# Training with Quantization Noise for Extreme Model Compression

**Angela Fan** [*]
Facebook AI Research, LORIA

**Pierre Stock** [*†]
Facebook AI Research, Inria

**Benjamin Graham**
Facebook AI Research

**Edouard Grave**
Facebook AI Research

**Rémi Gribonval** [†]
Inria

**Hervé Jégou**
Facebook AI Research

**Armand Joulin**
Facebook AI Research

## Abstract

We tackle the problem of producing compact models, maximizing their accuracy for a given model size. A standard solution is to train networks with Quantization Aware Training [1], where the weights are quantized during training and the gradients approximated with the Straight-Through Estimator [2]. In this paper, we extend this approach to work with extreme compression methods where the approximations introduced by STE are severe. Our proposal is to only quantize a different random subset of weights during each forward, allowing for unbiased gradients to flow through the other weights. Controlling the amount of noise and its form allows for extreme compression rates while maintaining the performance of the original model. As a result we establish new state-of-the-art compromises between accuracy and model size both in natural language processing and image classification. For example, applying our method to state-of-the-art Transformer and ConvNet architectures, we can achieve 82.5% accuracy on MNLI by compressing RoBERTa to 14 MB and 80.0% top-1 accuracy on ImageNet by compressing an EfficientNet-B3 to 3.3 MB.

## 1 Introduction

Many of the best performing neural network architectures in real-world applications have a large number of parameters. For example, the current standard machine translation architecture, Transformer [3], has layers that contain millions of parameters. Even models that are designed to jointly optimize the performance and the parameter efficiency, such as EfficientNets [4], still require dozens to hundreds of megabytes, which limits their applications to domains like robotics or virtual assistants.

Model compression schemes reduce the memory footprint of overparametrized models. Pruning [5] and distillation [6] remove parameters by reducing the number of network weights. In contrast, quantization focuses on reducing the bits per weight. This makes quantization particularly interesting when compressing models that have already been carefully optimized in terms of network architecture. Whereas deleting weights or whole hidden units will inevitably lead to a drop in performance, we demonstrate that quantizing the weights can be performed with little to no loss in accuracy.

Popular postprocessing quantization methods, like scalar quantization, replace the floating-point weights of a trained network by a lower-precision representation, like fixed-width integers [7]. These approaches achieve a good compression rate with the additional benefit of accelerating inference on supporting hardware. However, the errors made by these approximations accumulate in the computations operated during the forward pass, inducing a significant drop in performance [8].

[*]Equal contribution. Corresponding authors: `angelafan@fb.com`, `pstock@fb.com`
[†]Univ Lyon, Inria, CNRS, ENS de Lyon, UCB Lyon 1, LIP UMR 5668, F-69342, Lyon, France
Code available at `https://github.com/pytorch/fairseq/tree/master/examples/quant_noise`

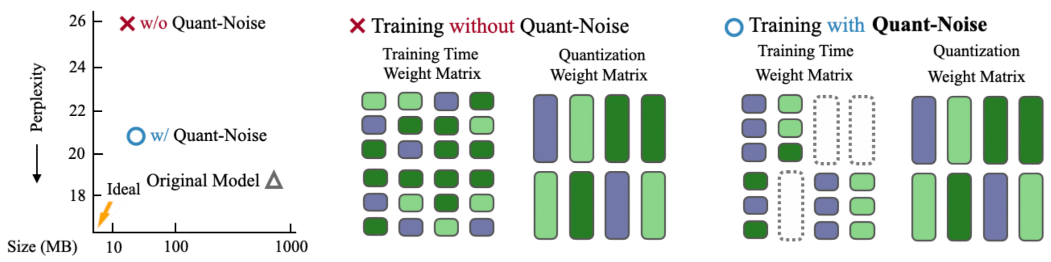

w/o Quant-Noise
w/ Quant-Noise
Ideal
Original Model
Perplexity
26
24
22
20
18
Size (MB)
10
100
1000
Training without Quant-Noise
Training Time Weight Matrix
Quantization Weight Matrix
Training with Quant-Noise
Training Time Weight Matrix
Quantization Weight Matrix

Figure 1: **Quant-Noise** trains models to be resilient to inference-time quantization by mimicking the effect of the quantization method during training time. This allows for extreme compression rates without much loss in accuracy on a variety of tasks and benchmarks.

A solution to address this drifting effect is to directly quantize the network during training. This raises two challenges. First, the discretization operators have a null gradient — the derivative with respect to the input is zero almost everywhere. This requires special workarounds to train a network with these operators. The second challenge that often comes with these workarounds is the discrepancy that appears between the train and test functions implemented by the network. Quantization Aware Training (QAT) [1] resolves these issues by quantizing all the weights during the forward and using a straight through estimator (STE) [2] to compute the gradient. This works when the error introduced by STE is small, like with `int8` fixed-point quantization, but does not suffice in compression regimes where the approximation made by the compression is more severe.

In this work, we show that quantizing only a subset of weights instead of the entire network during training is more stable for high compression schemes. Indeed, by quantizing only a random fraction of the network at each forward, most the weights are updated with unbiased gradients. Interestingly, we show that our method can employ a simpler quantization scheme during the training. This is particularly useful for quantizers with trainable parameters, such as Product Quantizer (PQ), for which our quantization proxy is not parametrized. Our approach simply applies a quantization noise, called Quant-Noise, to a random subset of the weights, see Figure 1. We observe that this makes a network resilient to various types of discretization methods: it significantly improves the accuracy associated with (a) low precision representation of weights like `int8`; and (b) state-of-the-art PQ. Further, we demonstrate that Quant-Noise can be applied to existing trained networks as a post-processing step, to improve the performance network after quantization.

In summary, this paper makes the following contributions:

- We introduce the Quant-Noise technique to learn networks that are more resilient to a variety of quantization methods such as `int4`, `int8`, and PQ;
- Adding Quant-Noise to PQ leads to new state-of-the-art trade-offs between accuracy and model size. For instance, for natural language processing (NLP), we reach 82.5% accuracy on MNLI by compressing RoBERTa to 14 MB. Similarly for computer vision, we report 80.0% top-1 accuracy on ImageNet by compressing an EfficientNet-B3 to 3.3 MB;
- By combining PQ and `int8` to quantize weights and activations for networks trained with Quant-Noise, we obtain extreme compression with fixed-precision computation and achieve 79.8% top-1 accuracy on ImageNet and 21.1 perplexity on WikiText-103.

## 2 Related Work

**Model compression.** Many compression methods focus on efficient parameterization, via weight pruning [5, 9, 10, 11], weight sharing [12, 13, 14] or with dedicated architectures [4, 15, 16]. Weight pruning is implemented during training [17] or as a fine-tuning post-processing step [18, 19]. Many pruning methods are unstructured, i.e., remove individual weights [5, 20]. On the other hand, structured pruning methods follow the structure of the weights to reduce both the memory footprint and the inference time of a model [9, 21, 22]. We refer the reader to Liu *et al.* [23] for a review of different pruning strategies. Others have worked on lightweight architectures, by modifying existing models [24, 25, 26] or developing new networks, such as MobileNet [16], ShuffleNet [15], and EfficientNet [4] in vision. Finally, knowledge distillation [6] has been applied to sentence representation [13, 27, 28, 29, 30], to reduce the size of a BERT model [31].

