Faculty of Engineering Sciences
Heidelberg University

Master Thesis
in Computer Engineering
submitted by
Sergej Bespalov
born in Moskau
02/06/2025

REDUCING GLOBAL MEMORY ACCESSES IN DNN TRAINING USING STRUCTURED WEIGHT MASKING

This Master thesis has been carried out by Sergej Bespalov
at the
Institute of Computer Engineering
under the supervision of
Prof. Dr. Holger Fröning

ABSTRACT

Training large deep neural networks (DNNs) is often constrained by memory bandwidth, with frequent global memory accesses representing a significant performance bottleneck. This thesis investigates the potential of dynamic structured weight masking to alleviate this bottleneck during training, focusing on the ResMLP architecture—a feedforward network composed exclusively of Multi-Layer Perceptrons. A novel framework implementing block-wise masking based on L2 norm magnitude and top-k selection was developed and evaluated on the CIFAR-10 dataset. The study systematically varied block sizes and sparsity ratios, analyzing the impact on classification accuracy, theoretical computational cost (FLOPs), and theoretical memory movement.

Results indicate that model accuracy remains robust up to approximately 50% sparsity when the mask is also applied during the backward pass; beyond this threshold, classification accuracy degradation is observed. Notably, larger blocks contribute to improved computational efficiency under masked backward conditions by offering hardware-friendly memory access patterns, whereas in unmasked backward passes, smaller blocks tend to perform more favorably in terms of maintaining accuracy. A key observation is the discrepancy between the substantial reduction in computationally active weights and the limited decrease in estimated memory movement, suggesting that tangible memory savings can only be achieved with hardware-aware implementations that bypass unnecessary data loads. Theoretical FLOPs decrease linearly with increasing sparsity, confirming the potential for computational efficiency gains.

Overall, this work contributes an empirical analysis of dynamic structured weight masking in MLP-based architectures, offering insights into the trade-offs between mask ratio, block granularity, and training stability. The findings underscore the importance of co-designing masking patterns to achieve improvements in both computational cost and memory access, while also highlighting considerations for maintaining training stability. Furthermore, they provide practical guidelines for the efficient training of DNNs on systems with limited memory or computational resources.

ZUSAMMENFASSUNG

Das Training großer Deep Neural Networks (DNNs) wird häufig durch die Memory Bandwidth eingeschränkt, wobei häufige Global Memory Accesses einen signifikanten Performance Bottleneck darstellen. Diese Arbeit untersucht das Potenzial von Dynamic Structured Weight Masking, um diesen Engpass während des Trainings zu mildern, mit Fokus auf die ResMLP Architecture – ein Feedforward Network, das ausschließlich aus Multi-Layer Perceptrons besteht. Ein neuartiges Framework, das Block-wise Masking auf Basis der L2 Norm Magnitude und Top-k Selection implementiert, wurde entwickelt und auf dem CIFAR-10 Dataset evaluiert. Die Studie variierte systematisch Block Sizes und Sparsity Ratios und analysierte deren Auswirkungen auf die Classification Accuracy, die Theoretical Computational Cost (FLOPs) sowie das Theoretical Memory Movement.

Die Ergebnisse zeigen, dass die Model Accuracy bis zu einer Sparsity von etwa 50% robust bleibt, sofern die Maske auch im Backward Pass angewendet wird; jenseits dieser Schwelle wird eine Verschlechterung der Classification Accuracy beobachtet. Bemerkenswerterweise tragen größere Blöcke unter Bedingungen eines maskierten Backward Pass zu einer verbesserten Computational Efficiency bei, indem sie Hardware-friendly Memory Access Patterns ermöglichen, während bei unmaskiertem Backward Pass kleinere Blöcke tendenziell vorteilhafter hinsichtlich der Erhaltung der Accuracy sind. Eine zentrale Beobachtung ist die Diskrepanz zwischen der erheblichen Reduktion der Computationally Active Weights und der begrenzten Abnahme des geschätzten Memory Movements. Dies deutet darauf hin, dass spürbare Speichereinsparungen nur durch Hardware-aware Implementations erzielt werden können, die unnötige Datentransfers umgehen. Die Theoretical FLOPs nehmen linear mit zunehmender Sparsity ab, was das Potenzial für Effizienzgewinne in der Berechnung bestätigt.

Insgesamt leistet diese Arbeit einen empirischen Beitrag zur Analyse von Dynamic Structured Weight Masking in MLP-basierten Architekturen und bietet Einblicke in die Kompromisse zwischen Mask Ratio, Block Granularity und Training Stability. Die Ergebnisse unterstreichen die Bedeutung einer sorgfältigen Gestaltung von Masking Patterns, um sowohl Verbesserungen bei den Computational Cost als auch beim Memory Access zu erzielen, und heben gleichzeitig Aspekte zur Wahrung der Training Stability hervor. Des Weiteren liefern sie praktische Richtlinien für das effiziente Training von DNNs auf Systemen mit begrenzten Speicher- oder Rechenressourcen.

Simplicity is the ultimate sophistication.

— Leonardo da Vinci

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to several individuals and institutions who have played a crucial role in the completion of this Master's thesis.

First and foremost, I extend my deepest appreciation to my supervisor, Daniel Barley. His invaluable guidance, unwavering support, and remarkable patience throughout this research journey were instrumental. I am particularly grateful for his willingness to always take the time to answer my questions in a competent and professional manner, which significantly contributed to my progress.

I am also profoundly grateful to Professor Holger Frönig, under whose esteemed leadership this thesis was undertaken. His oversight and academic direction provided a strong foundation for this work.

My thanks also go to all the members of the HAWAII Lab. The friendly, supportive, and intellectually stimulating atmosphere they cultivated made my research experience both enjoyable and productive.

Finally, I wish to acknowledge the ZITI at Heidelberg University. The academic environment and resources provided by ZITI were essential for conducting the studies that culminated in this thesis.

CONTENTS

1 INTRODUCTION AND MOTIVATION

The rapid advancement of deep neural networks (DNNs) has led to significant improvements across various domains, but it has also resulted in ever-increasing model sizes and computational complexities. Training these large-scale models requires vast computational resources and prolonged training times, with performance often limited by the capabilities of underlying hardware accelerators such as Graphics Processing Units (GPUs). A critical bottleneck in this process is the memory subsystem—specifically, the bandwidth and latency involved in accessing extensive weight tensors and intermediate activations stored in global memory [10]. The frequent transfer of data between the slower global memory and the faster on-chip memory hierarchies (e.g., shared memory, caches, registers) can dominate execution time, thereby impeding training efficiency and scalability. Mitigating this memory bottleneck is essential for developing and deploying larger, more powerful DNNs.

Sparsity, the practice of reducing the number of active parameters or operations, has emerged as a promising technique for enhancing DNN efficiency, particularly in the context of model inference [10]. However, applying sparsity during the training phase presents unique challenges and opportunities. This thesis investigates the potential of leveraging sparsity specifically to alleviate the memory bottleneck during DNN training. The core approach explored is **dynamic structured weight masking**. Unlike unstructured sparsity, which creates irregular patterns of zeroed weights that often do not map well to hardware, structured sparsity organizes the zeros into regular patterns (e.g., blocks). This regularity aligns more effectively with the parallel processing capabilities and memory access patterns of GPUs, potentially enabling more efficient data handling and coalesced memory accesses [10]. Moreover, by dynamically adapting the sparsity pattern during training via top-k selection based on the L2 norm within blocks, this method retains greater flexibility compared to static pruning techniques.

This research primarily focuses on applying dynamic structured weight masking to the **ResMLP** architecture [25], a feedforward network composed solely of Multi-Layer Perceptrons (MLPs) and designed for image classification, with comprehensive evaluation performed on the CIFAR-10 dataset. The ResMLP architecture is particularly well-suited for this investigation as it avoids the spatial dependencies inherent in convolutional networks and the complexity of attention mechanisms present in other vision models. Furthermore, to

explore the method's behavior on a significantly larger scale, preliminary experiments were also performed on the ImageNet dataset. These initial tests, conducted with limited epoch counts, showed that the masking strategies, as applied, did not achieve comparable accuracy on ImageNet, indicating that further adaptations, not explored in this thesis, would likely be necessary for effective performance on such large-scale, diverse datasets. The **primary objective** of this work is to assess the extent to which dynamic structured weight masking can **reduce global memory accesses** during the training of ResMLP networks, with the reduction of theoretical computational cost (FLOPs) serving as a secondary outcome.

To achieve this objective, a dedicated framework was developed and integrated into the ResMLP training pipeline. This framework facilitates the application of block-wise weight masking with variable block sizes and sparsity ratios. A systematic experimental methodology, including comprehensive benchmarking, was employed to analyze the trade-offs involved.

The key contributions of this thesis include:

- The development and implementation of a framework for applying dynamic structured weight masking (based on block L2 norm and top-k selection) during the training of ResMLP networks.
- Comprehensive benchmarking that evaluates the trade-offs among classification accuracy, theoretical computational cost (FLOPs) and **theoretical memory movement** across a wide range of block sizes and sparsity ratios.
- An in-depth analysis of the interplay between block granularity, sparsity levels, and model performance, providing empirically grounded insights for selecting appropriate masking parameters.
- The identification and examination of the critical role of applying the mask during the backward pass for maintaining training stability, particularly at high sparsity levels.
- A demonstration of the discrepancy between the significant reduction in computationally active parameters and the limited reduction in theoretically calculated global memory movement, highlighting the challenges in translating computational sparsity into memory access savings without hardware-aware implementations.

This thesis is structured as follows: Chapter 2 provides background information on machine learning concepts, neural network architectures including ResMLP, GPU hardware, and sparsity techniques. Chapter 3 reviews existing literature relevant to sparsity in deep learning, memory efficiency techniques, and MLP-based models. Chapter 4

details the methodological approach, including the experimental design, dataset preprocessing, masking implementation strategy, and evaluation metrics. Chapter 5 describes the technical implementation details of the framework and measurement tools. Chapter 6 presents the empirical results obtained from the benchmarking experiments. Chapter 7 interprets these results, discusses their implications and limitations, and relates them to the research objectives. Finally, Chapter 8 concludes the thesis by summarizing the key findings and suggesting directions for future work.

2 BACKGROUND

2.1 MACHINE LEARNING OVERVIEW

Machine learning (ML) is a subset of artificial intelligence (AI) that focuses on the development of algorithms enabling computers to learn patterns from data and make decisions or predictions without being explicitly programmed for each task. In traditional programming, input data is processed by a predefined set of rules to produce an output. Machine learning, however, works by allowing models to learn those rules from data itself, automating complex decision-making and pattern recognition processes [4, 16].

An illustration of this concept can be seen in Figure 2.1, which compares traditional programming and machine learning approaches. In traditional programming, a set of explicit rules is applied to input data, while in machine learning, the model learns patterns from examples (data) to make decisions or predictions.

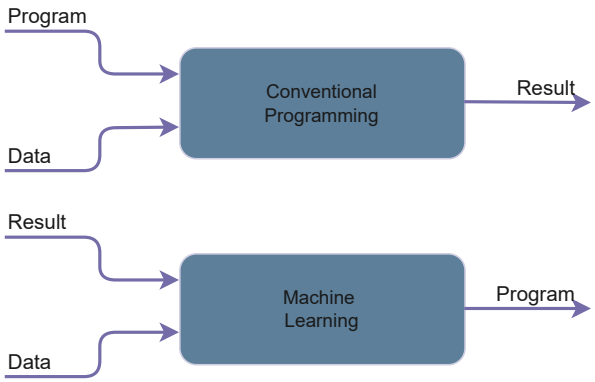


Figure 2.1: Comparison between Conventional Programming and Machine Learning: In conventional programming, data and a program are combined to produce a result, while in machine learning, data and results are used to generate a program through learning algorithms.

Deep neural networks (DNNs), a subset of machine learning (ML) models and the primary focus of this work, operate based on several key constituents. The specific learning paradigm employed in this thesis is supervised learning, which will be described first. Subsequently, the core operational components common in training such networks, namely the **forward pass**, **loss function**, and **backward**

pass (utilizing **backpropagation**), will be detailed. These components allow the model to iteratively improve its accuracy by minimizing the loss function.

2.1.1 Supervised Learning

Supervised learning involves training a machine learning model, such as a DNN, on labeled data, where each input is paired with the correct output (ground truth). The model learns to map inputs to outputs by minimizing the difference between its predictions and the actual labels, a difference quantified by a loss function. During training, processes like the forward pass and backward pass are used to iteratively adjust the model's parameters to reduce this loss. Supervised learning is most commonly employed for tasks like classification and regression and is a widely used technique in machine learning, with popular algorithms including decision trees, support vector machines, and neural networks [8].

In this work, supervised learning is the method applied to train the ResMLP model. For example, in the image classification task undertaken, the model is trained on a dataset where each image is labeled with the object it contains (e.g., dog, cat, car). During training, the model learns the distinguishing features of each class, allowing it to classify new, unseen images effectively.

2.1.2 Forward Pass

In the forward pass, the input $\mathbf{x}$ passes through the layers of the network. At each layer, the input undergoes transformations, typically involving a linear operation (such as matrix multiplication) followed by a non-linear activation function [6].

For a fully connected layer, the linear transformation of an input row vector $\mathbf{x}$ is mathematically expressed as $\mathbf{x}\mathbf{W}^T + \mathbf{b}$, yielding the output $\mathbf{z}$. In frameworks like PyTorch[1], for efficient matrix operations, the weight matrix $\mathbf{W}$ is often stored such that its transposed form, $\mathbf{W}^T$, is directly utilized in the computation. This aligns with typical memory layouts for optimized performance [6].

$$\mathbf{z} = \mathbf{x}\mathbf{W}^T + \mathbf{b} \tag{2.1}$$

where $\mathbf{x}$ is the input vector from the previous layer (or the input data for the first layer), $\mathbf{b}$ is the bias vector.

After computing $\mathbf{z}$, a non-linear activation function (such as ReLU, sigmoid, or tanh) is applied [4]:

$$\mathbf{a} = \sigma(\mathbf{z}) \tag{2.2}$$

1 PyTorch is a widely used deep learning framework that provides dynamic computational graphs and seamless integration with GPUs for efficient training. Official documentation is available at https://pytorch.org/.

where σ is the activation function, and **a** represents the activated output, which becomes the input for the next layer. The activation function is employed to introduce nonlinearity into the model, enabling the network to learn complex patterns[6]. This nonlinearity is crucial for the backpropagation algorithm, as it helps ensure that gradients are non-zero and can be propagated through the network, allowing for meaningful weight updates during the optimization steps.

The process repeats through all layers until the final output is obtained. The forward pass concludes with the model making a prediction, which is compared to the true label using the loss function. In the following subsection, the concept of the loss function will be explained in detail to provide a deeper understanding of its role in training.

2.1.3 Loss Function

The loss function $\mathcal{L}(\hat{y}, y)$ measures how well the model's prediction $\hat{y}$ matches the actual target (i.e., ground truth) y. For classification tasks, the **cross-entropy** loss is commonly used [6], which penalizes the model based on the discrepancy between its predictions and the true labels:

$$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{2.3}$$

where C is the number of classes, y_i is the true label (one-hot encoded), and $\hat{y}_i$ is the model's output for class i. Note that $\hat{y}_i$ is not directly a probability but is typically transformed into probabilities using a softmax function:

$$\hat{y}_i = \frac{\exp(z_i)}{\sum_{j=1}^{C} \exp(z_j)} \tag{2.4}$$

where z_i is the pre-softmax score (logit) for class i. The softmax function is crucial as it normalizes the logits into a probability distribution, ensuring that the predicted probabilities sum to 1, which is essential for interpreting the model's confidence in its predictions [6]. Alternatively, an argmax operation may be applied to determine the predicted class index.

These transformations are crucial for interpreting the model's output and computing the loss [6]. To understand how this loss is utilized to update the model parameters, the mechanics of the backward pass will be detailed in the following subsection.

2.1.4 Backward Pass

After computing the loss during the forward pass, the backward pass is used to update the model's parameters in order to minimize the loss.

This is accomplished through backpropagation, a process in which the gradient of the loss function with respect to each model parameter is calculated by propagating errors backward through the network using the chain rule of calculus [4].

For a parameter θ, the gradient of the loss function $\mathcal{L}$ with respect to θ is computed as:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}} \cdot \frac{\partial \mathbf{a}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \theta} \tag{2.5}$$

with the following components: $\frac{\partial \mathcal{L}}{\partial \mathbf{a}}$ is the gradient of the loss with respect to the layer's output, $\frac{\partial \mathbf{a}}{\partial \mathbf{z}}$ represents the gradient of the activation function, $\frac{\partial \mathbf{z}}{\partial \theta}$ denotes the gradient of the linear transformation.

Once the gradients are computed, the model parameters are updated using an optimization algorithm like stochastic gradient descent (SGD), through which parameters are adjusted iteratively based on the gradient derived from randomly selected mini-batches of training data [6]:

$$\theta \leftarrow \theta - \eta \cdot \frac{\partial \mathcal{L}}{\partial \theta} \tag{2.6}$$

where θ represents a model parameter, and this equation represents the gradient descent update rule, with η being the learning rate.

These two passes, forward and backward, allow the model to iteratively improve by reducing the loss function and getting closer to making accurate predictions.

2.1.5 Further Training Methods

In addition to supervised learning, other training methods such as unsupervised learning and reinforcement learning are widely used in machine learning. These methods differ in their approach to learning and the type of data they utilize.

Unsupervised learning is applied to unlabeled data, where the model identifies patterns, structures, or relationships without explicit guidance. Traditional loss functions based on prediction error cannot be used, as no predefined output is available. Instead, the model is optimized to uncover internal structures or patterns within the data. Common tasks include clustering, where similar data points are grouped together, and dimensionality reduction, where the number of features is reduced while preserving essential information. Techniques such as k-means clustering and principal component analysis (PCA) are frequently employed for these purposes [4]. For instance, unsupervised learning can be used to cluster customers based on purchasing behavior, identifying market segments such as frequent buyers or bargain hunters.

Reinforcement learning (RL), on the other hand, involves an agent that interacts with an environment to maximize cumulative rewards over time. The agent takes actions and receives feedback in the form of rewards or penalties, which are used to learn an optimal policy for decision-making [22]. Unlike supervised learning, RL does not rely on a loss function based on immediate accuracy. Instead, a reward function is utilized to guide the agent toward achieving long-term objectives. The learning process is characterized by trial and error, with adjustments to the policy being made based on accumulated rewards. Reinforcement learning is commonly applied in domains such as robotics, gaming, and autonomous vehicles, where sequential decision-making is critical.

