Tiled Bit Networks: Sub-Bit Neural Network Compression Through Reuse of Learnable Binary Vectors

Matt Gorbett
matt.gorbett@colostate.edu
Colorado State University
Fort Collins, CO, USA

Hossein Shirazi
hshirazi@sdsu.edu
San Diego State University
San Diego, CA, USA

Indrakshi Ray
indrakshi.ray@colostate.edu
Colorado State University
Fort Collins, CO, USA

Abstract

Binary Neural Networks (BNNs) enable efficient deep learning by saving on storage and computational costs. However, as the size of neural networks continues to grow, meeting computational requirements remains a challenge. In this work, we propose a new form of quantization to tile neural network layers with sequences of bits to achieve sub-bit compression of binary-weighted neural networks. The method learns binary vectors (i.e. tiles) to populate each layer of a model via aggregation and reshaping operations. During inference, the method reuses a single tile per layer to represent the full tensor. We employ the approach to both fully-connected and convolutional layers, which make up the breadth of space in most neural architectures. Empirically, the approach achieves near full-precision performance on a diverse range of architectures (CNNs, Transformers, MLPs) and tasks (classification, segmentation, and time series forecasting) with up to an 8x reduction in size compared to binary-weighted models. We provide two implementations for Tiled Bit Networks: 1) we deploy the model to a microcontroller to assess its feasibility in resource-constrained environments, and 2) a GPU-compatible inference kernel to facilitate the reuse of a single tile per layer in memory.

CCS Concepts

• **Computing methodologies** → **Neural networks**.

Keywords

Neural Network Quantization, Compression, Efficiency, On-Device Machine Learning, Edge Machine Learning, IoT

ACM Reference Format:
Matt Gorbett, Hossein Shirazi, and Indrakshi Ray. 2024. Tiled Bit Networks: Sub-Bit Neural Network Compression Through Reuse of Learnable Binary Vectors. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (CIKM '24), October 21–25, 2024, Boise, ID, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3627673.3679603

1 Introduction

The progress of modern machine learning can be largely attributed to the exponential growth of Deep Neural Networks (DNNs). Empirically, the capacity of DNNs is expanding at an astounding rate [3], a practice supported by theory showing that sufficiently over-parameterized models are in fact necessary for deep learning [1, 24]. Alongside this progress, the growing presence of resource-constrained machines (e.g. embedded devices, cell phones) has created unique opportunities to deploy increasingly large DNNs in novel environments. Consequently, maximizing the computational efficiency of neural networks is a relevant challenge at various scales of application.

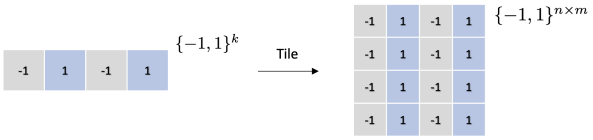

Figure 1: Tiling Illustration: A binary tile (left) of size $k = 4$ is replicated four time to create a weight matrix of size 16 (right). Tiling is used during the training process of Tiled Bit Networks (TBNs) to learn vectors for populating the parameters of a model (as illustrated above). During inference, only a single tile needs to be referenced per layer (left) – a specialized kernel can reuse the tile throughout layer computation for memory savings.

Efforts toward efficient deep learning span a broad range of techniques such as architectural design [23, 50], neural architecture search [33], knowledge distillation [22, 51], and quantization [6, 25, 62]. Quantization, which converts high precision neural network weights into discrete values, has achieved success in practical applications [40, 49], and has been applied down to the scale of Binary Neural Networks (BNNs) where the weights (and often activations) of a model are single bit values [7].

While BNNs have been established as a practical and extreme form of quantization, an emerging line of research has gone a step further with *sub-bit* neural network compression, which requires less than a single bit per model parameter. Wang et. al [58] first observed that the discrete set of binary convolutional kernels tend to cluster into a smaller subset; as a result, they devised a training regime to use a smaller set of kernels. Subsequent work has achieved improved compression by leveraging properties of binary convolutional kernels using minimum spanning trees [56] and sparse kernel selection [57].

Independent from previous approaches, this work proposes tiling neural networks with binary weights to achieve sub-bit memory and storage compression of model parameters. Tiled Bit Networks (TBNs) learn binary sequences (tiles) to fill in the weights of a DNNs layers during training. A tiling operation is depicted in Figure 1. The algorithm learns a condensed parameter representation for each layer by compressing the weight values using tensor reshaping and

CIKM '24, October 21–25, 2024, Boise, ID, USA

ACM ISBN 979-8-4007-0436-9/24/10
https://doi.org/10.1145/3627673.3679603

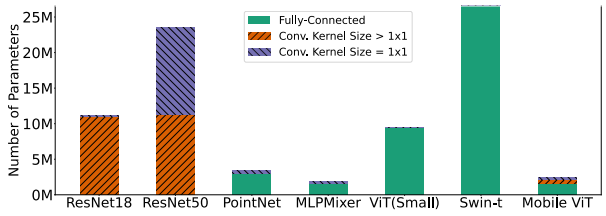

Figure 2: Composition of popular DNNs: The ResNet series is made up primarily of convolutional layers; MLPs (PointNet, MLPMixer) and Transformers (Swin-t, ViT, Mobile ViT) consist mostly of fully-connected parameters.

aggregation (Figure 3); a scalar is additionally applied to each layer or tile.

Unique from previous work that leverages the properties of convolutional kernels to achieve sub-bit compression, TBNs work on both fully-connected and convolutional layers, a relevant application for modern architectures as depicted in Figure 2. We test the approach on CNNs, Transformers, PointNets, and MLPs, enabling for the first time sub-bit DNN compression on models with high proportions of fully-connected parameters. Compared to previous approaches, TBNs achieve better or similar performance on image classification tasks. Empirically, across 2D and 3D image classification, TBNs achieve performance on par with Binary Weight Neural Networks (BWNNs) with only a fraction of the total parameters – less than a single bit is stored and accessed per model parameter. TBNs additionally achieve strong performance on semantic and part segmentation tasks as well as time series forecasting.

We provide two implementations for model inference in Section 5, which both require only a single tile per model layer in memory. First, we implement a lightweight TBN for deployment on a microcontroller, showing that the algorithm reduces memory and storage consumption compared to BWNNs. We also implement a GPU-compatible inference kernel using the Triton library [53], which allows memory savings through the reuse of a single tile in the highly parallelized setting. The TBN inference kernel requires 2.8x less peak memory (78.5MB vs. 222.5MB) compared to a standard kernel on ImageNet Vision Transformer (ViT) (Small) when both have full precision weights.

Our contributions are as follows:

- We achieve sub-bit memory and storage compression of neural network parameters by learning sequences of binary values (tiles) to populate the layers of DNNs. We apply the method to fully-connected and convolutional layers. To the best of our knowledge, this is the first work to show substantial sub-bit compression of fully-connected DNNs which are relevant in Transformers and MLPs (PointNet, MLPMixer).
- We provide two implementations that achieve sub-bit compression of model parameters by *reusing* a single tile per model layer: 1) we deploy a TBN to a microcontroller with a customized C kernel, and 2) we develop a specialized GPU kernel for fully-connected layers to leverage memory savings of tiled parameters during inference.