**Quantization.** There are extensive studies of scalar quantization to train networks with low-precision weightsand activations [32, 33, 34, 35]. These methods benefit from specialized hardware to also improve the runtime during inference [7]. Other quantization methods such as Vector Quantization (VQ) and PQ [36] quantize blocks of weights simulatneously to achieve higher compression rate [8, 37, 38, 39]. Closer to our work, several works have focused at simulatenously training and quantizing a network [1, 40, 41, 42]. Gupta *et al.* [41] assigns weights to a quantized bin stochastically which is specific to scalar quantization, but allows training with fixed point arithmetic. Finally, our method can be interpreted as a form of Bayesian compression [17], using the Bayesian intepretation of Dropout [43]. As opposed to their work, we select our noise to match the weight transformation of a target quantization methods without restricting it to a scale mixture prior.

## 3 Quantizing Neural Networks

In this section, we present the principles of quantization, several standard quantization methods, and describe how to combine scalar and product quantization. For clarity, we focus on the case of a fixed real matrix $\mathbf{W} \in \mathbf{R}^{n \times p}$. We suppose that this matrix is split into $m \times q$ blocks $\mathbf{b}_{kl}$:

$$\mathbf{W} = \begin{pmatrix} \mathbf{b}_{11} & \cdots & \mathbf{b}_{1q} \\ \vdots & \ddots & \vdots \\ \mathbf{b}_{m1} & \cdots & \mathbf{b}_{mq} \end{pmatrix}, \tag{1}$$

where the nature of these blocks is determined by the quantization method. A codebook is a set of $K$ vectors, i.e., $\mathcal{C} = \{\mathbf{c}[1], \ldots, \mathbf{c}[K]\}$. Quantization methods compress the matrix $\mathbf{W}$ by assigning to each block $\mathbf{b}_{kl}$ an index that points to a codeword $\mathbf{c}$ in a codebook $\mathcal{C}$, and storing the codebook $\mathcal{C}$ and the resulting indices (as the entries $\mathbf{I}_{kl}$ of an index matrix $\mathbf{I}$) instead of the real weights. During the inference, they reconstruct an approximation $\widehat{\mathbf{W}}$ of the original matrix $\mathbf{W}$ such that $\widehat{\mathbf{b}}_{kl} = \mathbf{c}[\mathbf{I}_{kl}]$.

We distinguish scalar quantization, such as `int8`, where each block $\mathbf{b}_{kl}$ consists of a single weight, from vector quantization, where several weights are quantized jointly.

### 3.1 Fixed-point Scalar Quantization

Fixed-point scalar quantization methods replace floating-point number representations by low-precision fixed-point representations. They simultaneously reduce a model's memory footprint and accelerate inference by using fixed-point arithmetic on supporting hardware.

Fixed-point scalar quantization operates on blocks that represent a single weight, i.e., $\mathbf{b}_{kl} = \mathbf{W}_{kl}$. Floating-point weights are replaced by $N$ bit fixed-point numbers [41], with the extreme case of binarization where $N = 1$ [32]. More precisely, the weights are rounded to one of $2^N$ possible codewords. These codewords correspond to bins evenly spaced by a scale factor $s$ and shifted by a bias $z$. Each weight $\mathbf{W}_{kl}$ is mapped to its nearest codeword $c$, i.e.,

$$\mathbf{c} = (\text{round}(\mathbf{W}_{kl}/s + z) - z) \times s, \tag{2}$$

where we compute the scale and bias as:

$$s = \frac{\max \mathbf{W} - \min \mathbf{W}}{2^N - 1} \quad \text{and} \quad z = \text{round}(\min \mathbf{W}/s).$$

We focus on this uniform rounding scheme instead of other non-uniform schemes [44, 45], because it allows for fixed-point arithmetic with implementations in PyTorch and Tensorflow (see Appendix). The compression rate is $\times 32/N$. The activations are also rounded to $N$-bit fixed-point numbers. With `int8` for instance, this leads to $\times 2$ to $\times 4$ faster inference on dedicated hardware. In this work, we consider both `int4` and `int8` quantization.

### 3.2 Product Quantization

Several quantization methods work on groups of weights, such as vectors, to benefit from the correlation induced by the structure of the network. In this work, we focus on Product Quantization for its good performance at extreme compression ratio [8].

**Traditional PQ.** In vector quantization methods, the blocks are predefined groups of weights instead of single weights. The codewords are groups of values, and the index matrix $\mathbf{I}$ maps groups of weights from the matrix $\mathbf{W}$ to these codewords. In this section, we present the Product Quantization framework as it generalizes both scalar and vector quantization. We consider the case where we apply PQ to the *columns* of $\mathbf{W}$ and thus assume that $q = p$.

Traditional vector quantization techniques split the matrix $\mathbf{W}$ into its $p$ columns and learns a codebook on the resulting $p$ vectors. Instead, Product Quantization splits each column into $m$ subvectors and learns the same codebook for each of the resulting $m \times p$ subvectors. Each quantized vector is subsequently obtained by assigning its subvectors to the nearest codeword in the codebook. Learning the codebook is traditionally done using $k$-means with a fixed number $K$ of centroids, typically $K = 256$ to store the index matrix $\mathbf{I}$ using `int8`. Thus, the objective function is written as:

$$\|\mathbf{W} - \widehat{\mathbf{W}}\|_2^2 = \sum_{k,l} \|\mathbf{b}_{kl} - \mathbf{c}[\mathbf{I}_{kl}]\|_2^2. \tag{3}$$

PQ shares representations between subvectors, which allows for higher compression rates than `intN`.

**Iterative PQ.** When quantizing a full network rather than a single matrix, extreme compression with PQ induces a quantization drift as reconstruction error accumulates [8]. Indeed, subsequent layers take as input the output of preceding layers, which are modified by the quantization of the preceding layers. This creates a drift in the network activations, resulting in large losses of performance. A solution proposed by Stock *et al.* [8], which we call **iterative PQ** (iPQ), is to quantize layers sequentially from the lowest to the highest, and finetune the upper layers as the lower layers are quantized, under the supervision of the uncompressed (teacher) model. Codewords of each layer are finetuned by averaging the gradients of their assigned elements with gradient steps of the form:

$$\mathbf{c} \leftarrow \mathbf{c} - \eta \frac{1}{|J_\mathbf{c}|} \sum_{(k,l) \in J_\mathbf{c}} \frac{\partial \mathcal{L}}{\partial \mathbf{b}_{kl}}, \tag{4}$$

where $J_\mathbf{c} = \{(k,l) \mid \mathbf{c}[\mathbf{I}_{kl}] = \mathbf{c}\}$, $\mathcal{L}$ is the loss function and $\eta > 0$ is a learning rate. This adapts the upper layers to the drift appearing in their inputs, reducing the impact of the quantization approximation on the overall performance.

### 3.3 Combining Fixed-Point with Product Quantization

Fixed-point quantization and Product Quantization are often regarded as competing choices, but can be advantageously combined. Indeed, PQ/iPQ compresses the network by replacing vectors of weights by their assigned centroids, but these centroids are in floating-point precision. Fixed-point quantization compresses both activations and weights to fixed-point representations. Combining both approaches means that the vectors of weights are mapped to centroids that are compressed to fixed-point representations, along with the activations. This benefits from the extreme compression ratio of iPQ and the finite-precision arithmetics of `intN` quantization.