2.2 NEURAL NETWORK ARCHITECTURES

Deep learning, a subfield of machine learning, fundamentally relies on the use of neural networks. These networks are characterized by multiple layers of interconnected processing units (neurons), which enable the automatic extraction of hierarchical features from data [14]. This capability for automated feature engineering makes deep learning particularly effective for complex tasks such as image recognition, natural language processing, and speech recognition, often eliminating the need for manual feature design. While early concepts in neural networks drew some high-level inspiration from simplified models of neural processing in the brain, modern deep learning architectures and their training mechanisms, such as backpropagation, operate on principles distinct from biological neural processes [29]. Depending on the specific task and data structure, different types of neural network architectures are employed:

2.2.1 Convolutional Neural Networks (CNNs)

CNNs are a type of feedforward network specifically designed for processing grid-like data structures, such as images. They are particularly effective at image-related tasks due to their ability to capture spatial hierarchies of features. CNNs use convolutional layers to automatically learn important features like edges, textures, and patterns in the data, progressively abstracting the information as it moves through deeper layers [6]. In the context of image classification, CNNs have been widely used in systems like facial recognition and medical imaging. The architecture is characterized by convolutional layers, pooling layers (which downsample the data), and fully connected layers at the end for classification [6].

2.2.2 Multi-Layer Perceptrons (MLPs)

MLPs are a type of feedforward network composed of multiple fully connected layers, where each neuron in one layer is connected to every neuron in the next layer. MLPs are capable of learning complex representations of data and are often used for general-purpose tasks like classification and regression [6]. While CNNs are specialized for spatial data, MLPs are used more broadly and often serve as final layers in hybrid architectures to perform tasks like decision-making or classification based on features extracted by earlier specialized layers.

2.2.3 Recurrent Neural Networks (RNNs)

RNNs, on the other hand, are designed for sequential data, with their looped architecture enabling the network to learn temporal dependencies [6]. This makes them particularly useful for natural language processing (NLP) tasks, such as language translation, speech recognition, and text generation. A specific type of RNN, Long Short-Term Memory (LSTM) networks, addresses issues of learning long-term dependencies in sequences [23].

2.2.4 ResMLP Architecture

The Residual Multi-Layer Perceptron (ResMLP) [25], utilized in this work, is a variant of the standard MLP architecture adapted for image classification tasks, simplifying the Vision Transformer (ViT) architecture. It processes an input image through a series of stages that transform raw image data into a final class prediction. An overview of these stages is provided below.

INPUT PROCESSING: PATCH EMBEDDING. In the initial stage, the input image is divided into a grid of non-overlapping patches. Each patch is flattened into a vector and then linearly projected into an embedding space of a specified dimension. This patch embedding layer converts the spatial structure of the image into a sequence of feature vectors that serve as inputs for subsequent layers. Figure 2.2 illustrates the linearization of patches for one RGB image [25].

FEATURE EXTRACTION: RESIDUAL BLOCKS The core of the ResMLP architecture consists of a stack of residual blocks that iteratively refine the patch embeddings. A key feature is the use of **residual connections**, first introduced in ResNets [6], which allow the network to bypass certain layers by passing the input directly to the output. These connections address the vanishing gradient problem, which can occur during backpropagation in deep networks [4, 16]. By enabling the gradient to flow through the network more easily, residual connections

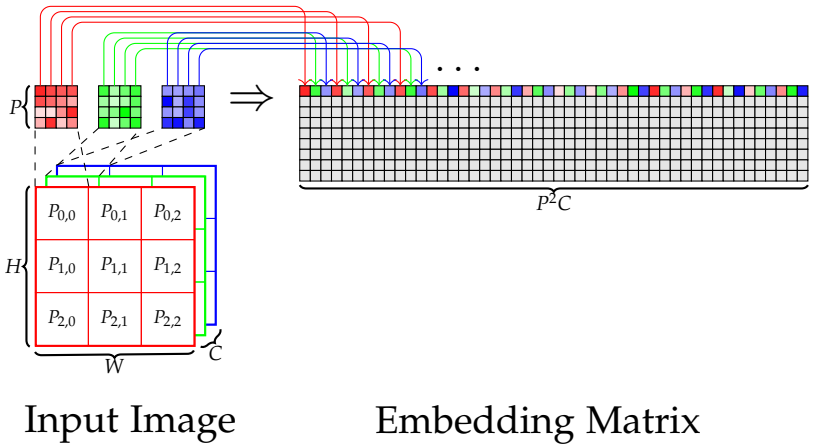


Figure 2.2: Visualization of the linearization of patches shown in detail for one patch of an RGB image. The patch is flattened along the row axis and the channel data is interleaved. The patch is thereby reduced by one dimension. Afterwards a linear layer scales the embeddings to the desired output dimension. The graphic implies sequential handling of patches only for visual clarity. Patches are processed in parallel in the actual implementation[1].

help improve convergence and allow the training of deeper architectures. In essence, residual connections allow the network to learn both an identity mapping and a more complex transformation, which leads to better optimization and more stable training dynamics [10]. Each block performs two primary operations:

Cross-Patch Mixing: This operation facilitates the exchange of information across different patches. It involves scaling the input embeddings using an affine transformation, transposing them, applying a linear layer that preserves the embedding shape, and finally transposing them back to their original configuration. A scalar factor is applied before adding the original input via a skip connection. Figure 2.3 demonstrates the forward computation of the cross-patch layer.

Cross-Channel Mixing: This operation focuses on transforming feature representations along the channel dimension within each patch. It typically expands the feature dimension by a certain factor (commonly 4) using a linear layer, applies a GELU [9] non-linearity, and then contracts the features back to the original dimension with a second linear layer. A residual connection adds the original input to the transformed output. Figure 2.4 provides a detailed view of the cross-channel layer's forward pass.

AGGREGATION AND CLASSIFICATION After feature extraction, the refined patch embeddings are aggregated to form a global representation of the image. This is achieved using global average pooling, which averages the features across all patches to produce a single feature vector. The aggregated vector is then passed through a final linear

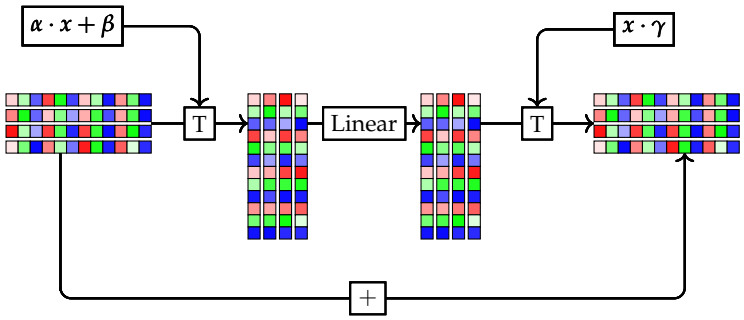


Figure 2.3: Forward computation of the cross-patch layer. The input embeddings are scaled using an affine transformation (element-wise multiplication with a scalar α and addition with a second scalar β) and transposed. The transposed embeddings are fed to a linear layer, which maintains the embeddings' shapes. The embeddings are then transposed back to their original shape and scaled element-wise by another scalar factor γ. Finally, the original input is added to the result, realizing the skip connection [1].

layer—the classification head—which projects the features onto the output classes, yielding the final prediction.

SUITABILITY FOR THIS WORK In summary, the ResMLP architecture converts an input image into a sequence of patch embeddings, refines these embeddings through residual blocks that employ both cross-patch and cross-channel mixing, and aggregates the learned features for classification. ResMLP suits this work particularly well because it offers excellent optimization potential, which aligns with the goals of this research focusing on memory efficiency. Several factors make ResMLP an ideal candidate for optimization through structured weight masking:

- Its reliance on MLP blocks, primarily involving matrix multiplications (linear layers), avoids the specific computational patterns and overhead associated with convolutional filters or complex self-attention mechanisms.
- The architecture is well-suited for sparsity-based optimizations. Unlike convolutional layers which operate with small, spatially structured kernels and maintain explicit spatial hierarchies in their feature maps, the linear layers in ResMLP process patch embeddings through large weight matrices. The absence of small, shared convolutional filters and the direct application of these large matrices to the full sequence of patch embeddings allows for a more direct and flexible application of structured block-masking patterns to the weight parameters.
- Its architectural design, which notably omits the complex self-attention mechanisms found in Vision Transformers (ViTs) [25], contributes to a potentially more favorable efficiency profile.

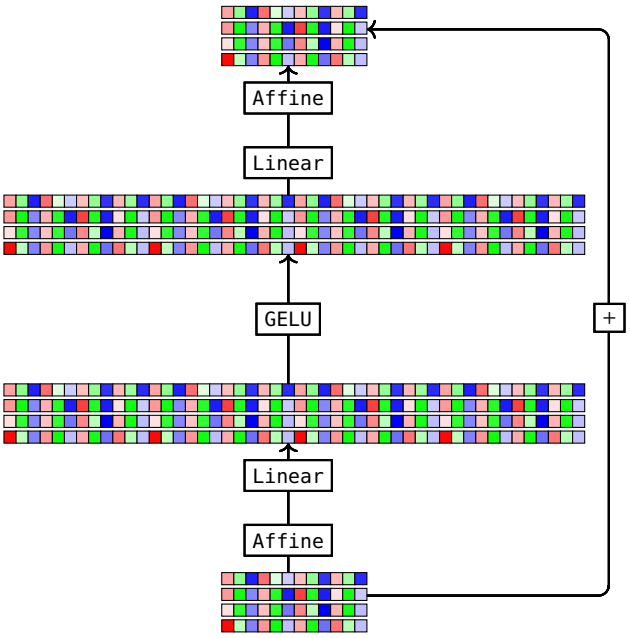


Figure 2.4: Cross-channel layer forward pass. As with the cross-patch layer there is pre- and post-layer affine scaling. The input is expanded by a factor of 4 by a first linear layer. A GELU [9] non-linearity is applied and a second linear layer contracts the patch data to the original size. Finally, the original input is added to the output[1].

While its total parameter count might be higher than that of CNNs due to the latter's use of weight sharing, the simpler per-parameter operations in ResMLP can be advantageous.

By leveraging these properties, this research focuses on optimizing the training process of the ResMLP model through the application of dynamic structured weight masking. The primary aim of this masking approach is to reduce memory movement during training, with the reduction of computational costs being a beneficial secondary effect [2, 3].

2.3 GPU ACCELERATION IN DEEP LEARNING

Graphics Processing Units (GPUs) are crucial for accelerating deep learning workloads due to their parallel processing capabilities. Neural networks involve extensive computations, such as matrix multiplications on large datasets, which are inherently parallelizable. [24].

A modern GPU architecture typically consists of many parallel processing units. In NVIDIA GPUs, these are referred to as CUDA cores[2],

2 CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA, allowing developers to leverage the power of GPUs for general-purpose computing. More details are available at https://developer.nvidia.com/cuda-zone.

which execute computations in parallel. GPUs manage thousands of concurrent threads, making them highly efficient for large-scale parallel data processing common in machine learning, such as matrix and tensor calculations. These threads are often organized into groups (e.g., "warps" in NVIDIA terminology, typically comprising 32 threads) that can execute the same instruction on different data (SIMD/SIMT paradigm) [11]. Such groups are scheduled onto larger execution units within the GPU, often called Streaming Multiprocessors (SMs) in NVIDIA architectures, which contain multiple processing cores. This hierarchical organization of threads, blocks of threads, and grids of blocks enables massive parallelism [18].

A critical factor for GPU performance is **coalesced memory access**, ensuring that threads within a concurrently executing group access contiguous memory locations. Coalesced accesses allow the GPU to consolidate multiple memory requests into fewer, larger transactions, thereby improving the effective utilization of memory bandwidth and reducing latency. This is vital because global memory latency on GPUs is significantly higher than on-chip computation speeds. Uncoalesced accesses lead to fragmented transactions, underutilizing bandwidth and increasing overall execution time [10]. Structured sparsity techniques, such as the block-wise masking explored in this thesis, inherently promote coalesced memory access by organizing active weights into contiguous blocks, which can mitigate irregular memory access patterns often associated with unstructured sparsity.

The NVIDIA A30 GPU, utilized in this work's experiments, is optimized for machine learning workloads. It features specialized units like Tensor Cores for accelerating matrix operations and is equipped with 24 GB of HBM2e memory, providing a peak memory bandwidth of up to 933 GB/s. This high bandwidth is essential for rapidly supplying data to the processing units. Furthermore, the GPU's memory system, including its cache hierarchy and defined cache line size (e.g., 128 bytes for the A30), influences memory efficiency. Accessing data aligned with cache lines allows the memory controller to fetch data in larger, more efficient chunks, potentially reducing the number of individual memory transactions and the overhead associated with unaligned or scattered data fetches, especially when data locality can be exploited. The A30 operates at a base frequency of 930 MHz, boostable to 1440 MHz, with memory running at 1215 MHz[3].

Frameworks like PyTorch simplify the utilization of GPU parallelism by abstracting many low-level details of parallel execution and memory management, allowing developers to more easily leverage GPU capabilities for deep learning.

3 Key specifications for the NVIDIA A30 PCIe GPU include 3584 CUDA cores and 224 Tensor Cores, a peak memory bandwidth of 933 GB/s, and theoretical peak performance of 10.3 TFLOPS (FP32), 82.6 TFLOPS (TF32), and up to 165 TFLOPS (FP16 with sparsity). Source: https://www.techpowerup.com/gpu-specs/a30-pcie.c3792.

2.4 DATASETS FOR IMAGE CLASSIFICATION

CIFAR-10 is a widely recognized benchmark dataset for image classification tasks. It consists of 60,000 low-resolution color images (32x32 pixels) divided into 10 distinct classes, with 50,000 images designated for training and 10,000 for testing[4]. The images in CIFAR-10 are characterized by their small size and depict a variety of common objects, including airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The low resolution of these images presents a challenge for classification algorithms, requiring models capable of capturing subtle visual features.

Another relevant benchmark is the CIFAR-100 dataset. While also containing 60,000 32x32 color images (split 50,000/10,000 for training/testing), it features 100 classes. This results in finer-grained categorization and a wider array of object types compared to CIFAR-10, making it a more complex classification challenge. Both CIFAR datasets are extensively used in the machine learning community for evaluating the performance of image classification models, such as CNNs and MLPs, due to their challenging nature and substantial sample sizes.

For evaluating models in more complex and realistic scenarios, the ImageNet dataset serves as a standard large-scale benchmark[5]. ImageNet comprises millions of high-resolution images distributed across thousand object categories, representing a significantly more demanding task for image classification models compared to the CIFAR datasets.

2.5 PRUNING AND MASKING TECHNIQUES

To optimize computational efficiency, reduce global memory accesses, and achieve theoretical floating-point operation (FLOP) savings during training, masking of weights is employed in this work. Masking, in this context, involves dynamically identifying and effectively ignoring certain elements (e.g., weights) during computation based on specific criteria. While the masked weights remain stored in memory, their contribution to the forward and potentially backward pass is nullified, thereby reducing the computational cost. This approach differs from typical pruning methods where parameters are permanently removed from the network structure [10], although pruning itself can also employ dynamic criteria during training. The masking method used here focuses on ephemeral sparsity, where sparse patterns are applied dynamically and non-destructively [2], offering flexibility by not inducing permanent structural changes.

4 CIFAR datasets official website: `https://www.cs.toronto.edu/ kriz/cifar.html`.

5 ImageNet is a large-scale image dataset containing millions of labeled images across thousand categories, commonly used for training and evaluating deep learning models. More information can be found at `https://image-net.org`.

By reducing the active elements involved in computation, masking directly contributes to theoretical FLOP savings, indicating potential improvements in the computational efficiency of deep learning models. Furthermore, if masking is implemented within custom compute kernels (e.g., in CUDA) designed to recognize and skip masked elements, it can substantially reduce memory movement by minimizing the data loaded from global memory during computations. This potential reduction in memory traffic is particularly useful for managing memory bottlenecks and computational overhead in models like ResMLP. Applying structured masking techniques, where blocks of weights are dynamically masked, not only reduces the theoretical FLOP count but also creates regular sparsity patterns. These patterns are potentially more amenable to hardware acceleration and optimized memory access strategies compared to unstructured sparsity, enhancing the prospects for practical memory efficiency gains and making it feasible to train larger models on GPUs like the NVIDIA A30 [3].

2.5.1 Structured Sparsity in Neural Networks

Structured sparsity in neural networks refers to the deliberate organization of weight sparsity into predefined patterns, such as blocks, rather than allowing arbitrary, unstructured sparse distributions. This approach aligns closely with the memory hierarchies and computational capabilities of modern hardware, particularly GPUs [10]. By grouping weights into blocks (e.g., 8×8 or 16×16), structured sparsity aims to reduce the overhead associated with indexing scattered sparse elements, thereby potentially lowering memory access costs and enabling GPU-friendly optimizations, which are central to this work. The efficiency gains from structured sparsity are particularly relevant for mitigating the performance bottleneck caused by global memory accesses in deep neural network (DNN) training [10].

In the context of MLP-based architectures like ResMLP, structured sparsity offers distinct advantages. Unlike CNNs, which have strong spatial inductive biases due to their convolutional kernels and locally connected layers, the linear layers in ResMLP operate on flattened sequences of patch embeddings using large weight matrices. The absence of these inherent, localized spatial dependencies and weight-sharing patterns found in convolutional layers means ResMLP's weight matrices can be partitioned into blocks with considerable flexibility, potentially with minimal disruption to the model's representational capacity [25].

This thesis explores how varying block sizes in such a structured sparsity scheme influences computational efficiency and accuracy when training ResMLP on the CIFAR-10 dataset. The potential benefits, such as reducing redundant memory transactions, could lead to lower energy costs for training and enable the deployment of larger models

on resource-constrained hardware. These outcomes directly align with the hardware-aware optimization objectives of this research [2]. By designing masking strategies that are cognizant of GPU memory access patterns, this work seeks to enhance the practical feasibility of deploying ResMLP efficiently on hardware like the NVIDIA A30, contributing to the overarching goal of resource-efficient DNN training [25].

2.5.2 Dynamic vs. Static Masking

Masking strategies in neural network training can be broadly categorized into static and dynamic approaches, each with distinct implications for model flexibility and performance. Static masking, often implemented as a form of pruning, involves the application of a fixed sparsity pattern to the network's weights before or during training, with the mask remaining unchanged thereafter [10]. While effective for reducing computational complexity and memory usage, static masking lacks adaptability, as it cannot respond to evolving training dynamics or input-specific patterns. This rigidity can lead to suboptimal accuracy, especially in tasks requiring nuanced feature representations, such as image classification with ResMLP.