2 Related Work

Quantized and Binary Neural Networks DNN quantization reduces full-precision weights and activations to discrete and lower precision values to enhance model storage, memory, and inference speed [31, 64]. The most extreme quantization was conventionally thought to be binarization, where weights can only be ±1 [43]. Binarization helps reduce computation, however, it often reduces model accuracy. Several works attempt to alleviate this issue such as XNOR-Net, which used channel-wise scaling factors for BWNNs and BNNs [47]. IR-Net [44] preserved the information by maximizing the entropy of the information while minimizing the quantization error. ReActNet used generalized activation functions to get within 3% of full-precision accuracy on ImageNet [36]; Shang et al. utilized contrastive learning to learn BNNs [52]. Xu et al. proposed FDA, which estimates sign function gradients in a Fourier frequency domain [59]; Xu et al. proposed ReCU which introduces a rectified clamp unit to address dead weights [60].

We note that the BNN research covered in this section uses binary activations as well as binary weights, and as a result, achieves significant memory and speed improvements. TBNs use full-precision activations, however, still achieve storage and memory improvements from using a single tile per layer. We denote BNNs that have full-precision activations as Binary Weight Neural Networks (BWNNs). We indicate whether previous BNN algorithms have binary activation's in benchmark experiments in Section 4.

Sub-Bit Quantization Sub-bit DNN compression reduces model sizes to less than a single bit per model parameter. Kim et al. [26] proposed a kernel decomposition to reduce computations in binary CNNs. FleXOR [29] used an encryption technique to store binary sequences. Wang et al. [58] observed that the full set of binary convolutional kernels tends to cluster into a subset; they formulate a training technique to find the best subsets of kernels. Lan et al. [27] stack convolutional filters to achieve sub-bit compression. Wang et al. [57] group kernels into binary codebooks for sparse kernel selection. Finally, Vo et al. [56] propose minimum spanning tree compression, which takes advantage of the observation that output channels in binary convolutions can be computed using another output channel and XNOR operations.

Previous sub-bit compression approaches are distinct from TBNs: initial work was based on removing redundancy and encrypting weights; CNN-based approaches are based on utilizing properties of binary convolutional kernels. TBNs, on the other hand, achieve substantial compression on both fully-connected and convolutional layers, and can be applied to multiple architectures (CNNs, Transformers, MLPs).

Efficient Machine Learning Model quantization is a sub-field of efficient deep learning, which encompasses multiple areas not covered in this work such as low rank factorization [5, 28, 30], structured and unstructured pruning [11, 17, 19, 21], knowledge distillation [22, 48], and memory efficiency through input patching [32] and attention tiling [8].

Embedded and On-Device Machine Learning The size and computational requirements of DNNs has motivated researchers to improve the compatibility of large models with hardware such as mobile phones and embedded devices (e.g. FGPAs, IoT Sensors) [4]. Architectural optimizations such as MobileNet [50], ShuffleNet

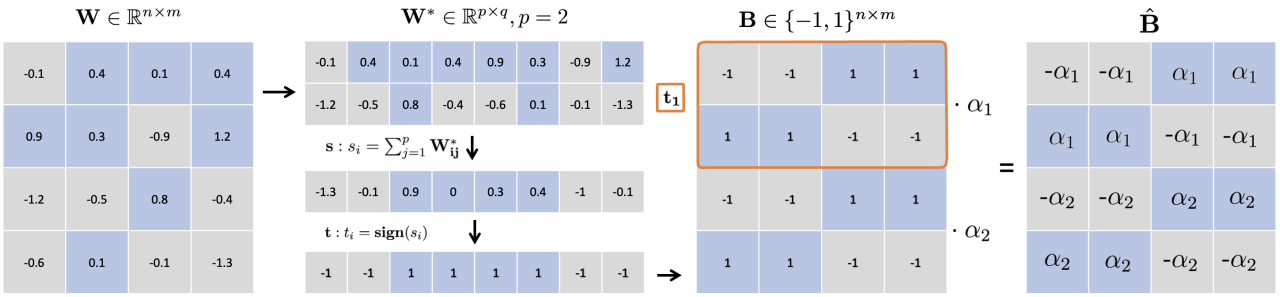

Figure 3: Tile Construction During Training: For each layer of a neural network, we train a standard weight tensor (W) (left). During the training, we compress the parameter by a factor of p by performing a reshaping (second column top) and then sum operation (second column middle). We use the straight-through estimator to binarize the vector s, creating the tile t (bottom of the second column). We next create binary weights B from the resulting binary vector by tiling vector t two times and reshaping it to an $n \times m$ tensor (third column). Finally, we apply a scalar α over each of the two tiles, resulting in the final weight tensor $\hat{\mathbf{B}}$. During the inference, only a single tile is needed, along with a small number of α scalars.

[37], and MCUNet [33, 34] have been achieved success, including to ease memory constraints via layer patching [32].

3 Method

Tiled Bit Networks are constructed from a standard neural network with layers $1, 2, l..., L$. We consider fully-connected and convolutional layers in this work since these layers generally make up the breadth of DNNs weights. We do not consider bias parameters in this work.

In this section, we describe the training process for TBNs, which involves learning full-precision parameters ($\mathbf{W}$) and applying aggregation and reshaping to create the tile vectors $\mathbf{t}$. We then describe our approach to tile-wise scaling, the second step of training TBNs. Finally, we describe training hyperparameters and their default settings.

Layer-Wise Tiling The key to our approach is that we learn tile vectors $\mathbf{t}^{[1]}, \mathbf{t}^{[l]}, ..., \mathbf{t}^{[L]}$ for each layer of our network. We initialize our model with full-precision values for each layer similar to standard training, creating a weight tensor $\mathbf{W}^{[l]} \in \mathbb{R}^{d_1 \times ... d_k}$ for layer l, where d_k is the dimensionality of the tensor (e.g., a fully-connected layer has $k = 2$). The total of elements in the tensor is $N = \prod_{i=1}^{k} d_i$. During training, we update $\mathbf{W}^{[l]}$ via stochastic gradient descent. Our goal is to *compress* $\mathbf{W}^{[l]}$ by a factor of p, where size N is divisible by p such that $p \times q = N$. To achieve this we reshape tensor $\mathbf{W}^{[l]}$ as a $p \times q$ dimensional matrix $\mathbf{W}^{[l]*}$ during forward propagation:

$$\mathbf{W}^{[l]} \in \mathbb{R}^{d_1, ... d_k} \rightarrow \mathbf{W}^{[l]*} \in \mathbb{R}^{p \times q} \tag{1}$$

We then sum the reshaped weight tensor $\mathbf{W}^{[l]*}$ along the p dimension to create a vector $\mathbf{s} \in \mathbb{R}^q$:

$$\mathbf{s} = \begin{bmatrix} \sum_{j=1}^{p} \mathbf{W}^{[l]*}_{\mathbf{1j}} \\ \sum_{j=1}^{p} \mathbf{W}^{[l]*}_{\mathbf{2j}} \\ \vdots \\ \sum_{j=1}^{p} \mathbf{W}^{[l]*}_{\mathbf{qj}} \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_q \end{bmatrix} \tag{2}$$

