Net [32], which reportedly had slightly better results when quantizing the network starting from a pretrained network. The methods of [8,14,16,28] quantize the network starting from a pretrained network. These methods start with a bias on the values of the parameters, which can limit how much they recover from the lost accuracy. A benefit of these methods though is that they can be quickly fine-tuned over a few epochs re-achieving state-of-the-art results. These methods are more interesting to us because of their quick deployment

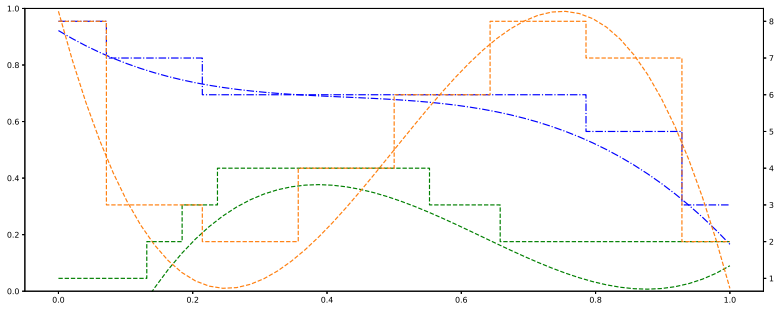

Fig. 2. Three examples of modified Chebyshev functions and their clamped versions. The continuous lines represent the values of the modified Chebyshev functions as function of the layer, which is then converted into bitwidths whose values are represented on the right hand axis. This is for two 8 layer VGG11s and a 20 layer ResNet corresponding to configurations 1)blue: 8,7,6,6,6,6,5,3 2) orange: 8,3,2,4,6,8,7,2 and 3) green: 1,1,1,2,3,4,4,4,4,4,4,3,3,2,2,2,2,2,2,2

cycle. It is worth noting that all of these methods use a uniform distribution of precision, meaning that all layers are quantized to the same number of bits.

3 Method

Our method consists of three parts: constraining, exploring, and sampling the search space. We first constrain the search space by assuming dependence between adjacent bit numbers. We do this by drawing bit distributions from a low-degree Polynomial (in the experiments we use a 2^{nd} degree Bezier curve and a 4^{th} order Chebyshev series). Given a these distributions, we quantize the network using the DSConv[16] method. We explore the space by placing a Gaussian prior over the polynomial parameters, and sampling / retraining a set of hyperparameters that gives the most information about the final payoff function. After exploring, we rank the configurations based on sampling the GP for accuracy, and choose the ones that are the most appropriate for our end-use. Each of these phases will be explained further in this section.

3.1 Constraining the Space

When trying to find the bits, from 1-8, for each layer, the search space will have size of 8^n, where n is the number of layers of the network. For a CNN of 50 layers, the search space will be $2^{150} \approx 10^{45}$, which is a similar size to a game of chess ($\approx 10^{50}$). The size of this space makes an exhaustive search prohibitive.

Our method for constraining the search space relies on the use of parameterised functions. We model a function of degree n with a few parameters, which describe the search space. We then discretise the function, such that a bit configuration of any layered size network can be sampled from a few parameteres alone.

We use two parameterised functions to illustrate our solution:

- We define the Bezier function $\mathbf{B}(x;\mathbf{w}) = \mathbf{w}^T\phi(x)$ for $x \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d,\ 0 \leq x \leq 1,\ 0 \leq w_i \leq 1\ \forall i \in \{i : i \in \mathbb{N}^+, i <= d\}$, where d is the degree of the polynomial. The vector $\phi(x)$ is the feature vector of the Bezier curve *i.e.* for Linear Bezier $\phi(x) = [1-x,\ x]^T$, for Quadratic Bezier $\phi(x) = [(1-x)^2,\ 2(1-x)x,\ x^2]^T$, *etc.*
- We define the modified Chebysehv function $\mathbf{T}(x;\mathbf{w}) = \frac{(\mathbf{w}-0.5)^T\phi_d(x)+1}{2}$ for $x \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d,\ -1 \leq x \leq 1,\ 0 \leq w_i \leq 1\ \forall i \in \{i : i \in \mathbb{N}^+, i <= d\}$, where d is the degree of the polynomial. The vector $\phi_d(x)$ is defined as $[T_0(x), T_1(x), T_2(x), \ldots, T_{d-1}(x)]^T$ where $T_0(x) = 1, T_1(x) = x,$ and $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$

The constraint function, $g(t)$, is then a clamped and rounded version of the chosen polynomial, $p(t)$, such that the bits, b, for each layer generated are between 1 and 8, and $b_i \in \mathbb{N}$. We can define then $g(t) = \lfloor \text{CLAMP}(8p(t)+1), 1, 8) \rceil$, where $\lfloor \cdot \rceil$ is the rounding function, and $\text{CLAMP}(f, a, b) = \min(\max(f, a), b)$.