In contrast, dynamic masking is employed in this work, where the sparsity pattern is recomputed adaptively during training. Specifically, a top-k sparsity criterion is applied per forward pass, selecting the k most significant weights (based on magnitude) for each input batch and masking the rest [2]. This dynamic adjustment allows the model to adapt to the input data, preserving flexibility and potentially capturing more relevant features compared to static methods. For instance, in the ResMLP architecture, where layers process global information without convolutional locality, dynamic masking ensures that the most impactful weights are retained for each specific image, enhancing the model's ability to generalize across datasets like CIFAR-10 [25].

The trade-off with dynamic masking lies in its computational overhead, as the mask must be recalculated periodically during training (e.g., each forward pass), unlike static approaches where the sparsity pattern is determined upfront or only once. However, in the structured approach used here, this overhead is associated with processing block-level statistics rather than individual weights, and the resulting block patterns are designed to be compatible with efficient GPU execution [3]. By combining dynamic adaptation with structured sparsity, a balance between flexibility and potential hardware efficiency is achieved. This dynamic, structured approach differs significantly from conventional static pruning techniques, which typically fix the sparsity pattern early on and often result in unstructured sparsity that is harder to accelerate [10]. The adaptability inherent in recalculating the mask is particularly valuable for maintaining accuracy across different

sparsity ratios, a key focus of the experimental analysis conducted in this work.

2.6 PTFLOPS

The **ptflops** library[6] was utilized in this project to calculate the theoretical floating-point operations (FLOPs) of the ResMLP network. **ptflops** is a lightweight Python tool designed for profiling PyTorch-based deep learning models [17]. It operates by attaching hooks to the layers of the network and recording the number of multiply-accumulate operations performed during a forward pass. This count is then typically converted to FLOPs (often by multiplying by two), providing a theoretical estimate of computational complexity rather than a direct measurement of hardware-specific execution time or energy [17].

While a direct measurement of FLOPs reduction through specialized sparse kernels was not implemented in this work, the **ptflops** library serves as a useful tool for theoretically quantifying the potential computational efficiency gains of the ResMLP network under different masking conditions. It is important to note that **ptflops** provides theoretical FLOPs based on the model architecture and the applied mask ratios; these figures may not perfectly reflect actual hardware performance improvements. Actual speedups are contingent on factors such as memory access patterns, cache utilization, specific hardware optimizations, and the availability of compute kernels that can effectively skip operations on masked elements. Nonetheless, the library supports standard PyTorch layers, including convolutions and linear operations, and allows custom definitions for unsupported layers [17]. Its straightforward API makes it adaptable for research focused on model optimization and resource efficiency. In this project, **ptflops** was employed to provide insights into the theoretical performance trade-offs associated with sparsity in the ResMLP network.

6 The `ptflops` library by Dmitry Nikolaev is available at https://github.com/sovrasov/flops-counter.pytorch.

3 LITERATURE REVIEW

This chapter reviews existing literature relevant to the core themes of this thesis: enhancing the efficiency of deep learning model training. The review focuses on techniques related to sparsity, particularly dynamic and structured approaches, memory efficiency strategies during training, and relevant advancements in MLP-based architectures pertinent to this work, such as ResMLP. The aim is to situate the contributions of this thesis within the context of current research, building upon foundational concepts introduced in Chapter 2.

3.1 WEIGHT MASKING AND DYNAMIC SPARSITY IN TRAINING

3.1.1 Principles and Applications of Weight Masking

Weight masking, in its general sense, refers to the selective deactivation or modulation of certain weights within a neural network. This is typically achieved by multiplying weights with a binary or continuous-valued mask, effectively removing or down-weighting their contribution to the network's computations for a given input or task. While this thesis focuses on dynamic structured weight masking for training efficiency, the principle of weight masking has been explored in literature for a diverse range of other purposes.

One significant application is the enhancement of model robustness against adversarial examples. For instance, Kubo et al. [12] proposed the Stochastic-Gated Partially Binarized Network (SGBN), where a gate module, itself a shallow convolutional network, learns input-dependent probabilities for stochastically masking (turning on or off) individual weights in the main network's convolutional filters. This dynamic, input-specific masking aims to make the model less susceptible to small, crafted perturbations designed to cause misclassification.

Another area where weight masking has been applied is machine unlearning, which focuses on making a model "forget" specific data it was trained on. Wang et al. [26] propose a method for class-specific unlearning where weights associated with the feature vectors of a target class to be forgotten are masked. Their approach identifies class-specific weights by analyzing feature vector statistics and aims to render the model unable to recognize the forgotten class without

retraining or significant additional computation, while preserving performance on retained classes.

Furthermore, weight masking techniques are being investigated to improve fairness and mitigate biases in machine learning models. Xue et al. [27] introduce Bias-based Weight Masking Fine-Tuning (BMFT), a post-processing method. BMFT first generates a mask to identify model parameters (weights) that contribute most significantly to biased predictions concerning sensitive attributes. It then employs a two-step fine-tuning strategy: initially fine-tuning the feature extractor on these identified bias-influenced weights to debias it, followed by fine-tuning a reinitialized classification layer to maintain discriminative performance on a small, group-balanced external dataset.

These examples illustrate the versatility of weight masking. However, they differ fundamentally from the focus of this thesis. The aforementioned methods employ masking for objectives such as adversarial defense, targeted data forgetting, or post-hoc bias mitigation, often involving different mask generation strategies or application scopes (e.g., post-processing). In contrast, this thesis investigates dynamic structured weight masking applied consistently during the training phase of DNNs, with the primary goal of improving training efficiency by reducing global memory accesses and, secondarily, computational cost.

3.1.2 Dynamic Sparse Training (DST) Methodologies

Dynamic Sparse Training (DST) encompasses a class of techniques that adaptively adjust a neural network's sparsity pattern throughout the training process. Unlike static pruning methods that determine sparsity before or after training, DST allows for the exploration of different sparse subnetworks during training, potentially leading to better performance-efficiency trade-offs by allowing connections to be pruned and regrown based on evolving criteria of importance [10]. This adaptability can be particularly beneficial as the significance of weights and connections can change during the learning process.

Several distinct approaches to DST have been proposed in the literature:

MAGNITUDE-BASED DST WITH CONNECTION REGROWTH: Many DST methods, including the one explored in this thesis, rely on weight magnitudes to prune less salient connections and often incorporate mechanisms to regrow connections. For example, RigL (Rigging the Lottery) [5] periodically prunes a fraction of weights with the smallest magnitudes and regrows the same number of connections by activating weights with the largest gradient magnitudes. Sparse Evolutionary Training (SET) [15] also prunes small-magnitude weights but regrows

connections randomly. These methods aim to find well-performing sparse subnetworks from scratch or during training.

STRUCTURED SPARSE-TO-SPARSE TRAINING: Building upon DST principles, Structured RigL (SRigL) [13] extends the RigL methodology to learn fine-grained structured N:M sparsity patterns. SRigL imposes a constant fan-in constraint (a specific case of N:M sparsity) and dynamically updates the sparse connectivity by pruning small-magnitude weights and regrowing connections based on gradient magnitudes, while maintaining the structural constraint. It can also incorporate neuron ablation to further improve performance at very high sparsities by effectively reducing layer width. This approach aims to achieve the benefits of DST while producing hardware-friendly sparse patterns.

SPARSIFYING ACTIVATIONS AND GRADIENTS: Other DST techniques focus on sparsifying not only weights but also activations or gradients. Dynamic Forward and Backward Sparse Training (DFBST) [19] proposes to dynamically sparsify both the forward pass (weights/activations) and the backward pass (gradients) using separate trainable masks. These masks are generated using trainable thresholds, and for the backward pass, a variance penalty term is included in the loss function to guide the learning of gradient thresholds. The goal is to achieve a completely sparse training schedule, reducing computational overhead in both passes.

DYNAMIC FILTER PRUNING DURING TRAINING: Instead of pruning individual weights, some methods dynamically prune entire filters (or channels) in convolutional neural networks during the training process. Roy et al. [21] propose a "pruning while training" strategy where filters are gradually pruned based on criteria like L1-norm magnitude at regular intervals throughout training. This avoids a separate retraining phase and allows the network to adapt to the reduced filter set as training progresses, aiming to minimize accuracy loss while achieving model compression.

SPARSE WEIGHT ACTIVATION TRAINING (SWAT): Another notable approach is Sparse Weight Activation Training (SWAT) introduced by Raihan and Aamodt [20]. SWAT aims to reduce training time and computational FLOPs by sparsifying weights in the forward pass, and both weights and activations during the computation of gradients in the backward pass. A distinguishing feature of SWAT is its update mechanism: it generates dense weight gradients which are then used to update all network weights, including those that were temporarily masked in the forward pass. This strategy facilitates dynamic exploration of the network topology during training. SWAT

can be applied to induce both unstructured and structured sparsity, such as channel pruning.

The dynamic structured weight masking approach employed in this thesis shares similarities with magnitude-based DST in its use of weight magnitudes (specifically, the Frobenius norm of blocks) for selection. However, it differs significantly from the aforementioned methods. Its primary distinction lies in its focus on creating block-structured sparsity via top-k selection of entire blocks of weights, with the main objective of reducing global memory accesses during training. This contrasts with SRigL's N:M patterns, DFBST's focus on trainable thresholds for both forward and backward pass sparsity, the filter-level granularity of dynamic filter pruning, and SWAT's emphasis on FLOP reduction and its specific gradient update strategy. While all aim for efficiency, the specific sparsity structure and the targeted benefit (memory access patterns for blocks in this thesis) differentiate the approach.

3.2 STRUCTURED SPARSITY AND MEMORY EFFICIENCY IN DNN TRAINING

Achieving true performance gains from sparsity often necessitates structured approaches, as unstructured sparsity can be challenging to accelerate effectively on parallel hardware like GPUs. This section reviews advancements in structured sparsity and techniques for improving memory efficiency during DNN training, focusing on key architectural patterns and their implications for hardware acceleration.

3.2.1 Advancements in Structured Sparsity

Structured sparsity refers to the organization of sparsity in predefined, regular patterns, such as blocks or N:M configurations, which can better align with hardware capabilities compared to arbitrary, unstructured sparsity [10]. This regularity is intended to facilitate more efficient memory access and computation on parallel processors like GPUs.

Key approaches in structured sparsity include:

- **Block Sparsity:** This involves partitioning weight matrices into blocks and treating entire blocks as units for pruning or masking. The method explored in this thesis is a form of dynamic block sparsity. The rationale is that operating on contiguous blocks can be more hardware-friendly than managing individual sparse weights.
- **N:M Fine-Grained Structured Sparsity:** This pattern mandates that within a small, contiguous group of M weights, a specific

number N must be non-zero. For instance, NVIDIA's Ampere architecture introduced support for a 2:4 sparsity pattern, where two out of every four weights can be pruned, allowing for specialized hardware acceleration [18].

Recent research has also focused on dynamically learning structured patterns during training. For example, Structured RigL (SRigL) [13] adapts dynamic sparse training principles to learn N:M structured patterns by iteratively adjusting connections while maintaining the structural constraint. Efficiency improvements have also been sought by targeting specific parts of the training pipeline; for instance, by using structured formats like Block Sparse Compressed Row (BSR) for compressing data, such as activations, during the backward pass to reduce memory overhead [2].

This thesis contributes by analyzing dynamic top-k block masking applied to ResMLP weights, evaluating its impact on accuracy and theoretical efficiency metrics.

3.2.2 Techniques for Memory-Efficient Training

The substantial memory requirements for training large DNNs, stemming from weights, activations, and optimizer states, present a significant bottleneck [10]. Several strategies aim to alleviate this:

- **Compression of Model Components:**
  - **Weight Compression:** Techniques such as NeuZip [7] aim to compress the model weights themselves by exploiting the statistical properties and entropy of their floating-point representations, thereby reducing storage and potentially data transfer costs.
  - **Activation Compression:** Similarly, methods have been developed to compress activation maps, particularly those stored for gradient computation in the backward pass, to reduce their significant memory footprint during training [3].
- **Sparsification Techniques:** As discussed, introducing sparsity through pruning or masking (whether structured or unstructured) inherently reduces the number of active parameters. While the focus of this thesis is on dynamic structured weight masking for its potential impact on memory access patterns related to weights, various sparsity approaches contribute to reducing the overall model complexity [10].

The dynamic structured weight masking explored in this thesis is presented as a strategy that complements these approaches by specifically targeting the memory accesses related to weight tensors during

the computational passes of training. It aims to reduce the amount of weight data that effectively participates in computation by dynamically selecting active blocks of weights.

3.3 ADVANCEMENTS IN MLP-BASED ARCHITECTURES

3.3.1 The Resurgence and Evolution of MLP Architectures

Recent advancements in computer vision have seen a renewed exploration of architectures built primarily from Multi-Layer Perceptrons (MLPs), moving beyond traditional convolutional designs. The ResMLP architecture, introduced by Touvron et al. [25], exemplifies this trend. It is designed as an architecture built entirely upon MLPs for image classification, aiming for simplicity and encoding little prior information about images.

The ResMLP model processes image patches through an initial linear projection followed by a sequence of residual operations. These operations consist of: (i) a cross-patch linear layer where image patches interact, applied independently and identically across channels, and (ii) a two-layer feed-forward network (a single-layer MLP in terms of hidden layers) where channels interact independently per patch [25]. This structure is inspired by Vision Transformers (ViTs) but introduces key simplifications. Notably, ResMLP replaces the self-attention sublayer of ViTs with a linear layer for cross-patch communication. According to Touvron et al. [25], this modification contributes to more stable training compared to ViTs under similar training schemes, allowing for the removal of batch-specific or cross-channel normalizations like BatchNorm or LayerNorm in favor of simpler affine transformations.

The authors demonstrate that ResMLP, when trained with modern strategies including heavy data augmentation and optional distillation, can achieve a surprisingly good accuracy/complexity trade-off on ImageNet, even when trained from scratch without pre-training on larger datasets, and without requiring complex normalization schemes [25]. The paper also notes compatibility with self-supervised learning and potential for adaptation to other domains like machine translation. The architectural simplicity of ResMLP, particularly its reliance on linear layers and basic MLP blocks for its core operations, makes it a pertinent choice for investigating efficiency enhancements such as the dynamic structured weight masking explored in this thesis.

3.3.2 Sparsely Activated Models: Mixture-of-Experts (MoE) in MLPs

A significant direction in enhancing the capacity and efficiency of large neural networks, including MLP-based architectures, involves the use of sparsely activated components, notably through the Mixture-of-Experts (MoE) paradigm. In general, an MoE layer consists of multiple "expert" sub-networks and a "gating" network or router that dynamically selects a small subset of these experts to process each input, allowing for increased model capacity without a proportional rise in computational cost per example.

Recent work by Yu et al. [28] has specifically explored sparsely activated all-MLP architectures for Natural Language Processing (NLP). They propose a "sparse all-MLP" (sMLP) where dense blocks in an existing all-MLP model (gMLP) are replaced with sparse MoE blocks. This approach aims to improve the expressiveness of the all-MLP architecture while keeping the compute cost constant. According to Yu et al. [28], their sMLP applies sparsity to two fundamental operations:

- **Hidden Dimension Operation (Feed-Forward Layers):** MoE structures are adopted for the feed-forward layers, drawing parallels with how MoEs are used in some Transformer-based models.
- **Token-Mixing Operations:** For token mixing, they design a new MoE module (sMoE) which processes chunks of hidden representations through different experts, with each expert performing a spatial projection.

The authors report that this sparse all-MLP with MoEs can improve language modeling perplexity and training efficiency compared to dense counterparts and some Transformer-based MoE models [28].

The sparsely activated expert paradigm, as exemplified by the sMLP in [28], shares a conceptual similarity with the dynamic weight masking explored in this thesis: both result in only a subset of the model's parameters being computationally active for a given forward pass. However, the underlying mechanisms and primary objectives differ. MoE architectures, like sMLP, are explicitly designed with distinct expert modules and a routing mechanism (which Yu et al. [28] discuss with deterministic and partial prediction strategies) to decide which expert(s) to activate. Sparsity is inherent in this architectural choice of activating entire expert sub-networks. In contrast, the dynamic structured weight masking investigated in this thesis is applied to an existing, typically dense, architecture (ResMLP). It does not involve separate expert modules or a learned input-dependent router; instead, it dynamically deactivates blocks of weights within the standard layers based on a global, magnitude-based heuristic (Frobenius norm and top-k selection). While both lead to a form of conditional computation,

MoEs as described by Yu et al. [28] primarily aim to increase model capacity and expressiveness efficiently, whereas the block masking in this work primarily targets the reduction of memory movement and computational cost for a given base architecture by making its existing layers dynamically sparse during training.

3.3.3 Situating this Thesis's Approach to MLP Efficiency

The renewed interest in MLP-based architectures, such as ResMLP [25], and efforts to enhance their efficiency and capacity, for instance through sparsely activated Mixture-of-Experts as explored by Yu et al. [28], form an important context for this thesis. While MoE approaches introduce sparsity by design through explicit expert sub-networks and learned or deterministic routing mechanisms to scale model capacity with controlled computational cost, the work undertaken in this thesis explores a different paradigm for improving the training efficiency of MLP-based models.

This research focuses specifically on the ResMLP architecture and investigates the application of dynamic structured weight masking directly to its existing, nominally dense layers. The primary goal is not to increase model capacity via sparsely activated experts, but rather to reduce the operational burden during the training of a given ResMLP model, particularly concerning global memory accesses and, secondarily, computational FLOPs. The mechanism employed—dynamic selection and masking of entire blocks of weights based on their magnitude (Frobenius norm)—differs from the input-dependent learned routing or structured expert selection characteristic of MoE systems. Therefore, this thesis contributes to the broader efforts in MLP efficiency by examining a distinct method of inducing and leveraging dynamic, structured sparsity within the layers of an established MLP architecture during its training phase.

4 METHODOLOGY

4.1 METHODOLOGICAL APPROACH

This research employs an experimental, quantitative methodology to explore the impact of dynamic structured weight masking on the performance of ResMLP networks for image classification tasks, with a primary focus on improving hardware efficiency by reducing memory traffic. Specifically, the study addresses the following question: **How do varying block sizes and masking ratios in dynamic structured weight masking, applied during ResMLP training, impact classification accuracy and theoretical memory movement?** The aim is to theoretically estimate the potential reduction in global memory movements achieved by this masking mechanism while ensuring that classification accuracy does not deteriorate significantly. This methodology was applied to experiments primarily on CIFAR-10. Additionally, an initial set of experiments on the ImageNet dataset was conducted to observe the approach's characteristics under more demanding conditions. A full optimization of the approach for ImageNet was beyond the scope of this work. The core masking, measurement, and analysis methodologies described herein were applied consistently to both the CIFAR-10 and the ImageNet experiments.

