DISSERTATION

TOWARDS FAIR AND EFFICIENT DISTRIBUTED INTELLIGENCE

Submitted by

Matt Gorbett

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2024

Doctoral Committee:

Advisor: Indrakshi Ray

Anura Jayasumana
Hossein Shirazi
Steve Simske

Copyright by Matt Gorbett 2024

All Rights Reserved

ABSTRACT

TOWARDS FAIR AND EFFICIENT DISTRIBUTED INTELLIGENCE

Artificial Intelligence is rapidly advancing the modern technological landscape. Alongside this progress, the ubiquitous presence of computational devices has created unique opportunities to deploy intelligent systems in novel environments. For instance, resource constrained machines such as IoT devices have the potential to enhance our world through the use of Deep Neural Networks (DNNs). However, modern DNNs suffer from high computational complexity and are often relegated to specialized hardware, a bottleneck which has severely limited their practical use. In this work, we contribute to improving these issues through the use of neural network compression. We present new findings for both model quantization and pruning, two standard techniques for creating compressed and efficient DNNs. To begin, we examine the efficacy of neural network compression for time series learning, an unstudied modality in model compression literature. We construct a generalized Transformer architecture for multivariate time series which applies both *binarization* and *pruning* to model parameters. Our results show that the lightweight models achieve comparable accuracy to dense Transformers of the same structure on time series forecasting, classification, and anomaly detection tasks while significantly reducing the computational burden. Next, we propose two novel algorithms for neural network compression: 1) Tiled Bit Networks (TBNs) and 2) Iterative Weight Recycling (IWR). TBNs present a new form of quantization to tile neural network layers with sequences of bits to achieve *sub-bit* compression of binary-weighted models. The method learns binary vectors (i.e. tiles) to populate each layer of a model via tensor aggregation and reshaping operations; during inference, TBNs use just a single tile per model layer. TBNs perform well across a diverse range of architecture (CNNs, MLPs, Transformers) and tasks (classification, segmentation) while achieving up to 8x reduction in size compared to binary-weighted models. The second algorithm, IWR, generates sparse neural networks from randomly initialized models by identifying

important parameters within neural networks for reuse. The approach enables us to prune 80% of ResNet50's parameters while still achieving 70.8% accuracy on ImageNet. Finally, we examine the feasibility of deploying compressed DNNs in practical applications. Specifically, we deploy Sparse Binary Neural Networks (SBNNs), TBNs, and other common compression algorithms on an embedded device for performance assessment, finding a reduction in both peak memory and storage size. By integrating algorithmic and theoretical advancements into a comprehensive end-to-end methodology, this dissertation contributes a new framework for crafting powerful and efficient deep learning models applicable in real-world settings.

ACKNOWLEDGEMENTS

This work has been partially supported by funding from NSF under award numbers DMS 2123761, CNS 2027750, CNS 1822118 and from the member partners of the NSF IUCRC Center for Cybersecurity Analytics and Automation – Statnett, Cyber Risk Research, AMI, NewPush, and ARL.

I would also like to thank my advisors Indrakshi Ray and Hossein Shirazi for contributing to every chapter of this dissertation. My growth as a researcher the past few years has been largely guided by their help and direction. Thank you to committee members Steve Simske and Anura Jayasumana for your ideas and contributions. Both of you have provided valuable insights and new research ideas which I intend to pursue after my dissertation. I would also like to thank Darrell Whitley for his contributions to the algorithm Iterative Weight Recycling. I would also like to thank my wife Phoebe for putting up with my rambling on about neural networks the last few years. Finally, I would like to dedicate this dissertation to the greatest cat of all time, Lou.

TABLE OF CONTENTS

SUMMARY ii
ACKNOWLEDGEMENTS iv
LIST OF TABLES vii
LIST OF FIGURES ix
LIST OF ALGORITHMS xi

1 Introduction 1
1.1 Background: Pruning and Quantization in Deep Learning 3

2 Sparse Binary Transformers for Multivariate Time Series Modeling 6
2.1 Introduction 6
2.2 Related Work 8
2.2.1 Transformers in Time Series 9
2.2.2 Compressed Neural Networks 10
2.2.3 Compressed and Efficient Transformers 11
2.3 Methodology 11
2.4 Experiments 16
2.4.1 Classification 16
2.4.2 Anomaly Detection 17
2.4.3 Forecasting 19
2.4.4 Architecture 22
2.5 Metrics 23
2.5.1 Metrics 23
2.5.2 Model Size Selection 25
2.5.3 Analysis 26
2.6 Ablation Studies 27
2.7 Additional Results 29
2.8 Discussion 29
2.8.1 Applications 30
2.8.2 Limitations and Future Work 31

3 Randomly Initialized Subnetworks with Iterative Weight Recycling 32
3.1 Summary 32
3.2 Introduction 32
3.3 Related Work 34
3.3.1 Neural Network Pruning and Compression 34
3.3.2 Randomized Neural Networks 35
3.3.3 Binary Neural Networks 35
3.4 Background 36
3.5 Methodology and Scope 38
3.5.1 Scope 39

3.6 Experimental Setup . . . 41
3.7 Results . . . 41

4 Tiled Bit Networks: Sub-Bit Neural Network Compression Through Reuse of Learnable Binary Vectors . . . 45
4.1 Summary . . . 45
4.2 Introduction . . . 45
4.3 Related Work . . . 48
4.4 Method . . . 50
4.5 Experiments . . . 53
4.5.1 CNN Architectures . . . 54
4.5.2 MLP-Based Architectures . . . 54
4.5.3 Vision Transformers . . . 56
4.5.4 Effect of Layer Size . . . 57
4.6 Implementation . . . 58
4.6.1 Microcontroller Deployment . . . 58
4.6.2 GPU Inference Kernel . . . 60
4.7 Ablation Study . . . 61
4.8 Discussion . . . 62

5 Applying Sparse and Quantized Neural Networks in Real-World Applications 64
5.1 Summary . . . 64
5.2 Introduction . . . 64
5.3 Background and Related Work . . . 65
5.4 Research Methodology . . . 69
5.4.1 Bit Packing and Unpacking . . . 69
5.4.2 Optimizing the model for real-world applications . . . 71
5.5 Research Scope . . . 72
5.5.1 Benchmarking Against Real-World Quantization Techniques . . . 72
5.5.2 Deploying Sparse Binary Neural Networks to MCUs . . . 73
5.6 Experimental Results . . . 74
5.6.1 SBNN’s and Quantization Performance Evaluation . . . 75
5.6.2 Compressed Neural Networks Arduino Deployment . . . 76

6 Discussion . . . 81
6.1 Future Work . . . 81
6.2 Conclusion . . . 82

Bibliography . . . 84

LIST OF TABLES

2.1 **Accuracy of time series classification models on five datasets.** Results are obtained from [1, 2]. Sparse Binary Transformer (SBT) models achieve higher accuracy than prior works (excluding the Dense Transformer) in each case, except for the Japanese Vowels dataset. Additionally, SBT models achieve accuracy within 2.7% of the Dense Transformer for each dataset. 17

2.2 Anomaly detection results with benign sample windows. We evaluate Precision (P), Recall (R), and the F1 score using both manual threshold and Peak Over Threshold (POT) threshold technique. We find that the single time step prediction window achieves high accuracy when each past time-step in w is benign. $w = 200$ for SMD and $w = 50$ for SMAP and MSL. These results indicate that when given time to stabilize after an anomalous event, our SBT framework can detect new anomalies with high accuracy. We evaluate our results using a manual threshold (τ=0.5% for SMD, 1% for others) and the POT automatic threshold selector. 19

2.3 **F1 scores of various time series anomaly detection models.** We compare our SBT framework with several state-of-the-art algorithms on the anomaly detection task. The table is ordered by average F1 accuracy across each dataset. We evaluate our algorithm using the traditional method (different from Table 2.2), where each sample can contain anomalous events in its input window. We use a manual threshold to report results for the SBT model. 20

2.4 A summary of time series forecasting models on three datasets. Each SBT model is run three times with different weight seeds and averaged. Standard deviation is less than 0.01. 21

2.5 **Non-zero FLOPs equations for various attention modules.** These calculations assume $\mathbf{Q}, \mathbf{K}$ and $\mathbf{V}$ are equal sized projections in $\mathbb{R}^{w \times d}$, and $d' = d/h$. $\mathbf{QV}^{\intercal}$ and $\mathbf{AV}$ are additionally multiplied by h. $\mathbf{Q}$-scaling and softmax FLOPs excluded from this table. 24

2.6 **Computational savings for Dense Transformers compared to SBTs**. SBT models achieve a substantial reduction in size and FLOPs count across all models. We denote parameters in thousands and size and FLOPs in millions, with savings calculated by dividing the Dense values by the SBT values. 26

2.7 We compare Biprop with Biprop plus random pruning on classification tasks. We find that random pruning of the attention activations does not hurt classifcation accuracy, and in fact helps it in the case of the Japanese Vowels dataset. 28

2.8 We compare Biprop plus the Step-T attention mask with two other methods. We find that Biprop with the Step-T mask performs similarly to using Biprop with full attention (Biprop Only). Biprop with an Identity Mask on the attention computation performs worse than the other two methods. We report results using MSE loss averaged across three runs. 28

2.9 Results of applying the Iterative Weight Recycling algorithm to the SBT model. We test the resutls on time-series forecasting tasks on both single-step forecasting (top) and 4-step forecasting (bottom). We run each experiment five times and report the mean MSE of each. 30

4.1 **CNN Results on CIFAR-10**: * 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). 55
4.2 **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. 56
4.3 **Vision Transformers trained on CIFAR-10**: We compare the performance of a Vision Transformer (ViT) (patch size 4) and the Swin-T model with TBN_4 and TBN_8 variations. Experiments are averaged over 3 runs (S.D. is less than 0.5). 57
4.4 We compare the performance of a binary-weight neural network (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 being allocated for the output activation. 60

5.1 Performance Comparison of Quantization Techniques. Each model is trained on MNIST and has a single hidden layer with 256 neurons and a ReLU activation. The storage size is measured in bits and the inference time is measured by the time it takes each model to execute 1,000 times (averaged over 3 runs). The maximum memory is computed by computing the memory required to execute the first layer of the MLP, which is substantially larger than the output layer. Overall, each quantization method provides different benefits. 76

LIST OF FIGURES

2.1 A sparse binary linear layer (left) and various attention modules (right). a) An example of a sparse and binary linear module, with binary weights $\mathbf{B}$ scaled to $\{-\alpha, \alpha\}$. b) A fully-connected attention module, where each point represents a time step ($w = 6$). c) The Step-T attention module, where each past time point attends to itself and the latest time point t attends to all past time points. d) An attention module with sparse Query (Q), Key (K), and Value (V) activations. 10
2.2 **Step-t Attention Mask** Left: For the forecasting task we mask inputs during training in order to simulate unknown future time points. Right: The Step-T attention mask used to calculate attention only at the current time-step versus past values. Using this mask rather than setting our Query dimension to one enables us to pass time window vectors along multiple encoder layers. 15
2.3 Time series predictions on the ETTm1 dataset for the Pyraformer (top) and Sparse Binary Transformer (bottom) . We show 600 predictions across each model for two features (HULL, LUFL). 22

3.1 **Performance of Iterative Weight Recycling at varying network depth**: We compare the accuracy of our algorithm against Edge-Popup, Biprop, and a densely trained baseline. We use varying prune rates and varying network depths on VGG-like architectures. We train and evaluate all algorithms on the CIFAR-10 dataset using the same hyperparameter's and training procedure. 42
3.2 **Effects of Varying Width at 50% Prune Rate** Results of Dense, Biprop, and Biprop+Iterative Weight Recycling models at varying network widths less than one. Using Iterative Weight Recycling in addition to Biprop yields winning tickets with a comparable accuracy to densely trained models at just 50% width factor in all architectures. 43
3.3 **Model Performance with Limited Parameters** We test the effect of high prune rates (>90%) on model performance, showing that Weight Recycling achieves high accuracy compared to IteRand, Edge-Popup, and Biprop. **Left:** Accuracies of various VGG architectures with prune rates greater than 90% and parameter count less than 500k. **Right:** ResNet18 with prune rates greater than 95%. With just under 600k parameters (95% prune rate), ResNet18 achieves higher accuracy than a dense baseline model (93.1%). 44

4.1 **Tiling Illustration:** A binary tile (top left) of size $k = 3$ is replicated 3 times to create a tiled vector of size 9 (bottom left). This vector is then reshaped to fill a $3x3$ weight tensor (right). Tiling and reshaping is used during the training process of TBNs to learn tiles for filling the parameters of a model (illustrated above). During inference, only a single tile needs to be referenced per layer (top left) – a specialized kernel can reuse the tile throughout layer computation for memory savings. 46
4.2 **Layer composition of DNN architectures:** The ResNet series is made up primarily of convolutional layers; Transformer (Swin-T, ViT) and MLP (PointNet, MLPMixer) models consist mostly of fully-connected layers. 48

4.3 **Tile Construction During Training:** For each layer of a neural network we train a weight tensor ($\mathbf{W}$) (left). During 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 vector $\mathbf{s}$, creating tile $\mathbf{t}$ (second column bottom). We next create binary weights $\mathbf{B}$ from the resulting binary vector by tiling vector $\mathbf{t}$ 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 inference, only a single tile is needed, along with a small number of α scalars. 49
4.4 We learn scalar α from tensor $\mathbf{A}$ by computing Equation 4.7 or 4.9 over its values ($\mathbf{W}$ can also be used in place of $\mathbf{A}$). The figure visualizes Equation 4.9 which calculates α over each tile. 52
4.5 **TBNs are vulnerable in models with low width layers:** We plot the performance of the ConvMixer and MLPMixer 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. 58
4.6 **GPU memory allocated during model inference:** We profile the memory of the ImageNet Vision Transformer (Left) and PointNet (right) during inference on a single input using our customized GPU kernel. The x-axis represents memory recorded within intermediate model layers during execution of a PyTorch model. The kernel achieves 2.8x memory reduction on the ViT and 1.2x reduction on PointNet. The green line represents a standard kernel with full-precision weights; other lines represent a tiled kernel with full-precision weights. 61
4.7 **Hyperparameter Configurations:** 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 $\mathbf{A}$ to calculate α compared to calculating α using just $\mathbf{W}$. We additionally show the benefit of using multiple αs (one per tile) rather than a single α. 62

5.1 (a) Output on Image 1 when running inference on a trained sparse binary neural network. We use this network to convert to C and run on a microcontroller. (b) Model output when running inference on Image 1 in our C program. Both models produce the same output. 78
5.2 The Arduino Integrated Development Environment. 79
5.3 The Arduino Nano 33 BLE microcontroller used for this dissertation. 79
5.4 The Arduino IDE with the sparse binary neural network sketch (top) and the output of the code on the microcontroller (bottom). The microcontroller uses a custom Arduino print method (Serial) to output and log things on a computer. 80

LIST OF ALGORITHMS

5.1 Packing and Unpacking a bit string into uint8 data type 69
5.2 Packing and Unpacking a neural network linear layer into uint8 data type (column wise) 70

Chapter 1

Introduction

Machine learning systems are becoming increasingly integrated into various aspects of our lives, ranging from healthcare and finance to transportation and communication [3,4]. In the last decade, DNNs have catapulted the broad field of machine learning into the forefront of technological progress [5]. Hardware tools such as the Graphics Processing Unit (GPU) [6] have accelerated our ability to train neural networks on large amounts of data. This achievement was first demonstrated by the seminal work of Krizhevsky et al. [7] who produced state-of-the-results in the field of computer vision.

The advent of the Transformer architecture, introduced by Vaswani et al. in 2017 [8], marked another significant milestone in machine learning. The Transformer architecture revolutionized natural language processing tasks and led to impressive advancements. However, it also brought an increase in the computational burden required for AI systems to function. The Transformer's attention mechanism, which allows models to capture contextual relationships across input sequences, demands substantial computation to process vast amounts of data. As a result, the deployment of advanced Transformer-based models, such as BERT [9], GPT [10], and their subsequent iterations, has strained hardware infrastructure and prompted researchers to explore techniques for optimizing computations.

Alongside the progress of large scale deep learning there is a growing need for DNNs in resource constrained environments [11]. The field, commonly referred to as edge AI, involves the deployment and execution of deep learning models on smaller "edge" devices such as microcontrollers, embedded systems, IoT devices, and smartphones, and enables data processing and intelligent decision-making at the source [12]. The decentralized approach offers advantages like reduced latency, enhanced privacy, efficient bandwidth usage [11], and model specificity catered to a device's customized task. Moreover, edge AI provides *democratization* of deep learning algorithms where any device can run a powerful deep learning model, an important property in today's age of

monopolized large language models such as OpenAI's GPT [10]. As a result, edge AI provides a meaningful alternative to large scale AI systems.

The high computational cost of modern deep learning coupled with the need for more efficient and lightweight DNNs has led researchers to explore methods to make deep algorithms more efficient [13]. Methods such as pruning [14, 15] and weight quantization [16, 17] provide opportunities to make neural networks smaller and less memory intensive, offering benefits in large-scale applications as well as in smaller edge AI use cases. Neural network pruning involves removing some parameters of the model by setting the values to zero. In effect, we create a "sparse" neural network, which can theoretically lead to a reduced computational cost. Quantization involves reducing the precision of the neural networks parameters and activations, such as reducing a models parameter from 32-bit values down to 8-bit values. For example, a 32-bit value of 3452.1432 can be quantized down to its 8-bit value 3452. While current pruning and quantization approaches offer substantial benefits, many often lead to performance degradation [18–20], or are impractical for applied use [21, 22].

In the following research we work towards addressing open problems in neural network compression research including both model quantization and pruning. In particular, we propose three distinct research areas which each tackle a particular problem. To begin, we propose a novel Transformer architecture, the SBT, which is a highly compressed (sparse and binary) Transformer capable of learning generalized multivariate time series learning tasks. Our proposed research is the first to show that neural networks are robust to extreme compression, particularly for multivariate time series learning. Our model is tested on time series classification, anomaly detection, and forecasting, and exhibits a substantial computational savings.

The second area of our work finds new algorithm for achieving neural network compression (including pruning and quantization). First we propose Iterative Weight Recycling, which improves our ability to find sparse neural networks within randomly initialized models. The algorithm achieves this by reusing high importance parameters within the pruned model structure, exhibiting higher sparsity and better test performance across architectures and datasets. In addition we explore another novel compression algorithm, which we call "Tiled Bit Networks". The algorithm tiles a neural

network with binary weight by repeating a sequence of bits into the neural networks structure for each layer. The algorithm is implemented across both convolutional and fully-connected layers, and is the first of its kind, achieving sub-bit neural network compression on a multitude of architectures (CNNs, Transformers, MLPs) and tasks (classification, segmentation) with significant compression compared to binary-weighted models.

In the final area of our work, we assess the feasibility of deploying Sparse Binary Neural Networks in real-world applications. The first two sections of our research employ a "sparse and binary" architecture for pruning *and* quantizing model's weights. In this section, we assess the viability of deploying this architecture in a limited resource environment. Specifically, we deploy a neural network on an Arduino microcontroller by first implementing the model with bit-packed linear layers and converting the model to C. In addition, we assess the sparse and binary neural network model against commonly used quantization approaches used in real world applications.

In the next section we provide a general overview of neural network pruning and quantization, with aim of providing a high-level background information related to each of our proceeding chapters. The rest of the proposal organizes each of our three topics into its own chapter: 1) Sparse Binary Transformers for Multivariate Time Series Modeling, 2) Enhanced Compression Algorithms for Neural Networks, and 3) Applying Sparse and Quantized Neural Networks in Real-World Applications. Each of the chapters contains an introduction, a detailed related work, methodology, experiments, and other analysis and research related to the topic. Finally, we wrap up the dissertation in our Conclusion section.

1.1 Background: Pruning and Quantization in Deep Learning

Deep learning models, while powerful, often suffer from large memory footprints and high computational costs. Model compression techniques have emerged as essential strategies to mitigate these challenges. This dissertation focuses on two key techniques: pruning and quantization.

Pruning Neural network pruning involves the removal of redundant connections in neural networks, leading to smaller and more efficient models. Weight pruning methods focus on eliminating

connections with low magnitudes, exploiting the observation that small weights have negligible impact on the network's output. Early techniques such as Optimal Brain Damage [14] proposed removing weights based on their second derivatives with respect to the loss function. Iterative pruning algorithms like Optimal Brain Surgeon [23] further refined this approach, incorporating Hessian information to identify and prune less significant connections. Recent advancements have introduced structured pruning, where entire channels or layers are pruned based on importance metrics such as L1-norm or saliency scores [24]. Moreover, sensitivity analysis methods [25] have gained popularity for their ability to identify redundant connections, achieving substantial compression ratios while preserving model accuracy.

Key to this dissertation is the work presented in the Lottery Ticket Hypothesis, introduced by Frankle and Carbin in 2018, which has significantly influenced the field of deep learning model optimization. This hypothesis challenges the conventional understanding of neural network training by proposing that within large, over-parameterized networks, there exist small subnetworks, termed "winning lottery tickets", which, when trained in isolation, can match the performance of the original network. Frankle and Carbin's groundbreaking work suggested that these sparse, trainable subnetworks could be identified through iterative pruning and retraining processes, leading to considerable model compression without sacrificing accuracy. Subsequent research efforts have delved deeper into understanding the characteristics and principles behind these winning tickets [26], exploring their role in transfer learning, exploring novel techniques for their identification [27], and applying them to various network architectures and tasks [28]. The Lottery Ticket Hypothesis has thus become a foundational concept in the realm of network pruning, offering valuable insights into the nature of deep learning models and paving the way for more efficient and effective training methodologies and model architectures.

Quantization

On the other front, quantization techniques focus on reducing the precision of model parameters, enabling compact representation without sacrificing performance. Gupta et al. [29] delved into the delicate balance between reduced numerical precision and model accuracy, offering insights into

the importance of precision control in deep learning applications. Additionally, Zhou et al. [30] introduced incremental quantization, a dynamic precision adjustment method during training, showcasing the significance of adaptive precision in maintaining model accuracy while significantly reducing computational demands. Common techniques for creating quantized networks include post-training quantization with retraining [31, 32], where we train a full-precision model and then quantize the weights after training, and quantization-aware training [29], where a DNN is trained with the knowledge that it will be quantized after training. Common techniques generally quantize model weights down to 8-bit parameters.