More precisely, for a given matrix, we store the `int8` representation of the $K$ centroids of dimension $d$ along with the $\log_2 K$ representations of the centroid assignments of the $m \times p$ subvectors. The `int8` representation of the centroids is obtained with Eq. (2). The overall storage of the matrix and activations during a forward pass with batch size 1 is

$$M = 8 \times Kd + \log_2 K \times mp + 8 \times n \text{ bits}. \tag{5}$$

In particular, when $K = 256$, the centroid assignments are also stored in `int8`, which means that every value required for a forward pass is stored in an `int8` format. We divide by 4 the `float32` overhead of storing the centroids, although the storage requirement associated with the centroids is small compared to the cost of indexing the subvectors for standard networks. In contrast to iPQ alone where we only quantize the weights, we also quantize the activations using `int8`. We evaluate this approach on both natural language processing and computer vision tasks in Section 5.

## 4 Method

Deep networks are not exposed to the noise caused by the quantization drift during training, leading to suboptimal performance. A solution to make the network robust to quantization is to introduce it during training. Quantization Aware Training (QAT) [1] exposes the network during training by quantizing weights during the forward pass. This transformation is not differentiable and gradients are approximated with a straight through estimator (STE) [2, 33]. STE introduces a bias in the gradients that depends on level of quantization of the weights, and thus, the compression ratio. In this section, we propose a simple modification to control this induced bias with a stochastic amelioration of QAT, called Quant-Noise. The idea is to quantize a randomly selected fraction of the weights instead of the full network as in QAT, leaving some unbiased gradients flow through unquantized weights. Our general formulation can simulate the effect of both quantization and of pruning during training.

## 4.1 Training Networks with Quantization Noise

We consider the case of a real matrix $\mathbf{W}$ as in Section 3. During the training of a network, our proposed Quant-Noise method works as follows: first, we compute blocks $\mathbf{b}_{kl}$ related to a target quantization method. Then, during each forward pass, we randomly select a subset of these blocks and apply some distortion to them. During the backward pass, we compute gradients for all the weights, using STE for the distorted weights.

More formally, given a set of tuples of indices $J \subset \{(k,l)\}$ for $1 \leq k \leq m$, $1 \leq l \leq q$ and a *distortion* or *noise* function $\varphi$ acting on a block, we define an operator $\psi(\cdot \mid J)$ such that, for each block $\mathbf{b}_{kl}$, we apply the following transformation:

$$\psi(\mathbf{b}_{kl} \mid J) = \begin{cases} \varphi(\mathbf{b}_{kl}) & \text{if } (k,l) \in J, \\ \mathbf{b}_{kl} & \text{otherwise.} \end{cases} \tag{6}$$

The noise function $\varphi$ simulates the change in the weights produced by the target quantization method (see Section 4.2 for details). We replace the matrix $\mathbf{W}$ by the resulting noisy matrix $\mathbf{W}_{\text{noise}}$ during the forward pass to compute a noisy output $\mathbf{y}_{\text{noise}}$, i.e.,

$$\mathbf{W}_{\text{noise}} = (\psi(\mathbf{b}_{kl} \mid J))_{kl} \quad \text{and} \quad \mathbf{y}_{\text{noise}} = \mathbf{W}_{\text{noise}}\mathbf{x}, \tag{7}$$

where $\mathbf{x}$ is an input vector. During the backward pass, we compute the gradient on the non-distorted weights and apply STE on the distorted weights, i.e.,

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \mathbf{y}_{\text{noise}}\mathbf{x}^\top. \tag{8}$$

Note that our approach is equivalent to QAT when $J$ containts all the tuples of indices. However, an advantage of Quant-Noise over QAT is that unbiased gradients continue to flow via blocks unaffected by the noise. As these blocks are randomly selected for each forward, we guarantee that each weight regularly sees gradients that are not affected by the nature of the function $\varphi$. As a side effect, our quantization noise regularizes the network in a similar way as DropConnect [46] or LayerDrop [22].

**Composing quantization noises.** As noise operators are compositionally commutative, we can make a network robust to a combination of quantization methods by composing their noise operators:

$$\psi(\mathbf{b}_{kl} \mid J) = \psi_1 \circ \psi_2(\mathbf{b}_{kl} \mid J). \tag{9}$$

This property is particularly useful to combine quantization with pruning operators during training, as well as combining scalar quantization with product quantization.

## 4.2 Adding Noise to Specific Quantization Methods

In this section, we propose several implementations of the noise function $\varphi$ for the quantization methods described in Section 3. We also show how to handle pruning with it.

**Fixed-point scalar quantization.** In `intN` quantization, the blocks are atomic and weights are rounded to their nearest neighbor in the codebook. The function $\varphi$ replaces weight $\mathbf{W}_{kl}$ with the output of the rounding function defined in Eq. (2), i.e.,

$$\varphi_{\texttt{intN}}(w) = (\text{round}(w/s + z) - z) \times s, \tag{10}$$

where $s$ and $z$ are updated during training. In particular, the application of Quant-Noise to `int8` scalar quantization is a stochastic amelioration of QAT.

| **Quantization Scheme** | **Language Modeling** 16-layer Transformer Wikitext-103 | | | **Image Classification** EfficientNet-B3 ImageNet-1k | | |
|---|---|---|---|---|---|---|
| | Size | Compression | PPL | Size | Compression | Top-1 |
| Uncompressed model | 942 | $\times$ 1 | 18.3 | 46.7 | $\times$ 1 | 81.5 |
| `int4` quantization | 118 | $\times$ 8 | 39.4 | 5.8 | $\times$ 8 | 45.3 |
| - trained with QAT | 118 | $\times$ 8 | 34.1 | 5.8 | $\times$ 8 | 59.4 |
| - trained with Quant-Noise | 118 | $\times$ 8 | **21.8** | 5.8 | $\times$ 8 | **67.8** |
| `int8` quantization | 236 | $\times$ 4 | 19.6 | 11.7 | $\times$ 4 | 80.7 |
| - trained with QAT | 236 | $\times$ 4 | 21.0 | 11.7 | $\times$ 4 | 80.8 |
| - trained with Quant-Noise | 236 | $\times$ 4 | **18.7** | 11.7 | $\times$ 4 | **80.9** |
| iPQ | 38 | $\times$ 25 | 25.2 | 3.3 | $\times$ 14 | 79.0 |
| - trained with QAT | 38 | $\times$ 25 | 41.2 | 3.3 | $\times$ 14 | 55.7 |
| - trained with Quant-Noise | 38 | $\times$ 25 | **20.7** | 3.3 | $\times$ 14 | **80.0** |
| iPQ & `int8` + Quant-Noise | 38 | $\times$ 25 | 21.1 | 3.1 | $\times$ 15 | 79.8 |

Table 1: **Comparison of different quantization schemes with and without Quant-Noise** on language modeling and image classification. For language modeling, we train a Transformer on the Wikitext-103 benchmark and report perplexity (PPL) on test. For image classification, we train a EfficientNet-B3 on the ImageNet-1k benchmark and report top-1 accuracy on validation and use our re-implementation of EfficientNet-B3. The original implementation of Tan *et al.* [4] achieves an uncompressed Top-1 accuracy of 81.9%. For both settings, we report model size in megabyte (MB) and the compression ratio compared to the original model.