To investigate this, a weight masking mechanism is integrated into the ResMLP architecture. Fixed masking parameters—namely, block sizes and mask ratios—are established for the duration of training, while the weight mask is dynamically applied via a top-k selection process during each forward pass. In practice, the weight masking forms an iterative loop within the training process, where the mask is recalculated and applied at every iteration. This design allows for a systematic analysis of the trade-offs between potential memory movement reduction and maintained accuracy by isolating the effects of varying block sizes and mask ratios within this dynamic, per-iteration masking framework.

The overall methodological workflow, illustrated conceptually in Figure 4.1, commences with data preprocessing. Subsequently, the core training process is executed, employing the ResMLP architecture integrated with the dynamic structured weight masking mechanism. This masking is applied iteratively throughout the training phase. The workflow concludes with an evaluation stage, where key performance metrics, including classification accuracy and theoretical memory movement, are systematically recorded and analyzed.

This work utilizes an existing ResMLP codebase. Key model parameters were adapted to operate on the CIFAR-10 dataset, which served as the primary benchmark. This dataset was selected not only for its relevance to image classification but also because its faster training times allow for more detailed measurements and a comprehensive exploration of the parameter space within a shorter timeframe. For the preliminary exploration on ImageNet, specific adaptations to the model and training procedure were also made, as detailed in subsequent sections. For CIFAR-10, the unmasked network was first tuned to serve as a baseline, achieving an accuracy exceeding 80% over 200 training epochs. This baseline performance provides a reference for evaluating the impact of the weight masking approach on both accuracy and theoretical memory movement for this dataset.

This work makes several important contributions:

- It provides a novel empirical evaluation of dynamic structured weight masking on a ResMLP network—an approach that has been primarily explored in other architectures—thereby offering fresh insights into its effectiveness on a purely MLP-based model for image classification, particularly concerning potential hardware efficiency gains.
- By investigating the interplay between block size, mask ratio, and accuracy, the research deepens our understanding of the efficiency-accuracy trade-offs inherent to dynamic structured masking.
- The systematic analysis of various block sizes and mask ratios contributes to determining the optimal ranges for these hyperparameters, thereby balancing computational efficiency with robust model performance.

Overall, this methodological framework, as depicted in Figure 4.1, facilitates a thorough investigation into how structured weight masking—applied dynamically as part of the training process—can potentially reduce global memory accesses, a key factor in hardware efficiency, without significantly compromising classification performance.

4.2 DATASET AND PREPROCESSING

This study primarily utilizes the CIFAR-10 dataset for a comprehensive evaluation of the masking techniques. Additionally, the ImageNet dataset was employed for preliminary scalability exploration.

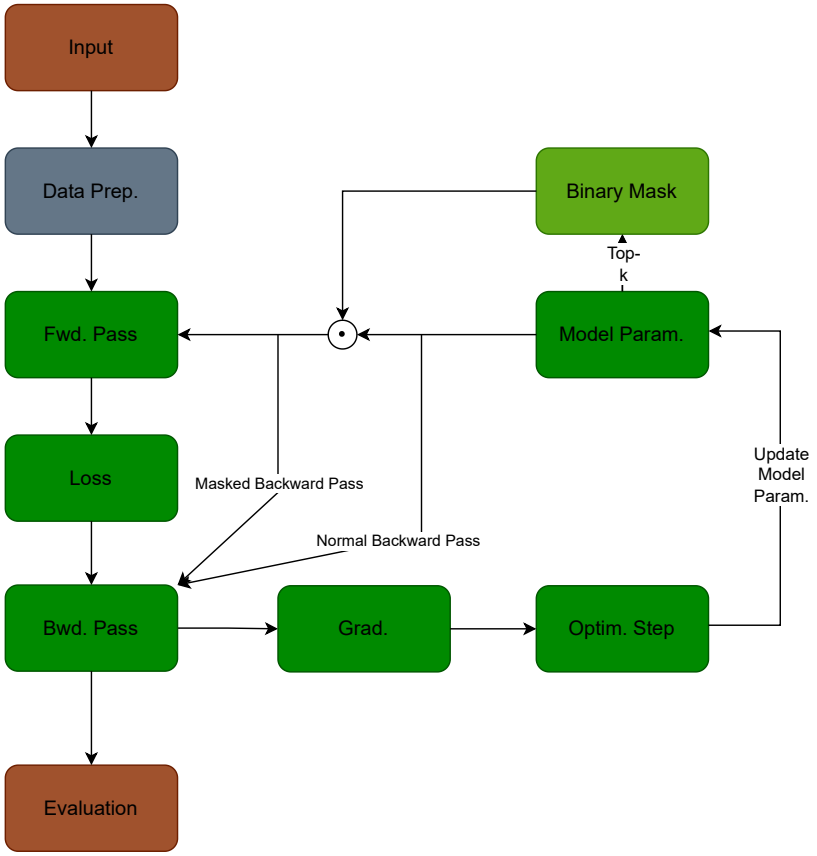


Figure 4.1: Conceptual overview of the training loop incorporating dynamic structured weight masking, highlighting the two alternative backward pass configurations investigated. The standard training steps include data preparation, forward pass, loss calculation, gradient computation (backward pass), and parameter optimization. Model parameters are used to dynamically generate a binary mask (via top-k selection of blocks based on their norm), which is then applied to the weights during the forward pass (indicated by the circled dot, representing element-wise multiplication). The subsequent backward pass for gradient computation is explored under two distinct, alternative configurations: (1) a **masked backward pass**, where gradients are computed considering the applied mask (arrow labeled "Masked Backward Pass"), or (2) an **unmasked backward pass**, where gradients are computed with respect to the original, full weights (arrow labeled "Normal Backward Pass"). Only one of these backward pass configurations is active in any given experiment. The optimization step then updates the original model parameters.

4.2.1 CIFAR-10 Dataset

The CIFAR-10 dataset served as the primary benchmark for this research. For training, it was prepared using a custom data loader, and the following specific preprocessing steps were applied:

4.2.1.1 Normalization for CIFAR-10

To ensure stable and efficient training, the pixel values of the CIFAR-10 images are normalized by subtracting the channel-wise mean and dividing by the channel-wise standard deviation. The normalization parameters used are the standard values for CIFAR-10 as provided by the PyTorch library:

- **Mean:** (0.4914, 0.4822, 0.4465) (for Red, Green, and Blue channels, respectively)
- **Standard Deviation:** (0.2470, 0.2435, 0.2616) (for Red, Green, and Blue channels, respectively)

4.2.1.2 Data Augmentation for CIFAR-10

To enhance the model's generalization ability and robustness on CIFAR-10, several data augmentation techniques are employed on the training set:

- **Random Horizontal Flipping:** Images are randomly flipped horizontally with a probability of 0.5.
- **Random Cropping:** Random crops of the resized images are taken to promote robustness to variations in object position and scale.
- **Mixup:** New synthetic training samples are generated by taking a weighted average of two randomly selected images and their corresponding labels, which encourages linear behavior between classes and reduces overfitting.
- **CutMix:** This technique replaces a random portion of an image with a patch from another image, with target labels adjusted proportionally to the area of the patch.

4.2.2 ImageNet Dataset for Scalability Exploration

For preliminary experiments to assess the scalability of the proposed masking approach, a large-scale ImageNet dataset comprising approximately 14 million training images across 1,000 object categories was utilized, along with a standard validation set. Input images for these experiments had dimensions of 224×224 pixels. Preprocessing steps, including channel-wise normalization and data augmentation (such as

random resized cropping and horizontal flipping), were applied in a manner conceptually similar to those used for CIFAR-10, but adapted to suit the characteristics and scale of the ImageNet dataset.

4.3 STRUCTURED WEIGHT MASKING

4.3.1 Mask Application

To evaluate the impact of structured sparsity on the ResMLP model, a weight masking technique is employed that selectively masks less important blocks of weights based on their magnitude. In this approach, the mask generation process uses a magnitude-based method where the Frobenius norm serves as the measure of a block's importance.

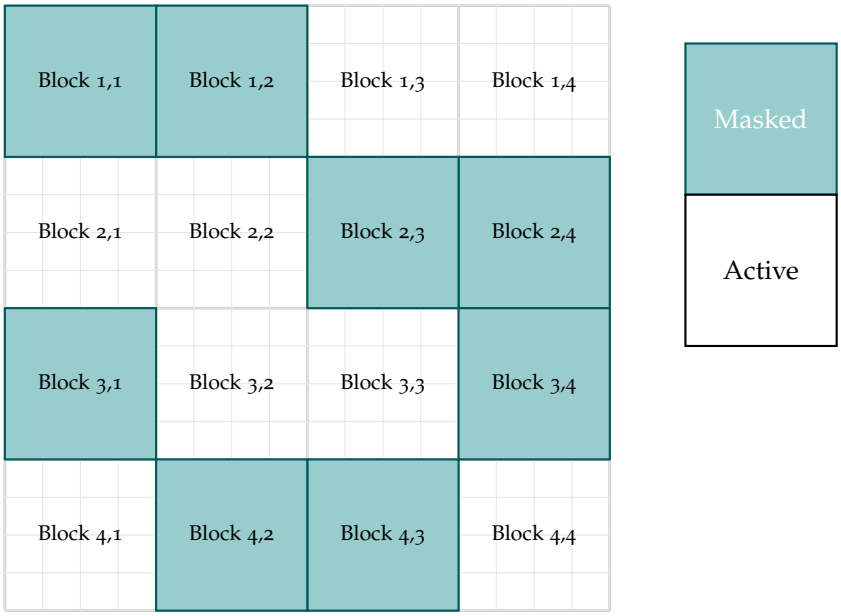
For each target weight tensor, the weights are partitioned into non-overlapping blocks (groups). Within each group, the Frobenius norm is computed. Based on the predetermined mask ratio, a top-k selection strategy is applied: only the weight groups (blocks) with the highest Frobenius norms are retained, while the weights within the remaining blocks are effectively set to zero by the mask. This binary mask, which matches the shape of the weight tensor, is generated dynamically during every forward pass.

During the forward pass, the binary mask is applied via element-wise multiplication, effectively nullifying the contributions of the weights within the masked blocks. Although this masking does not reduce memory accesses in the current implementation (since all weights are loaded from memory), it facilitates an evaluation of the model's performance with these weights computationally ignored, while the potential memory movement reduction is later calculated theoretically.

Figure 4.2 provides an illustrative example of the masking process. In this visualization, a 16×16 neural network weight matrix is partitioned into 4×4 blocks. A sparsity ratio of 50% is enforced via a top-k selection strategy based on the L2 norm, where the highest scoring blocks are retained and the remaining blocks are masked (displayed in teal).

4.3.2 Backward Pass with and without Masking

To better understand the role of masking during gradient propagation, two configurations were examined. In the first configuration, the binary weight mask generated and applied during the forward pass is also considered during the backward pass; consequently, weights that were masked (effectively treated as zero) in the forward pass do not contribute to the gradient computation for their own updates, and their values are not updated. In the second configuration, the



16×16 Weight Matrix with 4×4 Blocks and 50% Sparisty

Figure 4.2: 16×16 neural network weight matrix partitioned into 4×4 blocks. A sparsity ratio of 50% is enforced via a top-k selection strategy, where the highest scoring blocks are retained and the remaining blocks are masked (teal), facilitating hardware-efficient pruning while maintaining model performance.

mask was applied only during the forward pass, thereby allowing the gradients to be propagated through the full weight tensor, and all weights (including those masked in the forward pass) could receive updates during backpropagation.

This experimental setup was designed to enable a systematic analysis of how the application of the mask during the backward pass affected gradient computation and overall training dynamics. By comparing the two approaches, it was determined whether the treatment of masked weights during gradient computation influenced the model's convergence, stability, and final performance, or if allowing full gradient propagation for all weights yielded different results.

4.4 MEASUREMENT AND ANALYSIS STRATEGY

A comprehensive strategy was employed to evaluate the impact of the dynamic block-wise masking technique on both estimated computational resources and the performance of the trained ResMLP models. The evaluation encompassed several key aspects:

4.4.1 Computational Cost Estimation

The theoretical computational cost, measured in Floating Point Operations (FLOPs), was estimated for the trained models under various masking conditions. The estimation process began by calculating the FLOPs required for a standard, dense forward pass of the ResMLP architecture. This baseline value was then refined by adjusting the FLOP count on a layer-wise basis, specifically for the linear layers subject to masking. The adjustment accounted for the applied mask ratio for each configuration, providing an estimate of the effective FLOPs utilized during sparse training. The total theoretical computational cost over the entire training duration was subsequently estimated by scaling the per-epoch FLOPs accordingly.

4.4.2 Model Accuracy Evaluation

The primary metric for model performance was Top-1 classification accuracy, evaluated on the validation dataset at the end of each training epoch. To assess the influence of applying the mask during gradient computation and weight updates, two distinct experimental conditions were compared: one where the mask was applied during the backward pass, and one where it was not. The final validation accuracy for both conditions provides insight into the stability and effectiveness of the masking approach under different gradient handling strategies.

4.4.3 Masking Pattern Analysis

To gain qualitative insights into the behavior of the dynamic masking mechanism, visualizations of the masking patterns were generated. These visualizations, typically presented as heatmaps, illustrate the cumulative activity (i.e., the frequency of being non-masked) of weight blocks within the network's key linear layers throughout the training process. Analyzing these patterns helps in understanding the stability and evolution of the learned sparsity structures under different experimental conditions.

4.4.4 Theoretical Memory Movement Estimation

A theoretical analysis is performed to estimate the potential reduction in global memory movement attributable to dynamic structured weight masking during training. This estimation serves as an indicator of potential hardware efficiency gains by quantifying the volume of weight data theoretically accessed from global memory.

The underlying **memory model** for this estimation is built upon several key considerations. It accounts for the specific layer dimensions (input features, output features) and the total number of weights within each linear layer of the ResMLP model. The model also incorporates the masking configuration, namely the dimensions of the blocks used for partitioning weight matrices and the target mask ratio applied to these blocks. A fundamental hardware-related parameter influencing data transfer, the memory cache line size of the target GPU architecture, is a critical component of this model, as a cache line represents the smallest unit of data transferable between main memory and the GPU's cache. The estimation operates on the assumption that for each non-masked block of weights, all cache lines that this block occupies, even partially, must be loaded from global memory. Furthermore, a **worst-case alignment** scenario is assumed for these active (non-masked) blocks relative to cache line boundaries. This implies that even if only a small portion of a cache line is needed for an active block, the entire cache line is counted as accessed; similarly, if an active block spans multiple cache lines, all those cache lines are considered loaded. This conservative assumption provides a theoretical upper bound on the memory traffic generated by accessing the necessary weight elements.

By modeling these factors, the analysis calculates the total data volume corresponding to the cache lines occupied by all non-masked blocks across all relevant layers for each forward pass. This per-pass estimate is then aggregated over the total number of training iterations to provide an overall estimate of the total theoretical memory movement for weights under different masking configurations. This theoretical value helps in understanding the potential upper limits of

memory access savings before considering more complex run-time factors such as actual cache hit rates or specific memory controller behaviors.

4.4.5 Extraction of Key Performance Indicators

To facilitate quantitative comparisons across different masking configurations, key performance indicators were extracted post-training. These included the total number of computationally active (non-masked) weight elements aggregated over the entire training duration, providing a measure of the actual computational load reduction. Additionally, the final Top-1 validation accuracy achieved under each experimental condition (varying block sizes, mask ratios, and backward pass treatments) was systematically recorded for comparative analysis.

5 IMPLEMENTATION

5.1 INTRODUCTION

A detailed account of the technical implementation of the dynamic structured weight masking approach and the associated measurement tools described in the methodology is provided in this chapter. The primary aim is to present how dynamic masking is integrated within the ResMLP architecture and to explain the design of the profiling and visualization tools used to evaluate computational efficiency and model performance. In doing so, it is ensured that the experimental setup is both reproducible and clearly documented.

The chapter is organized to reflect the experimental workflow described in the methodology. Initially, the modifications made to the ResMLP model for the CIFAR-10 dataset are outlined, including adjustments to patch sizes, embedding dimensions, and training parameters. The dynamic structured weight masking mechanism is then described, with a focus on the generation and application of masks based on the Frobenius norm and a top-k selection strategy. Subsequent sections detail the measurement and profiling techniques employed to compute FLOPs, analyze memory movement, and evaluate model accuracy, as well as the data visualization strategies and experimental procedures, including hyperparameter sweeps and reproducibility measures.

5.1.1 Codebase Overview

The experimental framework for this thesis is built upon a modular Python codebase, which originates from and is maintained by the HAWAII Lab at ZITI, Heidelberg University. This codebase, designed for clarity and reproducibility, was adapted and extended for the specific investigations undertaken in this work. The core functionality is implemented in a set of well-organized scripts. For instance, the main training loop and model evaluation routines reside in `training.py`, while the profiling functions for computing FLOPs and monitoring memory usage are located in `measurement.py` and `memory_calculation.py`. Post-processing tasks, such as calculating the number of loaded elements and extracting final accuracy, are handled by `loaded_values.py` and `masked_accuracy.py`, respectively. Configuration parameters—including model settings, training hyperparameters, and grid search configurations—are maintained in YAML

files, such as `grid_cifar10.yaml`, located within the project's main directory.

Figure 5.1 on page 56 illustrates the directory structure of the project. This file tree delineates the hierarchical organization of the codebase, including key directories and scripts necessary for implementing the dynamic structured weight masking approach. Notably, a dedicated `masked` directory was created to contain all files that were modified for the masking experiments.