Specific to this work, binary weight quantization (binarization) restrict weights to binary values, $\{-1, 1\}$, significantly reducing memory usage and computational complexity. Binary neural networks [33] employ binary weights to simplify arithmetic operations to bitwise operations. One fundamental approach to BNNs involves novel training methods. Rastegari et al. [34] introduced XNOR-Net, a model incorporating binary weights and activations throughout training and inference, achieving significant model compression while maintaining accuracy. Zhou et al. [35] proposed using real-valued "centroids" in BNNs, enhancing convergence and overall performance. In terms of hardware implications, Courbariaux et al. [33] applied BNNs to reinforcement learning tasks, demonstrating the viability of binary networks in complex decision-making processes. Tang et al. [36] explored specialized hardware accelerators designed for efficient binary operations, maximizing computational efficiency. Furthermore, BNNs have found diverse applications. Hubara et al. [37] showcased the suitability of BNNs for energy-constrained environments, making them ideal for IoT devices and edge computing applications. Umuroglu et al. [38] extended BNNs for FPGA-based implementations, enabling real-time processing in embedded systems. In summary, quantization and binarization are powerful techniques for compressing neural networks, achieving empirical successful on a range of tasks. Despite this, current techniques for neural network binarization do not achieve the same accuracy as full precision models, leading to a trade-off between model accuracy and compression.

Chapter 2

Sparse Binary Transformers for Multivariate Time Series Modeling

Summary

Compressed Neural Networks have the potential to enable deep learning across new applications and smaller computational environments. However, understanding the range of learning tasks in which such models can succeed is not well studied. For the first chapter of our work, we examine *sparse* and *binary-weighted* Transformers for multivariate time series problems, examining whether the lightweight models can achieve accuracy comparable to that of dense floating-point Transformers of the same structure. We examine three time series learning tasks: *classification*, *anomaly detection*, and *single-step forecasting*. Additionally, to reduce the computational complexity of the attention mechanism, we propose applying two modifications, which show little to no decline in model performance: 1) in the classification task, we apply a fixed mask to the query, key, and value activations, and 2) for forecasting and anomaly detection, which rely on predicting outputs at a single point in time, we propose an attention mask to allow computation only at the current time step.

2.1 Introduction

The success of deep learning can largely be attributed to the availability of massive computational resources [39–41]. Models such as the Transformer [8] have changed machine learning in fundamental ways, producing state-of-the-art results across fields such as natural language processing (NLP), computer vision [28, 42], and time series learning [1]. Much effort has been aimed at scaling these models towards NLP efforts on large datasets [43, 44], however, such models cannot practically be deployed in resource-constrained machines due to their high memory requirements and power consumption.

Parallel to the developments of the Transformer, the Lottery Ticket Hypothesis [15] demonstrated that neural networks contain sparse subnetworks that achieve comparable accuracy to that of dense models. Pruned deep learning models can substantially decrease computational cost, and enable a lower carbon footprint and the democratization of AI. Subsequent work showed that we can find highly accurate subnetworks within randomly-initialized models without training them [27], including binary-weighted neural networks [26]. Such "lottery-ticket" style algorithms have mostly experimented with image classification using convolutional architectures, however, some work has shown success in pruning NLP Transformer models such as BERT [45–47].

In this work, we extend the Lottery Ticket Hypothesis to time series Transformers, showing that we can prune and binarize the weights of the model and still maintain an accuracy similar to that of a Dense Transformer of the same structure. To achieve this, we employ the Biprop algorithm [26], a state-of-the-art technique with proven success on complex datasets such as ImageNet [48]. The combination of weight binarization and pruning is unique from previous efforts in Transformer compression. Moreover, each compression technique offers separate computational advantages: neural network pruning decreases the number of non-zero floating point operations (FLOPs), while binarization reduces the storage size of the model. The Biprop algorithm's two compression methods rely on each other during the training process to identify a high-performing subnetwork within a randomly weighted neural network. The combination of pruning and weight binarization is depicted in Figure 2.1a.

We apply our approach to multivariate time series modeling. Research has shown that Transformers achieve strong results on time series tasks such as classification [1], anomaly detection [49, 50], and forecasting [51,52]. Time series data is evident in systems such as IoT devices [53], engines [54], and spacecraft [55, 56], where new insights can be gleaned from the large amounts of unmonitored information. Moreover, such systems often suffer from resource constraints, making regular deep learning models unrealistic – for instance, in the Mars rover missions where battery-powered devices are searching for life [57]. Other systems such as satellites contain thousands of telemetry channels that require granular monitoring. Deploying large deep learning models in each channel can be

extremely inefficient. As a result, lightweight Transformer models have the potential to enhance a wide variety of applications.

In addition to pruning and binarizing the Transformer architecture, we simplify the complexity of the attention mechanism by applying two modifications. For anomaly detection and forecasting, which we model using overlapping sliding window inputs, we apply an attention mask to only consider attention at the current time step instead of considering attention for multiple previous time steps. For classification tasks, we apply a static mask to the query, key, and value projections, showing that only a subset of activations is needed in the attention module to achieve the same accuracy as that obtained using all the activations.

Finally, we estimate the computational savings of the model in terms of parameters, storage cost, and non-zero FLOPs, showing that pruned and binarized models achieve comparable accuracy to dense models with substantially lower computational costs.

Our contributions are as follows:

- We show that sparse and binary-weighted Transformers achieve comparable accuracy to Dense Transformers on three time series learning tasks (classification, anomaly detection, forecasting). To the best of our knowledge, this is the first research examining the efficacy of compressed neural networks on time series related learning.
- We examine pruning and binarization jointly in Transformer-based models, showing the benefits of each approach across multiple computational metrics. Weight binarization of Transformer based architectures has not been studied previously.

These findings provide new potential applications for the Transformer architecture, such as in resource-constrained environments that can benefit from time series related intelligence.

2.2 Related Work

In this section, we describe existing research related to Transformers in time series modeling, neural network pruning and compression, and finally efficient Transformer techniques.

2.2.1 Transformers in Time Series

Various works have applied Transformers to time series learning tasks [58]. The main advantage of the Transformer architecture is the attention mechanism, which learns the pairwise similarity of input patterns. Moreover, it can efficiently model long-range dependencies compared to other deep learning frameworks such as LSTM's [51]. Zerveas et al. [1] showed that we can use unsupervised pretrained Transformers for downstream time series learning tasks such as regression and classification. Additional work in time series classification has proposed using a "two tower" attention approach with channel-wise and time-step-wise attention [59], while other work has highlighted the benefits of Transformers for satellite time series classification compared to both recurrent and convolutional neural networks [60].

For anomaly detection tasks, Transformers have shown favorable results compared to traditional ML and deep learning techniques. Notably, Meng et al. [61] applied the model to NASA telemetry datasets and achieved strong accuracy (0.78 F1) in detecting anomalies. TranAD [50] proposed an adversarial training procedure to exaggerate reconstruction errors in anomalies. Xu et al. [49] achieve state-of-the-art results in detecting anomalies in multivariate time series via association discrepancy. Their key finding is that anomalies have high association with adjacent time points and low associations with the whole series; using this property, the authors are able to accentuate anomalies.

Finally, Transformer variations have been proposed for time series forecasting to lower the attention complexity of long sequence time series [51, 52, 62, 63], add stochasticity [64], and incorporate traditional time series learning methods [63, 65]. Li et al. [62] introduce LogSparse attention, which allows each cell to attend only to itself and its previous cells with an exponential step size. The Informer method [52] selects dominant queries to use in the attention module based on a sparsity measurement. Pyraformer [51] introduces a pyramidal attention mechanism for long-range time series, allowing for linear time and memory complexity. Wu et al. [64] use a Sparse Transformer as a generator in an encoder-decoder architecture for time series forecasting, using a discriminator to improve the prediction.

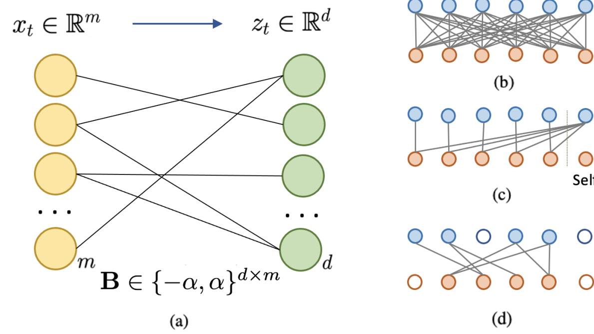

Figure 2.1: A sparse binary linear layer (left) and various attention modules (right). a) An example of a sparse and binary linear module, with binary weights $\mathbf{B}$ scaled to $\{-\alpha, \alpha\}$. b) A fully-connected attention module, where each point represents a time step ($w = 6$). c) The Step-T attention module, where each past time point attends to itself and the latest time point t attends to all past time points. d) An attention module with sparse Query (Q), Key (K), and Value (V) activations.

2.2.2 Compressed Neural Networks

Pruning unimportant weights from neural networks was first shown to be effective by Lecun et al. [14]. In recent years, deep learning has scaled the size and computational cost of neural networks. Naturally, research has been directed at decreasing size [13] and energy consumption [66] of deep learning models.

The Lottery Ticket Hypothesis [15] showed that randomly initialized neural networks contain sparse subnetworks that, when trained in isolation, achieve comparable accuracy to a trained dense network of the same structure. The implications of this finding are that over-parameterized neural networks are no longer necessary, and we can prune large models and still maintain the original accuracy.

Subsequent work found that we do not need to train neural networks at all to find accurate sparse subnetworks; instead, we can find a high performance subnetwork using the randomly initialized weights [27, 67–69]. Edge-Popup [27] applied a scoring parameter to learn the importance of each weight, using the straight-through estimator [70] to find a high accuracy mask over randomly initialized models. Diffenderfer and Kailkhuram [26] introduced the *Multi-Prize* Lottery Ticket Hypothesis, showing that 1) multiple accurate subnetworks exist within randomly initialized neural networks, and 2) these subnetworks are robust to quantization, such as binarization of weights. In

this work, we use the Biprop algorithm proposed in [26] to binarize the weights of Transformer models.

2.2.3 Compressed and Efficient Transformers

Large-scale Transformers such as the BERT (110 million parameters) are a natural candidate for pruning and model compression [46, 71]. Chen et al. [28] first showed that the Lottery Ticket Hypothesis holds for BERT Networks, finding accurate subnetworks between 40% and 90% sparsity. Jaszczur et al. [72] proposed scaling Transformers by using sparse variants for all layers in the Transformer. Other works have reported similar findings [73, 74], showing that sparsity can help scale Transformer models to even larger levels.

Other works have proposed modifications for more efficient Transformers aside from pruning [71]. Most research has focused on improving the $\mathcal{O}(n^2)$ complexity of attention, via methods such as fixed patterns [75], learnable patterns [76], low rank/kernel methods [77, 78], and downsampling [79, 80]. Various other methods have been proposed for compressing BERT networks such as pruning via post-training mask searches [81], block pruning [82], and 8-bit quantization [83]. We refer readers to Tay et al. [71] for details.

Despite the various works compressing Transformers, we were not able to find any research using both pruning and binarization. Utilizing both methods allows for more efficient computation (measured using FLOPs) as well as a significant decrease in storage (due to binary weights). Additionally, we find that our proposed model is still a fraction of the size of compressed NLP Transformers models when trained on time series tasks. For instance, TinyBERT [47] contains 14.5 million parameters and 1.2 billion FLOPs, compared to our models which contain less than 1.5 million **binary** parameters and 38 million FLOPs.

2.3 Methodology

In this section, we begin by describing the base Transformer architecture we will be using for our work. We will apply this base model to all three time series tasks, making small modifications

to the model in order to accomonodate the given task. Then, we will describe the modifications we make to the base model for quantization, including the attention modifications we make to simplify the model.

Our model consists of a Transformer encoder [8] with several modifications. We base our model off of Zerveas et al. [1], who propose using a common Transformer framework for several time series modeling tasks. To begin, we describe the base architecture of the Transformer as applied to multivariate time series. Subsequently, we describe the techniques used for pruning and binarization. Finally, we describe the two changes applied to the attention mechanism.

Dense Transformer We denote fully trained Transformers with no pruning and floating point 32 (FP32) weights as Dense Transformers. Let $\mathbf{X_t} \in \mathbb{R}^{w \times m}$ be a model input for time t with window size w and m features. Each input contains w feature vectors $\mathbf{x} \in \mathbb{R}^m : \mathbf{X_t} \in \mathbb{R}^{w \times m} = [\mathbf{x_{t-w}}, \mathbf{x_{t-w+1}}, ..., \mathbf{x_t}]$, ordered in time sequence of size w. In classification datasets w is predefined at the sample or dataset level. For anomaly detection and forecasting tasks, we fix w to 50 or 200 and use an overlapping sliding window as inputs.

The standard architecture (pre-binarization) projects m features onto a d-dimensional vector space using a linear module with learnable weights $\mathbf{W_p} \in \mathbb{R}^{d \times m}$ and bias $\mathbf{b_p} \in \mathbb{R}^d$. We use the standard positional encoder proposed by Vaswani et al. [8], and we refer readers to the original work for details. For the Dense Transformer classification models, we use learnable positional encoder [1]. Zerveas et al. [1] propose using batch normalization instead of layer normalization used in traditional Transformer NLP models. They argue that batch normalization mitigates the effects of outliers in time series data. We found that for classification tasks, batch normalization performed the best, while in forecasting tasks layer normalization worked better. For anomaly detection tasks we found that neither normalization technique was needed.

Each Transformer encoder layer consists of a multi-head attention module followed by ReLU layers. The self-attention module takes input $\mathbf{Z_t} \in \mathbb{R}^{w \times d}$ and projects it onto a Query ($\mathbf{Q}$), Key ($\mathbf{K}$), and Value ($\mathbf{V}$), each with learnable weights $\mathbf{W} \in \mathbb{R}^{d \times d}$ and bias $\mathbf{b} \in \mathbb{R}^d$. Attention is defined as $Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Softmax\left(\frac{\mathbf{QK^\intercal}}{\sqrt{d}}\right)\mathbf{V}$. Queries, keys, and values are projected

by the number of heads (h) to create multi-head attention. The resultant output $\mathbf{Z_t}'$ undergoes a nonlinearity before being passed to the next encoder layer. The Transformer consists of N encoder layers followed by a final decoder layer. For classification tasks, the decoder outputs l classification labels: $\mathbf{X_t'} \in \mathbb{R}^{w \times l}$, which are averaged over w. For anomaly detection and forecasting, the decoder reconstructs the full input: $\mathbf{X_t'} \in \mathbb{R}^{w \times m}$.

Sparse Binary Transformer Central to our binarization architecture is the Biprop algorithm [26], which uses randomly initialized floating point weights to find a binary mask over each layer. Given a neural network with weight matrix $\mathbf{W} \in \mathbb{R}^{i \times j}$ initialized with a standard method such as Kaiming Normal [84], we can express a subnetwork over neural network $f(x; \mathbf{W})$ as $f(x; \mathbf{W} \odot \mathbf{M})$, where $\mathbf{M} \in \{0, 1\}$ is a binary mask and $\odot$ is an elementwise multiplication.

To find $\mathbf{M}$, parameter $\mathbf{S} \in \mathbb{R}^{i \times j}$ is initialized for each corresponding $\mathbf{W} \in \mathbb{R}^{i \times j}$. $\mathbf{S}$ acts as a score assigned to each weight dictating the importance of the weights contribution to a successful subnetwork. Using backpropagation as well as the straight-through estimator [70], the algorithm takes pruning rate hyperparameter $p \in [0, 1]$, and on the forward pass computes $\mathbf{M}_k$ at layer m as

$$\mathbf{M}_k = \begin{cases} 1 & \text{if } |\mathrm{S}_k| \in \{\tau(k)_{k=1}^{l_m} \geq [l_m p]\} \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

where τ sorts indices $\{k\}_{k=1}^{l} \in \mathrm{S}$ such that $|S_{\tau(k)}| \leq |S_{\tau(k+1)}|$. Masks are computed by taking the absolute value of scores for each layer, and setting the mask to 1 if the value falls above the top p^{th} percentile.

To convert each layer to binary weights Biprop introduces gain term $\alpha \in \mathbb{R}$, which is common to Binary Neural Networks (BNN's) [85]. The gain term utilizes floating-point weights *prior* to binarization during training. During test-time, the alpha parameter scales the binarized weight vector. The parameter rescales binary weights $\mathbf{B} \in \{-1, 1\}$ to $\{-\alpha, \alpha\}$, and the network function becomes $f(x; \alpha(\mathbf{B} \odot \mathbf{M}))$. α is calculated as

$$\alpha = \frac{||\mathbf{M} \odot \mathbf{W}||_1}{||\mathbf{M}||_1} \tag{2.2}$$

with $\mathbf{M}$ being multiplied by α for gradient descent (the straight-through estimator is still used for backpropagation). This calculation was originally derived by Rastegari et al. [86].

In our approach we create sparse and binary modules for each linear and layer normalization layer. Our model consists of two linear layers at the top most level: one for projecting the initial input (embedding in NLP models) and one used for the decoder output. Additionally, each encoder layer consists of six linear layers: $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ projections, the multi-head attention output projection, and two additional layers to complement multi-head attention.

Attention Modifications In this section we describe two modifications made to the attention module to reduce its quadratic complexity. Several previous works have proposed changes to attention in order to lessen this bottleneck, such as Sparse Transformers [87], ProbSparse Attention [52], and Pyramidal Attention [51]. While each of these works present quality enhancements to the memory bottleneck of attention, we instead seek to evaluate whether simple sparsification approaches can retain the accuracy of the model compared to canonical attention. Our primary motivation for the following attention modifications are to test whether a compressed Transformer can retain the same accuracy as a Dense Transformer.

Fixed Q,K, and V Projection Mask To reduce the computational complexity of the matrix multiplications within the attention module, we apply random fixed masks to the $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ projections. We hypothesize that we can retain the accuracy of full attention by using this "naive" activation pruning approach, which requires no domain knowledge. We argue that the success of this approach provides insight into the necessity of full attention computations. In other words, Transformers are expressive and powerful enough for certain tasks that we can prune the models in an unsophisticated way and maintain accuracy. Moreover, many time series datasets and datasets generated at the edge are often times simplistic enough that we can apply this unsophisticated pruning [88, 89].

To apply this pruning, on model initialization we create random masks with prune rate $p_a \in \{0, 1\}$ for each attention module and each projection Q,K, and V. Attention heads within the same module inherit identical Q, K, or V masks. The mask is applied to each projection during train and

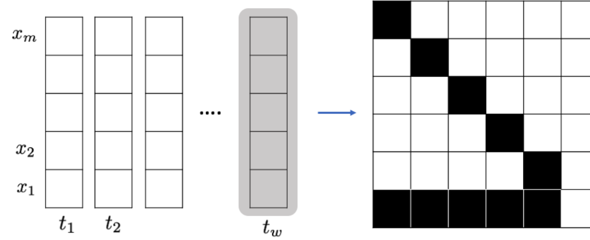

Figure 2.2: Step-t Attention Mask Left: For the forecasting task we mask inputs during training in order to simulate unknown future time points. Right: The Step-T attention mask used to calculate attention only at the current time-step versus past values. Using this mask rather than setting our Query dimension to one enables us to pass time window vectors along multiple encoder layers.

test. In each of our models we set the prune rate p_a of the attention module equal to the prune rate of the linear modules ($p_a = p$).

Step-t Attention Mask For anomaly detection and single-step forecasting tasks, the SBT algorithm relies on reconstructing or predicting outputs at the current time step t for each feature m, despite w time steps of data being provided to the model. Specifically, the SBT model is only interested in input vector $\mathbf{x_t} \in \mathbb{R}^m$. For anomaly detection, the model reconstructs $\mathbf{x_t}$ from the input, while in forecasting tasks the model masks $\mathbf{x_t} = 0$ prior to model input, reconstructs the actual values during training and inference.

In both tasks, vector $\mathbf{x_t}$ contains the only values necessary for the model to learn, and our loss function reflects this by only computing error for these values. As a result, computing attention for each other time step adds unnecessary computation. As depicted in Figure 2.2, we pass a static mask to the attention module to compute attention only at step-T. We additionally exclude attention computation at step-T with itself, forcing the variable to attend to historical time points for prediction. Finally, we add diagonal ones to the attention mask at all past time points to add stability to training. This masking method allows us to propagate the full input sample to multiple attention layers, helping us retain relevant historical information for downstream layers that would not be possible by changing the sizes of Q, K, and V to only model the t time step.

2.4 Experiments

In this section we detail our experiments for time series classification, anomaly detection, and forecasting. Common to each learning task, we normalize each dataset prior to training such that each feature dimension m has zero mean and unit variance. We use the Transformer Encoder, training each learning task and dataset using the Dense Transformer and the SBT to compare accuracy. Finally, we run each experiment three times with a different weight seed, and present the average result. For the SBT model, varying the weight seed shows evidence of the robustness to hyperparameters. Specific modifications to the model are made for each learning task, which we describe in the following sections. Additional training and architecture details can be found in the Appendix.

2.4.1 Classification

For our first time series learning task we select several datasets from the UCR Time Series Classification Repository [90]. The datasets contain diverse characteristics including varying training set size (204-30,000), number of features (13-200), and window size (30-405). We choose three datasets with the largest test set size (Insect Wingbeats, Spoken Arabic Digits, and Face Detection) as well as two smaller datasets (JapaneseVowels, Heartbeat). Each dataset contains a set window size except for Insect Wingbeats and Japanese Vowels, which contain a window size *up to* 30 and 29, respectively. In these datasets, we pad samples with smaller windows to give them consistent window sizes. The decoder in our classification architecture is a classification head, rather than a full reconstruction of the input as is used in anomaly detection and forecasting tasks. The SBT classification model is trained and tested using the fixed Q,K,V projection mask.

Results In Table 2.1, we show that SBTs perform as well as, or similar to, the Dense Transformer for each dataset at $p = 0.5$ and $p = 0.75$. Our models are averaged over three runs with different weight seeds. When comparing our model to state-of-the-art approaches, we find that the SBT achieves strong results across each dataset, with the highest reported performance on three out of

Table 2.1: Accuracy of time series classification models on five datasets. Results are obtained from [1, 2]. SBT models achieve higher accuracy than prior works (excluding the Dense Transformer) in each case, except for the Japanese Vowels dataset. Additionally, SBT models achieve accuracy within 2.7% of the Dense Transformer for each dataset.