Fig. 2 shows an example of a Chebyshev function and its clamped version. The y-axis in the left indicate the value of the Chebyshev function for different values of x. This is then clamped, rounded, and scaled such that it transforms into a discontinuous line that represents the bit chosen for each layer of a CNN. The bit value is indicated in the y-axis in the right.

By constraining the search space in this way, the minimization problem then shifts as follows:

$$\begin{array}{ll} \textbf{Naïve Approach} & \\ \min\limits_{\mathbf{b}} & \mathcal{L}(t;\mathbf{b}), \mathbf{b} \in \mathbb{N}^n \\ \text{s.t.} & 1 \leq b_i \leq 8,\ \forall i \in \{1, n\} \end{array} \qquad \begin{array}{ll} \textbf{Our Approach} & \\ \min\limits_{\mathbf{w}} & \mathcal{L}(t;\mathbf{w}), \mathbf{w} \in \mathbb{R}^d \\ \text{s.t.} & b_i = g(\frac{i}{n}),\ \forall i \in \{1, n\} \\ & 0 \leq w_j \leq 1,\ \forall j \in \{1, d\} \end{array} \tag{1}$$

where $\mathcal{L}$ is the loss function (to be introduced in Section 3.3).

The search then reduces to finding the parameters of the polynomial basis $\mathbf{w}$, which consequently define the bit distributions throughout the layers. The search space is then continuous and compatible with GPs, and significantly reduced to only d dimensions. Using this parameterisation, we are able to easily define a distance metric between configurations to be used when calculating the kernel function and predictive distribution from our Gaussian Process. So, with this setup, the search space can be sufficiently explored in a timely manner.

Quantization Strategy The method used for quantizing the CNN is DSConv. This is because our aim is to minimize time taken during training, and DSConv has consistently shown good accuracy properties in models, even before retrained.

In this method, both the activations and the weights are quantized, such that fast inference is possible. Each of the weight tensors are divided into blocks of size $B = 32$ depthwise. Each block holds B integers and one FP32 multiplier value.

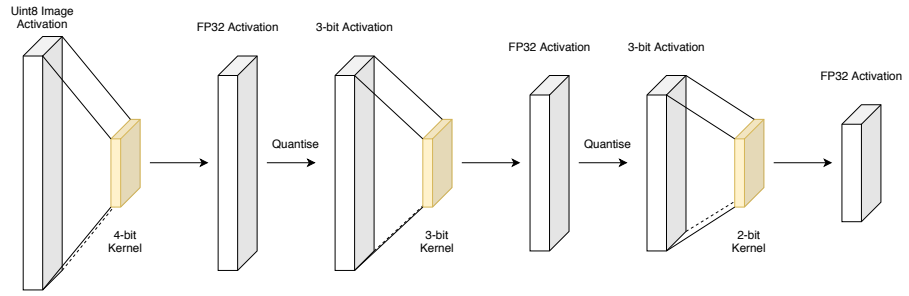

Fig. 3. Quantization given variable bit-widths. Notice that the input is the image, which is a `uint8` tensor (normalization can be dumped into a KDS tensor [16]), so it is not quantized. The quantization of activations is done before the convolution such that the convolution can be done using the same precision.

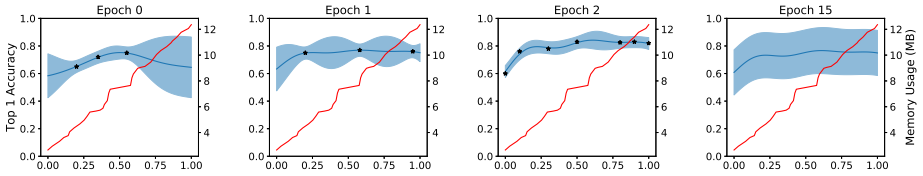

Fig. 4. Multi-Task Gaussian Process for inferring accuracy of quantized network. The quantization function is a Bezier Linear with the first parameter set to 0.5 *i.e.* $g(t; w_1) = 0.5 + t(w_1 - 0.5)$. For all figures, the x-axis is the value of w_0, the left y-axis is the accuracy on CIFAR10 of a toy CNN with 10 layers. The right y-axis (red line) shows the model size for a given value of w_0. The epoch correspondences for each task is [0, 1, 2, 15] respectively. After this exploration phase, the decision procedure is run on the predictive distribution of Task 4.

The integers are found by simply scaling each of the block from the original FP32 weight tensor by $\frac{2^{b-1}}{\text{MAX}(w)}$, and then flooring and cropping to range. The FP32 value is calculated by simply minimizing the L2 norm of the block with respect to the original corresponding block: $\xi = \frac{\sum_{i=0}^{B-1} w_i w_{qi}}{\sum_{i=0}^{B-1} w_{qi}^2}$. The activation tensor is quantized similarly, but using Block Floating Point (BFP) format in each of the blocks instead.