Listing 5.1 provides an example of such a YAML configuration file. This file details the hyperparameters, model settings, dataset parameters, and grid search configurations used in conjunction with the `LaunchPad` library to automatically generate SLURM job submission scripts for the experiments, ensuring the setup is transparent and reproducible.

```yaml
hp:
  model_config:
    - "\"ResMLP5 @ XChannel = masked 0.9 (1, 1); XPatch = masked 0.9 (1, 1);\""
    - "\"ResMLP5 @ XChannel = masked 0.9 (1, 2); XPatch = masked 0.9 (1, 2);\""
    - "\"ResMLP5 @ XChannel = masked 0.9 (1, 4); XPatch = masked 0.9 (1, 4);\""
    - "\"ResMLP5 @ XChannel = masked 0.9 (1, 8); XPatch = masked 0.9 (1, 8);\""
    - "\"ResMLP5 @ XChannel = masked 0.9 (16, 16); XPatch = masked 0.9 (16, 16);\""
    - "\"ResMLP5 @ XChannel = masked 0.9 (32, 32); XPatch = masked 0.9 (32, 32);\""
    - "\"ResMLP5 @ XChannel = masked 0.9 (64, 64); XPatch = masked 0.9 (64, 64);\""

meta:
  run: slurm
  gpus: 1
  mode: grid
  prefix: block_test_masked_backprop_masked_ratio_09
  repeat: 1
  sandbox: /csghome/tr312/Project/Run_model_master_mask
  script: "python /csghome/tr312/Project/Run_model_master_mask/training.py"

fixed:
  data_dir: /csghome/tr312/Project/data
  data_set: CIFAR10
  batch_size: 128
  num_classes: 10
  epochs: 200
  image_size: 32
  weight_decay: 0.0001
  lr: 0.01
  warmup_lr: 0.001
  opt: Adam
  patch_size: 4
  embedding_dim: 64
```

```
    mixup: 0.2
    cutmix: 0.8
    mixup_prob: 1
    pretrained_classes: 10

sbatch:
    partition: rivulet
    gpus: 1
    cpus-per-task: 8
    mem: 16G
    export: ALL
    time: "7-00:00:00"
    chdir: /csghome/tr312/Project/Run_model_master_mask
```

Listing 5.1: Training configuration for ResMLP experiments. This YAML file specifies the hyperparameters, model configurations, dataset parameters, and grid settings used in the experiments, ensuring consistency and reproducibility throughout the training process.

5.2 RESMLP MODEL MODIFICATIONS

5.2.1 Adaptation for CIFAR-10 and ImageNet

Modifications were made to adapt the original ResMLP model (typically designed for ImageNet-scale tasks) for the datasets used in this work. For the CIFAR-10 dataset, key parameters such as the patch size, embedding dimension, and network depth were adjusted to accommodate the smaller 32×32 images, as detailed in Table 5.1. Similarly, for the preliminary experiments on ImageNet, the ResMLP model parameters were configured to suit the larger dataset scale and complexity, with these specific settings outlined in Table 5.2. In order to manage and utilize the diverse training configurations defined in the YAML file for both datasets, a set of Python scripts was employed to parse these settings and instantiate the corresponding ResMLP models.

In the `helper.py` script, dataclasses define the different types of architectural modifications. The `BlockMasking` dataclass, central to this work, was added to accommodate the masked configurations defined in the YAML file. The corresponding implementation is shown in Listing 5.2.

```python
from dataclasses import dataclass, fields, asdict
from typing import Tuple, Optional

@dataclass
class BlockMasking:
    mask_ratio: float = 0.0
    block_size: Tuple[int, int] = (1, 1)

    def keys(self):
        return (f.name for f in fields(self))

```

```
        def __getitem__(self, item):
            return getattr(self, item)

        def as_dict(self):
            return asdict(self)
```

Listing 5.2: Dataclass for representing block-wise masking in the ResMLP model (helper.py).

Furthermore, the `factory.py` script contains the core logic for parsing model configuration strings from the YAML file. The relevant parts for handling the "masked" configuration are presented in Listing 5.3.

```
    import re
    from collections import defaultdict
    from typing import Tuple, Mapping, Any, Optional

    from .helper import BlockMasking

    # Regex patterns for model and layer configurations
    layer_pattern = r"""
        ([A-Za-z]+)(?:(\d+))?\s*=\s*
        (?:
            sparse\s+(-?\d+\.\d+)\s+(\(\s*-?\d+,\s*-?\d+\s*\))
        |
            compressed\s+(\(\s*-?\d+,\s*-?\d+\s*\))
            (?:\s+(\(\s*-?\d+,\s*-?\d+\s*\)))?
            (?:\s+(\w+))?
            (?:\s+(\w+))?
            (?:\s+(-?\d+\.\d+))?
        |
            masked\s+(-?\d+\.\d+)\s+(\(\s*-?\d+,\s*-?\d+\s*\))
        )
        \s*;
    """

    def _parse_model_configuration_string(
        config: str,
    ) -> Tuple[str, int, Mapping[str, Any]]:
        # ... (rest of the function until the layer parsing loop) ...

        # Parse each layer configuration
        for layer in layers:
            try:
                # ... (other parsing logic) ...
                mask_ratio = float(layer[9]) if layer[9] else None
                mask_block_size = tuple(map(int, layer[10].strip("()").
                    split(","))) if layer[10] else None

                # Create the appropriate block object based on the parsed
                    data
                # ... (other conditions) ...
                elif mask_ratio is not None:
                    current = BlockMasking(mask_ratio=mask_ratio,
                        block_size=mask_block_size)
                # ... (rest of the loop and function) ...
```

Listing 5.3: Relevant parts of the model factory for parsing masked configurations (factory.py).

In addition, fixed configuration settings in the training YAML file were adjusted to specify the general parameters for running ResMLP on the CIFAR-10 dataset instead of ImageNet, as shown in Listing 5.4. Specific model parameters for each dataset are detailed in their respective configuration tables.

```
fixed:
  data_dir: /csghome/tr312/Project/data
  data_set: CIFAR10
  batch_size: 128
  num_classes: 10
  epochs: 200
  image_size: 32
  weight_decay: 0.0001
  lr: 0.01
  warmup_lr: 0.001
  opt: Adam
  patch_size: 4
  embedding_dim: 64
  mixup: 0.2
  cutmix: 0.8
  mixup_prob: 1
  pretrained_classes: 10
```

Listing 5.4: Fixed configuration settings in the training YAML file that specify the parameters for running ResMLP on the CIFAR-10 dataset instead of ImageNet.

5.2.2 Model Configuration Details

The ResMLP model architecture, including its depth and any applied masking modifications, is defined via a configuration string provided through the `model_config` parameter within the YAML configuration files (see Listing 5.1 for an example). For instance, a configuration string such as

```
"ResMLP12 @ XPatch=masked 0.5 (2, 2); XChannel=masked
0.3 (1, 1);"
```

specifies that the base model is a ResMLP with a depth of 12 layers. It further defines a masking ratio of 0.5 applied to 2×2 blocks in the XPatch layers and a mask ratio of 0.3 applied to 1×1 blocks in the XChannel layers. Other key hyperparameters, such as embedding dimension and patch size, are set according to the specific dataset being used: parameters for the CIFAR-10 experiments are detailed in Table 5.1, and those for the ImageNet experiments are specified in Table 5.2.

Category	Parameters and Values
Model Parameters	Batch size: 128 Image size: 32 Patch size: 4 Embedding dimension: 64 Depth: 5 Number of classes: 10 Sparsity: 0.0 Sparsity at epoch: 10 Granularity: 64 Train/test: False
Learning Parameters	Epochs: 200 Weight decay: 0.0001 Decay rate: 0.1 Learning rate (lr): 0.01 Minimum lr: 1e-5 Warmup lr: 0.001 Warmup epochs: 5 Cooldown epochs: 10 Scheduler: cosine Optimizer: Adam Momentum: 0.005
Mixup Settings	Mixup: 0.2 Cutmix: 0.8 Cutmix min/max: None Mixup probability: 1.0 Mixup switch probability: 0.5 Mixup mode: batch Mixup off epoch: 0 Smoothing: 0.1

Table 5.1: Training Configuration Overview

5.2.3 Data Loading and Preprocessing

Data loading and preprocessing are managed using PyTorch's built-in dataset utilities, accessed via a dedicated `generate_data_loader` function within the codebase. The standard CIFAR-10 train-test split (50,000 training, 10,000 testing images) was utilized throughout the experiments. For ImageNet, the standard ILSVRC2012 dataset splits were

Category	Parameters and Values
Model Parameters	Batch size: 64 Image size: 224 Patch size: 14 Embedding dimension: 384 Depth: 12 Expansion factor: 4 Number of classes: 1000 Masking start epoch: 0 Train/test: False
Learning Parameters	Epochs: 300 Weight decay: 0.005 Decay rate: 0.1 Learning rate (lr): 0.001 Minimum lr: 1e-5 Warmup lr: 1e-5 Warmup epochs: 5 Cooldown epochs: 10 Scheduler: cosine Optimizer: AdamW Momentum: 0.005
Mixup Settings	Mixup: 0.8 Cutmix: 1.0 Cutmix min/max: None Mixup probability: 1.0 Mixup switch probability: 0.5 Mixup mode: batch Mixup off epoch: 0 Smoothing: 0.1

Table 5.2: Training Configuration Overview (Updated for ImageNet Run)

used. The overall data loading and preprocessing pipeline was largely inherited from the existing codebase. Minimal modifications were required, primarily in the adjustment of mixup and cutmix parameters, to ensure that the CIFAR-10 dataset was processed appropriately. For ImageNet, specific data loading and augmentation procedures common for this dataset were implemented. All other standard preprocessing steps—such as image resizing and normalization as detailed

in the Methodology chapter—were implemented as provided in the original framework or adapted as necessary for each dataset.

5.2.4 Data Loading and Preprocessing

Data loading and preprocessing are managed using PyTorch's built-in dataset utilities, accessed via a dedicated `generate_data_loader` function within the codebase. The standard CIFAR-10 train-test split (50,000 training, 10,000 testing images) was utilized throughout the experiments. The overall data loading and preprocessing pipeline was largely inherited from the existing codebase. Minimal modifications were required, primarily in the adjustment of mixup and cutmix parameters, to ensure that the CIFAR-10 dataset was processed appropriately. All other standard preprocessing steps—such as image resizing, normalization, and data augmentation as detailed in the Methodology chapter—were implemented as provided in the original framework.

5.3 STRUCTURED WEIGHT MASKING IMPLEMENTATION

The structured weight masking is implemented by generating a binary mask dynamically for each forward pass based on the current weight tensor values and the configuration specified in the YAML file (see Listing 5.1). The mask generation function resides in `prune.py` and is invoked for each weight tensor designated for masking.

5.3.1 Mask Generation

The binary mask is generated by the `get_prune_mask_block` function, which takes as input the weight tensor, the desired `mask_ratio`, the `block_size` (provided as a tuple, e.g., (2, 4) for the `XChannel` layer), and the `mask_order` (specifying the norm type). The parsing of the 'masked' configuration from the YAML file occurs in `factory.py` (see Listing 5.3).

The mask generation process involves the following steps:

BLOCK NORM CALCULATION (FROBENIUS NORM): For each block within the weight tensor, the Frobenius norm is computed using the `torch.linalg.norm` function. This calculation uses the specified order (typically 2, corresponding to the Frobenius norm for matrices)

and aggregates over the height and width dimensions of each block (`dim=(-2, -1)`). The Frobenius norm of a block $\mathbf{w}$ is computed as

$$\|\mathbf{w}\|_F = \sqrt{\sum_i \sum_j w_{ij}^2}, \tag{5.1}$$

which provides a measure of the magnitude of the block.

```
    norms = torch.linalg.norm(x_reshaped, ord=ord, dim=(-2, -1)).flatten
        ()
```

Listing 5.5: L2 norm calculation for block-wise masking (from prune.py).

TOP-k SELECTION: To achieve the desired `mask_ratio`, the top-k blocks are selected based on their computed Frobenius norms. The number of blocks to retain is determined by $k = \lceil \text{total\_blocks} \times (1 - \texttt{mask\_ratio}) \rceil$. The `torch.topk` function is used to obtain the threshold value corresponding to the k-th largest norm, and blocks with norms greater than or equal to this threshold are retained.

```
    k = ceil(total_blocks * (1 - mask_ratio))
    topk_threshold = torch.topk(norms, k=k, dim=-1, sorted=True).values.
        min()
    mask = norms >= topk_threshold
```

Listing 5.6: Top-k block selection for masking (from prune.py).

MASK CREATION: After selecting the top-k blocks, a binary mask is created where the elements corresponding to these blocks are set to 1 (active) and the remaining elements are set to 0 (masked). This binary mask is reshaped to match the original dimensions of the weight tensor by repeating the mask values for all elements within each block using `repeat_interleave`.

```
    mask = mask.view(blocks_y, blocks_x).repeat_interleave(block_size[1],
         dim=1).repeat_interleave(block_size[0], dim=0).float()
```

Listing 5.7: Binary mask creation and reshaping (from prune.py).

HANDLING BLOCK SIZES AND EDGE CASES: The `get_prune_mask_block` function ensures that the dimensions of the weight tensor are divisible by the specified `block_size`. Additionally, when `mask_ratio` is 0.0, a mask of all ones is returned, indicating that no weights are masked.

```
    import torch
    from typing import Tuple, Union
    from math import ceil

    @torch.no_grad()
    def get_prune_mask_block(
        x: torch.Tensor,
        mask_ratio: float,
```

```
        block_size: Tuple[int, int],
        ord: Union[int, str] = 2
    ) -> torch.Tensor:

        # Ensure input is 2D
        assert x.dim() == 2, f"Expected 2D tensor, got {x.dim()}D"

        # Check that both dimensions are divisible by block size
        assert (
            x.shape[0] % block_size[0] == 0 and x.shape[1] % block_size
                [1] == 0
        ), f"Shape {x.shape} incompatible with block size {block_size}"

        # No masking if mask_ratio is zero
        if mask_ratio == 0.0:
            return torch.ones_like(x)

        # Calculate number of blocks
        blocks_x = x.shape[1] // block_size[1]
        blocks_y = x.shape[0] // block_size[0]
        total_blocks = blocks_x * blocks_y

        # Calculate number of blocks to retain
        k = ceil(total_blocks * (1 - mask_ratio))

        # Divide tensor into blocks and calculate norms
        x_blocks_y = x.tensor_split(blocks_y, dim=0)
        x_reshaped = torch.stack(
            [torch.stack(b.tensor_split(blocks_x, dim=1), dim=0) for b in
                 x_blocks_y],
            dim=0
        )
        norms = torch.linalg.norm(x_reshaped, ord=ord, dim=(-2, -1)).
            flatten()

        # Select top-k blocks
        topk_threshold = torch.topk(norms, k=k, dim=-1, sorted=True).
            values.min()
        mask = norms >= topk_threshold
        mask = mask.view(blocks_y, blocks_x).repeat_interleave(block_size
            [1], dim=1).repeat_interleave(block_size[0], dim=0).float()

        return mask
```

Listing 5.8: Implementation of the block-wise mask generation function (from prune.py).

5.3.2 Mask Application

During the forward pass, the binary mask is applied to the weight tensor through element-wise multiplication. This operation effectively zeroes out the weights corresponding to the masked blocks, as demonstrated in the following code snippet.

```
    masked_weights = weight * weight_mask
```

```
    output = torch.matmul(input, masked_weights.mT)
```

Listing 5.9: Mask application in the forward pass (from linear.py).

5.3.3 Masking with and without Backward Pass

Two configurations for the backward pass are implemented:

BACKWARD PASS WITH MASKING: In this configuration, the mask is applied to the gradients during backpropagation. This means that the gradients corresponding to the masked weights are set to zero, preventing any updates to these weights during the optimization step. The implementation is shown in the following snippet:

```
    grad_input = torch.matmul(grad_output, weight * weight_mask)
    grad_weight = torch.matmul(grad_output.mT, input) * weight_mask
```

Listing 5.10: Backward pass with mask applied to gradients (from linear.py).

BACKWARD PASS WITHOUT MASKING: Alternatively, the mask may be applied only during the forward pass. In this case, gradients are computed with respect to the full weight tensor, and all weights are updated during optimization. The corresponding implementation is provided below:

```
    grad_input = torch.matmul(grad_output, weight)
    grad_weight = torch.matmul(grad_output.mT, input)
```

Listing 5.11: Backward pass without mask applied to gradients (from linear_umasked.py).

It was observed that training with the masked backward pass resulted in outcomes that differed significantly from those obtained when the backward pass was left unmasked. This observation indicates that the application of the mask during gradient computation plays a crucial role in guiding the learning process under masking constraints.
In the subsequent sections, the measurement, profiling, and visualization techniques used to assess the impact of this masking on computational efficiency and model performance will be described.

5.3.4 FLOPs Calculation

The theoretical FLOPs are computed using the `ptflops` library through the `Profiler` class in `measurement.py`. Initially, the dense FLOPs of the model are calculated, representing the computational cost without any masking, as demonstrated in Listing 5.12:

```
    macs, _ = get_model_complexity_info(
        self.model, tuple(self.sample_input.shape[1:]),
```

```
        as_strings=False, print_per_layer_stat=False
    )
    dense_total_flops = 2 * macs  # Dense FLOPs per epoch
```

Listing 5.12: Dense FLOPs calculation using ptflops (from measurement.py).

The dense FLOPs are then refined by iterating through the linear layers (such as `nn.Linear`, `MaskedLinear`, `SparseLinear`, and `CompressedLinear`) to compute their individual contributions, as shown in Listing 5.13:

```
    for module in self.model.modules():
        if isinstance(module, (nn.Linear, MaskedLinear, SparseLinear,
            CompressedLinear)):
            in_features = module.in_features
            out_features = module.out_features
            layer_dense_flops = 2 * in_features * out_features
            dense_linear_flops += layer_dense_flops
```

Listing 5.13: Dense FLOPs calculation for linear layers (from measurement.py).

To incorporate the effect of the applied masking configuration, the theoretical FLOPs for each masked linear layer are adjusted based on their respective target `mask_ratio` attributes. This adjustment is achieved in the calculation by multiplying the layer's dense FLOPs by $1 - \texttt{mask\_ratio}$, as illustrated in Listing 5.14. It is important to note that this calculation provides a theoretical estimate based on the target mask ratio. Due to the discrete nature of masking entire blocks, the actual number of active (non-masked) blocks, and thus the potentially achievable FLOP count, may exhibit step-like changes rather than varying strictly linearly with the target ratio.

```
    mask_ratio = getattr(module, "mask_ratio", 0.0)
    adjusted_linear_flops += layer_dense_flops * (1 - mask_ratio)
```

Listing 5.14: Adjusting FLOPs for sparsity (from measurement.py).

Subsequently, the total adjusted theoretical FLOPs for the model are computed by substituting the dense FLOPs of the linear layers with their adjusted counterparts (using the target mask ratio), as shown in Listing 5.15:

```
    adjusted_total_flops = dense_total_flops - dense_linear_flops +
        adjusted_linear_flops
```

Listing 5.15: Calculating total adjusted FLOPs (from measurement.py).