Model	Arabic Digits	Heart Beat	Insect W.B.	Japan. Vowels	Face Detect.	Mean
XGBoost	69.6	73.2	36.9	96.2	63.3	67.8
LSTM	31.9	72.2	17.6	79.7	57.7	51.8
Rocket [91]	71.2	75.6	-	86.5	64.7	74.5
Fran et al. [2]	95.6	75.6	16.0	98.9	52.8	67.8
DTW_D	96.3	71.7	-	94.9	52.9	79.0
Dense Trans.	98.0	76.6	63.4	**98.0**	56.0	78.8
$SBT_{p=0.5}$	98.2	77.2	**64.1**	95.3	**66.1**	**80.2**
$SBT_{p=0.75}$	**98.6**	**78.5**	61.3	85.3	65.8	77.9

the five datasets. Further, the SBT models perform consistently across datasets while models such as Rocket [91] and Fran et al. [2] have lower performance on one or more datasets.

Surprisingly, the SBT model achieves stronger average accuracy than the Dense Transformer (80.2% versus 78.8%), indicating that the pruned and binarized Transformer achieves a robust performance across datasets. Despite this, Insect Wingbeats and Japanese Vowels datasets achieved a slightly lower performance at $p = 0.5$ with a more substantial dropoff at $p = 0.75$, indicating the model may lose some of its power on certain tasks.

2.4.2 Anomaly Detection

For the anomaly detection task we test the SBT algorithm on established multivariate time series anomaly detection datasets used in previous literature: Soil Moisture Active Passive Satellite (SMAP) [92], Mars Science Labratory rover (MSL) [92], and the Server Machine Dataset (SMD) [55]. SMAP and MSL contain telemetry data such as radiation and temperature, while SMD logs computer server data such as CPU load and memory usage. The datasets contain benign samples in the training set, while the test set contains labeled anomalies (either sequences of anomalies or single point anomalies).

Our model takes sliding window data as input and reconstructs data at $\mathbf{x_t}$ given previous time points. We use MSE to reconstruct each feature in $\mathbf{x_t}$. We use the step-T attention mask as described in the previous section. To evaluate our results, we adopt an adjustment strategy similar to previous works [49, 50, 55, 93]: if *any* anomaly is detected within a successive abnormal segment of time, we consider all anomalies in this segment to have been detected. The justification is that detecting any anomaly in a time segment will cause an alert in real-world applications.

To flag anomalies, we retrieve reconstruction loss $\mathbf{x'_t}$ and threshold τ, and consider anomalies where $\mathbf{x'_t} > \tau$. Since our model is trained with benign samples, anomalous samples in the test set should yield a higher $\mathbf{x'_t}$. We compute τ using two methods from previous works: A manual threshold [49] and the POT method [94]. For the manual threshold, we consider proportion r of the validation set as anomalous. For SMD $r = 0.5\%$, and for MSL and SMAP $r = 1\%$. For the POT method, similar to OmniAnomaly [55] and TranAd [50], we use the automatic threshold selector to find τ. Specifically, given our training and validation set reconstruction losses, we use POT to fit the tail portion of a probability distribution using the generalized Pareto Distribution. POT is advantageous when little information is known about a scenario, such as in datasets with an unknown number of anomalies.

Results In Table 2.2 we report the unique findings of our single-step anomaly detection method using Precision, Recall, and F1-scores. Specifically, we find that when only considering inputs with fully benign examples in window w, both the SBT and the Dense Transformer achieve high accuracy on all three datasets (F1 between 90.6 and 100). In other words, we find that our model performance is best when we filter examples that have an anomalous sequence or data point in $[\mathbf{x_{t-w}}, \mathbf{x_{t-w+1}}, ..., \mathbf{x_{t-1}}]$. For SMD, $w = 200$ and for SMAP and MSL $w = 50$. This observation implies that the model needs time to stabilize after an anomalous period. Intuitively, if an anomaly occurred recently, new benign observations will have a higher reconstruction loss as a result of their difference with the anomalous examples in their input window. We argue that this validation metric is logical in real-world scenarios, where monitoring of a system after an anomalous period of time is necessary.

Table 2.2: Anomaly detection results with benign sample windows. We evaluate Precision (P), Recall (R), and the F1 score using both manual threshold and POT threshold technique. We find that the single time step prediction window achieves high accuracy when each past time-step in w is benign. $w = 200$ for SMD and $w = 50$ for SMAP and MSL. These results indicate that when given time to stabilize after an anomalous event, our SBT framework can detect new anomalies with high accuracy. We evaluate our results using a manual threshold (τ=0.5% for SMD, 1% for others) and the POT automatic threshold selector.

Dataset	Metric	Manual Threshold		POT Threshold	
		Dense	$SBT_{p=0.75}$	Dense	$SBT_{p=0.75}$
MSL	P	92.7	96.8	85.5	82.9
	R	100	100	100	100
	F1	96.2	96.8	92.1	91.1
SMD	P	85.4	85.3	99.9	100
	R	100	100	100	100
	F1	92.1	92.1	100	99.9
SMAP	P	93.9	93.7	85.9	84.9
	R	100	100	100	100
	F1	96.9	96.8	92.4	91.8

We additionally report F1-scores compared to state-of-the-art time series anomaly detection models in Table 2.3. To accurately compare our model against existing methods, we use the full test set without filtering out benign inputs with anomalies in the near past. SBT results are much more modest, with F1-scores between 70 and 88. Despite this, our method still performs stronger than non-temporal algorithms such as the Isolation Forest, as well as other deep-learning based approaches such as Deep-SVDD and BeatGan.

2.4.3 Forecasting

We test our method on single-step forecasting using the Step-T attention mask. Specifically, using the framework outlined by Zerveas et al. [1], we train our model by masking the input at the forecasting time-step t. For example, input $\mathbf{X_t}$ containing m features and t time-steps $[\mathbf{x_{t-w}}, \mathbf{x_{t-w+1}}, ..., \mathbf{x_t}]$ is passed through the network with $\mathbf{x_t} = 0$. We then reconstruct this masked input with the Transformer model, using mean squared error between the masked inputs

Table 2.3: F1 scores of various time series anomaly detection models. We compare our SBT framework with several state-of-the-art algorithms on the anomaly detection task. The table is ordered by average F1 accuracy across each dataset. We evaluate our algorithm using the traditional method (different from Table 2.2), where each sample can contain anomalous events in its input window. We use a manual threshold to report results for the SBT model.

Model	SMD	MSL	SMAP	Avg.
LOF	46.7	61.2	57.6	55.2
IsolationForest	53.6	66.5	55.5	58.5
OCSVM	56.2	70.8	56.3	61.1
DAGMM	57.3	74.6	68.5	66.8
VAR	74.1	77.9	64.8	72.3
MMPCACD	75.0	70.0	81.7	75.6
ITAD	79.5	76.1	73.9	76.5
Deep-SVDD	79.1	83.6	69.0	77.2
SBT$_{p=0.9}$	82.5	78.5	70.6	77.2
CL-MPPCA	79.1	80.4	72.9	77.5
BeatGAN	78.1	87.5	69.6	78.4
SBT$_{p=0.5}$	87	78.4	69.8	78.4
SBT$_{p=0.75}$	88.0	79.3	70.6	79.3
LSTM-VAE	82.3	82.6	78.1	81.0
OmniAnomaly	85.2	87.7	86.9	86.6
Anomaly Transformer	92.3	93.6	96.7	94.2

Table 2.4: A summary of time series forecasting models on three datasets. Each SBT model is run three times with different weight seeds and averaged. Standard deviation is less than 0.01.

Model	ECL		Weather		ETTm1	
	MSE	MAE	MSE	MAE	MSE	MAE
Informer	0.185	0.301	0.159	0.197	0.051	0.150
Pyraformer	0.149	0.305	-	-	0.081	0.214
Dense Trans.	0.182	0.299	0.173	0.225	0.070	0.201
$SBT_{p=0.5}$	0.198	0.316	0.166	0.216	0.059	0.171
$SBT_{p=0.75}$	0.221	0.333	0.168	0.218	0.070	0.191

reconstruction and the actual value. The masking method simulates unseen future data points during train time, making it compatible with the forecasting task during deployment.

We test our model on three datasets used in previous works: ECL contains electricity consumption of 321 clients in Kwh. The dataset is converted to hourly consumption values due to missing data. Weather contains data for twelve hourly climate features for 1,600 location in the U.S. ETTm1 (Electricity Transformer Temperature) contains 15-minute interval data including oil temperature and six additional power load features. Additional training details are available in the Appendix.

We compare our method against the Informer [52] and the Pyra-former [51] trained with single-step forecasting. Both are current state-of-the-art models that have shown robust results compared against a variety of forecasting techniques. Importantly, each method is compatible with multivariate time series forecasting as opposed to some research. We note that these models are built primarily for long-term time series forecasting (LSTF), which we do not cover in this work.

Results We evaluate results in Table 2.4 using MSE and MAE on the test set of each dataset. Results indicate that the SBT model achieves accuracy comparable to the Dense architecture in each dataset at $p = 0.5$. Interestingly, the Weather at ETTm1 SBT models achieved *better* accuracy than the dense model at $p = 0.5$. Both models additionally showed robustness to higher prune rates, with accuracy dropping off slowly. ECL on the other hand showed some sensitivity to prune rate, with a slight drop off when increasing the prune rate. We find that datasets with a higher dimensionality

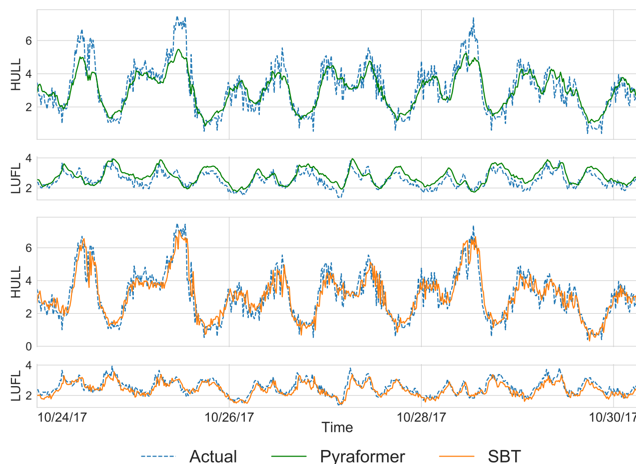

Figure 2.3: Time series predictions on the ETTm1 dataset for the Pyraformer (top) and Sparse Binary Transformer (bottom) . We show 600 predictions across each model for two features (HULL, LUFL).

performed the worst: ECL contains 321 features, while Insect Wingbeats contains 200. Increasing the dimensionality of the model (d) mitigated some of these effects, however it was at the cost of model size and complexity. Despite this, we find that the SBT model is able to predict the general trend of complex patterns in data, as depicted in Figure 2.3.

Compared to state-of-the-art approaches such as the Pyraformer and Informer architectures, our general purpose forecasting approach performs comparably, or slightly worse, on the single-step forecasting task. Metrics were not substantially different for any of the models except for the ECL dataset, where Pyraformer was easily the best model. Comparing the architectures, we find that the SBT model achieves substantially lower computational cost than both the Informer and Pyraformer models. For example, on the ECL dataset, Pyraformer contains 4.7 million parameters and the Informer 12.7 million parameters (both FP32, while the SBT model contains 1.5 million binary parameters.

2.4.4 Architecture

Each model in our framework consists of 2 encoder layers each with a multi-head attention module containing two heads. The feedforward dimensionality for each model is 256 with ReLU used for nonlinearity. Classification models had the best results using Batch Normalization layers, similar to [1], while forecasting models used Layer Normalization typical of other Transformer

models. For anomaly detection we did not use Batch or Layer Normalization. For the output of our models, anomaly detection and forecasting rely on a single decoder linear layer which reconstructs the output to size (m, w), while classification outputs size (d, $num.classes$) and takes the mean of d to formulate a final classification prediction. Further details are included in the Appendix and the code repository.

2.5 Metrics

In this section we estimate the computational savings achieved by using the SBT model. We will begin by introducing the metrics used to estimate computational savings, and will then summarize the results of these metrics for each model and task.

We note that several works have proposed modifications to the Transformer in order to make attention more efficient. In this section, we concentrate on the enhancements achieved by 1) creating a sparsely connected Transformer with binary weights, and 2) simplifying the attention module for time series specific tasks such as single-step prediction and classification. We argue that these enhancements are independent of the achievements made by previous works.

2.5.1 Metrics

FLOPs (Non-zero). In the field of network pruning, FLOPs, or the number of multiply-adds, is a commonly used metric to quantify the efficiency of a neural network [95]. The metric computes the number of floating point operations required for an input to pass through a neural network. We use the ShrinkBench tool to calculate FLOPs, a framework proposed by Blalock et al. [95] to perform standardized evaluation on pruned neural networks.

Our Transformer architecture contains FP32 activations at each layer along with binary weights scaled to $\{-\alpha, \alpha\}$. As a result, no binary operations are performed, and our total FLOPs count is a function of prune rate p. For example, a linear module with a standard FLOPs count of $d \times m$ has a new FLOPs count of $d \times m \times p$, where $p \in [0, 1]$. Linear layers outside of attention do not need window size added to the matrix multiply because the inputs are permuted such that batch size is

Table 2.5: Non-zero FLOPs equations for various attention modules. These calculations assume $\mathbf{Q}, \mathbf{K}$ and $\mathbf{V}$ are equal sized projections in $\mathbb{R}^{w \times d}$, and $d' = d/h$. $\mathbf{QV}^\intercal$ and $\mathbf{AV}$ are additionally multiplied by h. $\mathbf{Q}$-scaling and softmax FLOPs excluded from this table.

Attention Type	Q,K,V Proj.	$\mathbf{QV}^\intercal$	$\mathbf{AV}$
Canonical	d^2w	$d'w^2$	$d'w^2$
Step-T Mask	$(d^2w)p_a$	$(w-1)d'$	$2(w-1)d'$
Q,K,V Mask	$(d^2w)p_a$	$d'(wp_a)^2$	$(d'w^2)p_a$

the second dimension of the layer input. Each equation counts the number of *nonzero* multiply-adds necessary for the neural network.

Furthermore, we modify the FLOPs for the attention module to account for step-t attention mask and the fixed $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ mask, as summarized in Table 2.5. In the standard attention module where $\mathbf{Q}, \mathbf{K}$ and $\mathbf{V}$ are equal sized projections, matrix multiply operations ($\mathbf{QV}^\intercal$, $\mathbf{AV}$) for each head equate to $d'w^2$, where $d' = d/h$. For step-t attention, we only require computation at the current time step (the last row in Figure 2.2), while each each of the identities for past time steps equates to one. $\mathbf{AV}$ requires double the computations because $\mathbf{V}$ contains FP32 activations multiplied by the diagonal in $\mathbf{A}$. For the fixed mask, since $\mathbf{Q}$ and $\mathbf{K}$ are sparse projections, we only require $(wp_a)^2$ nonzero computations in the matrix multiply. Since $\mathbf{A}$ is a dense matrix, we require w^2 FLOPs to multiply sparse matrix $\mathbf{V}$.

A simplified equation for network FLOPs becomes $2L + N(2L + MHA)$, where L is a linear layer, N is the number of attention layers, and MHA is the multihead attention FLOPs (details described in Table 2.5). Several FLOP counts are omitted from this equation, which we include in our code, including positional encoding, Q-scaling, and layer and batch norm.

Storage Size. We measure the size of each model in total bits. Standard networks rely on weights optimized with the FP32 data type (32 bits). We consider each binarized module in our architecture to contain single bit weights with a single FP32 α parameter for each layer. Anomaly detection and classification datasets contain 14 binarized modules, and forecasting contains 18 with the additional binarization of the layer normalization. We note that the binarized quantities are only theoretical as

a result of the PyTorch framework not supporting the binary data type. Hardware limitations are also reported in other works [15].

2.5.2 Model Size Selection

Important to our work is tuning the size of each model. We analyze whether we can create a Dense Transformer with a smaller number of parameters and still retain a performance on par with a larger model. Our motivation for model size selection is two-fold: 1) Previous research has found that neural networks need to be sufficiently overparameterized to be pruned and retain the same accuracy of the dense model and 2) The time series datasets studied in this paper have a smaller number of dimensions than the vision datasets studied in most pruning and model compression papers. The effect of model overparameterization is that we need a dense model with enough initial parameters in order to prune it and still retain high performance. Theoretical estimates on the number of required parameters are proposed by the Strong Lottery Ticket Hypothesis [96, 97] and are further explored in other pruning papers [26, 68]. On the other hand, the limited features of some time series datasets (such as Weather with 7 features) leads us to wonder whether we could simply create a smaller model.

To alter the model size, we vary the embedding dimension d of the model. To find the ideal size of the model, we start from a small embedding dimension (such as 8 or 16), and increase the value in the Dense Transformer until the model performance on the validation set stops increasing. With this value of d, we test the SBT model.

Our results show that in each dataset, Dense Transformers with a smaller embedding dimension d either a) perform worse than the SBT at the optimized size, b) contain more parameters (as measured in total bits), c) have more FLOPs, or d) some combination of the above. In almost every dataset, the smaller Dense Transformer performs worse than the SBT while also requiring more size and FLOPs. The exception to this was Spoken Arabic Digits, where the smaller Dense Transformers ($d = 16$ and $d = 32$) performed slightly better than the SBT with $d = 64$. Additionally, these models had a lower FLOPs count. The advantage of the SBT model in this scenario was a substantially

Table 2.6: Computational savings for Dense Transformers compared to SBTs. SBT models achieve a substantial reduction in size and FLOPs count across all models. We denote parameters in thousands and size and FLOPs in millions, with savings calculated by dividing the Dense values by the SBT values.

					Dense Transformer			Sparse Binary Transformer				~Savings	
Type	Dataset	m	w	d	Params (FP32) (K)	Size (Bits) (Mil.)	FLOPs (Mil.)	p	Params (Binary) (K)	Size (Bits) (Mil.)	FLOPs (Mil.)	Size (Bits) ($\frac{\text{Dense}}{\text{SBT}}$)	FLOPs ($\frac{\text{Dense}}{\text{SBT}}$)
Classification	HB	61	405	64	169.6	5.4	52.7	0.5	102.3	0.1	21.6	×49.1	×2.4
	In.WB	200	30	128	555.5	17.8	5.4	0.5	420.1	0.4	2.7	×40.0	×2.0
	AD	13	93	64	167.1	5.3	2.8	0.5	100.0	0.1	1.3	×49.5	×2.2
	JV	12	29	32	75.5	2.4	0.3	0.5	41.6	0.04	0.2	×52.9	×2.1
	FD	144	62	128	414.9	13.3	8.3	0.5	281.3	0.3	4.0	×44.7	×2.1
Anomaly Detection	MSL	55	50	110	223.7	7.2	4.9	0.75	221.5	0.2	1.0	×32.3	×5.0
	SMAP	25	50	50	75.2	2.4	1.3	0.75	73.7	0.1	0.2	×32.6	×6.1
	SMD	38	200	76	132.8	4.2	19.5	0.75	129.8	0.1	1.9	×32.7	×10.5
Forecast.	ECL	321	200	350	1569.4	50.2	204.8	0.75	1563.9	1.6	74.5	×32.1	×2.7
	Weath	7	200	100	188.0	6.0	28.5	0.5	185.6	0.2	6.2	×32.4	×4.6
	ETTm1	12	200	64	102.0	3.3	15.5	0.5	100.0	0.1	2.6	×32.6	×5.9

lower storage cost than both smaller Dense models. Even if both Dense Transformer models were able to be quantized to 8-bit weights, the storage of the SBT would still be many times lower. The ETTm1 dataset additionally had high performance Dense Transformers with a smaller size $(d = 16, d = 32)$. However, both models were substantially more costly in terms of storage and additionally had a higher FLOPs count. Detailed results are provided in the Appendix.

2.5.3 Analysis

Results in Table 2.6 highlight the large computational savings achieved by SBT. We find that layer pruning reduces FLOPs count (due to the added nonzero computations), while binarization helps with the storage size.

Notably, all models have a FLOPs count at least two times less than the original Dense model. FLOPs are dramatically reduced in the anomaly detection and forecasting datasets, largely due to the step-t masking. Classification datasets have a dense attention matrix, leading to a smaller FLOPs reduction due to the softmax operation and the AV calculation (where V is sparse). We note that

using a higher prune rate can reduce the FLOPs more, however we include results at 50% prune rate for classification since these models achieved slightly better accuracy.

We highlight the storage savings of SBT models by measuring bit size and parameter count. Table 2.6 summarizes the substantial reduction in bit size for every model, with only two SBT models having a bit size greater than 1 million (Insect Wingbeats and ECL). The two models with a larger size also had the highest dimensionality m, and consequently d.

We note that SBT models contain a small number of FP32 values due to the single α parameter in each module. Additionally, we forego a learnable encoding layer in SBT classification models, leading to a smaller overall count. Finally, no bias term is added to the SBT modules, leading to a smaller number of overall parameters.

Compared to other efficient models, our model generally has a lower FLOPs count. For example, MobileV2 [98] has 16.4 million FLOPs when modeling CIFAR10, while EfficientNetV2 [99] has 18.1 million parameters.

2.6 Ablation Studies

We conduct two ablation studies testing the effects of removing the individual pruning mechanisms from the attention computation. We note that the attention pruning methods complement Biprop – Biprop mainly reduces the model size, whereas attention pruning does a better job at reducing the FLOPs. Each ablation experiment is averaged over three experimental runs with different seeds.

Table 2.7 highlights the effects of removing random pruning from the time series classification models. Notably, Biprop plus random pruning performs comparably to, or better than, Biprop on its own. Adding random pruning even outperforms using only Biprop with the Japanese Vowels dataset.

Table 2.8 highlights the results of attention variations for both anomaly detection and forecasting tasks. Specifically, we look at our proposed approach (Biprop+Step-T Mask), Biprop plus an

identity matrix mask in the attention layers, and finally Biprop only. We report results using mean squared error (MSE) loss averaged over three runs.

Results show that Biprop plus the Step-T mask performs comparably to using Biprop only. For anomaly detection tasks, the MSE is even lower compared to just using Biprop. Comparing both methods to the Biprop plus the identity matrix attention mask, we can see a significant difference in the results: the identity matrix attention mask attains a higher loss in each case.

Table 2.7: We compare Biprop with Biprop plus random pruning on classification tasks. We find that random pruning of the attention activations does not hurt classifcation accuracy, and in fact helps it in the case of the Japanese Vowels dataset.

Dataset	Biprop+Random Pruning	Biprop
Arabic Digits	98.2	98.2
Heartbeat	77.7	77.1
Insect Wingbeats	64.1	64
Japanese Vowels	95.3	84.4
Face Detection	66.1	65.9

Table 2.8: We compare Biprop plus the Step-T attention mask with two other methods. We find that Biprop with the Step-T mask performs similarly to using Biprop with full attention (Biprop Only). Biprop with an Identity Mask on the attention computation performs worse than the other two methods. We report results using MSE loss averaged across three runs.