We next create tile $\mathbf{t}^{[l]} = [t_1, t_2, t_i ... t_q]$ for a given layer by applying a threshold function to determine the binary value for each s_i in $\mathbf{s}$:

$$t_i = \begin{cases} 1, & \text{if } s_i > 0 \\ -1, & \text{otherwise} \end{cases} \tag{3}$$

Tile $\mathbf{t}^{[l]}$ is then replicated p times to create tile vector $\mathbf{b}^{[l]} \in \mathbb{R}^N$. Formally, let $\mathbf{1_N}$ be a vector of ones with size N. The tiling operation creates vector $\mathbf{b}^{[l]}$ as:

$$\mathbf{b}^{[l]} = \mathbf{1_N} \otimes \mathbf{t}^{[l]} \tag{4}$$

where $\otimes$ is the Kronecker product. We create our final binary weight tensor $\mathbf{B}^{[l]} \in \{-1, 1\}^{d_1, ... d_k}$ by reshaping vector $\mathbf{b}^{[l]}$:

$$\mathbf{B}^{[l]} = \mathbf{vec}^{-1}_{\mathbf{d_1, ... d_k}}(\mathbf{b}^{[l]}) \tag{5}$$

where $\mathbf{vec}^{-1}_{\mathbf{d_1, ... d_k}}(\cdot)$ denotes a vector to k-dimensional tensor operation.

We note that computing binary parameters $\mathbf{B}^{[l]}$ involves non-differentiable operations during forward propagation. As a result, we utilize straight-through gradient estimation, where the gradients of the model are passed-through the non-differentiable operator during backpropagation [2]. To achieve this we implement Equations (1) to (5) in the forward pass of a customized differentiation engine, and on backpropagation we pass the gradients through the customized module to update $\mathbf{W}^{[l]}$.

Putting it together, the tiled model $f(\cdot)$ can be trained with parameters $\mathbf{W}^{[l]}$ (to compute $\mathbf{B}^{[l]}$) and inputs x, producing an output y which serves as a continuous, differentiable approximation of a tiled neural network. In the context of straight-through gradient estimation, y is used during backpropagation to compute the gradient of loss $\mathcal{L}$ with respect to the parameter $\mathbf{W}^{[l]}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} = \frac{\partial \mathcal{L}}{\partial y^{[l]}} \cdot \frac{\partial y^{[l]}}{\partial \mathbf{W}^{[l]}}, \quad \frac{\partial y^{[l]}}{\partial \mathbf{W}^{[l]}} \approx \frac{\partial y^{[l]}}{\partial \mathbf{B}^{[l]}} \tag{6}$$

where $y^{[l]}$ is the output of layer l prior to the activation. $\frac{\partial y^{[l]}}{\partial \mathbf{B}^{[l]}}$ involves the thresholding, tiling, and reshaping operations.

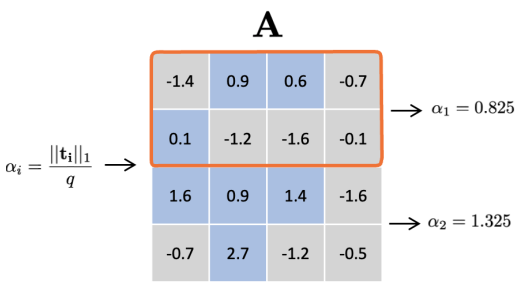

Figure 4: We learn scalar α from tensor A by computing Equation 7 or 9 over its values (W can also be used in place of A). The figure visualizes Equation 9 which calculates α over each tile.

Tile-wise Scalars Similar to XNORNet [47], we scale $\mathbf{B}^{[l]}$ by α. Rastegari et al. [47] derived the optimal scaling factor of a binary weight filter as the average absolute value of a weight, a method widely used in other research [9, 38, 43]. We can use parameter $\mathbf{W}^{[l]}$ to compute the scalar since its non-aggregated size is the same as a standard weight tensor:

$$\alpha = \frac{||\mathbf{W}^{[l]}||_1}{N} \tag{7}$$

We additionally experiment with an independent tensor, denoted $\mathbf{A}^{[l]} \in \mathbb{R}^{d_1 \times ... d_k}$, to exclusively compute the α scalar. We observe a slight performance benefit from using $\mathbf{A}^{[l]}$ in addition to $\mathbf{W}^{[l]}$. We add this option as a hyperparameter to our models.

Another hyperparameter setting for TBNs calculates one α for each tile $\mathbf{t_1}, \mathbf{t_2}...\mathbf{t_p}$ in layer l by utilizing the i^{th} set of the flattened tensor $\mathbf{A}^{[l]}$ (or $\mathbf{W}^{[l]}$), and calculating α_i using only these values. This represents the optimal scalar for that particular tile. To do this, we can reshape $\mathbf{A}^{[l]}$ to get the values corresponding to the p^{th} tile of layer l:

$$\mathbf{A}^{[l]} \in \mathbb{R}^{d_1,...d_k} \rightarrow \mathbf{A}^{[l]*} \in \mathbb{R}^{q \times p} \tag{8}$$

Next, similar to Equation 7 we calculate the 1-norm for each segment of values corresponding to the i^{th} tile in $\mathbf{A}^{[l]}$. We divide this number by q (the size of each tile) to give us $\alpha_1, \alpha_2, ...\alpha_p$:

$$\begin{bmatrix} \alpha_1^{[l]} \\ \vdots \\ \alpha_p^{[l]} \end{bmatrix} = \begin{bmatrix} \frac{||\mathbf{A_1}^{[l]*}||_1}{q} \\ \vdots \\ \frac{||\mathbf{A_p}^{[l]*}||_1}{q} \end{bmatrix} \tag{9}$$

Each α gets multiplied element-wise by its corresponding tile in $\mathbf{B}^{[l]}$. The resulting tensor, $\hat{\mathbf{B}}^{[l]}$ is used for the operation on the inputs. Equation 9 is depicted in Figure 4.

After training is complete, we save a vector of size q for each layer along with full-precision scalars (αs). We describe our implementation in Section 5.

Hyperparameter Settings We test our models with several hyperparameter configurations to assure the best performance. TBNs primarily contains three hyperparameters:

(1) Minimum layer size for tiling, λ. We set a minimum size N of a DNN layer required for tiling to be performed. Tiling smaller layers causes a drop in performance (Figure 7). **Default:** λ=64,000, ImageNet models: λ=150,000, Time-Series models: λ=32,000
(2) Parameter **W** for tiling and **A** for calculating α. **W** is used to learn a tile for each layer; it can also be used to calculate α scalars. Alternatively, we propose a separate parameter **A** to compute αs independently, which exhibits a small performance gain. **Default:** **A** for calculating α. For ImageNet, we use **W**.
(3) Tile-wise αs. We experiment with calculating a single α per layer as well as calculating α for each tile in a layer. In some settings multiple αs perform better. **Default:** Single α per layer.

4 Experiments

In this section we detail our experiments across a range of architectures, datasets, and tasks. We test TBNs on CNNs as well as fully-connected models such as PointNet and Transformers.

4.1 CNN Architectures

We next compare TBNs against previous sub-bit compression techniques for Convolutional Neural Networks (CNNs) including SNN [58], MST [56], and Spark [57]. CNN's are the only models which have achieved sub-bit compression in previous research and thus they are a strong choice for benchmarking TBNs. We note that TBNs also work on Transformers and MLPs, which we experiment with in the next sections.