**Product quantization.** As opposed to `intN`, codebooks in PQ require a clustering step based on weight values. During training, we learn codewords online and use the resulting centroids to implement the quantization noise. More precisely, the noise function $\varphi_{\text{PQ}}$ assigns a selected block $\mathbf{b}$ to its nearest codeword in the associated codebook $\mathcal{C}$:

$$\varphi_{\text{PQ}}(\mathbf{v}) = \text{argmin}_{\mathbf{c}\in\mathcal{C}} \|\mathbf{b} - \mathbf{c}\|_2^2. \tag{11}$$

Updating the codebooks online works well. However, empirically, running $k$-means once per epoch is faster and does not noticeably modify the resulting accuracy.

Note that computing the exact noise function for PQ is computationally demanding. We propose a simpler and faster alternative approximation $\varphi_{\text{proxy}}$ to the operational transformation of PQ and iPQ. The noise function simply zeroes out the subvectors of the selected blocks, i.e., $\varphi_{\text{proxy}}(\mathbf{v}) = 0$. As a sidenote, we considered other alternatives, for instance one where the subvectors are mapped to the mean subvector. In practice, we found that these approximations lead to similar performance, see Section 6. This proxy noise function is a form of Structured Dropout and encourages correlations between the subvectors. This correlation is beneficial to the subsequent clustering involved in PQ/iPQ.

**Adding pruning to the quantization noise.** The specific form of quantization noise can be adjusted to incorporate additional noise specific to pruning. We simply combine the noise operators of quantization and pruning by composing them following Eq. (9). We consider the pruning noise function of Fan *et al.* [22] where they randomly drop predefined structures during training. In particular, we focus on *LayerDrop*, where the structures are the residual blocks of highway-like layers [47], as most modern architectures, such as ResNet or Transformer, are composed of this structure. More precisely, the corresponding noise operator over residual blocks $\mathbf{v}$ is $\varphi_{\text{LayerDrop}}(\mathbf{v}) = 0$. For pruning, we do not use STE to backpropagate the gradient of pruned weights, as dropping them entirely during training has the benefit of speeding convergence [48]. Once a model is trained with LayerDrop, the number of layers kept at inference can be adapted to match computation budget or time constraint.

## 5 Results

We demonstrate the impact of Quant-Noise on the performance of several quantization schemes in a variety of settings (see Appendix - Sec. 8.1). We compare iPQ + Quant-Noise with existing work to demonstrate that Quant-Noise leads to extreme compression rates at a reasonable cost in accuracy.

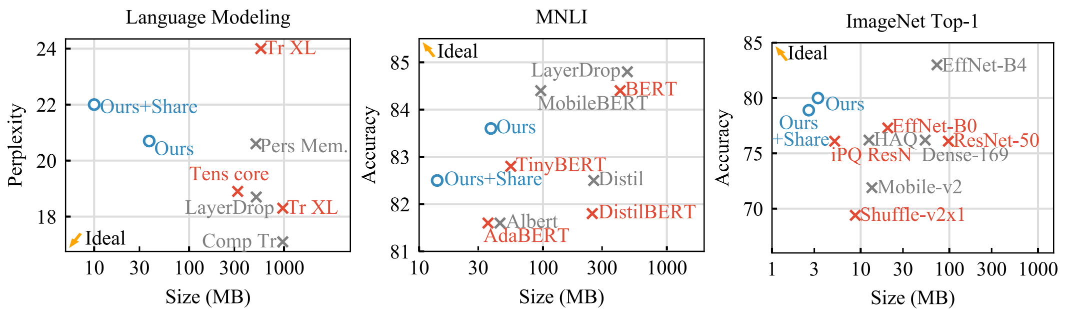


Figure 2: **Performance as a function of model size.** We compare models quantized with PQ and trained with the related Quant-Noise to the state of the art. **(a)** Test perplexity on Wikitext-103 **(b)** Dev Accuracy on MNLI **(c)** ImageNet Top-1 accuracy. Model size is shown in megabytes on a log scale. Red and gray coloring indicates existing work, with different colors for visual distinction.

| | **Language modeling** | | | **Sentence Representation** | | | **Image Classification** | | |
|---|---|---|---|---|---|---|---|---|---|
| | Comp. | Size | PPL | Comp. | Size | Acc. | Comp. | Size | Acc. |
| *Unquantized models* | | | | | | | | | |
| Original model | × 1 | 942 | 18.3 | × 1 | 480 | 84.8 | × 1 | 46.7 | 81.5 |
| + Sharing | × 1.8 | 510 | 18.7 | × 1.9 | 250 | 84.0 | × 1.4 | 34.2 | 80.1 |
| + Pruning | × 3.7 | 255 | 22.5 | × 3.8 | 125 | 81.3 | × 1.6 | 29.5 | 78.5 |
| *Quantized models* | | | | | | | | | |
| iPQ | × 24.8 | 38 | 25.2 | × 12.6 | 38 | 82.5 | × 14.1 | 3.3 | 79.0 |
| + Quant-Noise | × 24.8 | 38 | 20.7 | × 12.6 | 38 | 83.6 | × 14.1 | 3.3 | 80.0 |
| + Sharing | × 49.5 | 19 | 22.0 | × 34.3 | 14 | 82.5 | × 18 | 2.6 | 78.9 |
| + Pruning | × 94.2 | 10 | 24.7 | × 58.5 | 8 | 78.8 | × 20 | 2.3 | 77.8 |

Table 2: **Decomposing the impact of the different compression schemes. (a)** we train Transformers with Adaptive Input and LayerDrop on Wikitext-103 **(b)** we pre-train RoBERTA base models with LayerDrop and then finetune on MNLI **(c)** we train an EfficientNet-B3 on ImageNet. We report the compression ratio w.r.t. to the original model ("comp.") and the resulting size in MB.

### 5.1 Improving Compression with Quant-Noise

Quant-Noise is a regularization method that makes networks more robust to the target quantization scheme or combination of quantization schemes during training. We show the impact of Quant-Noise in Table 1 for a variety of quantization methods: `int8`/`int4` and iPQ.

We experiment in 2 different settings: a Transformer network trained for language modeling on WikiText-103 and a EfficientNet-B3 convolutional network trained for image classification on ImageNet-1k. Our quantization noise framework is general and flexible — Quant-Noise improves the performance of quantized models for every quantization scheme in both experimental settings. Importantly, Quant-Noise only changes model training by adding a regularization noise similar to dropout, with no impact on the convergence rate or training speed.

This comparison of different quantization schemes shows that Quant-Noise works particularly well with high performance quantization methods, like iPQ, where QAT tends to degrade the performances, even compared to quantizing as a post-processing step. In subsequent experiments in this section, we focus on applications with iPQ because it offers the best trade-off between model performance and compression, and has little negative impact on FLOPS.