In order to take advantage of the low bit multiplication speed, the activation tensor and the weight tensor need to have the same precision. Figure 3 shows how this is done. The activation tensor prior to a convolution layer is set to be quantized to the same bit precision as that layer. The first convolution is not quantized since the input image is already in `uint8` format. Also note that we quantize only the convolutional layers. The Fully Connected layers are all left in the original FP32 precision for training.

3.2 Exploring the Space

Next, we need a way of exploring the space in order to learn the accuracy of the network given a limited set of $\mathbf{w}$ points. We propose a Multi-Task Gaussian Process prior in the neural network, such that each task corresponds to the estimation of the accuracy of the quantized network given $\mathbf{w}$ after a certain number of epochs, *e.g.* task 1 corresponds to 0 epochs, task 2 to 1 epoch, task 3 to 2 epochs, task 4 to 15 epochs. Let there be m tasks, and a prior on f_l, $l \in \{0, m\}$, such that $f_l \sim \mathcal{GP}(\mu(t), k(t, t'))$. We also place a probability distribution $\mathcal{P}(f_0, f_1, ...f_m)$ over different tasks. Let $y_{(t,l)} = f_l(t) + \epsilon(t)$ be the observation at hyperparameter value t for task l, and let $\epsilon(t) \sim \mathcal{N}(0, \sigma^2; t)$ be the observation noise, which is normally distributed. This defines independent Gaussian Likelihoods $y_{(t,l)} \sim \mathcal{N}(f_l, \sigma^2; t)$. From this model, observations $\mathbf{y}$ are drawn, such that $\mathbf{y} = (y_{11}, ..., y_{s1}..., y_{12}, ..., y_{s2}, ..., y_{1m}, ..., y_{sm})$, where y_{il} is the i^{th} observation of the l^{th} task [23].

We used the The Intrinsic Correlation Model (ICM) of [5] and [3] for kernel calculation (in our experiments we made use of the squared exponential kernel). We can then define the mean and the correlation between tasks as:

$$\begin{aligned} \langle f_l(x) \rangle &= \mu_l(x) \\ \mathbb{C}(f_l(x), f_{l'}(x')) &= k^f(l, l')k^x(x, x') \end{aligned} \tag{2}$$

where k^f and k^x are positive semi definite functions, corresponding to the correlation between functions and the correlation between inputs respectively. From this it follows that the covariance is $K = K^f \otimes K^x$, where $\otimes$ is the Kronecker product, K^f is the matrix of correlations between the functions and K^x is the matrix of correlations between the inputs. For a new set of data points x_*, the mean prediction can then be calculated using the normal formula for the predictive distribution:

$$\begin{aligned} \overline{\mathbf{f}}(x_*) &= \mu_l(x_*) + (K^{x_*})^T \Sigma^{-1} \mathbf{y} \\ \Sigma &= K + D \otimes I \end{aligned} \tag{3}$$

where D is an $m \times m$ diagonal matrix where the $(l, l)^{th}$ term is σ_l^2.

Figure 4 shows an example of the Multi-Task setting with a 1D Bezier Curve for ease of visualization. Each plot shows the predictive mean and variance for each epoch after 14 data points have been collected, using the exploration algorithm explained in section 3.2. The idea is to predict what is the distribution of the last task given inputs in earlier tasks.

Exploration Phase In order to make decisions on what parameters to choose, we need to explore the space to predict the accuracy of the last task. The exploration phase for the multitask Gaussian Process follows the Low-Fidelity Search from [23]. The idea is to find the values of x, l such that it gives us maximal information $\mathbb{I}(y_{(x,l)}; f_m |\ \mathbf{y}) = \mathbb{H}(y_{(x,l)} \mid \mathbf{y}) - \mathbb{H}(y_{(x,l)} \mid \mathbf{y}, f_m)$, where $\mathbf{y}$ is the observation history, and (x, l) is the action to be performed. It is important to weight the information by a measure of the cost that it takes to perform that operation. So the exploration procedure chooses x, l that maximizes $\mathbb{I}(y_{(x,l)}; f_m |\ \mathbf{y})$ per unit

cost. This means that the parameter that has the most information about the payoff function will be picked.

Depending on the dataset and model chosen, the user can favour exploration on one fidelity over the other by decreasing the cost λ of running that particular task. Additionally, we set up a budget on the amount of time in unit cost or number of architectures that we are willing to explore. The Exploration Phase finishes when the Budget has been fully used. After this is finished, the user can run their preferred method of ranking configurations using the posterior of the trained GP.