We assess TBNs on both CIFAR-10 and ImageNet datasets using the ResNet18/34 models [20], similar to previous works. In addition, we include ResNet50 for CIFAR-10, which has 1×1 convolutional kernels. The SNN sub-bit compression algorithm was assessed with Resnet50 using a modified kernel selection technique specialized for 1×1 convolutions, enabling up to 8x compression. The technique, from Section C of the Appendix in [58], is the most similar approach to TBNs.

In Table 1 we compare the performance of TBNs to previous approaches using bit-width, number of parameters, and test set accuracy. Bit-width measures the number of bits per model parameter, with a full precision model having 32-bits per parameter and a binary model having 1-bit per parameter. In blue, we include the savings of TBNs compared to binary neural networks. In the next we column we denote the number of parameters required to save each model for inference, and finally the test set accuracy is the percentage of correct predictions by the model. Unique to the CNN architecture, we additionally include the number of bit operations required for each model in the results section.

Results. Table 1 highlights the results of TBNs compared to previous sub-bit compression techniques on the CIFAR-10 and ImageNet datasets. For CIFAR-10, we achieve sub-bit compression across ResNet architectures without a decrease in test performance at 4x compression. Experiments are run three times each and averaged. Compared to other methods, TBNs achieves a competitive performance with MST, the current state-of-the-art method for sub-bit compression. TBNs achieve similar performance at the same compression rates of MST for the CIFAR-10 models. For ImageNet,

TBNs achieves enhanced performance roughly 2x compression, with test accuracy within 1.5% of the performance of the binary-weighted IR-Net.

Model	Method	Bit-Width (Savings)	#Params (M-Bit)	Test Acc. (%)
CIFAR-10				
ResNet 18	Full-Precision	32	351.54	93.1
	IR-Net	1	10.99	92.9
	SNN	0.440 (2.3x)	4.88	92.1
	Sparks*	0.440 (2.3x)	4.88	90.8
	MST*	0.075 (13.3x)	0.81	91.6
	TBN_4	0.256 (3.9x)	2.85	93.1
	TBN_8	0.131 (7.7x)	1.46	92.4
	TBN_{16}	0.069 (14.5x)	0.77	91.2
ResNet 50	Full-Precision	32	750.26	95.4
	IR-Net	1	23.45	93.2
	SNN	0.35 (2.8x)	8.32	94.0
	TBN_4	0.259 (3.9x)	6.10	94.9
	TBN_8	0.136 (7.4x)	3.21	94.3
	TBN_{16}	0.075 (13.3x)	1.76	93.5
VGG Small	Full-Precision	32	146.24	92.7
	IR-Net	1	4.656	91.3
	SNN	0.440 (2.3x)	2.032	91.9
	Spark*	0.440 (2.3x)	2.032	90.8
	TBN_4	0.288 (3.5x)	1.340	92.6
	TBN_8	0.131 (7.7x)	0.722	91.5
	TBN_{16}	0.117 (8.6x)	0.520	90.2
ImageNet				
ResNet 34	Full-Precision	32	674.88	73.1
	IR-Net	1	21.09	70.4
	SNN	0.560 (1.8x)	11.71	66.9
	MST*	0.450 (2.2x)	9.51	65.4
	Sparks*	0.560 (1.9x)	11.71	67.6
	TBN_2	0.53 (1.9x)	11.13	68.9

Table 1: CNN Results on CIFAR-10 and ImageNet: * indicates model with binary activations. We denote the tiling compression p of each experiment as TBN_p. Savings (in blue) indicates the compression from a binary-weight model (1-bit per model parameter).

Dataset	Dataset	Full Precision	IR-Net	TBN
CIFAR-10	ResNet18	35.03	0.547	0.082 (6.7x)
	ResNet50	78.12	1.22	0.155 (7.9x)
ImageNet	ResNet34	225.66	3.526	0.58 (6.1x)

Table 2: Bit-Ops of ResNet architectures trained with a) full precision, b) IR-Net, and c) TBN. CIFAR-10 TBNs have $p = 4$, and ImageNet TBNs has $p = 2$.

In addition to the reduction in bit-width and parameter count, TBNs trained with our default training configurations (single α, no bias parameters) create replicated convolutional channels. For example a 2d convolutional layer with one input channel, two output channels, 3×3 kernel size, and a tile compression rate of $p = 2$ will create two identical output channels. This means that during inference TBNs lead to a substantial reduction in the number of required convolutional computations – only one of the tile computations need to be executed, and we can replicate output channels from the other tiles. Table 2 summarizes the bit-ops saving achieved by TBNs compared to both full precision on binary neural networks.

4.2 MLP-Based Architectures

In addition to CNN architectures, we consider MLP models which contain a high proportion of fully-connected and 1×1 convolutional layers. PointNet is a well-established model for unified tasks in classification, part segmentation, and semantic segmentation [42]. The model takes point cloud data from 3D representations. To assess PointNet we use datasets ModelNet40, Shapenet, and S3DIS, which are each designed for a specific task. We denote segmentation performance with Intersection over Union (IoU), and class average IoU. IoU is the ratio of the intersection area between the predicted and ground truth regions to the union area of both regions. Average IoU is calculated across all instances or regions in the dataset, whereas class average IoU is calculated separately for each class and then averaged across all classes.

Algorithm	Bit-Width (Params)	#Params (M-Bit)	Test Acc. (%)
Task: Classification, Dataset: ModelNet40			
Full-Precision	32	111.28	90.30
FDA*	1	3.48	81.87
BWNN	1	3.48	89.20
TBN_4	0.259 (3.9x)	0.90	88.67
TBN_8	0.136 (7.4x)	0.47	87.20

Algorithm	Bit-Width (Params)	#Params (M-Bit)	Instance Avg. IoU	Class Avg. IoU
Task: Part Segmentation, Dataset: ShapeNet				
Full-Precision	32	266.96	83.06	77.43
XNOR-Net*	1	8.34	-	60.87
BWNN	1	8.34	76.1	69.90
TBN_4	0.340 (2.9x)	2.68	76.3	70.20
TBN_8	0.207 (4.8x)	1.73	75.1	68.90

Algorithm	Bit-Width (Params)	#Params (M-Bit)	Test Acc.	Class Avg. IoU
Task: Semantic Segmentation, Dataset: S3DIS				
Full-Precision	32	112.96	78.27	42.20
BWNNs	1	3.53	69.50	31.30
TBN_4	0.431 (2.3x)	1.52	67.55	31.10
TBN_8	0.337 (3.0x)	1.19	65.70	29.55

Table 3: PointNet Results: We test TBNs on the fully-connected PointNet model. TBNs achieve performance close to the full-precision model on the classification benchmark and within 10% of full-precision performance on the Part Segmentation task. * indicates model results from BiBench with binary activations. We take BiBench binarization algorithm with the best results for both ModelNet40 and ShapeNet.

We derive MLP experiments from BiBench [45], who provide a diverse set of tasks to evaluate BNNs. We note that the benchmarks provided in BiBench assess binarizing pretrained models. Additionally BiBench only assesses models with binary weights and activations. In Table 3 we denote the best algorithms from BiBench. We additionally train a BWNN for each task for a stronger comparison.