**Fixed-Point Product Quantization.** Combining iPQ and `int8` as described in Section 3.3 allows us to take advantage of the high compression rate of iPQ with a fixed-point representation of both centroids and activations. As shown in Table 1, this combination incurs little loss in accuracy with respect to iPQ + Quant-Noise. Most of the memory footprint of iPQ comes from indexing and not storing centroids, so the compression ratios are comparable.

| Adaptive Input | PPL | RoBERTa | Acc. |
|---|---|---|---|
| Train without Quant-Noise | 25.2 | Train without Quant-Noise | 82.5 |
| + Finetune with Quant-Noise | 20.9 | + Finetune with Quant-Noise | 83.4 |
| Train with Quant-Noise | 20.7 | Train with Quant-Noise | 83.6 |

Table 3: **Quant-Noise: Finetuning vs training.** We report performance after quantization with iPQ. We use the $\phi_{\text{proxy}}$ noise function to train and finetune with Quant-Noise. We also use it during the transfer to MNLI for each RoBERTa model.

**Complementarity with Weight Pruning and Sharing.** We analyze how Quant-Noise is compatible and complementary with pruning ("+Prune") and weight sharing ("+Share"), see Appendix for details on weight sharing. We report results for Language modeling on WikiText-103, pre-trained sentence representations on MNLI and object classification on ImageNet-1k in Table 2. The conclusions are remarkably consistent across tasks and benchmarks: Quant-Noise gives a large improvement over strong iPQ baselines. Combining it with sharing and pruning offers additional interesting operating points of performance vs size.

## 5.2 Comparison with the state of the art

We now compare our approach on the same tasks against the state of the art. We apply our best quantization setup on competitive models and reduce their memory footprint by $\times 20 - 94$ when combining with weight sharing and pruning, offering extreme compression for good performance.

**Natural Language Processing.** In Figure 2, we examine the trade-off between performance and model size. Our quantized RoBERTa offers a competitive trade-off between size and performance with memory reduction methods dedicated to BERT, like TinyBERT, MobileBERT, or AdaBERT.

**Image Classification.** We compress EfficientNet-B3 from 46.7Mb to 3.3Mb ($\times 14$ compression) while maintaining high top-1 accuracy ($78.5\%$ versus $80\%$ for the original model). As shown in Figure 2, our quantized EfficientNet-B3 is smaller and more accurate than architectures dedicated to optimize on-device performance with limited size like MobileNet or ShuffleNet.

Incorporating pruning noise into quantization is also beneficial. For example, with pruning iPQ+Quant-Noise reduces size by $\times 25$ with only a drop of $2.4$ PPL in language modeling. Further, pruning reduces FLOPS by the same ratio as its compression factor, in our case, $\times 2$. By adding sharing with pruning, in language modeling, we achieve an extreme compression ratio of $\times 94$ with a drop of $6.4$ PPL with FLOPS reduction from pruning entire shared chunks of layers. For comparison, our $10$ MB model has the same performance as the $570$ MB Transformer-XL base.

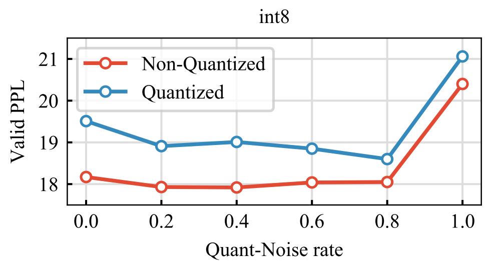
## 6 Ablations

In this section, we study the use of our approach as a post-processing step where a pre-trained model is finetuned with Quant-Noise. We also examine the impact of the level of noise during training as well as the impact of approximating iPQ during training.

### 6.1 Finetuning with Quant-Noise for Post-Processing Quantization

We explore taking existing pre-trained models and post-processing with Quant-Noise instead of training from scratch. For language modeling, we start with the Adaptive Inputs architecture and train for 10 additional epochs. For RoBERTa, we train for 25k more updates. We show that finetuning with Quant-Noise incorporates the benefits and almost matches training from scratch, see Table 3. For example, in language modeling, there is only a $0.2$ PPL difference after applying iPQ.

We further examine how to incorporate Quant-Noise more flexibly into pretraining RoBERTa for sentence classification tasks. We take an already pre-trained RoBERTa model and only incorporate Quant-Noise during the sentence classification task transfer learning step. We show in Table 3 that this is also effective at compressing while retaining accuracy after quantization with iPQ.



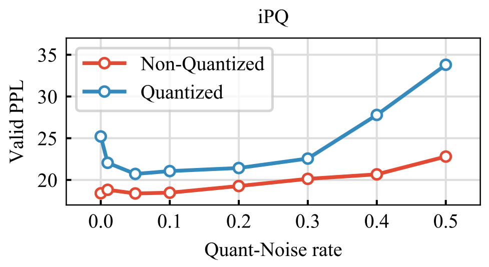


Figure 3: **Effect of Quantization Parameters**. We report the influence of the proportion of blocks to which we apply the noise. We focus on Transformer for Wikitext-103 language modeling. We explore two settings: iPQ and `int8`. For iPQ, we use $\varphi_{\text{proxy}}$.

| Noise | Blocks | PPL | Quant PPL |
|---|---|---|---|
| $\varphi_{\text{PQ}}$ | Subvectors | 18.3 | 21.1 |
| $\varphi_{\text{PQ}}$ | Clusters | 18.3 | 21.2 |
| $\varphi_{\text{proxy}}$ | Subvectors | 18.3 | 21.0 |
| $\varphi_{\text{proxy}}$ | Clusters | 18.4 | 21.1 |

Table 4: **Exact versus proxy noise function for different block selections with iPQ.** We compare exact ($\phi_{\text{PQ}}$ and the approximation $\phi_{\text{proxy}}$ with blocks selected from all subvectors or subvectors from the same cluster.

## 6.2 Impact of Noise Rate

We analyze the performance for various values of Quant-Noise in Figure 3 on a Transformer for language modeling. For iPQ, performance is impacted by high rates of quantization noise. For example, a Transformer with the noise function $\varphi_{\text{proxy}}$ degrades with rate higher than 0.5, i.e., when half of the weights are passed through the noise function $\varphi_{\text{proxy}}$. We hypothesize that for large quantities of noise, a larger effect of using proxy rather than the exact PQ noise is observed. For `int8` quantization and its noise function, higher rates of noise are slightly worse but not as severe. A rate of 1 for `int8` quantization is equivalent to the Quantization Aware Training of [40], as the full matrix is quantized with STE, showing the potential benefit of partial quantization during training.

## 6.3 Impact of Approximating the Noise Function

We study the impact of approximating quantization noise during training. We focus on the case of iPQ with the approximation described in Section 4.2. In Table 4, we compare the correct noise function for iPQ with its approximation $\varphi_{\text{proxy}}$. This approximate noise function does not consider cluster assignments or centroid values and simply zeroes out the selected blocks. For completeness, we include an intermediate approximation where we consider cluster assignments to apply noise within each cluster, but still zero-out the vectors. These approximations do not affect the performance of the quantized models. This suggests that increasing the correlation between subvectors that are jointly clustered is enough to maintain the performance of a model quantized with iPQ. Since PQ tends to work well on highly correlated vectors, such as activations in convolutional networks, this is not surprising. Using the approximation $\varphi_{\text{proxy}}$ presents the advantage of speed and practicality. Indeed, one does not need to compute cluster assignments and centroids for every layer in the network after each epoch. Moreover, the approach $\varphi_{\text{proxy}}$ is less involved in terms of code.