Dataset	Biprop+ Step-T	Biprop+ Identity Matrix	Biprop Only
Anomaly Detection			
MSL	0.277	0.364	0.357
SMAP	0.117	0.131	0.125
SMD	0.037	0.041	0.052
Forecasting			
ETTm1	0.059	0.068	0.070
ECL	0.198	0.204	0.182
Weather	0.166	0.180	0.166

2.7 Additional Results

Key to the cohesiveness of this dissertation, we test the SBT algorithm with Iterative Weight Recycling, an algorithm described in Chapter 4. Iterative Weight Recycling is an easily applied approach to improve the performance of sparse neural networks. It can be applied to both sparse-only models (e.g. Edge-Popup) and sparse and quantized models (Biprop). Since SBTs are built using part of the Biprop algorithm, we can easily apply Iterative Weight Recycling to the SBT model. We specifically test out the approach on time series forecasting tasks because it has the weakest results out of the three tasks.

Similar to previous approaches [68, 69] we use a recycling rate of 0.1. The rate provides enough of a change to make a difference, while not being too overpowering. We modify the recycling frequency with rates of 5 epochs, 10 epochs, and 20 epochs. Our models are trained for 100 epochs. Finally, we run each model five times and take the mean of each.

Results in Table 2.9 indicate that iterative weight recycling provides a marginal benefit across across all tasks compared to the baseline SBT model. All models except for ETTm1 at four-step forecasting get a small benefit from some form of weight-recycling. The frequency is a sensitive quantity, for example in the ECL dataset model in four-step forecasting, using a frequency of five makes the model performance drop drastically. However, the values are somewhat stable in other runs. Overall, Iterative Weight Recycling shows potential to improve SBT models across five out of six tasks and three datasets, highlighting the efficacy of its use.

2.8 Discussion

We show that Sparse Binary Transformers attain similar accuracy to the Dense Transformer across three multivariate time series learning tasks: anomaly detection, forecasting, and classification. We estimate the computational savings of SBT's by counting FLOPs as well as total size of the model.

Table 2.9: Results of applying the Iterative Weight Recycling algorithm to the SBT model. We test the resutls on time-series forecasting tasks on both single-step forecasting (top) and 4-step forecasting (bottom). We run each experiment five times and report the mean MSE of each.

Model	Forecasting Steps	IWR Freq. (Epochs)	ECL MSE	Weather MSE	ETTm1 MSE
Informer	1	-	0.1850	0.1590	0.0510
Pyraformer	1	-	0.1490	-	0.0810
Dense Transformer	1	-	0.2350	0.1633	0.0446
$SBT_{p=0.5}$	1	-	0.2164	0.1681	0.0561
$SBT_{p=0.5}$	1	5	0.2141	0.1672	0.0544
$SBT_{p=0.5}$	1	10	0.2062	0.1677	0.0553
$SBT_{p=0.5}$	1	20	0.2067	0.1661	0.0550
Dense Transformer	4	-	0.2385	0.2095	0.0776
$SBT_{p=0.5}$	4	-	0.2391	0.2121	0.0809
$SBT_{p=0.5}$	4	5	0.2651	0.2113	0.0848
$SBT_{p=0.5}$	4	10	0.2312	0.2105	0.0838
$SBT_{p=0.5}$	4	20	0.2350	0.2118	0.0816

2.8.1 Applications

SBTs retain high performance compared to dense models, coupled with a large reduction in computational cost. As a result, SBTs have the potential to impact a variety of new domains. For example, sensors and small embedded systems such as IoT devices could employ SBTs for intelligent and data-driven decisions, such as detecting a malicious actor or forecasting a weather event. Such devices could be extended into new areas of research such as environmental monitoring. Other small capacity applications include implantable devices, healthcare monitoring, and various industrial applications.

Finally, lightweight deep learning models can also benefit larger endeavors. For example, space and satellite applications, such as in the MSL and SMAP telemetry datasets, collect massive

amounts of data that is difficult to monitor. Employing effective and intelligent algorithms such as the Transformer could help in the processing and auditing of such systems.

2.8.2 Limitations and Future Work

Although SBTs theoretically reduce computational costs, the method is not optimized for modern libraries and hardware. Python libraries do not binarize weights to single bits, but 8-bit counts. Special hardware in IoT devices and satellites could additionally make implementation a burden. Additionally, while our implementation shows that sparse binarized Transformers exist, the Biprop algorithm requires backpropagation over a dense network with randomly initialized FP32 weights. Hence, finding accurate binary subnetworks requires more computational power during training than it does during deployment. This may be a key limitation in devices seeking autonomy. In addition to addressing these limitations, a logical step for future work would be to implement SBTs in state-of-the-art Transformer models such as the Pyramformer for forecasting and the Anomaly Transformer for time series anomaly detection.

SBTs have the potential to enable widespread use of AI across new applications. The Transformer stands as one of most powerful deep learning models in use today, and expanding this architecture into new domains provides promising directions for the future.

Chapter 3

Randomly Initialized Subnetworks with Iterative Weight Recycling

3.1 Summary

The Multi-Prize Lottery Ticket Hypothesis posits that randomly initialized neural networks contain several subnetworks that achieve comparable accuracy to fully trained models of the same architecture. However, current methods require that the network is sufficiently overparameterized. In this work, we propose a modification to two state-of-the-art algorithms (Edge-Popup and Biprop) that finds high-accuracy subnetworks with no additional storage cost or scaling. The algorithm, Iterative Weight Recycling, identifies subsets of important weights within a randomly initialized network for intra-layer reuse. Empirically we show improvements on smaller network architectures and higher prune rates, finding that model sparsity can be increased through the "recycling" of existing weights.

3.2 Introduction

The Lottery Ticket Hypothesis [15] demonstrated that randomly initialized DNNs contain sparse subnetworks that, when trained in isolation, achieve comparable accuracy to a fully-trained dense network of the same structure. The results of the hypothesis indicate that over-parameterized DNNs are no longer necessary; instead, finding "winning ticket" sparse subnetworks can yield high accuracy models. The consequences of winning tickets are abundant in practical use: we can train DNNs with a decreased computational cost [100] including memory consumption and inference time, and additionally enable wide-spread democratization of DNNs with a low carbon footprint.

Expanding on the Lottery Ticket Hypothesis, Ramanujan et al. [27] reported a remarkable finding: we do not have to train neural networks at all to find winning tickets. Their algorithm, Edge-

Popup, uncovered sparse subnetworks within *randomly initialized* DNNs that achieved comparable accuracy to fully trained models. This phenomena was mathematically proven in the Strong Lottery Ticket Hypothesis [67]. Practically, this finding showed that gradient-based weight optimization is not necessary for a neural network to achieve high accuracy. Moreover, it allows us to overcome difficulties of gradient-based sparsification, such as getting stuck at local minima and incompatible backpropagation [26]. Finally, randomly initialized "winning ticket" subnetworks have been shown to be more robust than other pruning methods [101].

Despite this fascinating discovery, it also marked a key limitation to existing work: randomly initialized DNNs require a large number of parameters in order to achieve high-accuracy. In other words, to reach the same level of performance as dense networks trained with weight-optimization, randomly initialized models need more parameters, and hence more memory space. Subsequent works have relaxed the bounds proposed by the Strong Lottery Ticket Hypothesis [96, 97], showing mathematically that network width needs to be only logarithmically wider than dense networks. Chijiwa et. al [68] proposed an algorithmic modification to Edge-Popup, iterative randomization (IteRand), showing that we can reduce the required network width for weight pruning to the same as a fully trained model up to constant factors.

In this work, we propose an algorithm to find accurate sparse subnetworks in randomly initialized DNNs and BNN's. Our approach exploits existing weights in a network layer, identifying subsets of trivial weights and replacing them with weights influential to a strong subnetwork. We demonstrate our results in Figure 3.1, showing improvements on a variety of architectures and prune rates.

Our contributions are as follows:

- We propose a new algorithm, Iterative Weight Recycling, which improves the ability to find highly accurate sparse subnetworks within randomly initialized neural networks. The algorithm is an improvement to both Edge-Popup (for DNNs) as well as Biprop (BNN's). The algorithm identifies k extraneous weights in a model layer and replaces them with k relevant weight values.
- We confirm the Multi-Prize Lottery Ticket Hypothesis with tighter overparameterization requirements, showing empirically that sparse subnetworks in limited parameter models

achieve high accuracy compared to their dense counterpart. We show that *recycling* weights in a randomly initialized network enables us to achieve this.

3.3 Related Work

In this section we review research related to neural network compression. We concentrate on work related to the Lottery Ticket Hypothesis (LTH) [15], as our proposed work aims to improve on this class of compression algorithms. The LTH has garnered a significant amount of attention in the research community in the last several years. We also introduce work related to Transformers [8], including Transformers for time series modeling [58] and efficient Transformers [71]. Finally, we introduce research related to empirical implementations of DNNs on resource-constrained machines [102].

3.3.1 Neural Network Pruning and Compression

The effectiveness of sparse neural networks was first demonstrated by Lecun et al. [14]. With the advent of deep learning, the size and efficiency of deep learning models quickly became a critical limitation. Naturally, research aimed at decreasing size [13, 103], and limiting power and energy consumption [66].

The Lottery Ticket Hypothesis found that dense networks contained randomly-initialized subnetworks that, when trained on their own, achieved accuracy comparable to the original dense model. However, the approach required training a dense network in order to identify winning tickets. Subsequent work identified strategies to prune DNNs *without* a pretrained model using methods such as greedy forward selection [104], mask distances [105], flow preservation techniques [106, 107], and channel importance [19].

Malach et al. [67] first showed mathematical justification for the LTH with the *Strong* LTH, showing that the width of a randomly-initialized network must exceed some target network by a polynomial term. Additionally they showed that the neural networks do not need to be trained at all.

Pensia et al. [96] and Orseau et al. [97] offered improvements on this theorem, showing that the network width needs to be only logarithmically wider than fully-trained dense networks.

Ramanujan et al. [27] provided substantial empirical results for the Strong LTH by showing that *randomly-initialized* neural networks contain high-accuracy subnetworks. They found high-performing neural networks masks across a range of architectures and datasets. They achieved this using their proposed Edge-Popup algorithm. Chijiwa et al. [68] improved on previous works, finding smaller *and* more accurate randomly initialized neural networks. They achieved this with *IteRand* (Iterative Randomization), an algorithm which re-randomizes neural networks weights iteratively in order to find highly-accurate subnetworks in smaller parameter spaces. They modified the Edge-popup algorithm described in [27] to empirically demonstrate their results. Finally, the recently proposed Multi-Prize Lottery Ticket poses that there are *many* randomly-initialized subnetworks which attain accuracies comparable to dense fully-trained models. They extend this finding to binary neural networks, detailed below.

3.3.2 Randomized Neural Networks

Relevant to work described in [27, 67, 68], randomized neural networks [108] have also been explored in shallow architectures. Several applications explore randomization, including random vector functional links [109–111], random features for kernel approximations [112–114], reservoir computing [115], and stochastic configuration networks [116]. Such applications employ unique training processes that can be much simpler than training a full architecture, while potentially losing some accuracy as a result.

3.3.3 Binary Neural Networks

BNN's studied in this work fall into the class of quantized neural networks. Like pruning, quantization is a natural approach for model compression [21], and involves creating lower precision weights than the standard 32-bit floating point weights used in DNNs. Diffenderfer and [26] employ binary quantization (binarization) of neural network weights, which they call MPT(1/32), and additionally network weights *and* activations (MPT(1/1)). Biprop searches for MPTs in both

scenarios, with strong results in MPT(1/32)'s and promising results in MPT(1/1). In this work, we consider networks with only binary weights MPT(1/32)'s). Common techniques for creating quantized networks include post-training quantization with retraining [31, 32] and quantization-aware training [117]. In this work, and in the Biprop algorithm proposed in [26], quantization-aware binary neural networks are trained with parameter α, which enables floating-point weights to learn a scale parameter prior to binarization [118].

3.4 Background

In this section we review current state-of-the-art methods for pruning randomly initialized and binary randomly initialized neural networks and then introduce our method, Iterative Weight Recycling.

Randomly Initialized DNNs Given a neural network $f(x;\theta)$ with layers 1,...L, weight parameters $\theta \in \mathbb{R}^n$ randomly sampled from distribution $\mathcal{D}$ over $\mathbb{R}$, and dataset x, we can express a subnetwork of $f(x;\theta)$ as $f(x;\theta \odot \mathrm{M})$, where $\mathrm{M} \in \{0,1\}^n$ is a binary mask and $\odot$ is the Hadamard product.

Edge-popup, proposed by Ramanujan et al. [27], finds M within a randomly-initialized DNN by optimizing weight scoring parameter $\mathrm{S} \in \mathbb{R}^n$ where $\mathrm{S} \sim \mathcal{D}_{score}$. S_i can be intuitively thought of as an *importance score* computed for each weight θ_i. The algorithm takes pruning rate hyperparameter $p \in [0,1]$, and on the forward pass computes M at M_i as

$$\mathrm{M}_i = \begin{cases} 1 & \text{if } |\mathrm{S}_i| \in \{\tau(i)_{i=1}^{k_j} \geq [k_j p/100]\} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

where τ sorts indices $\{i\}_{i=1}^{j} \in \mathrm{S}$ such that $|S_{\tau(i)}| \leq |S_{\tau(i+1)}|$. In other words, masks are computed at each weight by taking the absolute value of scores for each layer, and setting the mask to 1 if the absolute score value falls within the top 100*p%, otherwise they set the mask to zero. They use the straight-through estimator [70] to backpropagrate through the mask and update S via SGD.

Chijiwa et al. [68] improved on the Edge-Popup algorithm with their algorithm, titled IteRand. They show that by rerandomizing pruned network weights during training, better subnetworks can be found. They theoretically prove their results using an approximation theorem indicating rerandomization operations effectively reduce the required number of parameters needed to achieve high-accuracy randomly-initialized models.

The IteRand algorithm is mainly driven by two hyperparameters: K_{per} and re-randomization rate r. K_{per} drives the frequency weights will be re-randomized. The second hyperparameter, r, denotes a *partial* re-randomization of pruned weights. To achieve the best results, the authors set r to 0.1, meaning re-randomizing 10% of pruned weights (where $\mathrm{M} = 0$).

Algorithm 1 Edge-Popup with IteRand

1: **Require:** $\theta \sim \mathcal{D}_{weight}$, $\mathrm{S} \sim \mathcal{D}_{score}$, p, K_{per}, r
2: **Input:** Dataset(X,Y)
3: **function** EDGE-POPUP(S, M, $f(x)$)
4:     **for each** $l \in L$ **do**
5:         **if** $|s_i| \in$ top k $|\mathrm{S}_l|$ **then** $\mathrm{M}_i = 1$ **else** $\mathrm{M}_i = 0$
6:     **end for**
7:     **return** S, M
8: **end function**
9: **for** *i=1 ..., N-1* **do**
10:     $x, y \leftarrow$ MINIBATCH(X, Y)
11:     $\mathrm{S}, \mathrm{M} \leftarrow$ EDGE-POPUP$(\mathrm{S}, \mathrm{M}, f(x))$
12:     **if** $i \mod K_{per} = 0$ **then**
13:         $\theta \leftarrow Rerandomize(\theta, \mathrm{M})$
14:     **end if**
15: **end for**

Randomly-Initialized Binary Neural Networks (BNNs) Complementary to the findings reported in the previous section, Diffenderfer and Kailkhura [26] described a new method for finding high accuracy subnetworks within binary-weighted models. This finding provides us the ability to store discrete network weights rather than full-precision floating-point numbers, leading to substantial compression of large models and hence lower memory and storage costs. In this section, we summarize the Biprop algorithm.

We start with a modification of the function described in the previous section, replacing $\theta \in \mathbb{R}^n$ with binary weights $\mathcal{B} \in \{-1, 1\}$. The resulting network function becomes $f(x; \mathcal{B} \odot \mathrm{M})$, with

mask M over binary weights. Further, Biprop introduces scale parameter $\alpha \in \mathbb{R}$, which utilizes floating-point weights θ *prior* to binarization [118]. The learned parameter rescales binary weights to $\{-\alpha, \alpha\}$, with the resulting network function becoming $f(x; \alpha(\mathcal{B} \odot \mathrm{M}))$. Parameter α is updated with $||\mathrm{M} \odot \theta||_1/||\mathrm{M}||_1$, with M being multiplied by α for gradient descent (the straight-through estimator is still used for backpropagation). During test-time, the learned alpha parameter simply scales a binarized weight vector. As a result, only bit representations of the weights are needed at positive mask values (± 1 where $\mathrm{M} = 1$), substantially reducing memory, storage, and inference costs.

Empirically, Diffenderfer and Kailkhura [26] are able to produce high accuracy binary subnetworks using Biprop on a broad range of network architectures. They theoretically prove this result on models with sufficient over-parameterization. In the subsequent section we show how we can modify this algorithm, as well as Edge-Popup, with Weight Recycling to achieve high-performance on lower parameter architectures.

3.5 Methodology and Scope

In this section we detail Iterative Weight Recycling, first summarizing the methodology behind the approach, and subsequently detailing the experimental setup and results. Finally we perform empirical analysis on the algorithm, with results showing that Iterative Weight Recycling emphasizes keeping high norm weights values similar to the traditional L1 pruning technique.

We consider $f(x; \theta)$ as an l-layered neural network with ReLU activations, dataset $x \in \mathbb{R}^n$ with weight parameters $\theta \sim \mathcal{D}_{weight}$ and dataset x. We freeze θ and additionally turn off the bias term for each l. Our model is initialized similarly to Edge-Popup and Biprop: a score pararmeter S for each θ, where S_i learns the importance of θ_i. Additionally, we set a pruning rate $p \in [0, 1]$. $\mathcal{D}_{weight}$ is initialized using Kaiming Normal initialization (without scale fan) for Biprop and Signed Constant Initialization for Edge-Popup. Further, $\mathcal{D}_{score}$ is initialized with Kaiming Uniform with seed 0.

Weight recycling works on an iterative basis, similar to IteRand [68]. We define two hyperparameters, K_{per} and r, where K_{per} is the frequency with which we change weights, and r is the *recycling*

rate. During the recycling phase, we compute k as the number of weights we want to change in a given layer as $j * r$, where j is the size of S at layer l. We retrieve subsets $\mathrm{S}_l^{low}, \mathrm{S}_l^{high} \subset \mathrm{S}_l$ containing the lowest absolute k scores and highest absolute k scores at each layer:

$$\mathrm{S}_l^{low} = \{\tau(i)_{i=1}^{k}\}, \quad \mathrm{S}_l^{high} = \{\tau(i)_{i=j-k}^{j}\} \tag{3.2}$$

where τ sorts $\{i\}_{i=1}^{j} \in \mathrm{S}$ such that $|S_{\tau(i)}| \leq |S_{\tau(i+1)}|$. Here, $\{i\}_{i=1}^{j}$ equates to the index values associated with set $\{|S_\tau|\}_{i=0}^{j}$. Next, we retrieve weight values associated with S_l^{high} and S_l^{low}, with $\{i, ..., k\}_S = \{i, ..., k\}_\theta$. Finally, we set θ_l^{low}=θ_l^{high}. Effectively, the Iterative Weight Recycling algorithm finds S values and their associated index (where $i_{\mathrm{S}} = i_\theta$) and retrieves the weight value associated to the index, for both high and low S scores. The algorithm replaces low S weight values with high S weight values, discarding of low S weight values. Algorithm 2 denotes the equation in pseudo-code form.

Algorithm 2 Weight Recycling. Replace line 14 in Algorithm 1 with the following method

```
1: function WEIGHT-RECYCLE(S, θ)
2:     for each l ∈ L do                                  ▷ layer of size j
3:         k ← j * r                ▷ Calculate number of weights to change
4:         S_l^high ← highest k |S_l|    ▷ Indices of top k abs(score) values
5:         S_l^low ← lowest k |S_l|   ▷ Indices of bottom k abs(score) values
6:         θ_l[S_l^low] ← θ_l[S_l^high]         ▷ Replace low θ_l with high θ_l
7:     end for
8:     return S, M
9: end function
```

3.5.1 Scope

Next, we detail the scope of the research for our proposed algorithm Iterative Weight Recycling. Broadly, we will test the approach against previous benchmarks Edge-Popup [27], IteRand [68], and Biprop [26]. Previous works have mainly focused on test set accuracy to verify their results empirically. We will follow this same approach. While both IteRand and Biprop have theorems to go along with their empirical algorithms, we will concentrate on empirical results.

Model Architectures and Datasets We will use model architectures and datasets similar to the three previous works [26, 27, 68]. Conv-2 to Conv-8 are VGG-like CNNs [119] with depth d = 2 to

8. We additionally use their "wide" analogues, which introduces a scale parameter at each layer to influence the specific width of each layer width w = 0.1 to 1. Additionally we use ResNets [120], which utilize skip connections and batch normalization. The CIFAR-10 dataset is used for all experiments, with ImageNet experiments additionally included. Non-affine transformation is used for all CIFAR-10 experiments, and ResNets use a learned batch normalization similar to Diffenderfer and Kailkhura [26].

Prune Rates We will apply prune rates $\{0.2, 0.4, 0.5, 0.6, 0.8, 0.9\}$. These values are the same as previous works. We will additionally test our method at prune rates above 0.9.

Random Seeds To ensure reproducibility and accurate results, in Iterative Weight Recycling experiments we will use three different initializations, and report the average accuracy, with error bars denoting the lowest and highest accuracy.

Benchmarks We compare the performance of Iterative Weight Recycling to Edge-Popup, Biprop, and IteRand algorithms using the same hyperparameters. For each baseline algorithm, we use the hyperparameters that yielded the best results in the original papers: Signed Constant initialization for Edge-Popup/IteRand, and Kaiming Normal with scale fan for Biprop. For our algorithm, we use these same initialization strategies, except for Biprop with Weight Recycling we did not use scale fan as this yielded slightly better results. Additionally, for IteRand we use the same K_{per} and r as the paper: $K_{per} = 1$ (once per epoch), with $r = 0.1$.

In the research questions, we note that existing algorithms (Edge-Popup, IteRand, Biprop) find accurate subnetworks in randomly-intialized models exist in models which are at least 4 layers deep or have a width of roughly one.

Iterative Weight Recycling Hyperparameters For our algorithm, we choose $K_{per} = 10$ and $r = 0.2$ for all models. We find that less aggressive recycling yields better results, hypothesizing that recycling weights too frequently yielded highly redundant values.