3.3 Sampling the Space

The naïve goal is to find the highest accuracy per bit possible, which corresponds to finding the minimum of the loss function $\mathcal{L}(t; \mathbf{w}) = -\frac{y_{(t,m)}}{\sum_{i=1}^{n} b_i}$. However, there is a trade-off that must be considered. A model, *e.g.* ResNet20, using a total of 40 bits and achieving 80% accuracy (ratio of 2%/bit) is arguably worse than a model that uses 43 bits and achieves 85% accuracy (ratio of 1.97%/bit). The goal is instead to find a decision procedure that takes into account the regret of not using more bits based on a set of constraints. This relationship should be linear instead of inversely proportional. A better strategy is to assume that using 4-bits for all layers is the lowest uniform quantization scheme without loss of accuracy. Each bit used less than this should be a reward, and each bit used more than this should be a penalty, this is added (or subtracted) to the accuracy to get an "effective accuracy". We then define the effective accuracy as $\mathcal{E}(a, \mathbf{b}, n) = a - \frac{\sum_{i=1}^{n} b_i - 4n}{k}$, where a is the accuracy of the original network, and k is a constant of penalty per bit. Therefore, for $k = 100$ each bit used in addition to the average of 4-bits incurs a penalty of 1% in the effective accuracy. The reverse incurs a reward of 1% in the effective accuracy. The decision procedure becomes then to minimize the negative effective accuracy, $\mathcal{L} = -\mathcal{E}(a, \mathbf{b}, n)$. Once we have enough information about the GPs, we can rank configurations based on their loss in order to pick the most relevant for us.

4 Experiments and Results

We tested our method in a variety of configurations, using versions of the original VGG, ResNet, and GoogLeNet models, altered in order to take CIFAR10, and ImageNet32 as input. For training CIFAR10 and ImageNet32, we used data augmentation by cropping 32x32 image of the 4-pixel padded original. We used a SGD optimiser with momentum of 0.9, and weight decay of 5×10^{-5}. The learning rate started as 10^{-1}, and was divided by 10 after 150 and 250 epochs.

We ran the exploration procedure on ~ 65 configurations for each network using the multi-task algorithm outlined above. From these configurations, we could then use the mean of the gaussian to draw estimates of the accuracy of many different quantization schemes. Using the decision outlined above, we sorted the results by either accuracy, memory, or computational complexity, and

selected the points of interest for better visualization and intuition of what the general trend of the found configurations are.

Results on Accuracy using the CIFAR10 Dataset Results on CIFAR10 and ablation tests are displayed in Table 1. The configurations are color coded for clarity, with red representing higher bit counts and green representing lower bit counts. These configurations were selected based on the decision procedure outlined above, using the Bezier Linear polynomials.

For comparison, we show 6 configurations of each network: the first and third configurations were picked by our decision procedure; the second and fourth are simply the inverse order of the first and third configurations; the fourth and fifth rows use the traditional uniform distribution of bits for a fair comparison.

It is important to note that the decision to pick these configurations are based on the estimate of the GP rather than on the actual Top-1 results. In order to compare fairly, we also included the Top-1 score and standard deviation from 10 runs after properly training each of them for an additional 30 epochs using the same hyperparameters and optimiser that were used to train their FP32 version. We have also included a delta column which shows the difference between the Top-1 estimate from the GP and the Top-1 after fine-tuning the network. It is remarkable that most of the error in estimation is within 1%, which shows how the GP was able to generalize and interpolate properly as expected.

It can be seen that in general, using more bits in earlier layers yields more accurate, and lighter configurations. The higher accuracy can be explained numerically, since higher bits are used in earlier layers, the error propagation through the network is smaller. The lower memory usage is due to the fact that later layers have a higher number of channels, and therefore using lower precision in those layers yield a massive difference in memory need. For VGG16, the first configuration is both lighter, faster, and more accurate than using 3-bits for all layers. This pattern is repeated for the deeper VGG19 too, where the first configuration yielded superior results to the constant 3-bits for all layers, and also for ResNet18 as well. This "rule of thumb" is somewhat weaker in the GoogLeNet architecture though, even though there is still a clear correlation.

Results on Accuracy using Chebyshev Series In order to test robustness of the method in relation to the choice of prior functions, we chose to use a Chebyshev Series of fourth degree, which has a larger search space than the Bezier Linear model. We have tested the model using the CIFAR10 dataset as well, and the results are shown in Table 2.