## 7 Conclusion

We show that quantizing a random subset of weights during training maintains performance in the high quantization regime. We validate that Quant-Noise works with a variety of different quantization schemes on several applications in text and vision. Our method can be applied to a combination of iPQ and `int8` to benefit from extreme compression ratio and fixed-point arithmetic. Finally, we show that Quant-Noise can be used as a post-processing step to prepare already trained networks for subsequent quantization, to improve the performance of the compressed model.

## References

[1] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[2] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

[4] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2019.

[5] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *NIPS*, 1990.

[6] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[7] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. 2011.

[8] Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. And the bit goes down: Revisiting the quantization of neural networks. *CoRR*, abs/1907.05686, 2019.

[9] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[10] Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *CVPR*, 2018.

[11] Deepak Mittal, Shweta Bhardwaj, Mitesh M Khapra, and Balaraman Ravindran. Recovering from random pruning: On the plasticity of deep convolutional neural networks. In *WACV*, 2018.

[12] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers, 2018.

[13] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: The impact of student initialization on knowledge distillation. *arXiv preprint arXiv:1908.08962*, 2019.

[14] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations, 2019.

[15] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *CoRR*, 2017.

[16] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. *arXiv e-prints*, 2019.

[17] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through $l\_0$ regularization. *arXiv preprint arXiv:1712.01312*, 2017.

[18] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NIPS*, pages 1135–1143, 2015.

[19] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *ICLR*, 2016.

[20] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *ICML*, 2017.

[21] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *ICCV*, 2017.

[22] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

[23] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.

[24] Biao Zhang, Deyi Xiong, and Jinsong Su. Accelerating neural transformer via an average attention network. *arXiv preprint arXiv:1805.00631*, 2018.

[25] Felix Wu, Angela Fan, Alexei Baevski, Yann Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. In *ICLR*, 2019.

[26] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. *arXiv preprint arXiv:1905.07799*, 2019.

[27] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. 2019.

[28] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for bert model compression. *EMNLP*, 2019.

[29] Sanqiang Zhao, Raghav Gupta, Yang Song, and Denny Zhou. Extreme language model compression with optimal subwords and shared projections. *arXiv preprint arXiv:1909.11687*, 2019.

[30] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[32] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, 2015.

[33] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, 2016.

[34] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[35] Mark D. McDonnell. Training wide residual networks for deployment using a single bit for each weight, 2018.

[36] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *PAMI*, 2011.

[37] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[38] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. Fasttext.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016.

[39] Miguel A. Carreira-Perpiñán and Yerlan Idelbayev. Model compression as constrained optimization, with application to neural nets. part ii: quantization, 2017.

[40] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[41] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML*, 2015.

[42] Yinpeng Dong, Renkun Ni, Jianguo Li, Yurong Chen, Hang Su, and Jun Zhu. Stochastic quantization for learning accurate low-bit deep neural networks. *International Journal of Computer Vision*, 127(11-12):1629–1642, 2019.

[43] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[44] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[45] Yuhang Li, Xin Dong, and Wei Wang. Additive powers-of-two quantization: A non-uniform discretization for neural networks. *arXiv preprint arXiv:1909.13144*, 2019.

[46] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using DropConnect. In *ICML*, 2013.

[47] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[48] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *ECCV*, 2016.

[49] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[50] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[51] A. Adcock, V. Reis, M. Singh, Z. Yan, L. van der Maaten, K. Zhang, S. Motwani, J. Guerin, N. Goyal, I. Misra, L. Gustafson, C. Changhan, and P. Goyal. Classy vision. 2019.

[52] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer Sentinel Mixture Models. *arXiv*, abs/1609.07843, 2016.

[53] Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*, 2018.

[54] Adina Williams, Nikita Nangia, and Samuel R. Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of NAACL-HLT*, 2018.

[55] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. 2019. ICLR.

[56] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[57] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

[58] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proc. of ICML*, 2017.

[59] Edouard Grave, Armand Joulin, Moustapha Cisse, David Grangier, and Herve Jegou. Efficient softmax approximation for gpus. *arXiv*, abs/1609.04309, 2016.

[60] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[61] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[62] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. In *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*, 2014.

[63] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[64] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

[65] Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[66] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*, 2019.

[67] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*, 2016.

[68] Sainbayar Sukhbaatar, Edouard Grave, Guillaume Lample, Herve Jegou, and Armand Joulin. Augmenting self-attention with persistent memory. *arXiv preprint arXiv:1907.01470*, 2019.

[69] Xindian Ma, Peng Zhang, Shuai Zhang, Nan Duan, Yuexian Hou, Dawei Song, and Ming Zhou. A tensorized transformer for language modeling. *arXiv preprint arXiv:1906.09777*, 2019.

[70] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[71] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobile-bert: Task-agnostic compression of bert for resource limited devices.

[72] Daoyuan Chen, Yaliang Li, Minghui Qiu, Zhen Wang, Bofang Li, Bolin Ding, Hongbo Deng, Jun Huang, Wei Lin, and Jingren Zhou. Adabert: Task-adaptive bert compression with differentiable neural architecture search. *arXiv preprint arXiv:2001.04246*, 2020.

[73] Qingqing Cao, Harsh Trivedi, Aruna Balasubramanian, and Niranjan Balasubramanian. Faster and just as accurate: A simple decomposition for transformer models.

[74] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, 2015.

[75] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[76] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet V2: practical guidelines for efficient CNN architecture design. *CoRR*, 2018.

[77] Kuan Wang, Zhijian Liu, Yujun Lin andx Ji Lin, and Song Han. HAQ: hardware-aware automated quantization. *CoRR*, 2018.

## 8 Appendix

### 8.1 Experimental Setting

We assess the effectiveness of Quant-Noise on competitive language and vision benchmarks. We consider Transformers for language modeling, RoBERTa for pre-training sentence representations, and EfficientNet for image classification. Our models are implemented in PyTorch [49]. We use `fairseq` [50] for language modeling and pre-training for sentence representation tasks and `Classy Vision` [51] for EfficientNet.

**Language Modeling.** We experiment on the `Wikitext-103` benchmark [52] that contains 100M tokens and a vocabulary of 260k words. We train a 16 layer Transformer following Baevski *et al.* [53] with a LayerDrop rate of 0.2 [22]. We report perplexity (PPL) on the test set.

**Pre-Training of Sentence Representations.** We pre-train the base BERT model [31] on the `BooksCorpus + Wiki` dataset with a LayerDrop rate of 0.2. We finetune the pre-trained models on the MNLI task [54] from the GLUE Benchmark [55] and report accuracy. We follow the parameters in Liu *et al.* [56] training and finetuning.

**Image Classification.** We train an EfficientNet-B3 model [4] on the ImageNet object classification benchmark [57]. The EfficientNet-B3 of `Classy Vision` achieves a Top-1 accuracy of 81.5%, which is slightly below than the performance of 81.9% reported by Tan *et al.*[4].