Results. In Table 3 we summarize the PointNet model results across classification (ModelNet40), part segmentation (ShapeNet), and semantic segmentation (S3DIS). We find that classification performance is almost on par with the full-precision model, with less than a 2% drop in accuracy at 4x compression. For both part and semantic segmentation, TBNs exhibit some loss in accuracy compared to their full-precision counterpart. However, we note that in both cases TBNs perform on par with BWNNs. TBNs also significantly outperform XNOR-Net in part segmentation, the most successful BNN on the task in BiBench.

4.3 Transformers

In our next set of experiments, we assess TBNs on Transformers. It was noted in the BNN benchmark work BiBench [45] that Transformers perform poorly on BNNs. In particular, none of the binarization algorithms tested in BiBench achieved more than 70% of the performance of full-precision models. The research, which looks exclusively at models with binary weights *and* activations, noted that binarization of activations greatly affects the attention mechanism and leads to poor quality models. The equations between the query key, and value cause amplified information loss when model activations are binarized. In contrast to BiBench, we assess models with binary weights and full-precision activations which do not suffer from the same information loss as models with binary activations.

Model	Method	Bit-Width (Params)	#Params (M-Bit)	Test Acc. (%)
CIFAR-10				
ViT	Full-Precision	32	303.68	82.5
	BWNN	1	9.50	82.2
	TBN_4	0.253 (4.0x)	2.40	82.7
	TBN_8	0.129 (7.8x)	1.22	82.1
Swin-t	Full-Precision	32	851.14	86.8
	BWNN	1	26.60	85.8
	TBN_4	0.259 (3.9x)	6.88	85.8
	TBN_8	0.135 (7.4x)	3.61	84.6
ImageNet				
Swin-t	Full-Precision	32	873.60	81.3
	TBN_2	0.534 (1.9x)	14.7	77.3

Table 4: Vision Transformers trained on CIFAR-10 and ImageNet: We compare the performance of a ViT (patch size 4) and the Swin-t model with TBN_4 and TBN_8 variations for CIFAR-10 and TBN_2 for ImageNet. † The Swin-t TBN_2 ImageNet model was trained for 200 epochs with basic data augmentations.

We test Transformer TBNs on three tasks: 1) Vision Transformers, 2) Small NLP Transformers (encoders), and 3) time-series Transformers (encoders).

4.3.1 Vision Transformers. For Vision Transformers We train a ViT [10], which uses image patching along with the traditional attention, and Swin-t [35], which uses hierarchical partitioning of the image into patches with merging. We train the models from scratch on the CIFAR-10 and ImageNet datasets. Note that, training the models from scratch, rather than fine-tuning, can result in sub-optimal performance in Transformers [10].

Results. In Table 4 we summarize the results of ViTs trained from scratch. TBNs achieve performance within 2.5% of full-precision models across all CIFAR-10 experiments. Specifically, the TBN ViT can closely match the accuracy of the full-precision model at both 4x and 8x compression rates, while Swin-t has a performance degradation of just 1.2% at 4x and 2.5% at 8x. For ImageNet, the Swin-t TBN quickly converged to a stronger accuracy than the ResNet34 ImageNet model in just 200 epochs. Moreover, we only trained the model with basic data augmentations. In contrast, the full-precision Swin-t model was trained for 300 epochs with enhanced data augmentation techniques.

4.3.2 Time Series Transformers. In our next set of experiments we explore Transformer TBNs for multivariate time series forecasting. Transformer encoders have been shown to be robust to model compression for time series learning tasks including forecasting [17]; we aim to assess whether we can compress similar models using TBNs.

Our experiments utilize the ECL (Electricity) dataset and the Weather dataset similar to previous works [17, 63]. ECL has 321 features and Weather has 7 features. We use a Transformer encoder similar to previous work [61], using a dimensionality of 512 for the Electricity dataset and 128 for the Weather dataset. Similar to previous work, we assess the performance of the models use Mean Squared Error.

Results. We train each of the models five times and report the average MSE along with the standard deviation in Table 5. Notably, TBNs performed slightly better than full precision and binary models on the electricity dataset. On the weather dataset, we set λ=32,000 since all the layers are smaller than this. We find that the model is able to converge to full precision accuracy. However, when we lower λ to 16,000 model performance suffers.

Dataset	Method	Bit-Width (Params)	#Params (M-Bit)	MSE
Electricity	Full-Precision	32	145.2	0.212±0.10
	BWNN	1	4.5	0.210±0.05
	TBN_4	0.25 (4.0x)	1.1	0.209±0.13
Weather	Full-Precision	32	11.8	0.165±0.10
	BWNN	1	0.368	0.165±0.02
	TBN_4	0.54 (1.9x)	0.197	0.168±0.12

Table 5: Multivariate Time Series Forecasting: We compare TBNs to full precision neural networks and binary weighted neural networks (BWNNs) on two multivariate time series datasets. We find that TBNs achieve similar performance to larger models on single-step forecasting tasks.

5 Implementation

We cover two implementations of TBNs for model inference: 1) a C implementation which we deploy on a microcontroller, and 2) GPU-compatible inference kernels. Both methods implement *reuse* of a single tile per layer to achieve memory and storage savings.

5.1 Microcontroller Deployment

We first implement an inference engine to run on an Arduino microcontroller. The microcontroller has 1MB of storage and 250KB of memory, making it practical for a lightweight model. We program an MLP model trained on the MNIST dataset with 128 hidden neurons and a fused ReLU nonlinearity. We implement both a standard BWNN and a TBN. Our TBN has a compression rate of 4 and uses multiple αs (one for each tile). To implement the TBN we first train our model in PyTorch [41], and then convert the layer tiles and α scalars to C data types. We implement a fully-connected kernel in C as detailed in Algorithm 1. We develop a fully binarized kernel by packing binary weights into unsigned 8-bit integers and use bit-masking to extract the correct values during inference.

We assess the speed, memory, and storage space of each model. To assess speed, we report the Frames Per Second (FPS), which measures the number of times the program can execute on a sample per second. We measure FPS using a provided script which executes the compiled model 1000 times and reports the mean and standard deviation across five runs. We report memory as the maximum memory at any layer of the model. Finally, we report the storage space as the number of bits stored for each model. We expect speed to be the same across models, while memory and storage space should improve with TBNs.

Results. We summarize the results of the implementation across speed, memory, and storage space in Table 6. As expected, the speed of both models is roughly the same. The max memory usage, which happens during execution of the first fully-connected layer, is 58% less for the TBNs compared to the BWNN. Both models still require full-precision inputs and a full precision placeholder for the output, while the TBNs model requires roughly $\frac{1}{4}$ of the weights loaded in memory compared to the BWNN. Finally, storage for the TBN model is almost $\frac{1}{4}$ that of the BWNN. Since the classification layer only contains 1280 parameters, it is not tiled, accounting for more parameters in the tiled model.

Model	Speed (FPS)	Max Memory Usage (KB)	Storage (KB)
BWNN	704.5±3.3	16.20	12.70
TBN_4	705.1±3.6	6.80	3.32

Table 6: We compare the performance of a binary-weight neural network (BWNN) and a TBN deployed on a microcontroller. We implement an MLP with one hidden layer (size 128). The maximum memory usage corresponds to the full-precision image being processed by the first fully-connected layer, with additional memory allocated for the activation's.