As it can be seen, the 4^{th} degree introduced more flexibility as to what bit configurations the method is capable of finding. We found that with higher degree of polynomials, the number of architectures to search should also increase. In our experiments, we have searched for $\sim$ 150 configurations before finding good results. The table shows the expected result that more bits at the beginning compensate for the fewer bits at the end of the network. The ResNet-18 result resembles the configuration found in Table 4, even though it found a configuration that has more usage of 3-bits, but performs slightly worse. As also

Table 1. Results for many configurations on CIFAR10. VGG16 and VGG19 correspond to the architectures introduced in [20]. ResNet18 is the architecture from [9], and the GoogLeNet architecture is from [25]. The Configuration refers to the bit value for each layer of a given model, from earlier layers in the left to later layers in the right. They are color coded for clarity: red for higher bits and green for lower bits. It is important to note that we quantize only the convolutional layers, which means that VGG16 has 13 values, VGG19 has 16 values, ResNet18 has 20 values, and GoogLeNet has 64. Because of its size, the GoogLeNet values were represented by a subscript indicating the number of times that a given bit-width is used. The column "Delta" refers to the difference between the GP estimation of the Top1 accuracy and the actual mean Top1 accuracy ($n = 10$) after properly retraining that particular configuration.

CNN	Configuration (bits per layer)	Top 1 Estimate from GP	Mean Top1	Std	Delta	# Bits	Memory (in MB)
VGG16	32-bit Floating Point	-	93.7%	-	-	-	58.8
	6 555 44 333 22 11	(95.5%)	**93.7%**	0.2%	-1.8%	50	**4.84**
	11 22 333 44 555 6	(91.3%)	**87.7%**	0.2%	-3.6%	50	**9.50**
	7 66 55 44 33 222 1	(92.1%)	93.7%	0.1%	1.6%	50	5.26
	1 222 33 44 55 66 7	(90.1%)	91.5%	0.4%	1.4%	50	10.75
	4444444444444	(93.3%)	93.8%	0.1%	0.5%	52	8.28
	3333333333333	(92.9%)	93.5%	0.2%	0.6%	39	6.44
VGG19	32-bit Floating Point	-	93.9%	-	-	-	80.1
	6 555 4444 333 222 11	(94.4%)	**93.7%**	0.1%	-0.7%	54	**6.95**
	11 222 333 4444 555 6	(91.6%)	**89.6%**	0.4%	-2.0%	54	**12.04**
	5 4444 33333 2222 11	(93.9%)	93.5%	0.1%	-0.4%	46	6.14
	11 2222 33333 4444 5	(90.3%)	88.4%	1.2%	-0.9%	46	10.05
	3333333333333333	(92.9%)	93.4%	0.2%	0.5%	48	8.76
	2222222222222222	(92.1%)	92.2%	0.2%	0.1%	32	6.25
ResNet18	32-bit Floating Point	-	95.4%	-	-	-	44.6
	666 5555 4444 3333 2222 1	(96.3%)	**95.4%**	0.1%	-0.9%	75	**3.72**
	1 2222 3333 4444 5555 666	(95.9%)	**92.9%**	0.3%	-3.0%	75	**8.00**
	44444444 3333333333 22	(95.3%)	95.3%	0.1%	0.0%	60	4.34
	22 3333333333 44444444	(94.5%)	94.2%	0.2%	-0.3%	66	6.08
	33333333333333333333	(94.4%)	95.0%	0.1%	0.6%	60	4.90
	22222222222222222222	(93.1%)	93.3%	0.5%	0.2%	40	3.49
GoogLeNet	32-bit Floating Point	-	95.5%	-	-	-	24.32
	$4_{\times 21}$ $3_{\times 27}$ $2_{\times 16}$	(94.7%)	**95.3%**	0.1%	0.6%	207	**2.35**
	$2_{\times 16}$ $3_{\times 27}$ $4_{\times 21}$	(94.6%)	**94.2%**	0.1%	-0.4%	207	**2.98**
	$6_{\times 8}$ $5_{\times 13}$ $4_{\times 12}$ $3_{\times 13}$ $2_{\times 13}$ $1_{\times 5}$	(95.8%)	95.3%	0.1%	-0.5%	231	2.65
	$1_{\times 5}$ $2_{\times 13}$ $3_{\times 13}$ $4_{\times 12}$ $5_{\times 14}$ $6_{\times 8}$	(94.7%)	90.5%	0.2%	-4.2%	231	3.73
	$3_{\times 64}$	(94.7%)	95.1%	0.1%	0.4%	192	2.68
	$2_{\times 64}$	(93.4%)	93.5%	0.2%	0.1%	127	1.92

expected, when the bit distribution is inverted in the network, it results in both higher memory and lower accuracy.

The same behaviour is found with the VGGs, with the slight difference that as VGG11 is too shallow, it requires more bits to recover the accuracy. VGG16 is considerably deeper, and therefore our algorithm was able to compress it more significantly.

This results shows that our method can be used with a variety of basis. It is worth bearing in mind that the GP processing capability requires the inversion of a matrix, which is proportional to the degree of the polynomial chosen. Therefore our method will only work in a timely manner when using fewer hyperparameters to describe the function.

Table 2. Results of method when using Chebyshev Polynomials of 4^{th}degree.