Finally, the per-epoch theoretical FLOPs are scaled by the total number of training epochs, and the results are formatted and displayed using the `log_flops` method, as demonstrated in Listing 5.16:

```
    def log_flops(self):
        """Display the FLOP count details over the entire training run.
            """
        total_dense = self.dense_flops_per_epoch * self.num_epochs
        total_adjusted = self.adjusted_flops_per_epoch * self.num_epochs
```

```
        total_saved = self.saved_flops_per_epoch * self.num_epochs
        dense_str = self._format_flops(total_dense)
        adjusted_str = self._format_flops(total_adjusted)
        saved_str = self._format_flops(total_saved)
        print("\nFLOP Count (Over Entire Training):")
        print("-" * 50)
        print(f"Total Dense FLOPs:      {dense_str}")
        print(f"Total Adjusted FLOPs: {adjusted_str}")
        print(f"Total Saved FLOPs:      {saved_str}")
        print("-" * 50 + "\n")
```

Listing 5.16: Logging FLOPs (from measurement.py).

5.3.5 Accuracy Evaluation

The validation accuracy is computed at the end of each epoch by the `validate` function in `training.py`. Both top-1 and top-5 accuracies are evaluated on the validation dataset. Additionally, the `masked_accuracy.py` script is used to extract the top-1 validation accuracy from the training log files at the final epoch (epoch 199, corresponding to the 200th epoch in a 0-indexed count) and to aggregate these values for further analysis. The core logic for accuracy extraction is presented in Listing 5.17:

```
import os
import re
import pandas as pd
import numpy as np

# Base directory
base_dir = '/csghome/tr312/Project/Run_model_master_mask'

# List of block size directories
block_dirs = [
    'block_test_masked_backprop_mask_ratio_02',
    'block_test_masked_backprop_mask_ratio_05',
    'block_test_masked_backprop_mask_ratio_08',
    'block_test_masked_backprop_mask_ratio_09'
]

# Patterns to extract block size (A_B) and mask ratio (ZZ)
block_pattern = re.compile(r'(\d+_\d+)
    _block_test_masked_backprop_mask_ratio_\d{2}\.csv$')
mask_pattern = re.compile(r'_(\d{2})\.csv$')

def block_sort_key(block_str):
    w, h = map(int, block_str.split('_'))
    return (w, h)

def get_epoch_199_accuracy(file_path):
    try:
        df = pd.read_csv(file_path, nrows=201)
        acc_col = next((col for col in df.columns if 'acc@1' in col.
            lower()), None)
```

```
            if acc_col is None:
                return None

            df['Epoch'] = pd.to_numeric(df['Epoch'], errors='coerce')
            acc_row = df[df['Epoch'] == 199]
            return float(acc_row[acc_col].values[0]) if not acc_row.empty
                 else None
        except Exception as e:
            return None

    # Collect accuracy data
    accuracy_data = {}
    for dir_name in block_dirs:
        results_dir = os.path.join(base_dir, dir_name, 'results')

        if not os.path.exists(results_dir):
            continue

        for filename in os.listdir(results_dir):
            if filename.endswith('.csv'):
                block_match = block_pattern.search(filename)
                mask_match = mask_pattern.search(filename)

                if block_match and mask_match:
                    block_size = block_match.group(1)
                    mask_ratio = int(mask_match.group(1)) * 10  # Convert
                         '02' to 20%, etc.

                    csv_path = os.path.join(results_dir, filename)
                    acc = get_epoch_199_accuracy(csv_path)

                    if acc is not None:
                        accuracy_data.setdefault(block_size, {})[
                            mask_ratio] = acc

    if not accuracy_data:
        print("No valid data found.")
        exit()

    # Prepare data for CSV
    block_sizes = sorted(accuracy_data.keys(), key=block_sort_key)
    mask_ratios = sorted({mr for d in accuracy_data.values() for mr in d
        })

    rows =
    for bs in block_sizes:
        row = {'Block Size': bs.replace('_', '$\times$')}
        for mr in mask_ratios:
            row[f'{mr}%'] = accuracy_data[bs].get(mr, np.nan)
        rows.append(row)

    # Create DataFrame and save to CSV
    df = pd.DataFrame(rows)
    csv_path = os.path.join(base_dir, 'masked_accuracy.csv')
    df.to_csv(csv_path, index=False)
    print(f"Accuracy data saved to {csv_path}")
```

Listing 5.17: Extraction of accuracy data (from masked_accuracy.py).

5.3.6 Theoretical Memory Movement Calculation

The `memory_calculation.py` script implements the theoretical estimation of memory movement during the training process, the conceptual basis and assumptions of which are detailed in the Methodology chapter. This script utilizes the defined dimensions of both masked and non-masked layers, the specified block sizes and mask ratios, and considers hardware parameters like cache line size under worst-case assumptions.

Initially, the dimensions of the masked and non-masked layers are defined within the script, as shown in Listing 5.18:

```
    # Masked layers (block-sparse weights)
    masked_layers = [
        (64, 64, 5),     # XPatchLinear
        (256, 64, 5),   # XChannelLinear1
        (64, 256, 5),   # XChannelLinear2
    ]

    # Non-masked layers (full weights)
    non_masked_layers = [
        (64, 48, 1), (64, 1, 1),       # Embedding
        (64, 1, 10), (64, 1, 10),       # PreLayerAffine
        (64, 1, 10),                    # PostLayerScale
        (64, 1, 1), (64, 1, 1),       # Final Norm
        (64, 10, 1), (10, 1, 1),       # Head
    ]
```

Listing 5.18: Layer dimensions (from memory_calculation.py).

The block sizes and mask ratios to be analyzed are specified, as shown in Listing 5.19:

```
    block_sizes = [
        (1,1), (1,2), (1,4), (1,8), (1,16), (1,32), (1,64),
        (2,1), (4,1), (8,1), (16,1), (32,1), (64,1),
        (2,2), (4,4), (8,8), (16,16), (32,32), (64,64)
    ]

    sparsity_ratios = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
```

Listing 5.19: Block sizes and sparsity ratios (from memory_calculation.py).

The memory required for non-masked weights and activations is precomputed by the script, as illustrated in Listing 5.20:

```
    # Precompute non-masked weight memory (fixed per batch)
    non_masked_weight_bytes = sum(
        math.ceil(M * N * bytes_per_float / cache_line_size) *
            cache_line_size * count
        for M, N, count in non_masked_layers
    )

    # Precompute activation memory per batch (fixed)
    activation_bytes_per_batch = batch_size * (
        (3*32*32 + 64) +       # Embedding
        5 * (64 + 64) +         # XPatchLinear
```

```
        5 * (64 + 256) +        # XChannelLinear1
        5 * (256 + 64) +        # XChannelLinear2
        (64 + 10)             # Norm + Head
    ) * bytes_per_float
```

Listing 5.20: Precomputing non-masked memory (from memory_calculation.py).

Subsequently, the memory contribution from masked weights is calculated based on the block size and mask ratio, as demonstrated in Listing 5.21:

```
    for ratio in sparsity_ratios:
        kept_blocks = math.ceil(total_blocks * (1 - ratio))
        total_memory[ratio] += kept_blocks * cache_lines *
            cache_line_size * count
```

Listing 5.21: Calculating masked weight memory (from memory_calculation.py).

Finally, the total estimated memory movement over the training run is computed by summing the contributions from masked weights, non-masked weights, and activations, as shown in Listing 5.22:

```
    per_batch_bytes = (
        total_memory[ratio] +
        non_masked_weight_bytes +
        activation_bytes_per_batch
    )
    total_training_bytes = per_batch_bytes * total_batches
    total_gb = total_training_bytes / (1024 ** 3)

    # Cap memory at zero-sparsity baseline
    row[ratio] = min(total_gb, zero_sparsity_gb)
```

Listing 5.22: Calculating total training memory movement (from memory_calculation.py).

5.3.7 Extraction of Final Performance Metrics

Final performance metrics are extracted using two dedicated scripts. The `loaded_values.py` script computes the total number of loaded (non-masked) elements by iterating over the layers, calculating the kept elements based on the mask ratio and block size, and scaling the count by the total number of training batches. Its implementation is provided in Listing 5.23.

```
    import math
    import csv

    def generate_scaled_loaded_elements(
        masked_layers: list[tuple[int, int, int]],
        block_sizes: list[tuple[int, int]],
        sparsity_ratios: list[float],
        total_epochs: int = 200,
```

```
    batch_size: int = 128,
    train_samples: int = 50_000,
    csv_filename: str = "loaded_elements.csv"
):
    """
    Calculate total loaded elements scaled for full training duration.

    Args:
        masked_layers: List of (input_dim, output_dim, count) tuples
        block_sizes: List of (block_rows, block_cols) tuples
        sparsity_ratios: List of sparsity ratios (0.0-1.0)
        total_epochs: Number of training epochs
        batch_size: Samples per batch
        train_samples: Total training samples
        csv_filename: Output CSV filename
    """
    # Calculate total batches
    batches_per_epoch = math.ceil(train_samples / batch_size)
    total_batches = batches_per_epoch * total_epochs

    csv_rows = []

    for block_size in block_sizes:
        Bx, By = block_size
        row = {"Block Size": f"{Bx}x{By}"}

        for ratio in sparsity_ratios:
            total_loaded = 0

            for M, N, count in masked_layers:
                # Skip incompatible block sizes
                if M % Bx != 0 or N % By != 0:
                    continue

                # Calculate total blocks and the loaded (kept) blocks
                blocks_per_layer = (M // Bx) * (N // By)
                kept_blocks = math.ceil(blocks_per_layer * (1 - ratio))
                elements_per_block = Bx * By

                total_loaded += kept_blocks * elements_per_block * count

            # Scale to full training duration
            total_loaded_scaled = total_loaded * total_batches
            row[ratio] = int(total_loaded_scaled)

        csv_rows.append(row)

    # Write to CSV
    with open(csv_filename, 'w', newline='') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=['Block Size'] + sparsity_ratios)
        writer.writeheader()
        writer.writerows(csv_rows)
    print(f"Scaled loaded elements saved to {csv_filename}")
```

```
62
63      # Example usage for CIFAR-10 training
64      if __name__ == "__main__":
65          masked_layers = [
66              (64, 64, 5),    # XPatchLinear layers
67              (256, 64, 5),   # XChannelLinear1 layers
68              (64, 256, 5),   # XChannelLinear2 layers
69          ]
70
71          block_sizes = [
72              (1,1), (1,2), (1,4), (1,8), (1,16), (1,32), (1,64),
73              (2,1), (4,1), (8,1), (16,1), (32,1), (64,1),
74              (2,2), (4,4), (8,8), (16,16), (32,32), (64,64)
75          ]
76
77          sparsity_ratios = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
78
79          generate_scaled_loaded_elements(
80              masked_layers,
81              block_sizes,
82              sparsity_ratios
83          )
```

Listing 5.23: Calculation of scaled loaded elements (from loaded_values.py).

Additionally, the `masked_accuracy.py` script extracts and aggregates the top-1 validation accuracy from the training logs, with the results saved to a CSV file for further analysis. The extraction logic was previously presented in Listing 5.17.

5.4 EXPERIMENTAL PROCEDURE DETAILS

For the CIFAR-10 dataset, experiments were conducted by systematically varying two key hyperparameters: block size and mask ratio, with the specific ranges detailed in the Methodology chapter. To ensure reproducibility, a fixed random seed was used so that identical settings yield exactly the same results.

For the preliminary scalability exploration on the ImageNet dataset, experiments were conducted for a fixed configuration: a 32×32 block size with an 80% mask ratio. Training on ImageNet was performed for 90 epochs due to time constraints, unlike the 200 epochs used for CIFAR-10. A fixed random seed was also employed for these ImageNet experiments.

Job scheduling for all experiments was managed via SLURM, although a detailed discussion of SLURM configuration is beyond the scope of this work.

Additionally, external code contributions were incorporated into the experimental framework. In particular, the `LaunchPad`[1] Python library was utilized. This library leverages the YAML configuration

1 https://github.com/danielbarley/LaunchPad

file to automatically generate SLURM job submission scripts for each configuration specified, thereby streamlining the launch of multiple experiments across different parameter settings.

In summary, experiments on CIFAR-10 were systematically executed by varying block sizes and mask ratios. For ImageNet, a specific, promising configuration from the CIFAR-10 exploration was tested under a reduced epoch count to gain initial insights into scalability. Results for all experiments were logged and aggregated for subsequent analysis.

In this chapter, a detailed account was provided of the technical implementation of the dynamic structured weight masking approach and its associated measurement, profiling, and analysis tools. The modifications to the ResMLP model for CIFAR-10 were described. The implementation of block-wise masking using the Frobenius norm and top-k selection was detailed, along with the application of the mask during both forward and backward passes. A comprehensive measurement framework was developed to compute adjusted FLOPs, estimate theoretical memory movement, and enable the generation of visualizations for masking patterns and training progress. Additionally, the experimental procedure was briefly outlined, emphasizing the systematic variation of block sizes and mask ratios, the use of a fixed random seed for reproducibility, and the integration of a LaunchPad library to automate experiment setup. Overall, a robust framework has been established that provides the foundation for evaluating the effectiveness of the proposed masking techniques. The next chapter will present and analyze the experimental results.

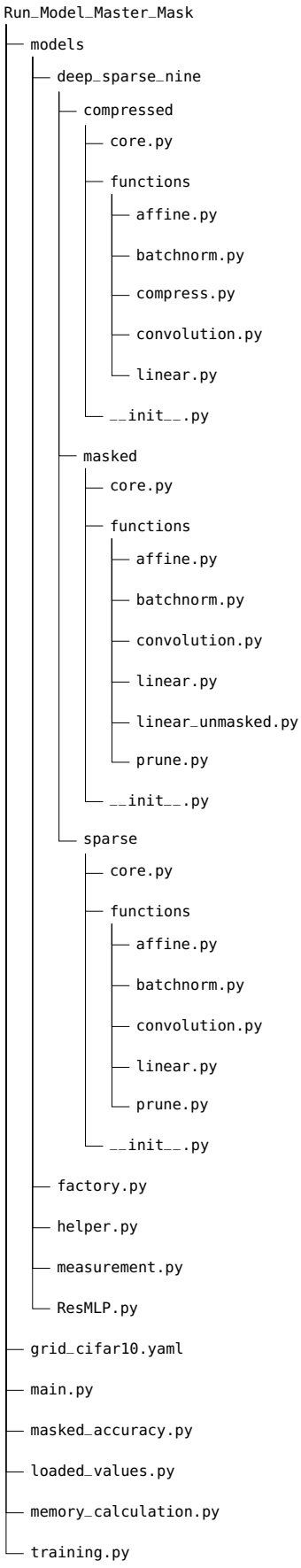


Figure 5.1: Directory structure of the Master Thesis project. This tree illustrates the organization of the codebase, including key directories such as `models`, and various measurement and training scripts. Note that the `masked` directory was newly created and contains all the adjusted files required for the experiments.

6 BENCHMARK AND RESULTS

6.1 INTRODUCTION

This chapter presents the empirical results obtained from benchmarking the ResMLP model integrated with the dynamic structured weight masking mechanism, as detailed in Chapters 4 and 5. The experiments were designed primarily to evaluate the trade-offs between model accuracy and theoretical memory movement when applying varying levels of weight sparsity using different block granularities during training on the CIFAR-10 dataset. The impact on theoretical computational cost (FLOPs) is also briefly examined. The analysis begins with an overview of the comprehensive results on CIFAR-10, followed by a focused investigation of a specific case exhibiting significant theoretical memory savings and interesting accuracy behavior. Additionally, preliminary findings from applying the approach to the ImageNet dataset, where maintaining performance under masking proved more challenging, will be briefly presented. The chapter concludes with a summary of the computational cost analysis.

This section details the experimental configuration used to obtain the benchmark results presented in this chapter. It covers the computational environment, core training procedures, key hyperparameter ranges explored for structured weight masking, and the metrics used for evaluation for both CIFAR-10 and the preliminary ImageNet experiments.

6.1.1 Computational Environment

The experiments were conducted on a system equipped with an NVIDIA A30 GPU running Linux. The software environment utilized Python 3.12.2, PyTorch 2.5.1, and CUDA 12.4. Table 6.1 provides a comprehensive summary of the system setup.

6.1.2 Training Procedure

For the CIFAR-10 dataset, the ResMLP model was trained for a total of 200 epochs using the Adam optimizer. An initial learning rate of 0.01 was employed and dynamically adjusted via a cosine annealing scheduler (warmup: 5 epochs at 0.001, cooldown: 10 epochs, minimum LR: 1e-5). Standard training practices were followed, including model initialization, optimizer/scheduler setup, and the use of a cross-entropy

Category	Details
System Information	Python Version: 3.12.2 PyTorch Version: 2.5.1 CUDA Version: 12.4
Operating System	Linux OS Version: #1 SMP PREEMPT_DYNAMIC Debian 6.1.112-1 (2024-09-30)
Hardware Details	Machine: x86_64 GPU: NVIDIA A30

Table 6.1: System Setup and Environment Details

loss function with label smoothing (0.1). The training framework facilitated experiments both with the standard (dense) model and with the dynamic structured weight masking mechanism central to this work.

For the preliminary ImageNet experiments, training was conducted for 90 epochs due to time constraints, utilizing the AdamW optimizer with parameters as detailed in Table 5.2. Other aspects of the training procedure, such as the learning rate scheduler and loss function, were consistent with the ImageNet-specific setup described in Chapter 5.

6.1.3 Hyperparameter Settings for Masking

To systematically evaluate the impact of dynamic structured weight masking on CIFAR-10, two key hyperparameters were explored: the masking block granularity (block size) and the target mask ratio.

BLOCK SIZES (CIFAR-10): The granularity of masking was controlled by varying the block size. A wide range of block dimensions were tested, encompassing fine-grained masking, vector-like blocks, and large square blocks. Specifically, the tested sizes included:

- Fine-grained: 1×1 (operating on individual weights).
- Vector-like: $1\times N$ and $N\times1$ where $N \in \{2, 4, 8, 16, 32, 64\}$.
- Square: $N\times N$ where $N \in \{2, 4, 8, 16, 32, 64\}$.

This selection allows for investigating how different masking structures influence performance. Finer granularities (like 1×1) offer maximum precision in selecting important weights but may lack structural benefits for hardware. Conversely, larger block sizes (e.g., 32×32, 64×64) enforce a coarser, more structured sparsity pattern. Such patterns, consisting of large contiguous masked regions, are potentially more amenable to hardware acceleration techniques that can skip memory transactions for zeroed blocks, thereby offering a potential pathway to improved memory efficiency.

MASK RATIOS (CIFAR-10): In parallel, for the CIFAR-10 experiments, the target mask ratio (determining the fraction of weight blocks to be masked) was varied across the set {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}.

SETTINGS FOR IMAGENET: For the preliminary ImageNet experiments, a fixed masking configuration was used: a block size of 32×32 and a target mask ratio of 0.8 (80%). This configuration was chosen as an illustrative example, and an exhaustive sweep of masking hyperparameters was not performed for ImageNet.