3.6 Experimental Setup

To begin, we use model architectures and datasets similar to the three previous works. Conv-2 to Conv-8 are VGG-like CNNs [119] with depth d = 2 to 8 We additionally use their "wide" analogues, which introduces a scale parameter at each layer to influence the specific width of each layer width w = 0.1 to 1. Additionally we use ResNets [120], which utilize skip connections and batch normalization. The CIFAR-10 dataset is used for all experiments, with ImageNet experiments included in the Appendix. Non-affine transformation is used for all CIFAR-10 experiments, and ResNets use a learned batch normalization similar to [26]. We apply similar pruning rates to previous works $\{0.2, 0.4, 0.5, 0.6, 0.8, 0.9\}$, and additionally test our method at prune rates above 0.9. In Iterative Weight Recycling experiments, we use three different initializations, and report the average accuracy, with error bars denoting the lowest and highest accuracy.

We compare the performance of Iterative Weight Recycling to Edge-Popup, Biprop, and IteRand algorithms using the same hyperparameters. For each baseline algorithm, we use the hyperparameters that yielded the best results in the original papers: Signed Constant initialization for Edge-Popup/IteRand, and Kaiming Normal with scale fan for Biprop. For our algorithm, we use these same initialization strategies, except for Biprop with Weight Recycling we did not use scale fan as this yielded slightly better results. Additionally, for IteRand we use the same K_{per} and r as the paper: $K_{per} = 1$ (once per epoch), with $r = 0.1$. For our algorithm, we choose $K_{per} = 10$ and $r = 0.2$ for all models. We found that less aggressive recycling yielded better results, hypothesizing that recycling weights too frequently yielded highly redundant values.

3.7 Results

In this section we test the effects of network overparameterization and prune rate on subnetwork performance, with the goal of empirically verifying the Iterative Weight Recycling compared to Edge Popup [27], Biprop [26], and IteRand [68] algorithms. We follow previous works and test neural networks with varying depth and width, and additionally test each algorithm at aggressive prune rates.

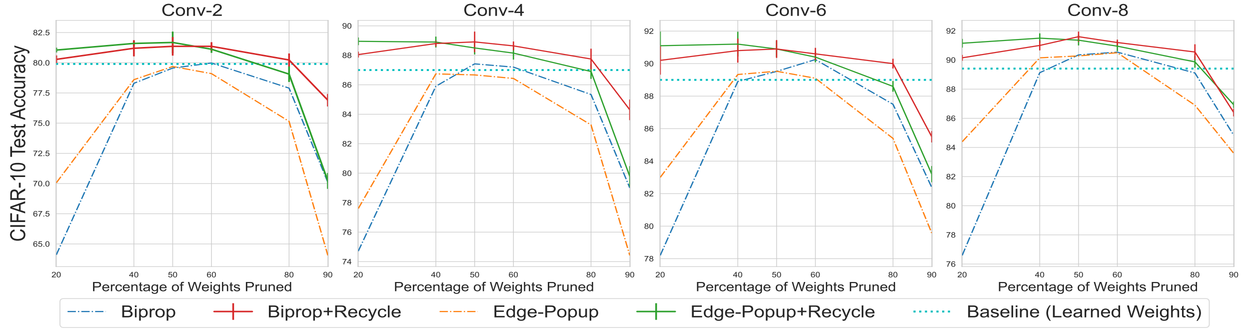

Figure 3.1: Performance of Iterative Weight Recycling at varying network depth: We compare the accuracy of our algorithm against Edge-Popup, Biprop, and a densely trained baseline. We use varying prune rates and varying network depths on VGG-like architectures. We train and evaluate all algorithms on the CIFAR-10 dataset using the same hyperparameter's and training procedure.

Varying Depth In Figure 3.1, we vary the depth of VGG architectures from 2 to 8 and compare the test accuracy at different pruning rates. We observe a clear advantage to using Biprop+Weight Recycling: at every prune rate and model architecture, Weight Recycling outperforms both Biprop and Edge-Popup. Additionally, Biprop with Weight Recycling generally outperforms Edge-Popup with Weight Recycling at higher prune rates. Iterative Weight Recycling outperforms dense models in each architecture except when 90% of the weights have been pruned. Notably, we discover that Iterative Weight Recycling is able to achieve accuracy exceeding the dense model in Conv-2 architectures. This is the first such observation on a low-depth model – Edge-Popup and Biprop research reported test accuracy *near* the dense model, however never clearly exceeded it. Further, Iterative Weight Recycling is able to achieve an 80.23% test accuracy with just 20% of the weights.

Varying Prune Rate In Figure 3.3, we demonstrate the results of Biprop, Edge-Popup, IteRand, and Iterative Weight Recycling (Biprop) on DNNs with prune rates above 80%. Iterative Weight Recycling shows favorable results with limited parameter counts. Notably, the algorithm consistently outperformed IteRand at aggressive prune rates between 80% and 99%. At more modest prune rates (20%-60%), Iterative Weight Recycling was comparable to IteRand.

In the ResNet18 architecture (11 million parameters), our algorithm was able to find winning tickets with just 5% of the random weights. These results are further evidence that overparameterization helps in the identification of high performing subnetworks.

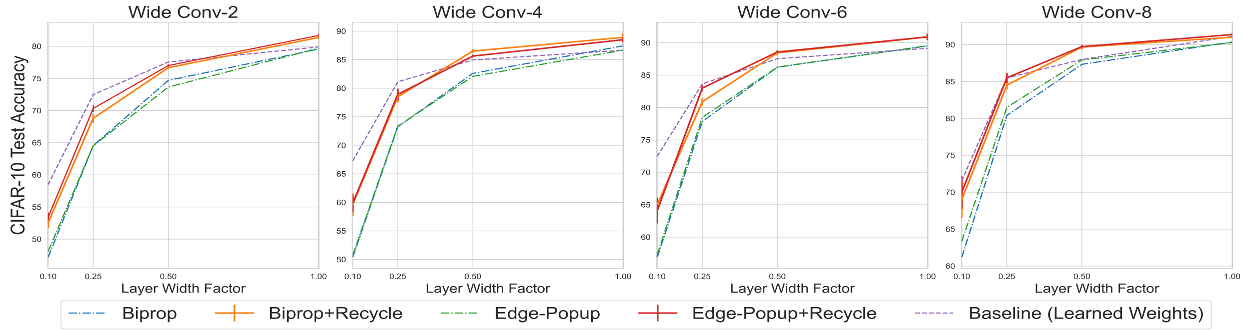

Figure 3.2: Effects of Varying Width at 50% Prune Rate Results of Dense, Biprop, and Biprop+Iterative Weight Recycling models at varying network widths less than one. Using Iterative Weight Recycling in addition to Biprop yields winning tickets with a comparable accuracy to densely trained models at just 50% width factor in all architectures.

Varying Width We also consider network width as a factor to control network parameterization. Previous showed that as we increase width, our chances of finding winning tickets increased. Ramanujan et al. [27] found winning tickets at width factors greater than 1, while Diffenderfer et al. [26] reported winning tickets around width factor 1.

In Figure 3.2, we demonstrate the efficacy of Iterative Weight Recycling on networks with width factors less than one. Results show that in each architecture, we can find winning tickets at just 50% width. In practical terms, in a Conv-4 architecture this computes to just 25% of the parameters as a Conv-4 with width factor 1 (600k vs. 2.4m). Additionally, our Conv-4 architecture with width factor 0.5 achieved an accuracy of 86.5% compared to 86.66% for a dense Conv-4 with width factor 1.

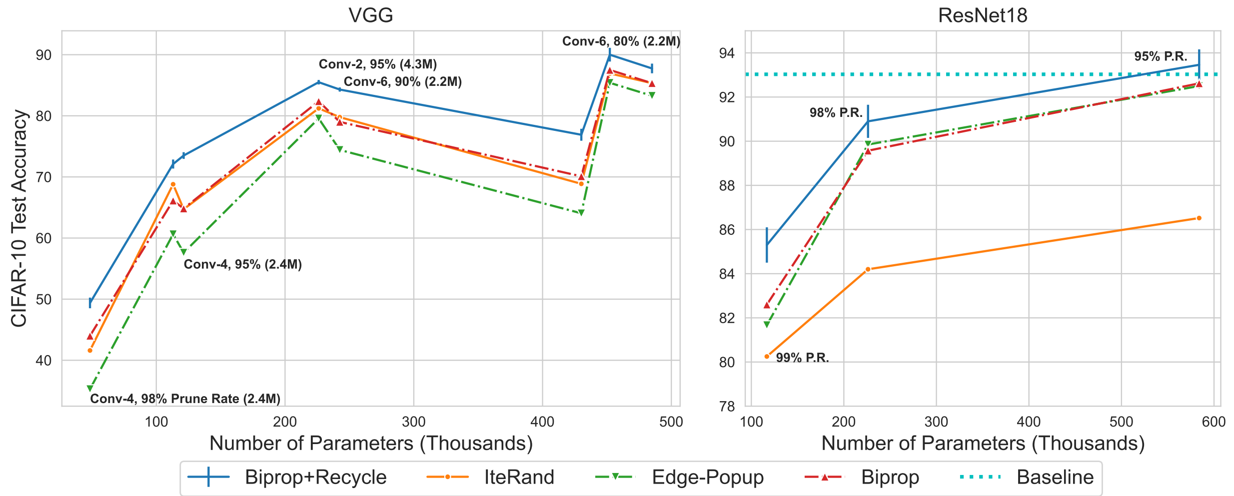

Figure 3.3: Model Performance with Limited Parameters We test the effect of high prune rates (>90%) on model performance, showing that Weight Recycling achieves high accuracy compared to IteRand, Edge-Popup, and Biprop. **Left:** Accuracies of various VGG architectures with prune rates greater than 90% and parameter count less than 500k. **Right:** ResNet18 with prune rates greater than 95%. With just under 600k parameters (95% prune rate), ResNet18 achieves higher accuracy than a dense baseline model (93.1%).

Chapter 4

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

4.1 Summary

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 on 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 model layer. 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) with a 4x 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 utilize the reuse of a single tile per layer in memory.

4.2 Introduction

The progress of modern machine learning can be largely attributed to the exponential growth of DNNs. Empirically, the capacity of DNNs is expanding at an astounding rate [121], a practice supported by theory showing that sufficiently over-parameterized models are in fact necessary for deep learning [122, 123]. Alongside this progress, the growing presence of resource-constrained

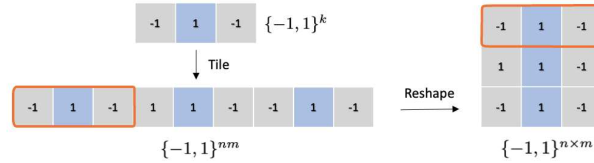

Figure 4.1: Tiling Illustration: A binary tile (top left) of size $k = 3$ is replicated 3 times to create a tiled vector of size 9 (bottom left). This vector is then reshaped to fill a $3x3$ weight tensor (right). Tiling and reshaping is used during the training process of TBNs to learn tiles for filling the parameters of a model (illustrated above). During inference, only a single tile needs to be referenced per layer (top left) – a specialized kernel can reuse the tile throughout layer computation for memory savings.

machines (e.g. cell phones, embedded devices) 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 many scales of application.

Efforts toward efficient deep learning span a broad range of techniques such as architectural design [98, 124], neural architecture search [102], knowledge distillation [125, 126], and quantization [35, 127, 128]. Quantization, which converts high precision neural network weights into discrete values, has achieved success in real-world applications [129, 130], and has been applied down to the scale of BNNs where the weights (and often activations) of a model are single bit values [131].

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 [132] 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 the properties of binary convolutional kernels using minimum spanning trees [133] and sparse kernel selection [134].

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 simple tiling operation is depicted in Figure 4.1. The algorithm learns a condensed parameter representation for each layer by compressing the bit values using tensor reshaping and

aggregation (Figure 4.3). Inspired by XNOR-Nets [86] we also make use of full-precision tensors to calculate a scalar over each 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 4.2. We test the approach on CNNs, Transformers, PointNets, and MLPs. Compared to previous approaches, TBNs achieve better or similar performance on image classification, exhibiting strong generalization at 4x compression rates. Moreover, the algorithm translates to MLP and Transformer models which contain high proportions of fully-connected parameters, enabling for the first time sub-bit DNN compression on these architectures. Empirically, across 2D and 3D image classification tasks TBNs achieve performance on par with a full precision model at roughly 4x compression compared to a standard BNN (i.e. we only require roughly $\frac{1}{4}$ of model parameters to be stored). We additionally achieve strong performance on semantic and part segmentation tasks.

We provide two implementations for model inference, 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. We also implement a GPU-compatible inference kernel using the Triton library [135], enabling memory savings through reuse of a single tile in the highly parallelized setting. The TBNs inference kernel requires 2.8x less peak memory (78.5MB vs. 222.5MB) compared to a standard kernel on the ImageNet Vision Transformer 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 sub-bit compression of fully-connected DNNs, which are relevant in Transformers and MLPs (PointNet, MLPMixer).

- We provide two implementations which 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) a specialized GPU kernel for fully-connected models to leverage memory savings of tiled layers during inference.

4.3 Related Work

Efficient deep learning encompasses multiple areas not covered in this work such as low rank factorization [78, 136], pruning [137], and memory efficiency through input patching [138] and attention tiling [139]. In this section we concentrate on binary and sub-bit neural networks.

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 [140, 141]. The most extreme quantization was conventionally thought to be binarization, where weights can only be ± 1 [85]. 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 [86]. IR-Net [142] preserved information by maximizing information entropy while minimizing the quantization error. ReActNet used generalized activation functions to get within 3% of full-precision accuracy on ImageNet [143]; Shang et al. utilized contrastive learning to learn BNNs [144]. Xu et al. proposed FDA, which estimates sign function gradients in a Fourier frequency domain [145]; Xu et al. proposed ReCU which introduces a rectified clamp unit to address dead weights [146].

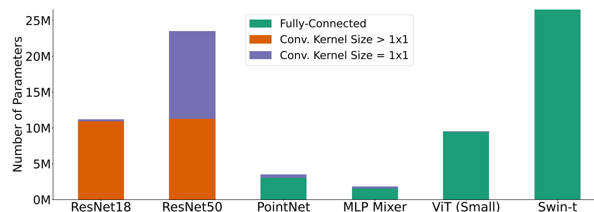

Figure 4.2: Layer composition of DNN architectures: The ResNet series is made up primarily of convolutional layers; Transformer (Swin-T, ViT) and MLP (PointNet, MLPMixer) models consist mostly of fully-connected layers.

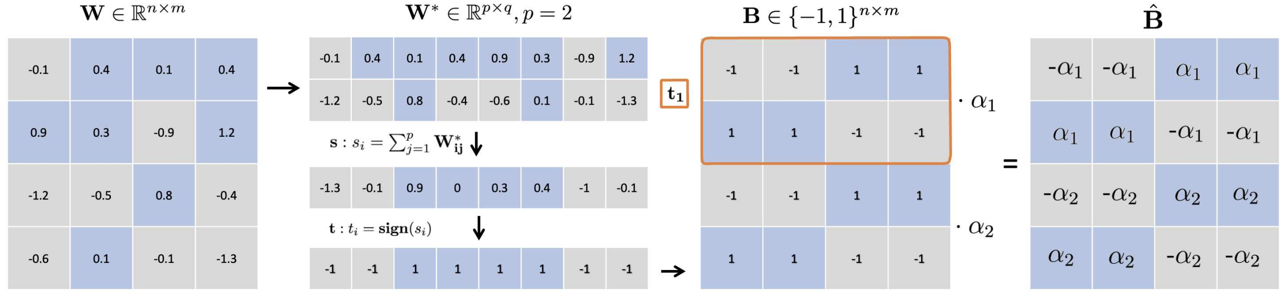

Figure 4.3: Tile Construction During Training: For each layer of a neural network we train a weight tensor ($\mathbf{W}$) (left). During training, we compress the parameter by a factor of p by performing a reshaping (second column top) and then sum operation (second column middle). We use the straight-through estimator to binarize vector $\mathbf{s}$, creating tile $\mathbf{t}$ (second column bottom). We next create binary weights $\mathbf{B}$ from the resulting binary vector by tiling vector $\mathbf{t}$ 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 inference, only a single tile is needed, along with a small number of α scalars.

We note that most BNN research uses binary activations as well as binary weights. 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 BWNNs, and indicate whether benchmark models have binary activations accordingly.

Sub-Bit Quantization Sub-bit DNN compression reduces model sizes to less than a single bit per model parameter. Kim et al. [147] proposed a kernel decomposition to reduce computations in binary CNNs. FleXOR [148] used an encryption technique for storing binary sequences. Wang et. al [132] observed that the full set of binary convolutional kernels tends to cluster into a subset; they formulate a training technique for finding the best subsets of kernels. Lan et al. [149] stack convolutional filters to achieve sub-bit compression. Wang et al. [134] group kernels into binary codebooks for sparse kernel selection. Finally, Vo et al. [133] 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; current approaches are based on utilizing properties of convolutional kernels. While convolutional layers enhance efficiency of neural networks, many architectures still rely on fully-connected layers (Transformers, PointNet, MLPs, etc.). For example, the lightweight Mobile ViT has significant fully-connected parameters [150] (see Figure 4.2).

4.4 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 a 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 In our approach 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 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 elements in the tensor is $N = \prod_{i=1}^{k} d_i$. During training we update $\mathbf{W}^{[\mathbf{l}]}$ via stochastic gradient descent. Our goal is to *compress* $\mathbf{W}^{[\mathbf{l}]}$ by a factor of p, where size N is divisible by p such that $p \times q = N$. To achieve this we reshape tensor $\mathbf{W}^{[\mathbf{l}]}$ as a $p \times q$ dimensional matrix $\mathbf{W}^{[\mathbf{l}]*}$ during forward propagation:

$$\mathbf{W}^{[\mathbf{l}]} \in \mathbb{R}^{d_1, ... d_k} \rightarrow \mathbf{W}^{[\mathbf{l}]*} \in \mathbb{R}^{p \times q} \tag{4.1}$$

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

$$\mathbf{s} = \begin{bmatrix} \sum_{j=1}^{p} \mathbf{W}^{[\mathbf{l}]*}_{\mathbf{1j}} \\ \sum_{j=1}^{p} \mathbf{W}^{[\mathbf{l}]*}_{\mathbf{2j}} \\ \vdots \\ \sum_{j=1}^{p} \mathbf{W}^{[\mathbf{l}]*}_{\mathbf{qj}} \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_q \end{bmatrix} \tag{4.2}$$

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

$$t_i = \begin{cases} 1, & \text{if } s_i > 0 \\ -1, & \text{otherwise} \end{cases} \tag{4.3}$$

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:

$$\mathbf{b^{[l]}} = \mathbf{1_N} \otimes \mathbf{t^{[l]}} \tag{4.4}$$

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

$$\mathbf{B^{[l]}} = \mathbf{vec^{-1}_{d_1,\dots d_k}}(\mathbf{b^{[l]}}) \tag{4.5}$$

where $\mathbf{vec^{-1}_{d_1,\dots d_k}}(\cdot)$ 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 [151]. To achieve this we implement Equations 4.1-4.5 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 $\mathbf{W^{[l]}}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} = \frac{\partial \mathcal{L}}{\partial y^{[l]}} \cdot \frac{\partial y^{[l]}}{\partial \mathbf{W}^{[l]}}, \quad \frac{\partial y^{[l]}}{\partial \mathbf{W}^{[l]}} \approx \frac{\partial y^{[l]}}{\partial \mathbf{B}^{[l]}} \tag{4.6}$$

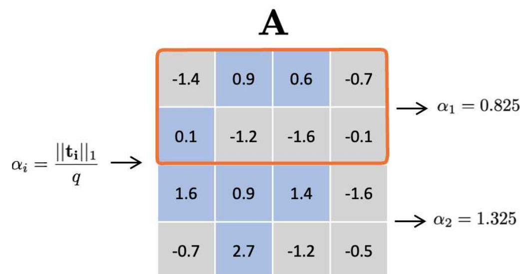

Figure 4.4: We learn scalar α from tensor $\mathbf{A}$ by computing Equation 4.7 or 4.9 over its values ($\mathbf{W}$ can also be used in place of $\mathbf{A}$). The figure visualizes Equation 4.9 which calculates α over each tile.

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.

Tile-wise Scalars Similar to XNORNet [86], we scale $\mathbf{B}^{[l]}$ by α. Rastegari et al. [86] 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 [85, 152, 153]. We can use parameter $\mathbf{W}^{[l]}$ to compute the scalar, since its non-aggregated size is the same as a standard weight tensor:

$$\alpha = \frac{||\mathbf{W}^{[l]}||_1}{N} \tag{4.7}$$

We additionally experiment with an independent tensor, denoted $\mathbf{A}^{[l]} \in \mathbb{R}^{d_1 \times \ldots d_k}$, 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 $\mathbf{t_1}, \mathbf{t_2} \ldots \mathbf{t_p}$ in layer l by utilizing the ith 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 pth tile of layer l:

$$\mathbf{A}^{[l]} \in \mathbb{R}^{d_1, \ldots d_k} \rightarrow \mathbf{A}^{[l]*} \in \mathbb{R}^{q \times p} \tag{4.8}$$

Next, similar to Equation 4.7 we calcuate the 1-norm for each segment of values corresponding to the ith tile in $\mathbf{A}^{[l]}$. We divide this number by q (the size of each tile) to give us $\alpha_1, \alpha_2, \ldots \alpha_p$:

$$\begin{bmatrix} \alpha_1^{[l]} \\ \vdots \\ \alpha_p^{[l]} \end{bmatrix} = \begin{bmatrix} ||\mathbf{A_1^{[l]*}}||_1 q \\ \vdots \\ ||\mathbf{A_p^{[l]*}}||_1 q \end{bmatrix} \tag{4.9}$$

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 input data. Equation 4.9 is depicted in Figure 4.4.

After training is complete, we save a vector of size N/p for each layer along with full-precision scalars (αs).

Hyperparameter Settings We test our models with several hyperparameter configurations to assure the best performance. TBNs primarily contain 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. **Default:** λ=64,000, ImageNet models: λ=150,000

2. Parameter $\mathbf{W}$ for tiling and $\mathbf{A}$ for calculating α. $\mathbf{W}$ is used to learn a tile for each layer; it can also be used to calculate α scalars. Alternatively, we propose a separate parameter $\mathbf{A}$ to compute αs independently, which exhibits a small performance gain. **Default:** $\mathbf{A}$ for calculating α. For ImageNet, we use $\mathbf{W}$.

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.