Method	Network	Bitwidths	Accuracy Loss	Memory (as a % of original)
Ours	ResNet18	6 4 33 222 33333333 22	-0.6%	8.0%
		22 33333333 222 33 4 6	-1.2%	11.5%
Ours	VGG11	7 666 7 6 5 4	-0.1%	16.8%
		4 5 6 7 666 7	-0.5%	19.7%
Ours	VGG16	4 3 222 3333 22 11	-0.8%	6.3%
		11 22 3333 222 3 4	-2.4%	8.3%

Results on Network Size Figure 5 shows the result of the GP-estimated accuracy of different configurations by their model size. The solid purple line links the uniform configurations, starting with all 1s and finishing with all 6s. Therefore, any point that lies above that line is an interesting point, since it gives better accuracy by using the same amount of memory of its uniform counterpart. We have highlighted a number of different interesting configurations with red stars and labelled them from A-M in order to better visualise what each point represent.

As it can be seen in the figure, the choice of bit-usage throughout the network plays an important role in both the accuracy and the memory usage. Even though there is a clear trend that links model size and accuracy, there are a handful of configurations which can perform well on both fronts. It can be seen that, in general, points that are above the purple line are linearly decreasing with bit-usage whereas the ones that are below the purple line are linearly increasing with bit-usage.

The surprising result is that, in the CIFAR10 experiments, even though using uniformly 1-bit for all layers achieves bad results, by just introducing a couple of bits in the first three quarters of the network (such as in points A, C and F), the memory increase is almost negligible, but the accuracy recovery is significant. Adding bits at the end of the network however, achieves the opposite effect. It

can also be noticed that points A, C, E, and F, achieve better accuracy than the uniformly 2s configuration whilst using 50% less memory. This is even more evident in point E, in which we used up to 6 bits in the first layers, but still achieved less memory usage due to the usage of 1-bit in the bigger kernels at the end of the network.

In the ImageNet32 experiments, we also see some improvement, albeit less dramatic than the CIFAR10 experiments. The overall massage is still the same, as it can be seen in points H, J and L, for which adding bits in the first layers has achieved good accuracy with small memory increases. It is still noteworthy that even with a dataset as challenging as ImageNet32, due to its substantial decrease of information when compared to the default ImageNet, the GP could find good configurations without needing more datapoints. This shows that this method can be robust to changes in dataset.

Brief Comparison with ReLeQ One of the other papers that touched in this subject was ReLeQ [6]. As explained in the literature review section, they use a reinforcement learning approach to find optimum bit-distributions over the network. Whilst their quantization methodology varies greatly from that used in this paper, it is worth comparing their results to ours. Their results for the CIFAR10 dataset in two of the networks are shown in Table 3. It can be seen that we achieve similar results for ResNet, though with different mean bits. Since ReLeQ's method does not use the same constraint as our method, it could find more varied solutions. This is a limitation to our method which allows it to find solutions to the network faster whilst using less computational power, but reduces the freedom of choice.

Table 3. Comparison of our accuracy results with ReLeQ's method [6] on CIFAR10. The authors in [6] did not provide their models size in memory, so we estimated using both our and the original author's quantisation scheme to make a fair comparison.

Method	Network	Bitwidths	Accuracy Loss	Model size (MB)	
				DSConv	WRPNx1
ReLeQ [6]	ResNet-20	8 22 3 222 3 2 333 222 3 2222 8	0.12%	3.88	3.25
Ours	ResNet-20	666 5555 4444 3333 2222 1	0.1%	3.54	2.91
ReLeQ [6]	VGG-11	8 5 8 5 6666 8	0.17%	6.86	6.61
Ours	VGG-11	777 666 55	0.14%	6.35	5.42
ReLeQ [6]	VGG-16	888 6 8 6 8 6 8 6 8 6 8 6 8 6 88	0.1%	13.32	12.54
Ours	VGG16	6 555 44 333 22 11	0.1%	4.62	3.74

Results on ImageNet using ResNet For completeness, we have included some of the results found by our algorithm on the more challenging ImageNet dataset [19]. This was trained using an Adam Optimizer [12], with learning rate of 10^{-5}.