It should be noted that for all experiments, while the binary mask is dynamically recalculated via a top-k selection strategy during each forward pass based on block norms, the target mask ratio remains fixed throughout the training process after an initial warmup phase (e.g., the first 10 epochs of training).

6.1.4 Evaluation Metrics

Model performance was primarily evaluated on the respective test/-validation sets (CIFAR-10 test set, ImageNet validation set) using top-1 classification accuracy. Training accuracy was also monitored during the training process to assess convergence and identify potential overfitting. These metrics provide a comprehensive view of the model's classification performance and generalization capability under different masking configurations and datasets.

6.2 OVERALL PERFORMANCE LANDSCAPE: ACCURACY AND MEMORY MOVEMENT

To provide a comprehensive overview of the model's behavior across the explored hyperparameter space, the final validation accuracy and the estimated total memory movement are presented together. This allows for an initial assessment of the interplay between block size, mask ratio, and the two backward pass configurations (mask applied vs. not applied during gradient computation).

6.2.1 Presentation and High-Level Observations

In the heatmaps (Figure 6.1 and 6.2), the y-axis represents block size and the x-axis represents the mask ratio. Each cell shows the final Top-1 validation accuracy (percentage and color) and the theoretical memory movement (GB) after 200 epochs.

Several key observations emerge from these heatmaps:

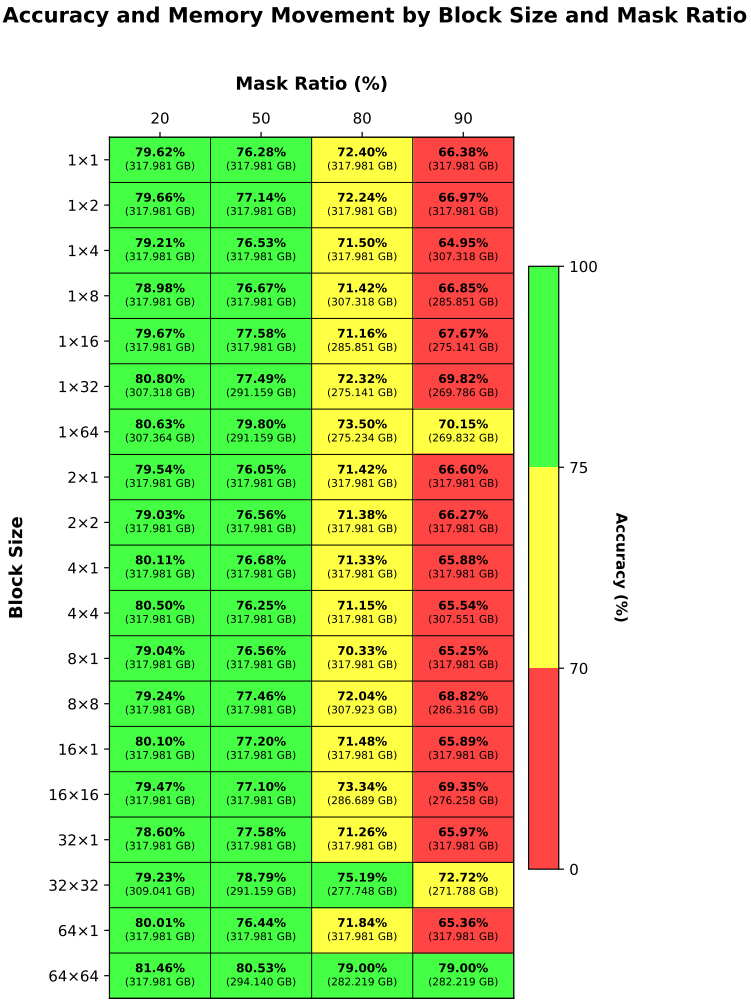


Figure 6.1: Final Top-1 Validation Accuracy (color scale and percentage) and Theoretical Memory Movement (GB) for varying Block Sizes and Mask Ratios, with the mask applied during the backward pass.

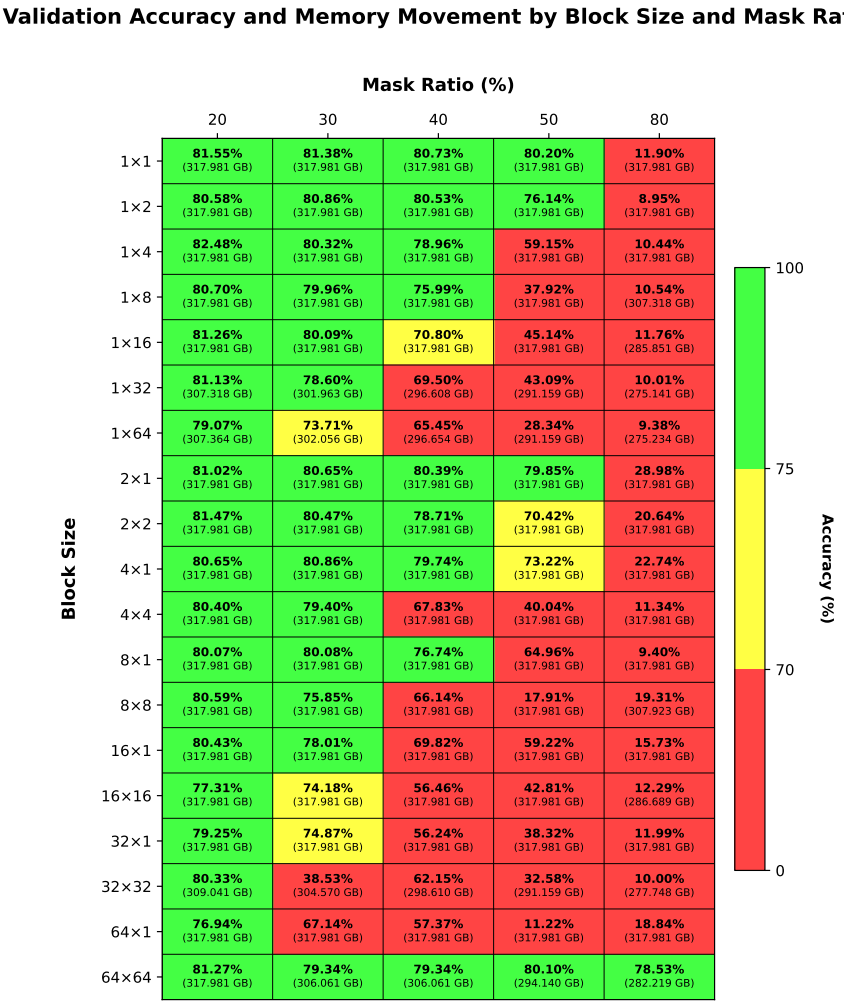


Figure 6.2: Final Top-1 Validation Accuracy (color scale and percentage) and Theoretical Memory Movement (GB) for varying Block Sizes and Mask Ratios, without applying the mask during the backward pass.

- When the mask is applied during the backward pass, accuracy generally remains high for mask ratios up to approximately 50%, after which it tends to decline.
- The rate and extent of accuracy degradation at higher mask ratios are strongly dependent on the block size. Larger block sizes sometimes maintain accuracy better at moderate mask ratios but can degrade sharply later.
- Theoretical memory movement reductions are most apparent at high mask ratios (e.g., 80%, 90%) combined with large block sizes (e.g., 32x32, 64x64). Many other configurations show minimal theoretical savings based on the calculation method.
- A noticeable difference in final accuracy exists between the masked and unmasked backward pass conditions, particularly visible in certain high-mask-ratio, large-block configurations like 32x32 at 80% mask ratio.

These trends motivate a closer look at the relationship between loaded elements, memory movement, and the impact of the backward pass condition, particularly in regimes where memory savings appear possible.

6.2.2 Preliminary Results on ImageNet

To gain initial insights into the scalability of the dynamic structured masking approach to larger and more complex datasets, preliminary experiments were conducted on the ImageNet ILSVRC2012 dataset. As detailed in the Benchmarking Setup, these experiments utilized a ResMLP model adapted for ImageNet (Table 5.2) and were run for 90 epochs.

The baseline ResMLP model, when trained densely (without any masking) on ImageNet for 90 epochs, achieved a top-1 validation accuracy of 40.60%.

For the masked experiment on ImageNet, a configuration employing 32×32 block sizes with an 80% target mask ratio was evaluated. Under these conditions, the model achieved a top-1 validation accuracy of 16.51% after 90 epochs.

These preliminary results indicate a significant drop in accuracy when applying the tested masking configuration to ImageNet compared to the dense baseline on the same dataset, suggesting that the specific masking strategy and parameters require further adaptation for effective performance on this larger scale.

6.3 ANALYSIS OF MEMORY MOVEMENT, LOADED ELEMENTS, AND MASKING GRANULARITY

A primary goal of this work is to investigate the potential for reducing global memory accesses during the forward pass. The memory movement values presented in the overview heatmaps (Figures 6.1 and 6.2) quantify the theoretically estimated data transferred for all layers in the forward pass, including both masked and unmasked weight layers.

A predominant observation from the heatmaps is the frequent occurrence of the maximum calculated value (approximately 318 GB in many cells). When this maximum is reached, it indicates that the theoretical memory movement, as calculated, is not reduced by masking under those specific configurations and assumptions.

To illustrate the actual reduction in the number of **computationally active elements**, Table 6.2, included below, presents the total number of loaded (non-masked) weight elements, scaled over the entire training duration.

Block Size	Sparsity								
	10%	20%	30%	40%	50%	60%	70%	80%	90%
1x1	12.97B	11.53B	10.09B	8.65B	7.21B	5.77B	4.32B	2.88B	1.44B
1x2	12.97B	11.53B	10.09B	8.65B	7.21B	5.77B	4.33B	2.88B	1.44B
1x4	12.97B	11.53B	10.09B	8.65B	7.21B	5.77B	4.33B	2.89B	1.44B
1x8	12.98B	11.54B	10.09B	8.65B	7.21B	5.77B	4.33B	2.89B	1.45B
1x16	12.98B	11.54B	10.10B	8.66B	7.21B	5.77B	4.34B	2.89B	1.45B
1x32	12.99B	11.55B	10.11B	8.67B	7.21B	5.78B	4.34B	2.90B	1.46B
1x64	13.01B	11.56B	10.13B	8.68B	7.21B	5.81B	4.35B	2.93B	1.48B
2x1	12.97B	11.53B	10.09B	8.65B	7.21B	5.77B	4.33B	2.88B	1.44B
4x1	12.97B	11.53B	10.09B	8.65B	7.21B	5.77B	4.33B	2.89B	1.44B
8x1	12.98B	11.54B	10.09B	8.65B	7.21B	5.77B	4.33B	2.89B	1.45B
16x1	12.98B	11.54B	10.10B	8.66B	7.21B	5.77B	4.34B	2.89B	1.45B
32x1	12.99B	11.55B	10.11B	8.67B	7.21B	5.78B	4.34B	2.90B	1.46B
64x1	13.01B	11.56B	10.13B	8.68B	7.21B	5.81B	4.35B	2.93B	1.48B
2x2	12.97B	11.53B	10.09B	8.65B	7.21B	5.77B	4.33B	2.89B	1.44B
4x4	12.98B	11.54B	10.10B	8.66B	7.21B	5.77B	4.34B	2.89B	1.45B
8x8	13.01B	11.56B	10.13B	8.68B	7.21B	5.81B	4.35B	2.93B	1.48B
16x16	13.11B	11.71B	10.21B	8.81B	7.21B	5.91B	4.50B	3.00B	1.60B
32x32	13.61B	12.01B	10.81B	9.21B	7.21B	6.41B	4.80B	3.60B	2.00B
64x64	14.41B	14.41B	11.21B	11.21B	8.01B	8.01B	8.01B	4.80B	4.80B

Table 6.2: Loaded weights (in billions) for various block sizes and sparsity levels from a 200-epoch training run of the ResMLP network on the CIFAR-10 dataset.

As clearly demonstrated in Table 6.2, increasing the mask ratio consistently and significantly reduces the total number of loaded elements required for computation, largely independent of block size

(minor variations notwithstanding). This reduction directly reflects the potential for computational savings. However, a disparity exists between this substantial reduction in loaded elements and the limited reduction observed in the calculated memory movement for many configurations, highlighting a challenge in directly translating computational sparsity into theoretical memory access savings using this model.

The process of masking discrete blocks also introduces granularity effects. The actual number of masked elements may not increase linearly with the target mask ratio, especially for large blocks where masking a single block removes a large chunk of weights. This effect is illustrated in Figure 6.3, which plots the actual number of masked elements versus the target mask ratio for different block sizes. Small block sizes (e.g., 2x2, left plot) result in a near-linear increase in masked elements as the target ratio increases. Intermediate block sizes (e.g., 32x32, center plot) show noticeable steps, while large block sizes (e.g., 64x64, right plot) exhibit pronounced steps and plateaus.

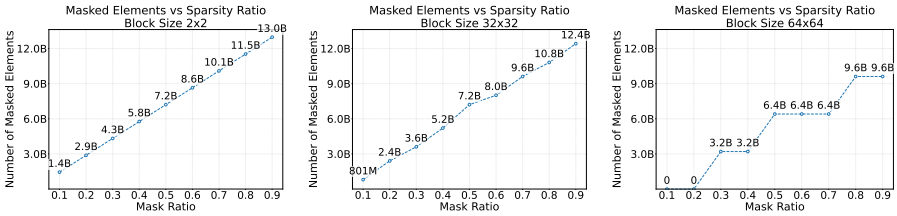


Figure 6.3: Comparison of Masked Elements vs. Target Mask Ratio for different block sizes: 2x2 (Left), 32x32 (Center), and 64x64 (Right). Larger block sizes exhibit more pronounced step-like increases due to block granularity.

These steps occur because masking proceeds block by block based on the top-k selection using the ceiling function (`math.ceil`) to determine the number of blocks to keep. For large blocks like 64x64, increasing the target mask ratio slightly (e.g., from 50% to 60%, or 80% to 90%) might not be sufficient to change the discrete number of blocks that are actually masked. Consequently, the total number of masked elements plateaus within these ranges, as seen in the rightmost plot of Figure 6.3.

6.4 IMPACT OF BACKWARD PASS MASKING AT HIGH MASK RATIOS

Given the observed variations in accuracy across different block sizes, mask ratios, and backward pass conditions, particularly the interaction between block size and mask ratio highlighted in the heatmaps, this section delves into the impact of masking gradients during the backward pass.

Figure 6.4 provides a comparative view of the validation accuracy progression over epochs for all tested block sizes at a fixed, high mask ratio of 80%. The left plot shows the results when the mask is applied during the backward pass, while the right plot shows the results when it is not.

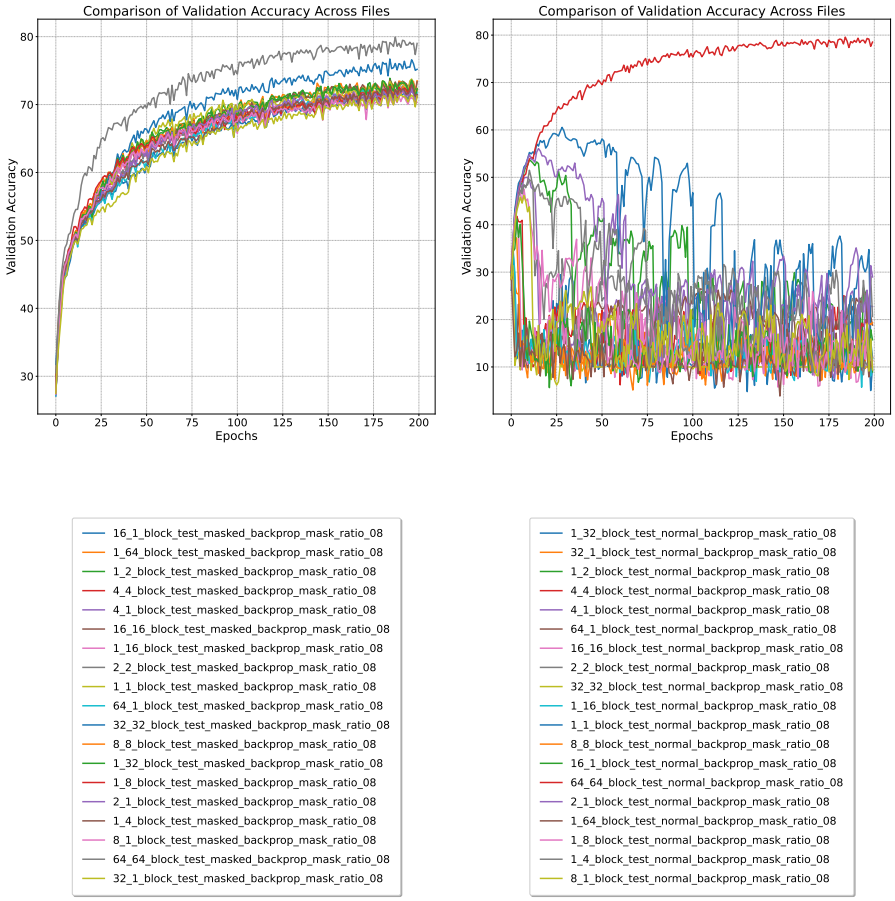


Figure 6.4: Comparison of Validation Accuracy Across All Block Sizes at 80% Mask Ratio. Left: Mask applied during backward pass. Right: Mask not applied during backward pass. Each line represents a different block size configuration.

While the density of plots in Figure 6.4 makes detailed inspection of individual block size performance challenging, the overall trend is strikingly clear. When the mask is applied during the backward pass (left plot), most configurations, despite the high 80% mask ratio, manage to learn and achieve varying levels of final accuracy, generally converging towards values between 70% and 79% as seen in the heatmap (Figure 6.1). However, when the mask is **not** applied during the backward pass (right plot), the training behavior is drastically different. Most configurations exhibit highly unstable accuracy curves, often collapsing to very low values, and fail to converge effectively. This strongly indicates that allowing gradients to propagate through masked weights at high mask ratios is detrimental to the learning pro-

cess across almost all block granularities. This observation motivates the subsequent detailed case study focusing on a configuration where this effect is particularly pronounced.

6.5 CASE STUDY: 32X32 BLOCK SIZE AT 80% MASK RATIO

To delve deeper into the observed phenomena, particularly the impact of the backward pass condition in a regime with potential memory savings, this section focuses on the configuration using a 32x32 block size with an 80% mask ratio. This case is selected because Figures 6.1 and 6.2 show a substantial theoretical memory movement reduction (to approx. 277 GB) and, as reinforced by Figure 6.4, a striking difference in final accuracy between the masked backward pass (approx. 75.19%) and the unmasked backward pass (approx. 10.00%).