4.5 Experiments

We next detail our experiments across a range of architectures, datasets, and tasks. We test our approach on CNNs as well as fully-connected models such as PointNet and Transformers. Additional results can be found in the Appendix.

4.5.1 CNN Architectures

In this section we compare TBNs against previous sub-bit compression techniques for Convolutional Neural Networks (CNNs) including SNN [132], MST [133], and Spark [134]. We assess the performance of the techniques on both CIFAR-10 datasets using the ResNet series of models [154] and the VGG-Small model [142], similar to previous works. In addition to the models used in previous research we include ResNet-50 for CIFAR-10, which has 1×1 convolutional kernels. The SNN sub-bit compression algorithm was assessed with Resnet-50 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 [132], is the most similar approach to TBNs.

Results. Table 4.1 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 architectures without a decrease in test set performance at 4x compression. Experiments are run three times each and averaged. Compared to other methods, TBNs achieve a competitive performance with MST, the current state-of-the-art method for sub-bit compression. TBNs achieve similar performance at roughly the same compression rates of MST for the CIFAR-10 models. Finally, our approach allows for more extreme compression rates. Results in Table 4.1 show only small drops in performance at up to 16x compression compared to a full precision model.

4.5.2 MLP-Based Architectures

In addition we consider MLP models which contain a high proportion of fully-connected and 1×1 convolutional layers. PointNet [155] is a well-established model for unified tasks in classification, part segmentation, and semantic segmentation. 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.

Table 4.1: CNN Results on CIFAR-10: * 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).

CIFAR-10				
Model	Method	Bit-Width (Savings)	#Params (M-Bit)	Test Acc. (%)
ResNet 18	Full-Precision	32	351.54	93.1
	IR-Net	1	10.990	92.9
	SNN	0.440 (2.3x)	4.882	92.1
	Sparks*	0.440 (2.3x)	4.880	90.8
	MST*	0.075 (13.3x)	0.814	91.6
	TBN_4	0.256 (3.9x)	2.850	93.1
	TBN_8	0.131 (7.7x)	1.463	92.4
	TBN_{16}	0.069 (14.5x)	0.768	91.2
ResNet 50	Full-Precision	32	750.26	95.4
	IR-Net	1	23.450	93.2
	SNN	0.35 (2.8x)	8.321	94.0
	TBN_4	0.259 (3.9x)	6.100	94.9
	TBN_8	0.136 (7.4x)	3.210	94.3
	TBN_{16}	0.075 (13.3x)	1.760	93.5

We derive MLP experiments from BiBench [156], 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, while TBNs have full-precision activations. In Table 4.2 we denote the best algorithms from the BiBench. We also train a BWNN for each task, since it provides us with abetter comparison (models with full-precision activations).

Results. In Table 4.2 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 and 4x compression. For both part and semantic segmentation, TBNs exhibit 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, which was the most successful BNN on the task in BiBench.

Table 4.2: 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.

Task: Classification, Dataset: ModelNet40			
Algorithm	Bit-Width (Params)	#Params (M-Bit)	Test Acc./ (%)
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

Task: Part Segmentation, Dataset: ShapeNet				
Algorithm	Bit-Width (Params)	#Params (M-Bit)	Instance Avg. IoU	Class Avg. IoU
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

Task: Semantic Segmentation, Dataset: S3DIS				
Algorithm	Bit-Width (Params)	#Params (M-Bit)	Test Acc.	Class Avg. IoU
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

4.5.3 Vision Transformers

In our next set of experiments we assess TBNs on Transformer models. It was noted in BiBench that Transformers perform poorly in BNNs, with none of the binarization algorithms from the benchmark research achieving 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. In particular, the multiplications between the query, key, and value cause amplified information loss when model activations are binarized. The authors show that the local structure of CNNs has the least amount of error compared to Transformers and MLPs.

Table 4.3: Vision Transformers trained on CIFAR-10: We compare the performance of a ViT (patch size 4) and the Swin-T model with TBN_4 and TBN_8 variations. Experiments are averaged over 3 runs (S.D. is less than 0.5).

Model	Method	Bit-Width (Params)	#Params (M-Bit)	Test Acc. (%)
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

In contrast to BiBench we address models with binary weights and full-precision activations, which do not suffer from the same information loss as models with binary activations. Due to the difficulty of training large language models from scratch, we instead turn back to Computer Vision, where Transformers have been established as state-of-the-art architectures. We train Swin-T [157] and ViT [158] models from scratch on the CIFAR-10 dataset and compare the models to a full-precision model of the same structure. ViT's use image patching along with the traditional attention mechanism. Swin-T uses hierarchical partitioning of the image into patches and then merges them as the network gets deeper.

Results. In Table 4.3 we summarize the results of ViTs trained on the CIFAR-10 dataset. TBNs achieve performance within 2.5% of Full-Precision models across all experiments. Specifically, the TBN ViT is able to 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.

4.5.4 Effect of Layer Size

In our final set of experiments we look at MLPMixers [159] and ConvMixers [160] 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 4.5 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 elements. Many layers have

less than 65k parameters and dont't fulfill our minimum layer size for compression. This results in limited reductions in parameter count along with high performance degradation. MLPMixer, on the other hand, had layers with 131k elements, and resulted in a more modest performance degradation at higher parameter compression.

Assessing other architectures, we found that the convolutional, PointNet, and Transformer models contained layers with many features (often in the millions).

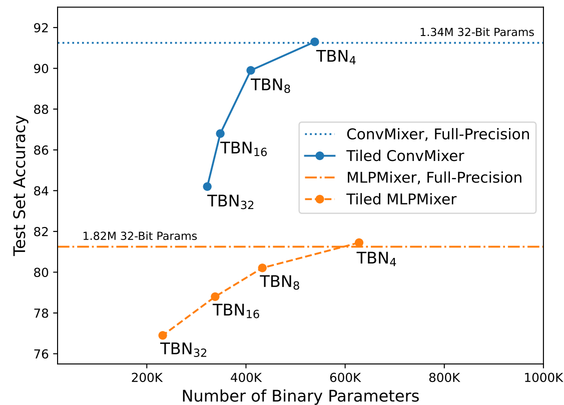

Figure 4.5: TBNs are vulnerable in models with low width layers: We plot the performance of the ConvMixer and MLPMixer at various compression rates/number of parameters compared to its full-precision baseline (horizontal line). The ConvMixer accuracy degrades quickly as a result of smaller layers: its 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.

.

4.6 Implementation

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

4.6.1 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

Algorithm 3 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), $\alpha_1, \alpha_2, ...\alpha_p$

```
t_i ← 0,    α_i ← α_1
for i ← 1 to m do y[i] ← 0 α_i = α_i + 1
    for j ← 1 to n do
        y[i] ← y[i] + t[t_i] · x[j] · α_i
        if t_i = q then
            t_i ← 0                          ▷ Move to beginning of tile vector
            α_i ← α_{i+1}                    ▷ Get next tiles α
        else
            t_i ← t_i + 1                    ▷ Increment tile index
        end if
    end for
    y[i] ← max(0, y[i])                      ▷ Fused ReLU
end for
```

Output: Output vector $\mathbf{y}$ of size m

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 [161], 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 3. We develop a fully binarized kernel by packing weights into unsigned 8-bit integers and use bit-masking to extract the correct values.

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 is able to 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 4.4. 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 TBN 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.

Table 4.4: 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 being allocated for the output activation.

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

4.6.2 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 GPU. Standard PyTorch operations do not allow us to reuse a single tile without allocating new memory for the full layers parameters (multiple tiles); leveraging Triton provides more control over lower level memory allocation.

We implement a fully connected module using full precision (32-bit) parameters. We argue that comparing a tiled 32-bit model with a standard 32-bit model simulates the comparison of a tiled 1-bit model with a standard 1-bit model. Moreover, tiled networks work with full-precision parameters, and may be relevant to other levels of weight precision (see Appendix).

We implement the fully connected layer using the matrix multiplication functionality provided by Triton. It uses block-level matrix multiplication and achieves similar performance to the optimized cuBLAS library. Our implementation coverts an $m \times n$ matrix to $m \times r$, where r equals n/p (we compress the second dimension). For pointer arithmetic we point to the n/p block of weights. In other words we implement fully connected tiling with *matrix-to-matrix* pointers rather than *matrix-to-vector* pointers.

Results We measure the GPU memory usage for inference on a single image in Figure 4.6. The two figures profile the memory usage through each layer of the model. For the ViT ImageNet model, peak memory usage is 222.5MB for the standard kernel and 78.5MB for the TBN_4 kernel. The

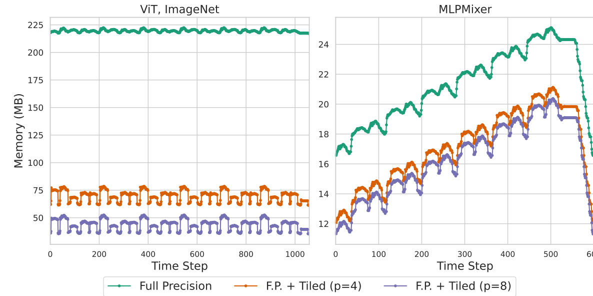

Figure 4.6: GPU memory allocated during model inference: We profile the memory of the ImageNet Vision Transformer (Left) and PointNet (right) during inference on a single input using our customized GPU kernel. The x-axis represents memory recorded within intermediate model layers during execution of a PyTorch model. The kernel achieves 2.8x memory reduction on the ViT and 1.2x reduction on PointNet. The green line represents a standard kernel with full-precision weights; other lines represent a tiled kernel with full-precision weights.

MLPMixer model offers more modest reductions as a result of holding its intermediate activations. The standard kernel has a peak memory of 25.1MB compared to 21.1MB for the TBN_4.

The results in Figure 4.6 reflect the memory savings by only loading a *single* tile per model layer. Specifically, during inference the PyTorch library loads weights for *all* layers into memory. It allocates new memory for the input and output activations to each layer, and deallocates these activations when they are no longer needed. For example, the ViT model has six attention layers with roughly 8.4 million parameters each (the Q, K, V parameters 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, which make up 34MB per attention layer.

4.7 Ablation Study

We study the effects of three hyperparameters of TBNs: 1) λ (minimum layer size for tiling), 2) $\mathbf{W}$ for tiling and $\mathbf{A}$ 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 second hyperparameter, we test two settings: the first uses $\mathbf{W}$ for both learning the binary tile and calculating α; the second uses parameter $\mathbf{A}$ for calculating α. We denote this setting as $\mathbf{W}$

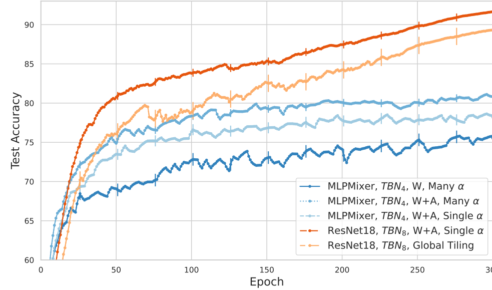

Figure 4.7: Hyperparameter Configurations: Test set accuracy across training for the MLPMixer and ResNet18 models with various hyperparameter configurations. For ResNet18, we show how tiling every convolutional layer, rather than layers of a certain size, leads to performance loss (red/orange). In blue we show the effects of using a separate parameter $\mathbf{A}$ to calculate α compared to calculating α using just $\mathbf{W}$. We additionally show the benefit of using multiple αs (one per tile) rather than a single α.

or $\mathbf{W} + \mathbf{A}$. For the third parameter, we test a single α per layer and compare it to computing α for each tile.

Figure 4.7 shows the effects of the different hyparameters on both the ResNet50 and MLPMixer models on the test accuracy throughout training. We show that a global tiling factor (no λ) on ResNet50 causes a significant decrease in performance. This was observed across most models, with some CIFAR-10 models failing to reach even 50% test accuracy. Next, we find the model converges best when given a separate parameter $\mathbf{A}$ to calculate α. Moreover, when calculating an α for each tile, we observe a slight increase in performance. We note that the latter two hyperparameters exhibit varying results across architectures, thus we do not adhere to a single configuration for these approaches.

4.8 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. The method performs well across multiple computer vision datasets and tasks.

Future Work A natural direction for future work is to apply TBNs to models with binary weights and binary activations. BNNs with binary activations maximize memory savings and

improve speed. We would also like to test the approach on larger scale models (e.g. LLMs), where additional algorithmic modifications (such as full-precision parameter tiling) may enhance performance. Finally, specialized kernels could be implemented to maximize the efficiency of TBN with regards to parallelization, including convolutional kernels and kernels better optimized for re-usability of tiles.

Chapter 5

Applying Sparse and Quantized Neural Networks in Real-World Applications

5.1 Summary

This research focuses on two primary objectives: benchmarking sparse and binary neural networks against conventional quantization techniques and assessing the feasibility of deploying these sparse binary models on microcontrollers, specifically the Arduino Nano 33 BLE Sense Rev2. The first goal involves a comprehensive evaluation, comparing sparse binary neural networks with post-training quantization and quantization-aware training methods across metrics such as test accuracy, FLOPs, memory, and model size. Concurrently, the study outlines an implementation strategy for deploying sparse binary models on microcontrollers. Our ultimate aim is to demonstrate the practical viability of sparse binary neural networks in real-world applications by successfully deploying them on resource-constrained microcontrollers, ensuring a holistic understanding of their utility and limitations.

5.2 Introduction

In this chapter we delve into two objectives surrounding the application of neural network quantization and compression. Our first goal revolves around benchmarking sparse and binary neural networks against conventional methodologies prevalent in practical applications. Specifically, we scrutinize sparse binary neural networks in comparison with widely adopted post-training quantization and quantization-aware training techniques. The assessment is multi-faceted and focuses on parameters such as test accuracy, computational complexity (measured through FLOPs), maximum memory requirements, and model size. Such metrics are essential for comprehending the impact of quantization and compression on performance.

The second facet of this research endeavors to explore the feasibility of deploying sparse binary models on resource-constrained microcontrollers. Specifically, we deploy our models on the Arduino Nano 33 BLE Sense Rev2. This includes the development of a feedforward network for the MNIST dataset, employing techniques such as bit packing and unpacking, model compilation using the PyTorch C++ runtime, and implementation of standard quantized models. Our ultimate goal is to demonstrate the viability of sparse binary neural networks in real-world applications, gauged through successful deployment on microcontrollers, ensuring a comprehensive evaluation of their practical utility.

5.3 Background and Related Work

In this section we describe the related work in both industry and research related to implementations of compressed neural networks.

DNNs on Resource-Constrained Machines

Compressed neural networks have shown potential in both research [102, 138, 162] and industry [163]. The last few years, work out of the Han lab at MIT has enabled neural network deployment on MCUs, which we will cover first in this section. In addition, companies such as EdgeImpulse have started specifically targeting "edge AI", which we define as neural networks being deployed on resource constrained machines such as microcontrollers and mobile devices. EdgeImpulse's library, TensorFlow Lite Micro [164, 165], is a lightweight neural network library which is compatible with small devices such as the Arduino [166] MCU. In the rest of this section, we will cover the advances made with the MCUNet at the Han Lab, as well as other recent advances in deploying neural networks on microcontrollers.

"MCUNet: Tiny Deep Learning on IoT Devices" [102] is a research paper publisehd at NeurIPS in 2020. The paper focuses on the development of an innovative deep learning framework, MCUNet, specifically designed for Internet of Things (IoT) devices with constrained resources. The authors address the challenge of deploying complex deep learning models on resource-limited IoT devices, which typically have limited memory, processing power, and energy consumption capacities.

Specifically, MCUNet is a software-system co-design (similar to Apple products) which enables ImageNet scale inference on tiny devices. Lin et al. propose optimizing the deep learning model design (TinyNAS) and the inference library (TinyEngine) to reduce the memory usage. Specifically, they use neural architecture search and provide the algorithms with constraints common to microcontrollers. Impressively, the authors achieve over 70% accuracy on the ImageNet test set running on a microcontroller.

In follow-up work, the authors propose memory-efficient inference by "patching" to control peak memory usage of neural networks during inference [138]. Specifically, the authors find that common convolutional neural networks have an imbalanced memory usage throughout their layers. They balance out this memory usage by applying "patch-by-patch" memory usage. In their most recent publication, the authors propose training neural networks directly on microcontrollers via a novel training method which uses sparse gradients and quantization awareness techniques [167]. Specifically, the authors propose "Quantization-Aware Scaling", a method which calibrates gradient scales and stabilize 8-bit training. They additionally use a sparse update to reduce the gradient computation.

Other Research and Frameworks Novac et al. [168] focus on quantization and deployment of DNNs on 32-bit microcontrollers. They propose an alternative inference engine to TensorFlow Lite Micro called MicroAI. The framework offers execution on 32-bit floating points as well as 8-bit and 16-bit integers. They evaluate their framework on several microcontrollers, such as the ARM Cortex-M4F.

One last framework we consider in our work is PyTorch, specifically PyTorch for mobile [169]. The engine offers its QNNPack, a quantization engine, XNNPACK, a floating point kernel library engine for Arm CPUs, as well as efficient mobile interpreters for Android and iOS. However, PyTorch's mobile library is too large to deploy on microcontrollers (which have significantly less memory than phones. To counter this, we investigate PyTorch's Tracing Based selective build tool [170], which prunes the necessary files from PyTorch C++ library to compile into a lighter weight compilation for deploying. We hypothesize that successfully deploying a PyTorch model on

a microcontroller could offer outstanding benefits to the broader community, since PyTorch is the main deep learning library in use today.

Other research has examined specific aspects of neural architecture search for deploying neural networks on MCUs. MicroNets [171] propose a differentiable neural network architecture search where operation count is treated as a proxy for latency. Finally, Xie et al. [172] propose a weight-sharing mechanism to improve optimization.

Post-Training Quantization

Post-training quantization techniques have emerged as the standard for compressing neural networks in real-world applications. For example, both PyTorch Quantization [20] and TensorFlow [173] provide post-training quantization as standard tools. Below we review work related to post-training quantization. The standard technique involves first training a model using either 32-bit or 16-bit floating point integers, and then converting the integers to their lower precision 8-bit counterpart after training has completed. This typically reduces the performance, but it is a simple and effective approach. Next we review several related works for post-training quantization.

Gupta and Jacob [17] provided a comprehensive survey on *quantization and training of neural networks*, covering post-training quantization and discussing the broader landscape of quantization techniques. Courbariaux et al. [16] introduced the concept of quantization and training of neural networks for efficient integer-arithmetic-only inference. Their approach demonstrated that by representing both weights and activations as low bit-width integers, significant reductions in memory footprint and computation cost could be achieved. Rastegari et al. [34] extended the concept of quantization to edge devices in their work on *quantization and training of neural networks for efficient inference on edge devices*. By optimizing models for edge deployments, they showcased the potential to unlock powerful AI capabilities on devices with limited resources. In the context of binary neural networks, Li et al. [174] proposed techniques to create accurate binary convolutional neural networks. Their exploration of quantization to binary values (1-bit) demonstrated how training and quantization of networks to binary weights and activations could yield compact models with promising accuracy. Notably, the technique required five layers of bit-size matrices for each

standard layer to achieve performance similar to a dense model. In other words they needed 5-bits for each parameter in the model to achieve high accuracy on ImageNet.

Quantization-Aware Training

Quantization Aware Training (QAT) has gained significant attention as a method to improve the performance of quantized neural networks. Both PyTorch and TensorFlow offer QAT in their quantization libraries, and the method offers a competitive alternative to post-training quantization. QAT trains a neural network in a way that prepares it to work efficiently with limited numerical precision. For example, the model may be trained with 32-bit floating point numbers, but QAT prepares the model to use lower precision (typically 8-bit) numbers.

In more technical detail, quantizing a model from 32-bit weights to 8-bit weights introduces an error, e.g. a number may go from 8.123 to 8. QAT introduces "fake" quantization layers after certain operations, simulating what would happen if the model was using lower-precision numbers. These layers calculate quantization effects on gradients while keeping the actual weights and activations in 32-bit. QAT additionally introduces a *quantization loss*, which is the difference between the actual 32-bit output and the quantized output. This loss is added to the total loss during training. This loss is used for backpropagation.

Related research has introduced a differentiable quantization framework for neural networks [175]. In particular, Gong et al. enabled end-to-end training of quantized models by introducing a relaxation technique that bridges the gap between discrete quantization and continuous optimization. Jacob et al. [176] proposed a quantization technique that integrates directly into the training process. Their method employs a quantization function that mimics the quantization behavior during inference. Polino et al. [177] introduced the concept of loss-aware quantization during training. Mishra et al. [178] proposed the "Apprentice" framework for quantization-aware training. Their approach involves training a student network to mimic the behavior of a larger, well-trained teacher network.

5.4 Research Methodology

In this section we describe our proposed approach implementing a sparse binary weighted neural network in practice. This includes bit packing and unpacking into higher precision data types as well as optimizing the model in C. Neither of these things have been achieved with this particular algorithm, thus our motivation is studying its practical use.

5.4.1 Bit Packing and Unpacking

To correctly binarize a neural network, we need to perform several special operations as a result of higher level programming language. Specifically, Python and C have data types as low as a single *byte*, rather than single bit, so we will need to perform bit-packing to save and deploy our model post-training, and bit-unpacking for model inference on an edge device. Bit packing involves taking a set of 8 bits and "packing" them into a byte data type. For example, a single byte consists of 8-bits, which can be represented as an unsigned integer in most common computer languages. The bit string "00000000" equals integer 0 and "11111111" equals integer 256. We can represent any value between 0 and 256 with 8-bit strings. The integer value (0-256) is saved as an "uint8" in most common programming languages. The "u" in uint8 means unsigned. We can also represent positive and negative values with a signed integer data type between -128 and 127.