Table 4 shows the results. As expected, the same pattern of decreasing precision downstream holds across datasets. Comparing these results with the results

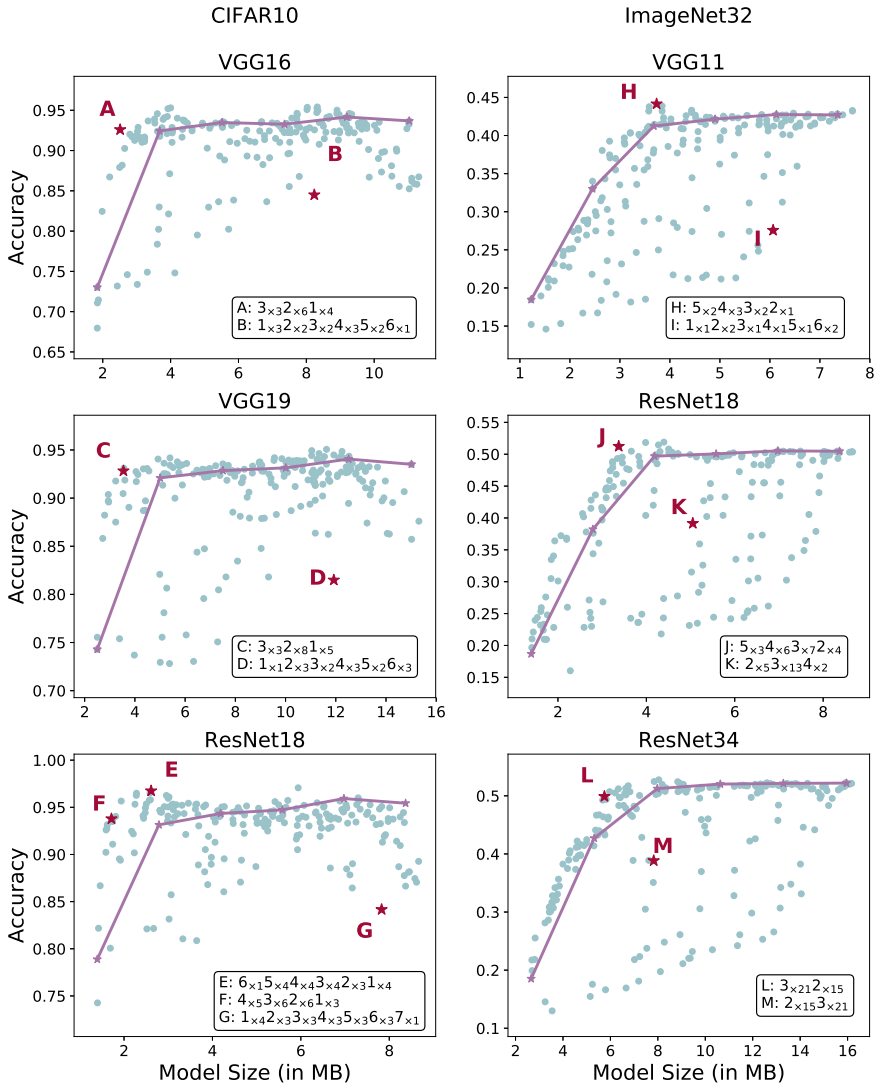

Fig. 5. Scatter plot of the effect on accuracy versus model size of different bit configurations. The left three plots use the CIFAR10 dataset and the right three plots use the ImageNet32 dataset. Note that this is the plot of the estimate as given by the trained GP, and not the actual accuracy given proper training. The solid line refers to the uniform configurations, starting with all 1s and ending with all 6s. Points A-M highlight different configurations as shown in the text boxes. The string of numbers shown refers to the bit size on each layer of the given network. Note that VGG11 has 8 convolutional layers, and therefore points A and B have only 8 numbers. This applies to VGG16 (13 layers), VGG19 (16 layers), ResNet18 (20 layers), and ResNet34 (36 layers) as well.

from DSConv [16], we can see that a decreasing bit-width throughout the architecture, decreasing from 6 bits to 2, is superior to the "all 4s" and "all 3s".

Table 4. Results of our method using the ImageNet dataset with the ResNet architecture.

Method	# of Layers	Bitwidths	Acc. Loss	Size (MB)
Ours	18	6×3 5×5 4×5 3×5 2×2	0.2%	**4.89**
DSConv [16]	18	4	0.0%	5.88
DSConv [16]	18	3	0.8%	4.55
Ours	50	6×6 5×15 4×14 3×15 2×3	0.6%	**11.89**
DSConv [16]	50	4	0.0%	14.54
DSConv [16]	50	3	0.9%	11.74
HAQ [29]	50	*flexible*	0.0%	12.14

As with ReLeQ's method, HAQ's method has a weaker constraint on bit distribution, which means it would be able to find configurations that our method would not;However, even with our very strong constraint, we were still able to find configurations that are competitive in memory requirements to those found by HAQ. This shows the strength of the conclusion that later layers require lower precision than earlier layers to maintain the same accuracy.

5 Conclusion

In this paper, we demonstrate that a uniform distribution over bit-widths throughout a CNN is likely not the most efficient way to quantize a neural network. In order to demonstrate this, we used a Multi-Task Gaussian Process prior over different training epochs, and a Bayesian Optimization exploration procedure based on Information Maximization that estimated the accuracy of different configurations.