5.2 GPU Inference Kernel

Our second implementation utilizes the open-source Triton library, which enables us to write customized CUDA kernels in Python to run on GPUs. Native PyTorch does not allow the reuse of a single tile without allocating new memory for the full layers parameters (p tiles) – memory reuse between tensors with incompatible strides is not possible in standard operations. Leveraging Triton provides more control over lower-level memory allocation.

We implement fully-connected kernels using both full-precision (32-bit) weights as well as binary (1-bit) weights. For the former use case, we experiment tiling layers with full-precision weights to compare against standard 32-bit models. For the latter, we pack bit parameters into unsigned 8-bit integers, and unpack them during inference. Bit values are packed by *row*, converting an $m \times n$ matrix to size $(\frac{m}{8}) \times n$.

We implement both kernels using the matrix multiplication functionality provided by Triton. It uses block-level matrix multiplication and has a similar performance to the optimized cuBLAS library. Our tiling implementation converts an $m \times n$ matrix to $m \times q$ (we compress the second dimension). For pointer arithmetic, we reuse the same $m \times q$ tile for multiple computations. In other words, fully-connected tiling is a *matrix-to-matrix* implementation rather than *matrix-to-vector* implementation.

Algorithm 1 FC Layer with Tiling, Many αs, Forward Pass

Inputs: Tile vector $\mathbf{t}$ of size q, input vector $\mathbf{x}$ of size n, layer size metadata (m, n), $\alpha_1, \alpha_2, \ldots, \alpha_p$
$t_i \leftarrow 0$, $\alpha_i \leftarrow \alpha_1$
for $i \leftarrow 1$ to m **do**
  $\mathbf{y}[i] \leftarrow 0$
  $\alpha_i \leftarrow \alpha_i + 1$
  for $j \leftarrow 1$ to n **do**
    $\mathbf{y}[i] \leftarrow \mathbf{y}[i] + \mathbf{t}[t_i] \cdot \mathbf{x}[j] \cdot \alpha_i$
    if $t_i = q$ **then**
      $t_i \leftarrow 0$ { Move to beginning of tile vector}
      $\alpha_i \leftarrow \alpha_{i+1}$ { Get next tiles α}
    else
      $t_i \leftarrow t_i + 1$ { Increment tile index}
    end if
  end for
  $\mathbf{y}[i] \leftarrow \max(0, \mathbf{y}[i])$ {Fused ReLU}
end for
Output: Output vector $\mathbf{y}$ of size m

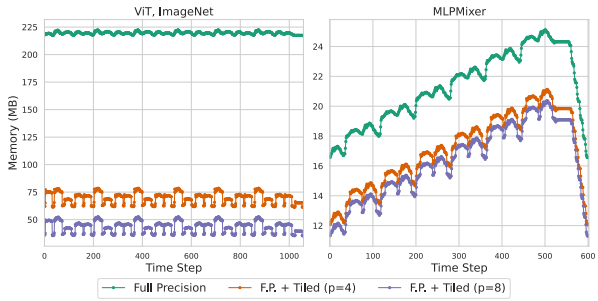

Figure 5: GPU memory allocated during model inference: We profile the memory of the ImageNet ViT (left) and PointNet (right) during inference using a customized GPU kernel with full-precision weights. The x-axis represents memory recorded within intermediate model layers during execution of a PyTorch model. The tiled kernel achieves 2.8x memory reduction on the ViT and 1.2x reduction on PointNet.

Model	Peak Memory (MB)	Parameter Mem. (MB)	% Param. Mem. (MB)
Full Precision	222.5	208.0	93.5%
FP, Tiled$_4$	78.5 (2.8x)	52.0 (4.0x)	66.2%
BWNN	18.4	6.5	35.3%
TBN$_4$	13.4 (1.4x)	1.6 (4.1x)	11.9%

Table 7: Memory assessment during inference on ImageNet ViT: Tiled kernels reduce the memory occupied by model weights, leading to a lower overall memory footprint. The final column is the % of the peak memory used by weights, which is substantially lower for TBNs.

Results. We measure the GPU memory usage for inference on a single image in Figure 5 and Table 7. Figure 5 profiles GPU memory usage through each layer of the model, while Table 7 examines the peak memory and memory occupied by model parameters in an ImageNet ViT.

The results in Figure 5 reflect the memory savings by only loading a *single* tile per model layer for the standard full-precision kernel as well as the tiled full-precision kernel. Specifically, during inference the PyTorch library loads weights for *all* layers into memory. It allocates new memory for the input and output activations of each layer, and deallocates the activations when they are no longer needed. For example, ViT has six attention layers with roughly 8.4 million parameters each (Q, K, V and the feed-forward parameters) for a total of 54.6 million parameters. The input and output activations to the attention layers have a size of 65,536. As a result, we can see the small spikes in the graph, which represent the allocation and de-allocation of activations. However, most of the memory is a result of the weights (34MB per attention layer).

Table 7 examines the memory savings of TBNs on both peak memory and memory allocated by model weights for the ImageNet ViT. We observe a 2.8x reduction in peak memory for the full-precision kernel and 1.4x reduction in peak memory for a binary kernel. When assessing the memory occupied specifically by model weights, we observe that TBNs reduce memory by 4x, and as a result require a substantially smaller proportion of peak memory compared to their standard counterparts: in the binary weight setting, TBN weights occupy just 11.9% of the total peak memory.

6 Ablation Studies

We perform ablation study's on TBNs, first exploring the effect of layer size on MLPMixers and ConvMixers, and then analyzing the effects of hyperparameters in MLPMixers and ResNets.

6.1 Effects of Layer Size

We first look at MLPMixers [54] and ConvMixers [55] for the CIFAR-10 classification task. ConvMixer has shown potential to compete with ViTs on complex tasks, while MLPMixers provide us with another opportunity to test fully-connected models.

In Figure 6 we visualize the performance of TBNs at tiling compression rates up to 32x for both models. We find that the ConvMixer accuracy degrades quickly after 4x compression. When analyzing the architecture, we find that the largest layer has just 65,536 parameters. Moreover, many layers have less than 65k parameters and don't fulfill the minimum layer size for compression (λ), resulting in limited reductions in parameter count along with high performance degradation. MLPMixer, on the other hand, has layers with 131k elements, and resulted in a more modest performance degradation at higher compression.

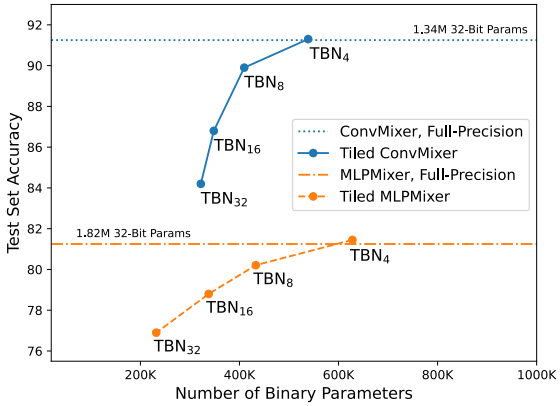

Figure 6: TBNs are vulnerable in models with low width layers: We plot the performance of the ConvMixer and MLP-Mixer at various compression rates/number of parameters compared to its full-precision baseline (horizontal line). The ConvMixer accuracy degrades quickly as a result of smaller layers: its maximum layer size is 65k. The MLPMixer has layer sizes of 131k. Both models achieve near full-precision performance at 4x compression, and degrade at varying rates thereafter.