Below, we demonstrate how to convert the binary maxtrix [1, 1, 1, 1, 1, 1, 1, 1] into the packed 8-bit integer value 255. To do this, we create a uint8 tensor with the packed value, and then unpack it back to the original binary data. The resulting output will show you the original binary data, packed value, packed tensor, unpacked binary, and unpacked data.

```
import torch

# Original binary data
binary_data = torch.tensor([1, 1, 1, 1, 1, 1, 1, 1], dtype=torch.int8)

# Packing
packed_value = int(''.join(map(str, binary_data.tolist())), base=2)
# Convert binary (with base 2) to integer

# Create a uint8 tensor with the packed value
packed_tensor = torch.tensor([packed_value], dtype=torch.uint8)
```

```
# Unpacking
unpacked_binary = bin(packed_tensor.item())[2:].zfill(8)
# Convert packed value to binary, remove '0b', zero-fill to 8 bits

unpacked_data = torch.tensor(list(map(int, unpacked_binary)),dtype=torch.int8)
# Convert binary back to tensor

print("Original Binary Data:", binary_data)
[1, 1, 1, 1, 1, 1, 1, 1]
print("Packed Value:", packed_value)
255
print("Packed Tensor:", packed_tensor)
tensor([255], dtype=torch.uint8)
print("Unpacked Binary:", unpacked_binary)
11111111
print("Unpacked Data:", unpacked_data)
tensor([1, 1, 1, 1, 1, 1, 1, 1], dtype=torch.int8)
```

Listing 5.1: Packing and Unpacking a bit string into uint8 data type

To apply this to a neural network, we will pack and unpack weights the weights of a linear layer specifically. Since the Biprop algorithm does not use bias in its linear layer computation, we will focus on the weight matrix of the linear layer. A linear layer performs the computation

$$y = x \cdot W \tag{5.1}$$

where x is the input to the layer, such as data or the values output from the previous layer), and W is the weights of the layer. W is a 2d matrix of size MxN. For example, an MNIST sample is 784 pixels, which we input into a linear layer of size 512. W will therefore be size 784x512.

We will pack the linear layer column-wise (using the N dimension) or row-wise (using the M dimension). In either situation we'll need to consider columns or rows that are not divisible by 8, leading to potential trailing values in the uint8 data type to fill the extra allocated space. For example, a matrix with 8 columns can be packed down to 1 uint8 column, while a matrix with 13 columns must be packed into 2 uint8 columns with 3 trailing values. We implement this in code below:

```
import torch
import math
import numpy as np
```

```python
# Original binary data, a 3x13 matrix with binary values
binary_data = torch.tensor([[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                             [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                             [1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1]], dtype=
                                torch.int8)

num_cols = 13
pack_columns = binary_data.size(1) / 8 if binary_data.size(1) / 8 == 0 else
    math.floor(binary_data.size(1) / 8) + 1 #2

# Packing
packed_values = []
for row in binary_data.numpy().tolist():
    row.extend([0] * (pack_columns*8-num_cols))
    result_list = [int(''.join(map(str, row[i:i + 8])),2) for i in range(0,
        len(row), 8)]
    print(result_list)
    packed_values.append(result_list)

packed_tensor = torch.tensor(packed_values, dtype=torch.uint8)


#Unpacking
matrix=packed_tensor.numpy()
num_bits = 8  # Number of bits to extract
shifted_matrix = matrix[:, :, np.newaxis] >> np.arange(num_bits - 1, -1, -1)
extracted_bits_matrix = shifted_matrix & 1
extracted_bits_matrix=extracted_bits_matrix.reshape(extracted_bits_matrix.
    shape[0], -1)[:,:num_cols]

unpacked_binary_data=torch.tensor(extracted_bits_matrix)


print("Packed Tensor:\n", packed_tensor)
tensor([[170, 168],
        [ 85,  80],
        [227, 136]], dtype=torch.uint8)
print("Unpacked Tensor:\n", unpacked_binary_data)
tensor([[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
        [1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1]])
```

Listing 5.2: Packing and Unpacking a neural network linear layer into uint8 data type (column wise)

5.4.2 Optimizing the model for real-world applications

We additionally consider activation fusing. Activation fusing involves combining the computation of a layers activations with its weight matrix multiply computations, leading to a more efficient model.

5.5 Research Scope

Our research scope consists of two primary goals. The first goal involves benchmarking sparse and binary neural networks compared to typical approaches which are common in the wild. The second goal is to assess the feasibility of deploying sparse binary models on microcontrollers. Together, these two goals will help us understand the overall benefit and feasibility of sparse binary neural networks in real-world applications.

5.5.1 Benchmarking Against Real-World Quantization Techniques

We benchmark sparse binary neural networks against both post-training quantization as well as models trained with quantization awareness. These are the two most commonly used approaches in industry and practice. Our goal is to assess the performance across several metrics to understand the feasibility of sparse binary models.

Metrics To begin, we assess sparse binary neural networks with common approaches by comparing the test accuracy of each model. This metric is important to understand the degradation of performance quantization and model compression will incur. For our other metrics, we will use FLOPs and storage size as described in Chapter 1. FLOPs provide a reasonable metric on computational complexity, and is used to measure computational performance including processing power and efficiency. Model size gives us an idea of how much space the model will take up on a given device.

Our final metric, maximum memory, measures the most amount of RAM that will be required for model inference. Neural networks work by passing an input layer by layer through the model until a final output is reached. As a result, in general only two parts of a given neural network are needed in memory at any given time: the input to a layer and the layer itself. In addition to these two things, we also need to allocate memory for the output of the layer. The input to a layer can be the data the model is using for inference, or the activation from the previous layer in the model. The layer itself is a multidimensional weight matrix that is learned during training. The memory allocated to the output should be the expected size of the output once the layer performs

its operations on the input. Some models such as ResNet architectures hold additional tensors in memory in order to propagate the activations deeper into the network; doing so helps with vanishing gradient problem where gradients diminish as they are backpropagated through the model. In this work, we do not consider these types of architectures.

Experiments: Datasets and Architectures We run experiments using MNIST, a dataset used heavily throughout literature. We will use a two layer feed forward architecture for baseline experiments as well as sparse binary neural network experiments. We will report test accuracy, FLOPs, and storage size of three models each with different compression techniques (quantization, quantization aware training, and sparse binary neural networks) so as to provide reasonable comparison of sparse binary models against real-world commonly used algorithms.

5.5.2 Deploying Sparse Binary Neural Networks to MCUs

Our main goal for this chapter of the research is to deploy a sparse binary model to an MCU. We first detail hardware, then summarize our model and dataset, and finally detail our implementation.

Hardware We use an Arduino Nano 33 BLE Sense Rev2 for this application. The Arduino model has 256KB of memory (SRAM) and 1MB of flash storage. It connects to a computer with a Micro USB and has sensors to detect color, proximity, motion, temperature, humidity, audio and more. Arduino is compatible with C++ and C programming languages.

Model and Dataset We exclusively experiment with the simplest dataset and task for this project: a simple feedforward network (otherwise known as a multilayer perceptron) using MNIST. MNIST is a dataset of images of handwritten digits (numbers 0 to 9), and the job of the model is to classify the handwritten digit into the actual number. The MNIST dataset is commonly used for simple experiments and to quickly iterate on ideas. The feedforward network is used as a non-linear architecture that we can perform gradient descent on. The feedforward network will consist of 784 inputs (MNIST is 28x28 pixels, which we flatten), a hidden layer with size 256 neurons, a ReLU nonlinear activation, and a classification output (there are 10 classes).

Implementation Our implementation consists of a model which performs bit packing and unpacking as well as compiling the model using C. We additionally deploy a quantized model on a microcontroller.

For the sparse binary neural network, we develop a feed forward network which packs and unpacks bits into the larger precision 8-bit integer data type, as described in the previous section. *To the best of our knowledge parameter packing has not been implemented for this algorithm.* Previous works implemented a theoretical models for the sparse binary framework [26, 27], which use full-precision data types that *simulate* binary weights.

After the packing/unpacking implementation is completed, we convert the model to C for compilation and deployment. Specifically, we create a feed forward model which does the following (in order):

1. We first train a sparse binary model on MNIST.

2. After training completion, we *pack* the binary weight values into the higher precision 8-bit integer data type in PyTorch. We save this model.

3. We next test model inference by performing weight *unpacking*. To do this, we load the model and a test sample in python and run the sample through the model. We perform unpacking of the model into higher precision binary weights, layer by layer.

4. We convert the model to C and build it using a mobile selective build.

5. We perform any optimizations on the model, such as deleting layer pointers as we traverse through the model.

5.6 Experimental Results

In this section we detail our experimental results, starting with summarizing how sparse binary neural networks performed compared to other approaches, and then detailing the process of deploying the algorithms on an Arduino microcontroller.

5.6.1 SBNN's and Quantization Performance Evaluation

Table 5.1 offers a comparative analysis of several quantization techniques applied to neural networks trained on the MNIST dataset. Each model features a single hidden layer with 256 neurons and a ReLU activation function. The models evaluated include a Dense Neural Network, Post-Training Quantization, Quantization-Aware Training, and SBNN.

The provided table offers a comprehensive comparison of various quantization techniques applied to neural networks trained on the MNIST dataset, each featuring a single hidden layer with 256 neurons and a ReLU activation function. The quantization methods assessed include a Dense Neural Network (Dense NN), Post-Training Quantization (PTQ), Quantization-Aware Training (QAT), and SBNN.

The dense neural network serves as the baseline, achieving a high test accuracy of 98.5% but demanding substantial computational resources with 101,632 FLOPs and a large memory requirement of 806.98 KB. Post-Training Quantization (PTQ) significantly reduces computational complexity to 101,632 FLOPs and storage to 1,626,112 bits, showcasing a notable reduction in both memory (204.86 KB) and inference time (6.178 seconds). Quantization-Aware Training (QAT) achieves the highest accuracy among the evaluated methods at 98.33%, maintaining the same computational and memory efficiency as PTQ. The SBNN excels in memory optimization, requiring only 54.33 KB of memory and reducing storage significantly to 406,528 bits. However, this memory efficiency comes at a slight cost, with an inference time of 7.215 seconds.

In summary, each quantization technique presents distinct advantages. PTQ and QAT offer an excellent balance between accuracy, computational complexity, and memory usage, making them suitable for various applications. SBNN stands out for its remarkable memory optimization, while still providing competitive accuracy, making it ideal for scenarios where memory efficiency is paramount, and a minor trade-off in execution speed is acceptable. The choice among these techniques should align with the specific requirements and constraints of the application. We note that further code optimization may be achieved through other techniques unknown to us

Table 5.1: Performance Comparison of Quantization Techniques. Each model is trained on MNIST and has a single hidden layer with 256 neurons and a ReLU activation. The storage size is measured in bits and the inference time is measured by the time it takes each model to execute 1,000 times (averaged over 3 runs). The maximum memory is computed by computing the memory required to execute the first layer of the MLP, which is substantially larger than the output layer. Overall, each quantization method provides different benefits.

Quant. Type	Test Acc.	FLOPs	Max. Memory (KB)	Storage (Bits)	Inf. Time (S)
Dense NN	98.5%	101,632	806.98	6,504,448	-
PTQ	97.05%	101,632	204.86	1,626,112	**6.178**
QAT	**98.33%**	101,632	204.86	1,626,112	6.295
SBNN	97.57%	101,632	**54.33**	**406,528**	7.215

for the sparse binary neural network model. This work is meant to serve as a starting point for implementation; we leave further improvements (such as speed enhancements) to future work.

5.6.2 Compressed Neural Networks Arduino Deployment

Next we summarize the process of deploying SBNN's and 8-bit quanitzation models on an Arduino microcontroller. For this section, we desired to achieve two things: 1) deploy our model to a microcontroller for demonstration, and 2) learn and familiarize ourselves with the design of lower level neural network programs.

We explored several libraries for achieving this such as TensorFlow Lite Micro [173] and Pytorch C++ Mobile Selective Build [20]. TensorFlow Lite Micro has a robust functionality which could have suited our needs, however, we were concerned that the codebase was established enough that we wouldn't have achieved point 2 (learning and familiarization). Moreover, it is written in C++, which is not compatible with all microcontrollers and embedded systems. Finally, we would have still needed to implementation sparse binary neural networks on top of this codebase. PyTorch C++ Mobile Selective Build had potential, however, its stripped down C++ code on even the simplest models was still substantially larger than the storage space available on a microcontroller (over four megabytes).

Tensor1 Instead of using a pre-existing library, we elect to build our own. This allowed us flexibility and the ability to learn each step of the engineering. We named our repository

Tensor1, which stands for tensors with 1 bit parameters. Our code repository is available at www.github.com/mattgorb/tensor1.

Detailed Implementation In this section we describe step-by-step how we implemented sparse-binary neural networks. Our Python step-by-step scripts are available at www.github.com/mattgorb/tensor1/tree/main/python_quantization/sbnn_mlp, and our C library is available at www.github.com/mattgorb/tensor1/tree/main/models/sbnn_mlp and www.github.com/mattgorb/tensor1/tree/main/src.

Steps to implement sparse binary neural networks on a microcontroller are as follows:

1. We train a Sparse Binary Neural Network using the full precision weights to simulate quantization during training in "1_sbnn_train.py".

2. We reconfigure the model to drop the scoring parameter, binarize the weights, and compute a single alpha. We then save the model with these new configurations in Python. Code is located in "2_sbnn_reconfigure.py".

3. Next, we convert the simulated binarized weights as well as the sparse mask into bit packed data types. Current approaches binarize a parameter to plus/minus 1 for each weight, but keep the value in a 32-bit tensor. For example, we pack 8 binary weight values into a single byte tensor. Code is located in "3_sbnn_packed.py".

4. For the final part of our Python code we write our neural networks weights to C files. We save each layer's weights in C data types. In our previous packing file, we save bits into unsigned bytes. Here, we take those bytes and save them as uint8_t data types in C. This is achieved writing the data types layer by layer into a C file via Python. This file produces two files in C: sbnn_weights.c and sbnn_256_weights.h. Code is located in "4_write_params_fp.py".

5. We can build out our model definition in C. To do this, we first create a main function in C, then define the two layers of our model in a separate file. Code is located at www.github.com/mattgorb/tensor1/tree/main/models/sbnn_mlp.

```
tuna:~/sbnn/python_quantization/sbnn_mlp$ python 3_sbnn_packed.py

Test set: Average loss: 0.3088, Accuracy: 9753/10000 (98%)

model prediction:
[[-1419.2297    -4785.825       -97.8613      604.3575    -3694.5579
   -1612.4872    -6695.3203     3626.692    -1694.5974        1.3012085]]
```

(a)

```
tuna:~/sbnn$ make build-model MODEL=sbnn_mlp
rm -f sbnn_mlp
rm -rf build
rm -f build/kernels.o build/tensor1.o build/util.o build/main.o build/sbnn.o build/sbnn_256_weights.o
mkdir -p build
gcc -Wall -Iinclude -c src/kernels.c -o build/kernels.o
mkdir -p build
gcc -Wall -Iinclude -c src/tensor1.c -o build/tensor1.o
mkdir -p build
gcc -Wall -Iinclude -c src/util.c -o build/util.o
mkdir -p build
gcc -Wall -Iinclude -c models/sbnn_mlp/main.c -o build/main.o
mkdir -p build
gcc -Wall -Iinclude -c models/sbnn_mlp/sbnn.c -o build/sbnn.o
mkdir -p build
gcc -Wall -Iinclude -c models/sbnn_mlp/sbnn_256_weights.c -o build/sbnn_256_weights.o
gcc -Wall -Iinclude build/kernels.o build/tensor1.o build/util.o build/main.o build/sbnn.o build/sbnn_256_weights.o -o sbnn_mlp
tuna:~/sbnn$ ls
1_two_working.zip  build    Makefile  ptq_mlp              qat_mlp    run_exec.sh  scripts  tensor1   weights
arduino_libs       include  models    python_quantization  README.md  sbnn_mlp     src      TODO.txt
tuna:~/sbnn$ sbnn_mlp
Output: -1419.217773, -4785.862305, -97.857529, 604.357971, -3694.641357, -1612.514038, -6695.446289, 3626.694580, -1694.594116, 1.267365,
```

(b)

Figure 5.1: (a) Output on Image 1 when running inference on a trained sparse binary neural network. We use this network to convert to C and run on a microcontroller. (b) Model output when running inference on Image 1 in our C program. Both models produce the same output.

6. We finally build out specific kernels for each model. Since our model only has linear layers, this was the only kernel built. We included an option to **fuse** ReLU nonlinearity into the layer: www.github.com/mattgorb/tensor1/blob/main/src/kernels.c#L193

For both standard quantization methods, we perform the same steps: train the models in Python, convert them to their quantized versions, and then convert the quantized parameters to C. From here, we are able to write the model defintions and specific kernels in C for each model.

Validation We validate each of our three models by confirming that the model inference on a single input image is the same in both Python and C. We take screenshots, which are shown in Figure 5.1

Deployment to Arduino Microcontroller To make the C model compatible with Arduino, we include all of our .c and .h files in a single folder. This way the Arduino development environment is able to recognize the files as a single library. We import this library into Arduino, and all of its methods are then usable. The Arduino IDE is pictured in Figure 5.4

In Arduino, we then simply include our code from the main function into the Arduino setup method. We connect our microcontroller to a labtop where our Arduino IDE is, and press "Deploy"

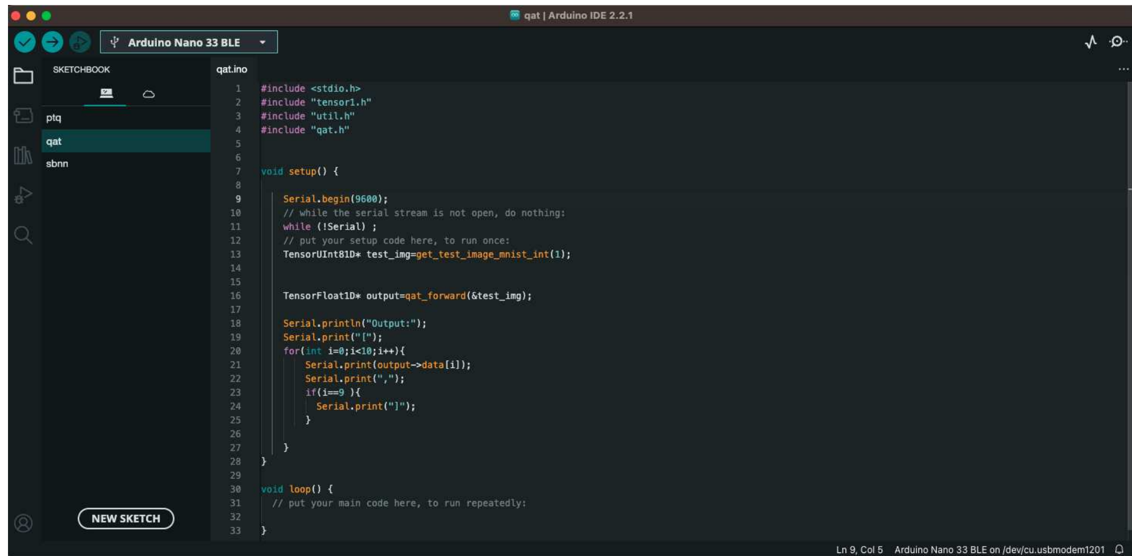

Figure 5.2: The Arduino Integrated Development Environment.

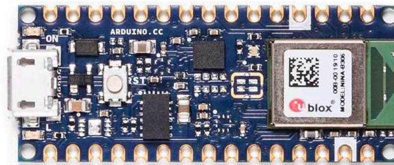

Figure 5.3: The Arduino Nano 33 BLE microcontroller used for this dissertation.

in the Arduino IDE. This compiles and deploys the code to the Arduino microcontroller. We can then print our model's output from the microcontroller.

In conclusion, we were able to deploy the sparse binary neural network successfully to the Arduino microcontroller. Initially, we attempted to write our model in C++, however we had an easier time using C. C++ has in-built libraries that aren't all available on the microcontroller. C, on the other hand, is a light enough programming language that we were able to build and use functionality available to the tiny MCU.

```
sbnn.ino
1   #include <stdio.h>
2   #include "tensor1.h"
3   #include "util.h"
4   #include "sbnn_256_weights.h"
5   #include "sbnn.h"
6
7   void setup() {
8
9       Serial.begin(9600);
10      // while the serial stream is not open, do nothing:
11      while (!Serial) ;
12
13      TensorFloat1D* test_img=tensor_test_image_mnist();
14      TensorFloat1D* output=sbnn_forward(&test_img);
15
16      Serial.println("Output:");
17      Serial.print("[");
18      for(int i=0;i<10;i++){
19          Serial.print(output->data[i]);
20          Serial.print(",");
21          if(i==9 ){
22            Serial.print("]");
23          }
24
25      }
26
27  }
28
29
30
Output   Serial Monitor ×
Message (Enter to send message to 'Arduino Nano 33 BLE' on '/dev/cu.usbmodem11201')
Output:
[-1419.22,-4785.86,-97.86,604.36,-3694.64,-1612.51,-6695.45,3626.69,-1694.59,1.27,]
```

Figure 5.4: The Arduino IDE with the sparse binary neural network sketch (top) and the output of the code on the microcontroller (bottom). The microcontroller uses a custom Arduino print method (Serial) to output and log things on a computer.

Chapter 6

Discussion

6.1 Future Work

The research in this dissertation contains novel quantization objectives across new architectures (Transformers) and algorithms (Iterative Weight Recycling, Tiled Bit Networks), and additionally includes a low level implementation of the compression methods achieved in this work. The research opens up several new paths which we hope to explore after the completion of this work.

To begin, the Transformer architecture in Chapter 1 has potential for additional research along two paths. First, we'd like to explore an implementation of the model on a microcontroller. This would be the first, to our knowledge, implementation of a Transformer on such a small device. We would like to explore the memory and storage constraints of deploying Transformers on tiny devices. This goes hand in hand with Chapter 4 of our dissertation, where we explore implementation of quantized models on microcontrollers.

Next, we'd like to explore building an generalized time series architecture for multi-task time series learning. Multi-task learning is an ML field which aims to utilize useful information contained in multiple related tasks to help improve the generalization performance of all the tasks [179–181]. In our architecture in Chapter 1, we utilize the general Transformer framework of Zerveas et al. [1], which is capable of learning pre-trained multivariate time series representations. Zerveas et al. use a pretraining method which learns to impute values of multivariate time-series representations. It is then fine-tuned for downstream tasks such as classification and regression on the same dataset used for pretraining. We instead aim to use this architecture to learn multiple different time series tasks at once. In the original paper, the authors found that the architecture could be used for classification and regression. We extended the architecture to anomaly detection and single-step forecasting as well. Our goal in future work is to tune this model to learn multiple tasks (such as classification and forecasting) at once from the same data. To achieve this, we will review existing literature

on time-series multi task learning as well as glean information from other fields, such as vision Transformers, which have been show to be capable of multi-task learning with just a few samples of data [182].

Similar to multi-task learning for time-series data, we also propose exploring multi-task learning for computer vision problems using Vision Transformers. For example, Kim et al. use few-shot learning [182] to learn dense prediction tasks. We believe these models could be quantized and pruned substantially. Our goal is to achieve a highly compressed model from quantization while additionally compressing the model by packing multiple learning tasks into the single model.