We have observed that starting a CNN with higher bit-widths and decreasing precision in later layers yield better accuracy and better memory usage than the traditional uniformly distributed bit-width. This can be interpreted either numerically (as in less error being propagate down the network), or can be interpreted as the functionality of each layer in the network (earlier layers are concerned with feature extraction and later layers are concerned with classification).

Acknowledgements

This research was supported by Intel and the EPSRC, and we thank our colleagues from the Programmable Solutions Group who greatly assisted in this work.

References

1. Baker, B., Gupta, O., Naik, N., Raskar, R.: Designing neural network architectures using reinforcement learning. arXiv preprint arXiv:1611.02167 (2016)
2. Bergstra, J., Yamins, D., Cox, D.D.: Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In: Proceedings of the 30th International Conference on Machine Learning. vol. 28, pp. I–115. JMLR.org (2013)
3. Bonilla, E.V., Chai, K.M., Williams, C.: Multi-task gaussian process prediction. In: Advances in neural information processing systems. pp. 153–160 (2008)
4. Cai, Z., He, X., Sun, J., Vasconcelos, N.: Deep learning with low precision by half-wave gaussian quantization. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (Jul 2017). https://doi.org/10.1109/cvpr.2017.574
5. Chai, K.M.: Multi-task learning with gaussian processes. Ph.D. thesis, The University of Edinburgh (2010)
6. Elthakeb, A.T., Pilligundla, P., Yazdanbakhsh, A., Kinzer, S., Esmaeilzadeh, H.: Releq: A reinforcement learning approach for deep quantization of neural networks. arXiv preprint arXiv:1811.01704 (2018)
7. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Advances in Neural Information Processing Systems 28. pp. 2962–2970 (2015)
8. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149 (2015)
9. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)
10. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: Training neural networks with low precision weights and activations. The Journal of Machine Learning Research **18**(1), 6869–6898 (2017)
11. Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., Xing, E.P.: Neural architecture search with bayesian optimisation and optimal transport. In: Advances in Neural Information Processing Systems. pp. 2016–2025 (2018)
12. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
13. Kitano, H.: Designing neural networks using genetic algorithms with graph generation system. Complex systems **4**(4), 461–476 (1990)
14. Lin, X., Zhao, C., Pan, W.: Towards accurate binary convolutional neural network. In: Advances in Neural Information Processing Systems 30. pp. 345–353 (2017)
15. Liu, H., Simonyan, K., Vinyals, O., Fernando, C., Kavukcuoglu, K.: Hierarchical representations for efficient architecture search. arXiv preprint arXiv:1711.00436 (2017)
16. Nascimento, M.G.d., Fawcett, R., Prisacariu, V.A.: Dsconv: Efficient convolution operator. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 5148–5157 (2019)
17. Olah, C., Mordvintsev, A., Schubert, L.: Feature visualization. Distill (2017). https://doi.org/10.23915/distill.00007, https://distill.pub/2017/feature-visualization
18. Rasmussen, C.E., Williams, C.K.I.: Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning). The MIT Press (2005)

19. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV) **115**(3), 211–252 (2015). https://doi.org/10.1007/s11263-015-0816-y
20. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
21. Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. In: Advances in neural information processing systems. pp. 2951–2959 (2012)
22. Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M., Prabhat, M., Adams, R.: Scalable bayesian optimization using deep neural networks. In: International conference on machine learning. pp. 2171–2180 (2015)
23. Song, J., Chen, Y., Yue, Y.: A general framework for multi-fidelity bayesian optimization with gaussian processes. In: The 22nd International Conference on Artificial Intelligence and Statistics. pp. 3158–3167 (2019)
24. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary computation **10**(2), 99–127 (2002)
25. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 1–9 (2015)
26. Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., Le, Q.V.: Mnasnet: Platform-aware neural architecture search for mobile. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 2820–2828 (2019)
27. Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 847–855. KDD '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2487575.2487629, `http://doi.acm.org/10.1145/2487575.2487629`
28. Vanhoucke, V., Senior, A., Mao, M.Z.: Improving the speed of neural networks on cpus. In: in Deep Learning and Unsupervised Feature Learning Workshop, NIPS. Citeseer (2011)
29. Wang, K., Liu, Z., Lin, Y., Lin, J., Han, S.: Haq: Hardware-aware automated quantization with mixed precision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 8612–8620 (2019)
30. Zhang, D., Yang, J., Ye, D., Hua, G.: Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In: Proceedings of the European Conference on Computer Vision (ECCV). pp. 365–382 (2018)
31. Zhong, Z., Yan, J., Wu, W., Shao, J., Liu, C.L.: Practical block-wise neural network architecture generation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 2423–2432 (2018)
32. Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., Zou, Y.: Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv preprint arXiv:1606.06160 (2016)
33. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016)
34. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 8697–8710 (2018)