6.2 Hyperparameter Configuration Analysis

We study the effects of three hyperparameters of TBNs as described in Section 3: 1) λ (minimum layer size for tiling), 2) **W** for tiling and **A** for computing α, and 3) Multiple versus single α scalars. For λ, we test global tiling, where all layers are compressed. We compare this to our default training which sets λ to 64k. For the

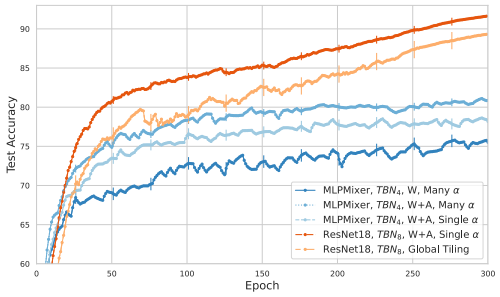

Figure 7: Hyperparameter Configurations: We perform an ablation study to the assess the test set accuracy across training for the MLPMixer and ResNet18 models with various hyperparameter configurations. For ResNet18, we show how tiling every convolutional layer, rather than layers of a certain size, leads to performance loss (red/orange). In blue we show the effects of using a separate parameter A to calculate α compared to calculating α using just W. We additionally show the benefit of using multiple αs (one per tile) rather than a single α.

second hyperparameter, we test two settings: the first uses **W** for both learning the binary tile and calculating α; the second uses parameter **A** for calculating α. We denote this setting as **W** or **W + A**. For the third parameter, we test a single α per layer and compare it to computing α for each tile.

Figure 7 shows the effects of the different hyperparameters on both the ResNet50 and MLPMixer models on the test accuracy throughout training. We show that a global tiling factor on ResNet50 causes a decrease in performance. Next, we find the model converges best when given a separate parameter **A** to calculate α. Finally, an α for each tile leads to a small increase in performance.

7 Discussion

We propose Tiled Bit Networks for learning binary vectors to fill in the weights of a neural networks layers during training, achieving sub-bit compression on model parameters during inference. TBNs work on both convolutional and fully-connected layers, and are applicable to CNNs, MLPs, and Transformer-based architectures.

Tiled Bit Networks have the potential to democratize deep learning by providing resource constrained devices access to larger scale models. For example, TBNs compress the 23.5 million parameter ResNet50 model to under 6.1 million bits, a size which can fit into a microcontroller with 1MB of storage. Moreover, the algorithm unlocks potential for stronger sub-bit compression on other architectures, particularly those with fully-connected layers. The innovation not only increases the accessibility of deep learning use but also has the potential to contribute to environmental sustainability by reducing the carbon footprint associated with model size and computational complexity.

Future Work A natural direction for future work is to test the approach on larger models (e.g. LLMs), where additional algorithmic modifications (such as full-precision parameter tiling) may enhance performance. Additionally, specialized kernels could be implemented to maximize the efficiency of TBNs with regards to parallelization, including convolutional kernels and kernels better optimized for the reusability of tiles. In addition to direct algorithmic enhancements, TBNs could also be assessed in parallel fields of machine learning such as adversarial detection [12, 13], dataset complexity and [14, 15, 18], and federated learning [16, 39].

Acknowledgements

This work was supported in part by funding from NIST under Award Number 60NANB23D152 and NSF under Award Numbers CNS 2335687, ATD 2123761, CNS 1822118, NIST, ARL, Statnett, AMI, NewPush, and Cyber Risk Research.

Appendix

A Training Details

CNN's trained with CIFAR-10 ResNet and VGG models are trained with similar hyperparameters on the CIFAR-10 dataset. They use an SGD optimizer with learning rate of 0.1, weight decay 0.0001, and momentum 0.9. Batch size is 128. We use a cosine learning rate policy. Layers are created with kaiming normal initialization with scale fan. We additionally use standard batch normalization. We standardize the dataset. We initialize **W** and **A** with different seeds for each run. TBN hyperparameters: We use multiple α's by default, **W**+**A**. Our CIFAR-10 and ImageNet models are based off of the Edge-Popup repository.

CNN's trained with ImageNet We train ResNet on the ImageNet dataset using TBNs. We use default hyperparameters from Edge-Popup [46]. We train our model for 600 epochs, using an SGD optimizer, learning rate of 0.256, a weight decay of 3e-5, momentum of 0.875, and batch size 128. We use a 0.1 value for label smoothing. We additionally use a learning rate warmup length of 5.

PointNet Classification For our three PointNet tasks, we use an existing repository which you can find here. We use a batch size of 24, and train the model for 500 epochs. An Adam optimizer is used with a learning rate of 0.001. We use a decay rate of 1e-4.

PointNet Part Segmentation We train our model for 250 epochs using a batch size of 16. We use an Adam optimizer with a learning rate of 0.001 and a decay rate of 0.0001. We adjust the learning rate by 0.5 every 20 steps. We use 2048 points for the ShapeNet dataset.

PointNet Semantic Segmentation We train our model for 32 epochs with batch size 16. We use Adam with a learning rate 0.001, and a weight decay of 0.0001. We use 4096 points for the S3DIS dataset. We decay the learning rate by 0.7 every 10 steps.

Transformers and Mixer Models trained with CIFAR-10 We use an Adam optimizer with a learning rate of 0.001 and batch size 128. We train for 500 epochs. We use random cropping, random horizontal flip, and standard normalization for CIFAR-10. MLPMixer has a depth of 6, 512 output size, and a patch size of 4. Our Vision Transformer uses a patch size of 4, head of size 512, 8 heads, an MLP head of 512, and an embedding dropout of 0.1. For ConvMixer we use a convolutional kernel of 8, dimension of 256, and depth 16.

Swin-t with ImageNet We mostly follow the same training protocol as the original paper except with more epochs. We train for 600 epochs, using an AdamW optimizer, a cosine decay learning rate scheduler, and 20 epochs of linear warm-up. We use a batch size of 350, an initial learning rate of 0.001, and a weight decay of 0.05. We use the same data augmentations as ResNet models.

B Additional Ablation Study

We include an additional ablation in Figure 8. We look at the ResNet50 model with different configurations such as global tiling versus minimum layer size tiling, **W** for calculating α versus **W** for tiling and **A** for calculating α, and multiple versus one α parameter.

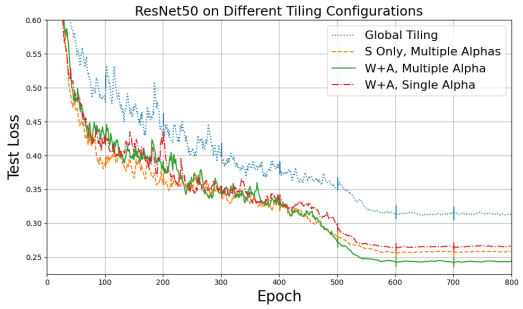

Figure 8: Test Loss on various ResNet-50 tiling configurations with compression rate 4. Using minimum layer size for compression, multiple αs, and a separate parameter A for calculating αs performed the best (green line). However, the only clear performance benefit across hyperparameters came from using a minimum layer size for tiling λ.