### 8.2 Training Details

**Language Modeling** To handle the large vocabulary of Wikitext-103, we follow [58] and [53] in using adaptive softmax [59] and adaptive input for computational efficiency. For both input and output embeddings, we use dimension size 1024 and three adaptive bands: 20K, 40K, and 200K. We use a cosine learning rate schedule [53, 60] and train with Nesterov's accelerated gradient [61]. We set the momentum to 0.99 and renormalize gradients if the norm exceeds 0.1 [62]. During training, we partition the data into blocks of contiguous tokens that ignore document boundaries. At test time, we respect sentence boundaries. We set LayerDrop to 0.2. We set Quant-Noise value to 0.05. During training time, we searched over the parameters (0.05, 0.1, 0.2) to determine the optimal value of Quant-Noise. During training time, the block size of Quant-Noise is 8.

**RoBERTa** The base architecture is a 12 layer model with embedding size 768 and FFN size 3072. We follow [56] in using the subword tokenization scheme from [63], which uses bytes as subword units. This eliminates unknown tokens. We train with large batches of size 8192 and maintain this batch size using gradient accumulation. We do not use next sentence prediction [64]. We optimize with Adam with a polynomial decay learning rate schedule. We set LayerDrop to 0.2. We set Quant-Noise value to 0.1. We did not hyperparameter search to determine the optimal value of Quant-Noise as training RoBERTa is computationally intensive. During training time, the block size of Quant-Noise is 8.

During finetuning, we hyperparameter search over three learning rate options (1e-5, 2e-5, 3e-5) and batchsize (16 or 32 sentences). The other parameters are set following [56]. We do single task finetuning, meaning we only tune on the data provided for the given natural language understanding task. We do not perform ensembling. When finetuning models trained with LayerDrop, we apply LayerDrop and Quant-Noise during finetuning time as well.

**EfficientNet** We use the architecture of EfficientNet-B3 defined in `Classy Vision` [51] and follow the default hyperparameters for training. We set Quant-Noise value to 0.1. During training time, we searched over the parameters (0.05, 0.1, 0.2) to determine the optimal value of Quant-Noise. During training time, the block size of Quant-Noise is set to 4 for all $1 \times 1$ convolutions, 9 for depth-wise $3 \times 3$ convolutions, 5 for depth-wise $5 \times 5$ convolutions and 4 for the classifier. For sharing, we shared weights between blocks 9-10, 11-12, 14-15, 16-17, 19-20-21, 22-23 and refer to blocks that share the same weights as a *chunk*. For LayerDrop, we drop the chunks of blocks defined previously with probability 0.2 and evaluate only with chunks 9-10, 14-15 and 19-20-21.

| **Model** | **MB** | **PPL** |
|---|---|---|
| Trans XL Large [65] | 970 | 18.3 |
| Compressive Trans [66] | 970 | 17.1 |
| GCNN [58] | 870 | 37.2 |
| 4 Layer QRNN [67] | 575 | 33.0 |
| Trans XL Base [65] | 570 | 24.0 |
| Persis Mem [68] | 506 | 20.6 |
| Tensorized core-2 [69] | 325 | 18.9 |
| Quant-Noise | **38** | 20.7 |
| Quant-Noise + Share + Prune | 10 | 24.2 |

Table 5: **Performance on Wikitext-103.** We report test set perplexity and model size in megabytes. Lower perplexity is better.

| **Model** | **MB** | **MNLI** |
|---|---|---|
| RoBERTa Base + LD [22] | 480 | 84.8 |
| BERT Base [31] | 420 | 84.4 |
| PreTrained Distil [13] | 257 | 82.5 |
| DistilBERT [70] | 250 | 81.8 |
| MobileBERT* [71] | 96 | 84.4 |
| TinyBERT† [30] | 55 | 82.8 |
| ALBERT Base [14] | 45 | 81.6 |
| AdaBERT† [72] | 36 | 81.6 |
| Quant-Noise | 38 | 83.6 |
| Quant-Noise + Share + Prune | 14 | 82.5 |

Table 6: **Performance on MNLI.** We report accuracy and size in megabytes. * indicates distillation using BERT Large. † indicates training with data augmentation. Work from Sun *et al.* [28] and Zhao *et al.* [29] do not report results on the dev set. Cao *et al.* [73] do not report model size. Higher accuracy is better.

### 8.3 Scalar Quantization Details

We closely follow the methodology of PyTorch 1.4. We emulate scalar quantization by quantizing the weights and the activations. The scales and zero points of activations are determined by doing a few forward passes ahead of the evaluation and then fixed. We use the `Histogram` method to compute $s$ and $z$, which aims at approximately minimizing the $L_2$ quantization error by adjusting $s$ and $z$. This scheme is a refinement of the `MinMax` scheme. Per channel quantization is also discussed in Table 9.

### 8.4 iPQ Quantization Details

**Language Modeling** We quantize FFN with block size 8, embeddings with block size 8, and attention with block size 4. We tuned the block size for attention between the values (4, 8) to find the best performance. Note that during training with apply Quant-Noise to all the layers.

**RoBERTa** We quantize FFN with block size 4, embeddings with block size 4, and attention with block size 4. We tuned the block size between the values (4, 8) to find the best performance. Note that during training with apply Quant-Noise to all the layers.

**EfficientNet** We quantize blocks sequentially and end up with the classifier. The block sizes are $4$ for all $1 \times 1$ convolutions, $9$ for depth-wise $3 \times 3$ convolutions, $5$ for depth-wise $5 \times 5$ convolutions and $4$ for the classifier. Note that during training with apply Quant-Noise to all the weights in InvertedResidual Blocks (except the Squeeze-Excitation subblocks), the head convolution and the classifier.

| Model | MB | Acc. |
|---|---|---|
| EfficientNet-B7 [4] | 260 | 84.4 |
| ResNet-50 [74] | 97.5 | 76.1 |
| DenseNet-169 [10] | 53.4 | 76.2 |
| EfficientNet-B0 [4] | 20.2 | 77.3 |
| MobileNet-v2 [75] | 13.4 | 71.9 |
| Shufflenet-v2 $\times 1$ [76] | 8.7 | 69.4 |
| HAQ 4 bits [77] | 12.4 | 76.2 |
| iPQ ResNet-50 [8] | 5.09 | 76.1 |
| Quant-Noise | 3.3 | 80.0 |
| Quant-Noise + Share + Prune | 2.3 | 77.8 |

Table 7: **Performance on ImageNet.** We report accuracy and size in megabytes. Higher accuracy is better.

| $p$ | 0 | 0.2 | 0.4 | 0.6 | 0.8 | 1 |
|---|---|---|---|---|---|---|
| Top-1 | 80.66 | 80.83 | 80.82 | 80.88 | 80.92 | 80.64 |

Table 8: **Effect of Quantization Parameters**. We report the influence of the Quant-Noise rate $p$ with Scalar Quantization (`int8`). We focus on EfficientNet for ImageNet classification.

### 8.5 Details of Pruning and Layer Sharing

We apply the *Every Other Layer* strategy from Fan*et al.* [22]. When combining layer sharing with pruning, we train models with shared layers and then prune chunks of shared layers. When sharing layers, the weights of adjacent layers are shared in chunks of two. For a concrete example, imagine we have a model with layers A, B, C, D, E, F, G, H. We share layers A and B, C and D, E and F, G and H. To prune, every other chunk would be pruned away, for example we could prune A, B, E, F.