Finally, we would also like to expand our research on tiled bit parameters. Specifically, we would like to complete our experiments on larger scale models, such as LLM's like BERT and ResNet models trained on ImageNet. Further, we would like to implement the approach on resource constrained machines such as microcontrollers. We believe we can theoretically deploy a ResNet50 model (which has 25 million parameters) onto a microcontroller with less than 1 MB of storage capacity.

6.2 Conclusion

In this work we propose several novel algorithms and findings in the field of neural network compression. First, we examine the efficacy of compressing Transformer models down to single bit weights overlaid with a sparse single bit matrix. This work is the first to look at quantizing Transformers down to single bits. Additionally, we apply the approach to three time series learning tasks (classification, anomaly detection forecasting) which are relevant to resource-constrained environments. To the best of our knowledge, this is the first work to examine model compression specifically for time series tasks. The second objective of our research proposes a new method for creating high accuracy subnetworks within randomly-initialized neural networks. We find that the method, Iterative Weight Recycling, can achieve higher accuracy on smaller models compared to previous approaches. Our second algorithm, Tiled Bit Parameters, learns sequences of bits to tile through each layer, leading to a substantial decrease in storage space. Finally, we implement model

binarization and sparsification in a real-world application. Specifically, since modern libraries do not support single bit data types, we pack single-bit weights into a neural network which contains 8-bit integer data types. This is the first work to implement the end-to-end sparse binary model structure in a real application. We deploy the model to a small microcontroller to show the efficacy of sparse and binary models. Overall, this research contributes too several categories of neural network compression and quantization, and we argue that each approach can provide value to both the academia and industry.

Bibliography

[1] George Zerveas, Srideepika Jayaraman, Dhaval Patel, Anuradha Bhamidipaty, and Carsten Eickhoff. A transformer-based framework for multivariate time series representation learning. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 2114–2124, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Jean-Yves Franceschi, Aymeric Dieuleveut, and Martin Jaggi. Unsupervised scalable representation learning for multivariate time series. *Advances in Neural Information Processing Systems*, 32, 2019.

[3] Kyunghyun Cho. The promise and perils of ai in healthcare. *Harvard Public Health Review*, 26:1–4, 2019.

[4] Yuheng Zheng, Shuai Zhang, Ruilin Lyu, and Jian Huang. Artificial intelligence in finance: A state-of-the-art survey. *International Journal of Finance & Economics*, 2(4):204–217, 2020.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[6] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. Graphics processing units. *Communications of the ACM*, 45(8):109–116, 2007.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 30–38, 2017.

[9] 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.

[10] Alec Radford and Tim Salimans. Improving language understanding by generative pretraining. 2018.

[11] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 52(10):30–39, 2019.

[12] Weisong Shi, Qing Cao, Jie Zhang, and You Li. Edge computing: Vision and challenges. In *IEEE Internet of Things Journal*, volume 3, pages 637–646. IEEE, 2016.

[13] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both Weights and Connections for Efficient Neural Networks. October 2015. arXiv: 1506.02626.

[14] Yann LeCun, John Denker, and Sara Solla. Optimal Brain Damage. In *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

[15] Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. March 2019. arXiv: 1803.03635.

[16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *Advances in Neural Information Processing Systems*, 2016.

[17] Suyog Gupta and Michael Jacob. Quantization and training of neural networks: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[18] Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in Neural Information Processing Systems*, 33:6377–6389, 2020.

[19] Yulong Wang, Xiaolu Zhang, Lingxi Xie, Jun Zhou, Hang Su, Bo Zhang, and Xiaolin Hu. Pruning from Scratch. *arXiv:1909.12579 [cs]*, September 2019. arXiv: 1909.12579.

[20] Pytorch quantization. https://pytorch.org/docs/stable/quantization.html.

[21] James O' Neill. An Overview of Neural Network Compression. *arXiv:2006.03669 [cs, stat]*, August 2020. arXiv: 2006.03669.

[22] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*.

[23] Babak Hassibi, David G Stork, and Gregory J Wolff. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in Neural Information Processing Systems*, 5:164–171, 1993.

[24] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Lin, Shouhong Huang, and Anton van den Hengel. Sparse convolutional neural networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 806–814, 2017.

[25] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[26] James Diffenderfer and Bhavya Kailkhura. Multi-prize lottery ticket hypothesis: Finding accurate binary neural networks by pruning a randomly weighted network. In *International Conference on Learning Representations*, 2021.

[27] Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. What's Hidden in a Randomly Weighted Neural Network? In *Computer Vision and Pattern Recognition (CVPR)*, 2020.

[28] Hanting Chen, Yunhe Wang, Tianyu Guo, Chang Xu, Yiping Deng, Zhenhua Liu, Siwei Ma, Chunjing Xu, Chao Xu, and Wen Gao. Pre-Trained Image Processing Transformer. In

2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pages 12294–12305, Nashville, TN, USA, June 2021. IEEE.

[29] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *arXiv preprint arXiv:1502.02551*, 2015.

[30] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[31] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(11):5784–5789, November 2018. Conference Name: IEEE Transactions on Neural Networks and Learning Systems.

[32] Tim Dettmers. 8-Bit Approximations for Parallelism in Deep Learning. *arXiv:1511.04561 [cs]*, February 2016. arXiv: 1511.04561.

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

[34] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Quantization and training of neural networks for efficient inference on edge devices. *arXiv preprint arXiv:1611.07145*, 2016.

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

[36] Wenlin Tang, Guoqiang Hua, Bo Hu, and Jian Wang. Train high accuracy deep neural networks with low precision weights and activations. *arXiv preprint arXiv:1705.08974*, 2017.

[37] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in Neural Information Processing Systems*, 2016.

[38] Yusuf Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, Kees A Vissers, and George A Constantinides. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM, 2017.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[40] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. September 2014.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Las Vegas, NV, USA, June 2016. IEEE.

[42] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Herve Jegou. Training data-efficient image transformers & distillation through attention. In *Proceedings of the 38th International Conference on Machine Learning*, pages 10347–10357. PMLR, July 2021. ISSN: 2640-3498.

[43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics:*

Human Language Technologies, Volume 1 (Long and Short Papers), pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[44] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[45] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. *Advances in Neural Information Processing Systems*, 33:15834–15846, 2020.

[46] Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Hassan Sajjad, Preslav Nakov, Deming Chen, and Marianne Winslett. Compressing large-scale transformer-based models: A case study on bert. *Transactions of the Association for Computational Linguistics*, 9:1061–1080, 2021.

[47] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. TinyBERT: Distilling BERT for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, Online, November 2020. Association for Computational Linguistics.

[48] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[49] Jiehui Xu, Haixu Wu, Jianmin Wang, and Mingsheng Long. Anomaly transformer: Time series anomaly detection with association discrepancy. In *International Conference on Learning Representations*, 2022.

[50] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. TranAD: deep transformer networks for anomaly detection in multivariate time series data. *Proceedings of the VLDB Endowment*, 15(6):1201–1214, February 2022.

[51] Shizhan Liu, Hang Yu, Cong Liao, Jianguo Li, Weiyao Lin, Alex X Liu, and Schahram Dustdar. Pyraformer: Low-complexity pyramidal attention for long-range time series modeling and forecasting. In *International Conference on Learning Representations*, 2021.

[52] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11106–11115, 2021.

[53] Andrew A Cook, Göksel Mısırlı, and Zhong Fan. Anomaly detection for iot time-series data: A survey. *IEEE Internet of Things Journal*, 7(7):6481–6494, 2019.

[54] Pankaj Malhotra, Anusha Ramakrishnan, Gaurangi Anand, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. Lstm-based encoder-decoder for multi-sensor anomaly detection. *arXiv preprint arXiv:1607.00148*, 2016.

[55] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. Robust Anomaly Detection for Multivariate Time Series through Stochastic Recurrent Neural Network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2828–2837, Anchorage AK USA, July 2019. ACM.

[56] Sriram Baireddy, Sundip R Desai, James L Mathieson, Richard H Foster, Moses W Chan, Mary L Comer, and Edward J Delp. Spacecraft time-series anomaly detection using transfer

learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1951–1960, 2021.

[57] Kshitij Bhardwaj and Maya Gokhale. Semi-supervised on-device neural network adaptation for remote and portable laser-induced breakdown spectroscopy. *arXiv preprint arXiv:2104.03439*, 2021.

[58] Qingsong Wen, Tian Zhou, Chaoli Zhang, Weiqi Chen, Ziqing Ma, Junchi Yan, and Liang Sun. Transformers in Time Series: A Survey. March 2022. Number: arXiv:2202.07125 arXiv:2202.07125 [cs, eess, stat].

[59] Minghao Liu, Shengqi Ren, Siyuan Ma, Jiahui Jiao, Yizhou Chen, Zhiguang Wang, and Wei Song. Gated transformer networks for multivariate time series classification. *arXiv preprint arXiv:2103.14438*, 2021.

[60] Marc Rußwurm and Marco Körner. Self-attention for raw optical satellite time series classification. *ISPRS journal of photogrammetry and remote sensing*, 169:421–435, 2020.

[61] Hengyu Meng, Yuxuan Zhang, Yuanxiang Li, and Honghua Zhao. Spacecraft anomaly detection via transformer reconstruction error. In *International Conference on Aerospace System Science and Engineering*, pages 351–362. Springer, 2019.

[62] Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *Advances in Neural Information Processing Systems*, 32, 2019.

[63] Tian Zhou, Ziqing Ma, Qingsong Wen, Xue Wang, Liang Sun, and Rong Jin. FEDformer: Frequency enhanced decomposed transformer for long-term series forecasting. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 27268–27286. PMLR, 17–23 Jul 2022.

[64] Sifan Wu, Xi Xiao, Qianggang Ding, Peilin Zhao, Ying Wei, and Junzhou Huang. Adversarial sparse transformer for time series forecasting. *Advances in Neural Information Processing Systems*, 33:17105–17115, 2020.

[65] Haixu Wu, Jiehui Xu, Jianmin Wang, and Mingsheng Long. Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting. *Advances in Neural Information Processing Systems*, 34:22419–22430, 2021.

[66] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. April 2017. arXiv: 1611.05128.

[67] Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. Proving the Lottery Ticket Hypothesis: Pruning is All You Need. In *Proceedings of the 37th International Conference on Machine Learning*, pages 6682–6691. PMLR, November 2020. ISSN: 2640-3498.

[68] Daiki Chijiwa, Shin' ya Yamaguchi, Yasutoshi Ida, Kenji Umakoshi, and Tomohiro INOUE. Pruning Randomly Initialized Neural Networks with Iterative Randomization. In *Advances in Neural Information Processing Systems*, volume 34, pages 4503–4513. Curran Associates, Inc., 2021.

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

[70] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *arXiv:1308.3432 [cs]*, August 2013. arXiv: 1308.3432.

[71] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Comput. Surv.*, apr 2022. Just Accepted.

[72] Sebastian Jaszczur, Aakanksha Chowdhery, Afroz Mohiuddin, Lukasz Kaiser, Wojciech Gajewski, Henryk Michalewski, and Jonni Kanerva. Sparse is enough in scaling transformers. *Advances in Neural Information Processing Systems*, 34:9895–9907, 2021.

[73] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[74] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *J. Mach. Learn. Res*, 23:1–40, 2021.

[75] Jiezhong Qiu, Hao Ma, Omer Levy, Scott Wen-tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding. *arXiv preprint arXiv:1911.02972*, 2019.

[76] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[77] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

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

[79] Hang Zhang, Yeyun Gong, Yelong Shen, Weisheng Li, Jiancheng Lv, Nan Duan, and Weizhu Chen. Poolingformer: Long document modeling with pooling attention. In *International Conference on Machine Learning*, pages 12437–12446. PMLR, 2021.

[80] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[81] Woosuk Kwon, Sehoon Kim, Michael W. Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. A fast post-training pruning framework for transformers. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[82] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M Rush. Block pruning for faster transformers. *arXiv preprint arXiv:2109.04838*, 2021.

[83] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE, 2019.

[84] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

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

[86] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV*, pages 525–542. Springer, 2016.

[87] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[88] Matt Gorbett, Hossein Shirazi, and Indrakshi Ray. Local intrinsic dimensionality of iot networks for unsupervised intrusion detection. In *Data and Applications Security and Privacy XXXVI: 36th Annual IFIP WG 11.3 Conference, DBSec 2022, Newark, NJ, USA, July 18–20, 2022, Proceedings*, pages 143–161. Springer, 2022.

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

[90] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery*, 31:606–660, 2017.

[91] Angus Dempster, François Petitjean, and Geoffrey I Webb. Rocket: exceptionally fast and accurate time series classification using random convolutional kernels. *Data Mining and Knowledge Discovery*, 34(5):1454–1495, 2020.

[92] Kyle Hundman, Valentino Constantinou, Christopher Laporte, Ian Colwell, and Tom Soderstrom. Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 387–395, 2018.

[93] Lifeng Shen, Zhuocong Li, and James Kwok. Timeseries anomaly detection using temporal hierarchical one-class network. *Advances in Neural Information Processing Systems*, 33:13016–13026, 2020.

[94] Alban Siffer, Pierre-Alain Fouque, Alexandre Termier, and Christine Largouet. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1067–1075, 2017.

[95] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

[96] Ankit Pensia, Shashank Rajput, Alliot Nagle, Harit Vishwakarma, and Dimitris Papailiopoulos. Optimal Lottery Tickets via Subset Sum: Logarithmic Over-Parameterization

is Sufficient. In *Advances in Neural Information Processing Systems*, volume 33, pages 2599–2610. Curran Associates, Inc., 2020.

[97] Laurent Orseau, Marcus Hutter, and Omar Rivasplata. Logarithmic Pruning is All You Need. In *Advances in Neural Information Processing Systems*, volume 33, pages 2925–2934. Curran Associates, Inc., 2020.

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

[99] Mingxing Tan and Quoc Le. Efficientnetv2: Smaller models and faster training. In *International Conference on Machine Learning*, pages 10096–10106. PMLR, 2021.

[100] Ari Morcos, Haonan Yu, Michela Paganini, and Yuandong Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[101] James Diffenderfer, Brian R Bartoldson, Shreya Chaganti, Jize Zhang, and Bhavya Kailkhura. A Winning Hand: Compressing Deep Networks Can Improve Out-Of-Distribution Robustness. 2021.

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

[103] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. March 2014. arXiv: 1503.02531.

[104] Mao Ye, Chengyue Gong, Lizhen Nie, Denny Zhou, Adam Klivans, and Qiang Liu. Good Subnetworks Provably Exist: Pruning via Greedy Forward Selection. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, pages 10820–10830. PMLR, November 2020. ISSN: 2640-3498.

[105] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G. Baraniuk, Zhangyang Wang, and Yingyan Lin. Drawing Early-Bird Tickets: Towards More Efficient Training of Deep Networks. *arXiv:1909.11957 [cs, stat]*, February 2022. arXiv: 1909.11957.

[106] Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking Winning Tickets Before Training by Preserving Gradient Flow. September 2019.

[107] Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. In *Advances in Neural Information Processing Systems*, volume 33, pages 6377–6389, 2020.

[108] Claudio Gallicchio and Simone Scardapane. Deep Randomized Neural Networks. In Luca Oneto, Nicolò Navarin, Alessandro Sperduti, and Davide Anguita, editors, *Recent Trends in Learning From Data: Tutorials from the INNS Big Data and Deep Learning Conference (INNSBDDL2019)*, Studies in Computational Intelligence, pages 43–68. Springer International Publishing, Cham, 2020.

[109] Deanna Needell, Aaron A. Nelson, Rayan Saab, and Palina Salanevich. Random Vector Functional Link Networks for Function Approximation on Manifolds. *arXiv:2007.15776 [cs, math, stat]*, July 2020. arXiv: 2007.15776.

[110] Yoh-Han Pao, Gwang-Hoon Park, and Dejan J. Sobajic. Learning and generalization characteristics of the random vector functional-link net. *Neurocomputing*, 6(2):163–180, April 1994.

[111] Y.-H. Pao and Y. Takefuji. Functional-link net computing: theory, system architecture, and functionalities. *Computer*, 25(5):76–79, May 1992. Conference Name: Computer.

[112] Quoc Le, Tamas Sarlos, and Alexander Smola. Fastfood - Computing Hilbert Space Expansions in loglinear time. In *Proceedings of the 30th International Conference on Machine Learning*, pages 244–252. PMLR, May 2013. ISSN: 1938-7228.

[113] Ali Rahimi and Benjamin Recht. Random Features for Large-Scale Kernel Machines. In *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.

[114] Raffay Hamid, Ying Xiao, Alex Gittens, and Dennis DeCoste. Compact Random Feature Maps. page 9.

[115] Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, August 2009.

[116] Dianhui Wang and Ming Li. Stochastic Configuration Networks: Fundamentals and Algorithms. *IEEE Transactions on Cybernetics*, 47(10):3466–3479, October 2017. Conference Name: IEEE Transactions on Cybernetics.

[117] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. page 10, 2015.

[118] Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. Training Binary Neural Networks with Real-to-Binary Convolutions. March 2020. arXiv: 2003.11535.

[119] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. Technical Report arXiv:1409.1556, arXiv, April 2015. arXiv:1409.1556 [cs] type: article.

[120] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. Technical Report arXiv:1512.03385, arXiv, December 2015. arXiv:1512.03385 [cs] type: article.

[121] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

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

[123] Zeyuan Allen-Zhu, Yuanzhi Li, and Yingyu Liang. Learning and generalization in overparameterized neural networks, going beyond two layers. *Advances in Neural Information Processing Systems*, 32, 2019.

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

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

[126] 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.

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

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

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

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

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

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

[133] Quang Hieu Vo, Linh-Tam Tran, Sung-Ho Bae, Lok-Won Kim, and Choong Seon Hong. Mst-compression: Compressing and accelerating binary neural networks with minimum spanning tree. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6091–6100, 2023.

[134] Yikai Wang, Wenbing Huang, Yinpeng Dong, Fuchun Sun, and Anbang Yao. Compacting binary neural networks by sparse kernel selection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 24374–24383, 2023.

[135] Philippe Tillet, H. T. Kung, and David Cox. 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*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[136] Yuanzhi Li and Yingyu Liang. Learning overparameterized neural networks via stochastic gradient descent on structured data. *Advances in Neural Information Processing Systems*, 31, 2018.

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

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

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

[140] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. Towards effective low-bitwidth convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7920–7928, 2018.

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

[142] Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, and Jingkuan Song. Forward and backward information retention for accurate binary neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2250–2259, 2020.

[143] Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. 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*, pages 143–159. Springer, 2020.

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

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

[146] Zihan Xu, Mingbao Lin, Jianzhuang Liu, Jie Chen, Ling Shao, Yue Gao, Yonghong Tian, and Rongrong Ji. Recu: Reviving the dead weights in binary neural networks. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 5178–5188, 2021.

[147] Hyeonuk Kim, Jaehyeong Sim, Yeongjae Choi, and Lee-Sup Kim. A kernel decomposition architecture for binary-weight convolutional neural networks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.

[148] Dongsoo Lee, Se Jung Kwon, Byeongwook Kim, Yongkweon Jeon, Baeseong Park, and Jeongin Yun. Flexor: Trainable fractional quantization, 2020.

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

[150] Sachin Mehta and Mohammad Rastegari. Mobilevit: Light-weight, general-purpose, and mobile-friendly vision transformer. In *International Conference on Learning Representations*, 2021.

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

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

[153] James Diffenderfer and Bhavya Kailkhura. Multi-prize lottery ticket hypothesis: Finding accurate binary neural networks by pruning a randomly weighted network. In *International Conference on Learning Representations*, 2020.

[154] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[155] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 652–660, 2017.

[156] Haotong Qin, Mingyuan Zhang, Yifu Ding, Aoyu Li, Zhongang Cai, Ziwei Liu, Fisher Yu, and Xianglong Liu. Bibench: Benchmarking and analyzing network binarization. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

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

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

[159] Ilya O Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, et al. Mlp-mixer: An all-mlp architecture for vision. *Advances in Neural Information Processing Systems*, 34:24261–24272, 2021.

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

[161] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

[162] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory, 2022.

[163] Edge impulse. edgeimpulse.com.

[164] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

[165] Tensorflow lite micro. https://github.com/tensorflow/tflite-micro-arduino-examples/tree/main.

[166] Arduino TF Lite Micro. https://docs.arduino.cc/hardware/nano-33-ble-sense.

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

[168] Pierre-Emmanuel Novac, Ghouthi Boukli Hacene, Alain Pegatoquet, Benoit Miramond, and Vincent Gripon. Quantization and deployment of deep neural networks on microcontrollers. *Sensors*, 21(9):2984, 2021.

[169] Pytorch mobile. https://pytorch.org/mobile/home/.

[170] Pytorch selective build. https://pytorch.org/blog/pytorchs-tracing-based-selective-build/.

[171] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3:517–532, 2021.

[172] Lingxi Xie, Xin Chen, Kaifeng Bi, Longhui Wei, Yuhui Xu, Lanfei Wang, Zhengsu Chen, An Xiao, Jianlong Chang, Xiaopeng Zhang, et al. Weight-sharing neural architecture search: A battle to shrink the optimization gap. *ACM Computing Surveys (CSUR)*, 54(9):1–37, 2021.

[173] Tensorflow lite. https://www.tensorflow.org/lite.

[174] Zhaofan Li, Qiang Chang, and Zhongyue Liu. Towards accurate binary convolutional neural network. *arXiv preprint arXiv:1711.11294*, 2017.

[175] Mingming Gong, Weiming Liu, Yu Yang, Wenzhen Diao, Chen Qian, Yang Wu, and Baoquan Yu. Differentiable quantization: Bridging the gap between optimization and quantization for training accurate low-bit deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

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

[177] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. In *International Conference on Learning Representations*, 2018.

[178] Alok Mishra, Tushar Krishna, Praneeth Netrapalli, and K Sai Praneeth. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. In *International Conference on Learning Representations*, 2017.

[179] Yu Zhang and Qiang Yang. A survey on multi-task learning. *IEEE Transactions on Knowledge and Data Engineering*, 34(12):5586–5609, 2022.

[180] Trevor Standley, Amir Zamir, Dawn Chen, Leonidas Guibas, Jitendra Malik, and Silvio Savarese. Which tasks should be learned together in multi-task learning? In *International Conference on Machine Learning*, pages 9120–9132. PMLR, 2020.

[181] Yu Zhang and Qiang Yang. An overview of multi-task learning. *National Science Review*, 5(1):30–43, 2018.

[182] Donggyun Kim, Jinwoo Kim, Seongwoong Cho, Chong Luo, and Seunghoon Hong. Universal few-shot learning of dense prediction tasks with visual token matching. In *The Eleventh International Conference on Learning Representations*, 2022.