References

[1] Zeyuan Allen-Zhu, Yuanzhi Li, and Yingyu Liang. 2019. Learning and generalization in overparameterized neural networks, going beyond two layers. *Advances in neural information processing systems* 32 (2019).

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

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017).

[5] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. 2020. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794* (2020).

[6] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. 2019. Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, 3009–3018.

[7] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[9] James Diffenderfer and Bhavya Kailkhura. 2020. Multi-Prize Lottery Ticket Hypothesis: Finding Accurate Binary Neural Networks by Pruning A Randomly Weighted Network. In *International Conference on Learning Representations*.

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[11] Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635* (2018).

[12] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[13] Matt Gorbett and Nathaniel Blanchard. 2022. Utilizing network features to detect erroneous inputs. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 34–43.

[14] Matt Gorbett, Hossein Shirazi, and Indrakshi Ray. 2022. Local intrinsic dimensionality of iot networks for unsupervised intrusion detection. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 143–161.

[15] Matt Gorbett, Hossein Shirazi, and Indrakshi Ray. 2022. WiP: the intrinsic dimensionality of IoT networks. In *Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies*. 245–250.

[16] Matt Gorbett, Hossein Shirazi, and Indrakshi Ray. 2023. Cross-Silo Federated Learning Across Divergent Domains with Iterative Parameter Alignment. In *2023 IEEE International Conference on Big Data (BigData)*. IEEE, 5233–5242.

[17] Matt Gorbett, Hossein Shirazi, and Indrakshi Ray. 2023. Sparse binary transformers for multivariate time series modeling. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 544–556.

[18] Matt Gorbett, Caspian Siebert, Hossein Shirazi, and Indrakshi Ray. 2023. The intrinsic dimensionality of network datasets and its applications 1. *Journal of Computer Security* Preprint (2023), 1–26.

[19] Matt Gorbett and Darrell Whitley. 2023. Randomly initialized subnetworks with iterative weight recycling. *arXiv preprint arXiv:2303.15953* (2023).

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[21] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*. 1389–1397.

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

[23] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[24] Xia Hu, Lingyang Chu, Jian Pei, Weiqing Liu, and Jiang Bian. 2021. Model complexity of deep learning: A survey. *Knowledge and Information Systems* 63 (2021), 2585–2619.

[25] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

[26] Hyeonuk Kim, Jaehyeong Sim, Yeongjae Choi, and Lee-Sup Kim. 2017. A kernel decomposition architecture for binary-weight Convolutional Neural Networks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062189

[27] Weichao Lan and Liang Lan. 2021. Compressing deep convolutional neural networks by stacking low-dimensional binary convolution filters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 8235–8242.

[28] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. 2014. Low-Rank Factorization of Weight Matrices in Neural Networks. *arXiv preprint arXiv:1511.06441* (2014).

[29] Dongsoo Lee, Se Jung Kwon, Byeongwook Kim, Yongkweon Jeon, Baeseong Park, and Jeongin Yun. 2020. FleXOR: Trainable Fractional Quantization. arXiv:2009.04126 [cs.LG]

[30] Yuanzhi Li and Yingyu Liang. 2018. Learning overparameterized neural networks via stochastic gradient descent on structured data. *Advances in neural information processing systems* 31 (2018).

[31] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*. PMLR, 2849–2858.

[32] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. 2021. Mcunetv2: Memory-efficient patch-based inference for tiny deep learning. *arXiv preprint arXiv:2110.15352* (2021).

[33] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. 2020. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems* 33 (2020), 11711–11722.

[34] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-device training under 256kb memory. *Advances in Neural Information Processing Systems* 35 (2022), 22941–22954.

[35] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*. 10012–10022.

[36] Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. 2020. Reactnet: Towards precise binary neural network with generalized activation functions. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XIV 16*. Springer, 143–159.

[37] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*. 116–131.

[38] Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. 2020. Training binary neural networks with real-to-binary convolutions. *arXiv preprint arXiv:2003.11535* (2020).

[39] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*. PMLR, 1273–1282.

[40] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. 2021. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295* (2021).

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[42] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. 2017. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 652–660.

[43] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. 2020. Binary neural networks: A survey. *Pattern Recognition* 105 (2020), 107281.

[44] Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, and Jingkuan Song. 2020. Forward and backward information retention for accurate binary neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2250–2259.

[45] Haotong Qin, Mingyuan Zhang, Yifu Ding, Aoyu Li, Zhongang Cai, Ziwei Liu, Fisher Yu, and Xianglong Liu. 2023. BiBench: Benchmarking and Analyzing Network Binarization. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1177, 38 pages.

[46] Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. 2020. What's Hidden in a Randomly Weighted Neural Network?. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 11890–11899. https://doi.org/10.1109/CVPR42600.2020.01191

[47] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*. Springer, 525–542.

[48] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2014. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550* (2014).

[49] Manuele Rusci, Alessandro Capotondi, and Luca Benini. 2020. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. *Proceedings of Machine Learning and Systems* 2 (2020), 326–335.

[50] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[51] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[52] Yuzhang Shang, Dan Xu, Ziliang Zong, Liqiang Nie, and Yan Yan. 2022. Network binarization via contrastive learning. In *European Conference on Computer Vision*. Springer, 586–602.

[53] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[54] Ilya O Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, et al. 2021. Mlp-mixer: An all-mlp architecture for vision. *Advances in neural information processing systems* 34 (2021), 24261–24272.

[55] Asher Trockman and J Zico Kolter. 2022. Patches are all you need? *arXiv preprint arXiv:2201.09792* (2022).

[56] Quang Hieu Vo, Linh-Tam Tran, Sung-Ho Bae, Lok-Won Kim, and Choong Seon Hong. 2023. MST-compression: Compressing and Accelerating Binary Neural Networks with Minimum Spanning Tree. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 6091–6100.

[57] Yikai Wang, Wenbing Huang, Yinpeng Dong, Fuchun Sun, and Anbang Yao. 2023. Compacting Binary Neural Networks by Sparse Kernel Selection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 24374–24383.

[58] Yikai Wang, Yi Yang, Fuchun Sun, and Anbang Yao. 2021. Sub-bit neural networks: Learning to compress and accelerate binary neural networks. In *Proceedings of the IEEE/CVF international conference on computer vision*. 5360–5369.

[59] Yixing Xu, Kai Han, Chang Xu, Yehui Tang, Chunjing Xu, and Yunhe Wang. 2021. Learning frequency domain approximation for binary neural networks. *Advances in Neural Information Processing Systems* 34 (2021), 25553–25565.

[60] Zihan Xu, Mingbao Lin, Jianzhuang Liu, Jie Chen, Ling Shao, Yue Gao, Yonghong Tian, and Rongrong Ji. 2021. ReCU: Reviving the Dead Weights in Binary Neural Networks. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. 5178–5188. https://doi.org/10.1109/ICCV48922.2021.00515

[61] George Zerveas, Srideepika Jayaraman, Dhaval Patel, Anuradha Bhamidipaty, and Carsten Eickhoff. 2021. A transformer-based framework for multivariate time series representation learning. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 2114–2124.

[62] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044* (2017).

[63] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. 2021. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 11106–11115.

[64] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. 2018. Towards effective low-bitwidth convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7920–7928.