### 8.6 Numerical Results for Graphical Diagrams

We report the numerical values displayed in Figures 2 and **??** in Table 5 for language modeling, Table 6 for BERT, and Table 7 for ImageNet.

### 8.7 Further Ablations

#### 8.7.1 Impact of Quant-Noise for the Vision setup

We provide another study showing the impact of the proportion of elements on which to apply Quant-Noise in Table 8.

#### 8.7.2 Impact of the number of centroids

We quantize with 256 centroids which represents a balance between size and representation capacity. The effect of the number of centroids on performance and size is shown in Figure 4 (a). Quantizing with more centroids improves perplexity — this parameter could be adjusted based on the practical storage constraints.

#### 8.7.3 Effect of Initial Model Size

Large, overparameterized models are more easily compressed. In Figure 5, we explore quantizing both shallower and skinnier models. For shallow models, the gap between quantized and non-quantized perplexity does not increase as layers are removed (Figure 5, left). In contrast, there is a larger gap in performance for models with smaller FFN (Figure 5, right). As the FFN size decreases, the weights are less redundant and more difficult to quantize with iPQ.

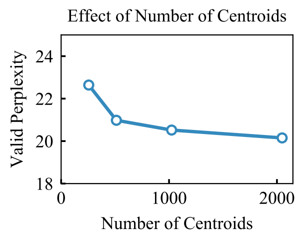


Figure 4: **Quantizing with a larger number of centroids**. Results are shown on Wikitext-103 valid.

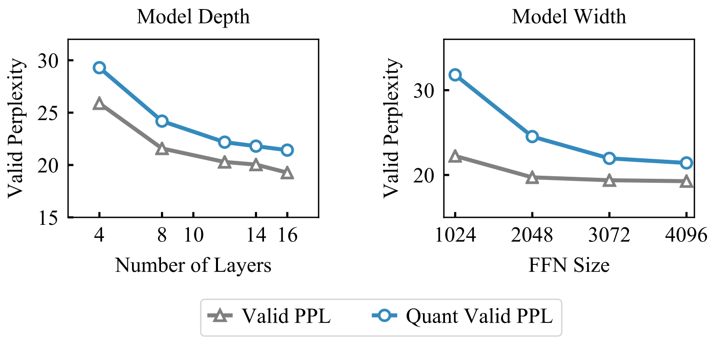


Figure 5: **(a)** Effect of Initial Model Size for more shallow models **(b)** Effect of Initial Model Size more skinny models

### 8.7.4 Difficulty of Quantizing Different Model Structures

Quantization is applied to various portions of the Transformer architecture — the embedding, attention, feedforward, and classifier output. We compare the quantizability of various portions of the network in this section.

**Is the order of structures important?** We quantize specific network structures first — this is important as quantizing weight matrices can accumulate reconstruction error. Some structures of the network should be quantized last so the finetuning process can better adjust the centroids. We find that there are small variations in performance based on quantization order (see Figure 6). We choose to quantize FFN, then embeddings, and finally the attention matrices in Transformer networks.

**Which structures can be compressed the most?** Finally, we analyze which network structures can be most compressed. During quantization, various matrix block sizes can be chosen as a parameter — the larger the block size, the more compression, but also the larger the potential reduction of performance. Thus, it is important to understand how much each network structure can be compressed to reduce the memory footprint of the final model as much as possible. In Figure 6, we quantize two model structures with a fixed block size and vary the block size of the third between 4 and 32. As shown, the FFN and embedding structures are more robust to aggressive compression, while the attention drastically loses performance as larger block sizes are used.

### 8.7.5 Approach to `intN` Scalar Quantization

We compare quantizing per-channel to using a histogram quantizer in Table 9. The histogram quantizer maintains a running min/max and minimizes L2 distance between quantized and non-quantized values to find the optimal min/max. Quantizing per channel learns scales and offsets as vectors along the channel dimension, which provides more flexibility since scales and offsets can be different.

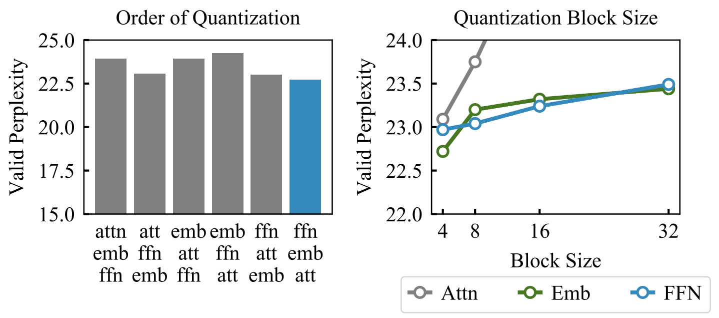


Figure 6: **Effect of Quantization on Model Structures.** Results are shown on the validation set of Wikitext-103. **(a)** Quantizing Attention, FFN, and Embeddings in different order. **(b)** More Extreme compression of different structures.

| Quantization Scheme | Language Modeling 16-layer Transformer Wikitext-103 | | | Image Classification EfficientNet-B3 ImageNet-1K | | |
|---|---|---|---|---|---|---|
| | Size | Compress | Test PPL | Size | Compress | Top-1 Acc. |
| Uncompressed model | 942 | $\times 1$ | 18.3 | 46.7 | $\times 1$ | 81.5 |
| Int4 Quant Histogram | 118 | $\times 8$ | 39.4 | 5.8 | $\times 8$ | 45.3 |
| + Quant-Noise | 118 | $\times 8$ | 21.8 | 5.8 | $\times 8$ | 67.8 |
| Int4 Quant Channel | 118 | $\times 8$ | 21.2 | 5.8 | $\times 8$ | 68.2 |
| + Quant-Noise | 118 | $\times 8$ | 19.5 | 5.8 | $\times 8$ | 72.3 |
| Int8 Quant Histogram | 236 | $\times 4$ | 19.6 | 11.7 | $\times 4$ | 80.7 |
| + Quant-Noise | 236 | $\times 4$ | 18.7 | 11.7 | $\times 4$ | 80.9 |
| Int8 Quant Channel | 236 | $\times 4$ | 18.5 | 11.7 | $\times 4$ | 81.1 |
| + Quant-Noise | 236 | $\times 4$ | 18.3 | 11.7 | $\times 4$ | 81.2 |

Table 9: **Comparison of different approaches to `int4` and `int8` with and without Quant-Noise** on language modeling and image classification. For language modeling, we train a Transformer on the Wikitext-103 benchmark. We report perplexity (PPL) on the test set. For image classification, we train a EfficientNet-B3 on the ImageNet-1K benchmark. We report top-1 accuracy on the validation set. For both setting, we also report model size in megabyte (MB) and the compression ratio compared to the original model.

### 8.7.6 LayerDrop with STE

For quantization noise, we apply the straight through estimator (STE) to remaining weights in the backward pass. We experiment with applying STE to the backward pass of LayerDrop's pruning noise. Results are shown in Table 10 and find slightly worse results.

| Model | MB | PPL |
|---|---|---|
| Quant-Noise + Share + Prune | 10 | 24.2 |
| Quant-Noise + Share + Prune with STE | 10 | 24.5 |

Table 10: **Performance on Wikitext-103 when using STE in the backward pass of the LayerDrop pruning noise.**