6.5.1 Training Dynamics Comparison

Figure 6.5 isolates the Top-1 validation accuracy and training loss curves for the 32x32 @ 80% configuration, directly contrasting the two backward pass conditions.

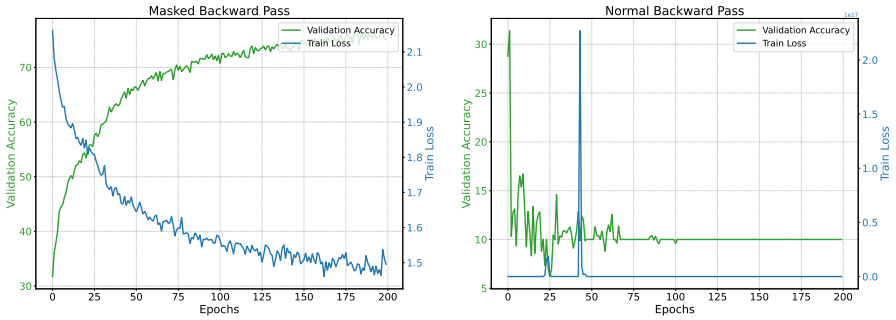


Figure 6.5: Comparison of Training Dynamics for 32x32 Block Size at 80% Mask Ratio. Left: Mask applied during backward pass. Right: Mask not applied during backward pass. The plots show Top-1 Validation Accuracy (green) and Training Loss (blue) over 200 epochs.

The contrast in training dynamics is pronounced. When the mask is applied during the backward pass (Figure 6.5, left), the model trains relatively well, achieving over 75% final accuracy. The training loss decreases steadily, and the validation accuracy increases consistently, indicating stable learning despite the high mask ratio. Conversely, when the mask is not applied during the backward pass (Figure 6.5, right), the training becomes highly unstable. The training loss exhibits large, erratic spikes, and the validation accuracy fails to improve significantly, remaining near random chance level (around 10%). This

instability is attributed to gradients being computed with respect to the full weight tensor, including contributions from weights that were masked (and thus did not contribute) during the forward pass. Such gradients, derived from inactive weights, can be noisy or disproportionately scaled, leading to detrimental parameter updates that hinder effective learning.

6.5.2 Masking Pattern Visualization

Visualizing the distribution of active (non-masked) blocks within the network layers can offer qualitative insights into how the masking mechanism operates under the different backward pass conditions. Figure 6.6 presents heatmaps of the cumulative block activity counts over the training duration for the three main linear layer types within the first ResMLP block (Block 0) for the 32x32 with 80% mask ratio configuration. The left column corresponds to the stable training run (mask applied during backward pass), and the right column corresponds to the unstable run (mask not applied).

Comparing the heatmaps in Figure 6.6, a visual difference in the masking dynamics emerges. In the stable run (left column, masked backward pass), the patterns of active blocks (brighter colors) appear somewhat more structured or consistent over time. Conversely, the unstable run (right column, unmasked backward pass) exhibits patterns that seem more varied or potentially erratic, suggesting that the set of active blocks might be changing more frequently or less predictably. This qualitative observation aligns with the idea that unstable gradient updates in the unmasked backward pass case could interfere with the top-k selection mechanism, leading to less stable sparsity patterns compared to the masked backward pass condition where training is stable.

While these visualizations are qualitative, they provide supporting evidence suggesting that the backward pass masking not only stabilizes gradient updates but might also lead to more consistent dynamic sparsity patterns during training, especially under challenging high-mask-ratio conditions.

6.6 COMPUTATIONAL COST ANALYSIS (THEORETICAL FLOPS)

For completeness, the theoretical computational cost reduction is presented. Figure 6.7 plots the calculated theoretical FLOPs per epoch against the mask ratio.

The plot illustrates the linear decrease in theoretical FLOPs inherent in the calculation methodology as the mask ratio increases, a direct consequence of the model's assumption that computational cost scales

Figure 6.6: Comparison of Block Activity Heatmaps for Block 0 Layers (32x32 Block Size, 80% Mask Ratio). Left column: Mask applied during backward pass. Right column: Mask not applied during backward pass. Rows correspond to XPatch Linear, XChannel Linear 1, and XChannel Linear 2 layers, respectively. Colors indicate the cumulative count of times each block was active (non-masked) over 200 epochs.

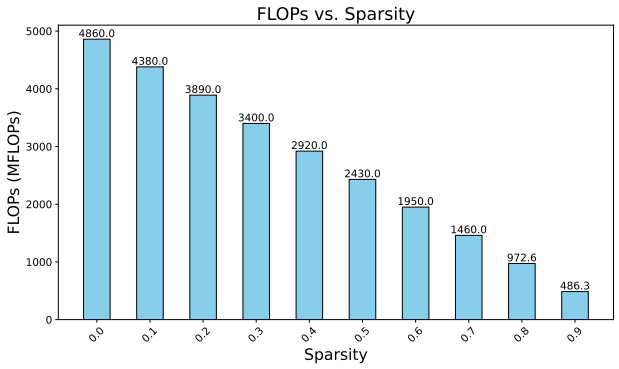


Figure 6.7: Theoretical FLOPs per epoch as a function of Mask Ratio. The plot demonstrates a linear decrease in computational operations as the mask ratio increases, reflecting the reduction in active weight elements.

with the number of active (non-masked) elements (Table 6.2). This simplified theoretical calculation indicates that at an 80% mask ratio, the total theoretical FLOPs are reduced by approximately 5x compared to the dense equivalent. While practical speedups depend on hardware/software support, this quantifies the computational efficiency potential estimated by this model.

The theoretical FLOPs for the CIFAR-10 experiments, as analyzed, demonstrated the expected linear reduction in computation with increasing mask ratio. Overall, the results presented in this chapter emphasize the critical role of co-designing the mask ratio and block granularity to balance accuracy and efficiency, and highlight the distinction between reducing computation and reducing memory traffic in theoretical assessments. The handling of gradients during the backward pass also emerged as a critical factor for training stability at high mask ratios, particularly evident in the CIFAR-10 experiments. Furthermore, preliminary tests on ImageNet indicated that applying this masking approach to larger, more complex datasets presents substantial challenges requiring dataset-specific adaptations.

7 DISCUSSION

This chapter interprets the empirical results presented in Chapter 6, focusing on the interplay between dynamic structured weight masking, model accuracy, theoretical memory movement, and computational cost within the ResMLP architecture. The findings are discussed in relation to the research objectives, situated within the context of existing literature, and considered for their practical implications and limitations.

7.1 OVERVIEW OF KEY FINDINGS

The benchmarking experiments yielded several notable insights regarding the effects of dynamic structured weight masking during the training of ResMLP networks. For the CIFAR-10 dataset, key findings include: First, when employing a masked backward pass, model accuracy on CIFAR-10 remained robust for mask ratios up to approximately 50%, beyond which accuracy typically deteriorated. The onset and degree of this degradation were further modulated by the granularity of the block size, as illustrated in Figure 6.1 (masked backward pass) and contrasted with the behavior observed with an unmasked backward pass (Figure 6.2). Second, a marked discrepancy was observed between the substantial reduction in loaded (non-masked) elements and the modest decrease in estimated theoretical memory movement. While the number of active elements decreased nearly linearly with increasing mask ratio (see Table 6.2), the calculated memory movement showed significant reductions only for configurations employing large block sizes at high mask ratios (Figures 6.1 and 6.2). Third, the strategy for handling gradients during the backward pass proved critical. Specifically, applying the mask during backpropagation was essential for ensuring training stability and preserving final accuracy—especially at high mask ratios (e.g., 80%)—whereas allowing gradients to propagate through masked weights frequently led to instability or collapse (Figures 6.4 and 6.5). Finally, the theoretical computational cost for CIFAR-10 experiments, quantified in FLOPs, decreased linearly with increasing mask ratio, directly reflecting the reduction in active weight elements (Figure 6.7).

Furthermore, preliminary experiments on ImageNet, limited to 90 epochs and a specific masking configuration (32×32 blocks, 80% mask ratio), demonstrated that this setup led to a marked decrease in accuracy from 40.60% (dense) to 16.51% (masked). This underscores

the increased difficulty of applying the tested masking strategies effectively to more complex datasets without further, dataset-specific adjustments.

7.2 INTERPRETATION OF FINDINGS

7.2.1 Accuracy-Mask Ratio Trade-off and Block Granularity

The decline in classification accuracy observed at mask ratios exceeding approximately 50% when employing a masked backward pass is primarily attributable to the removal of a substantial portion of weight parameters, leading to a loss of critical information necessary for the model to capture complex data features. This 50% threshold for maintaining robust accuracy is a key characteristic of the masked backward pass configuration. In this scenario, the model tolerates higher mask ratios, and employing larger blocks can even help capture important features more effectively. In contrast, when the backward pass is unmasked, accuracy degrades at lower mask ratios, and the selective elimination provided by fine-grained (smaller block) masking performs better. Additionally, larger blocks result in more hardware-friendly memory access due to their contiguous structure, which facilitates coalesced memory transactions on GPUs. This observation underscores the critical role of the backward pass masking strategy: with backward pass masking, larger blocks yield superior performance, whereas without it, smaller blocks are preferable. Furthermore, an additional observation is that for larger block configurations, increasing the target mask ratio yields little change in the number of masked elements until a critical threshold is reached, at which point a sudden jump occurs (Figure 6.3).

7.2.2 Theoretical Memory Movement versus Loaded Elements

A critical observation from the results is the discrepancy between the substantial reduction in loaded (non-masked) elements (Table 6.2) and the relatively modest decrease in estimated theoretical memory movement, particularly for smaller block sizes or lower mask ratios (Figures 6.1 and 6.2). This disparity likely stems from the conservative assumptions underpinning the theoretical memory movement calculations used in this work. Specifically, factors such as cache line granularity—where accessing even a single element within a cache line may necessitate loading the entire line—and assumptions about worst-case alignment of data blocks relative to cache lines can lead to overestimations of the required memory traffic. Consequently, unless large, contiguous memory regions corresponding to full cache lines or multiples thereof are masked, the calculated reduction in data transfer

remains limited. Such a condition, where theoretical memory savings become apparent, is met primarily in configurations employing large block sizes (e.g., 32x32, 64x64) at high mask ratios.

Furthermore, the granularity effect observed in the actual number of masked elements (Figure 6.3) can also contribute to the behavior seen in the theoretical memory movement estimations. The plateauing of actual achieved mask level for large blocks across certain target mask ratio ranges means the number and configuration of loaded blocks may not change significantly. If the access patterns for these loaded blocks remain similar, the calculated memory movement may also stay constant or change minimally across those ranges, despite increases in the target mask ratio.

This analysis highlights that while structured masking significantly reduces the number of active parameters (Table 6.2) and thus potential computation, achieving corresponding reductions in global memory accesses, as estimated theoretically here, is more challenging. The theoretical memory model suggests that realizing tangible memory access savings likely requires hardware-aware optimizations. These might include the use of specialized sparse matrix libraries or custom compute kernels designed to explicitly bypass memory loads for masked regions and to exploit the coalesced memory access patterns potentially offered by structured sparsity. The potential for such savings appears strongly linked to using large block sizes at high mask ratios, where both the granularity effect is pronounced and the potential alignment with memory transaction units (like cache lines) is greater.

7.2.3 The Critical Role of Backward Pass Masking

The experiments unequivocally demonstrate that applying the weight mask during the backward pass is crucial for training stability, particularly at high mask ratios (Figures 6.4 and 6.5). When the mask was omitted during backpropagation, gradients were computed with respect to the entire weight tensor, including contributions from masked weights that did not influence the forward pass. These spurious gradients, calculated for weights that made no contribution to the forward pass output and thus to the computed loss, effectively introduce erroneous information into the learning process. Such gradients can be noisy, as they do not reflect a genuine contribution to the error; disproportionately scaled, as the optimization algorithm might assign undue importance to changes in these inactive weights; or otherwise misleading, by suggesting parameter adjustments that are not pertinent to the actual network behavior observed during the forward pass. Consequently, the optimizer may perform pathological updates, adjusting parameters based on this flawed information, which can disrupt the learning trajectory, prevent convergence, and ultimately destabilize the entire training process. In contrast, applying the mask

during the backward pass ensures that only gradients corresponding to the active weights are propagated. This alignment between forward and backward computations yields a more stable and meaningful learning signal.

The approach taken by Sparse Weight Activation Training (SWAT) [20] provides an interesting counterpoint, particularly as its evaluations were primarily conducted on Convolutional Neural Networks (CNNs). SWAT also sparsifies weights and activations for the computation of gradients in the backward pass but critically, it computes dense weight gradients which are then used to update all network weights, including those that were inactive during the forward pass. Raihan and Aamodt [20] report that this method allows for stable training and enables dynamic exploration of the sparse topology within these CNN architectures. The stability reported by SWAT, despite updating all weights, suggests that the method of gradient derivation and the consistent application of these gradients to all parameters (active or inactive) are key factors. This contrasts with the instability observed in the 'unmasked backward pass' configuration of this study, where gradients were also computed for all weights. The differing outcomes might be attributed to the specific update strategy of SWAT, the distinct architectural contexts (CNNs vs. the ResMLP used here), or the interaction with the top-k selection mechanism employed in this work. This highlights that the nuanced details of gradient computation, its application, and the underlying network architecture are paramount when weights involved in the forward pass differ from those being updated.

Further qualitative support for this interpretation comes from comparing the masking pattern visualizations from the case study (Figure 6.6). In the stable run (left column, masked backward pass), the patterns of active blocks (brighter colors) appear somewhat more structured or consistent over time. Conversely, the unstable run (right column, unmasked backward pass) exhibits patterns that seem more varied or potentially erratic, suggesting that the set of active blocks might be changing more frequently or less predictably. This qualitative observation aligns with the idea that unstable gradient updates in the unmasked backward pass case could interfere with the top-k selection mechanism, leading to less stable masking patterns compared to the masked backward pass condition where training is stable. While these visualizations are qualitative, they provide supporting evidence suggesting that the backward pass masking not only stabilizes gradient updates but might also lead to more consistent dynamic masking patterns during training, especially under challenging high-mask-ratio conditions.

7.2.4 Interpreting Scalability Challenges on ImageNet

The preliminary experiments on ImageNet, which showed a significant drop in accuracy for the masked model (16.51%) compared to its dense counterpart (40.60%) even with a configuration (32×32 blocks, 80% mask ratio) that was relatively successful on CIFAR-10, highlight the substantial challenges in scaling dynamic structured masking to larger and more complex datasets. Several factors likely contribute to this observed performance.

Firstly, the ImageNet dataset is vastly larger and more diverse than CIFAR-10, demanding greater model capacity and potentially different architectural considerations for the ResMLP model to learn effectively, even before masking is applied. The baseline dense accuracy of 40.60% after 90 epochs itself suggests that the model might have been undertrained or that the base architecture requires further scaling for optimal ImageNet performance.

Secondly, the masking configuration tested on ImageNet was not specifically tuned for this dataset but was chosen as an illustrative example from the space explored for CIFAR-10. It is plausible that the optimal block granularity and mask ratio for ImageNet differ significantly from those suitable for CIFAR-10. Aggressively masking 80% of a model that may already be struggling with capacity or training duration on a complex task could disproportionately harm its ability to learn critical features.

Finally, the limitation to 90 training epochs for the ImageNet experiments, due to time constraints, further complicates the interpretation. Models trained on ImageNet typically require hundreds of epochs to converge fully. The observed accuracy for both dense and masked models might be far from their potential with more extensive training.

Therefore, the lower accuracy of the masked model on ImageNet should be interpreted with caution. It primarily indicates that a direct application of the tested masking setup is insufficient for this challenging task and that achieving effective performance would necessitate further, dataset-specific adaptations. This could involve adjustments to the base model architecture, a dedicated search for optimal masking hyperparameters for ImageNet, and more extensive training, all of which were beyond the scope of this thesis's preliminary ImageNet exploration.

7.3 RELATION TO RESEARCH OBJECTIVES AND LITERATURE

The primary objective of this study was to investigate the potential for reducing global memory accesses in deep neural network training through dynamic structured weight masking. The findings indicate

that while the technique significantly reduces the number of active parameters, the translation of these reductions into decreased theoretical memory movement is constrained by the underlying memory model assumptions. Nevertheless, the observed reduction in active elements highlights the potential for actual memory savings, provided that hardware-aware optimization strategies are employed. Unlike conventional pruning methods, which focus on static sparsity for inference, this work advances the understanding of dynamic masking during training. By situating these results alongside studies on activation compression [2, 3] and unstructured dynamic sparsity [13, 19], this thesis offers a nuanced perspective on block-based dynamic weight masking within an MLP architecture. A notable contribution is the empirical demonstration of the necessity for backward pass masking to ensure training stability under high sparsity conditions.

A relevant baseline for dynamic sparsity in training is the Sparse Weight Activation Training (SWAT) methodology [20]. While both this thesis and SWAT explore dynamic sparsity during training, their primary objectives and specific mechanisms differ. SWAT's main goal is the reduction of training time via decreased computational FLOPs, achieved by sparsifying weights in the forward pass and both weights and activations during backward pass gradient computations, followed by updating all network weights using the derived dense gradients. This contrasts with the primary objective of this thesis, which is to reduce global memory accesses through dynamic structured block-wise masking. Furthermore, the approach in this thesis focuses on a specific form of structured sparsity (blocks) and investigates the critical impact of how gradients for masked elements are handled, particularly the finding that not updating masked weights (masked backward pass) is essential for stability. SWAT, on the other hand, reports stable training while updating all weights, enabling dynamic topology exploration, and can be applied to both unstructured and structured (e.g., channel-level) sparsity. Thus, while SWAT provides a valuable reference for achieving computational speedups via dynamic sparsity, this work investigates a distinct strategy focused on memory access reduction and the intricacies of maintaining training stability with block-structured dynamic masking.

7.4 IMPLICATIONS OF THE STUDY

The practical implications of this research are multifaceted. For the ResMLP architecture on CIFAR-10, maintaining a mask ratio of up to approximately 50% appears to strike an optimal balance between preserving model accuracy and reducing computational load. Moreover, the choice of block size is a critical hyperparameter that interacts with the treatment of the backward pass. When the backward pass

is masked, larger blocks can better preserve essential features while also facilitating hardware-friendly memory access, thereby enhancing training stability even at high mask ratios. In contrast, when the backward pass is unmasked, the more selective elimination provided by smaller blocks tends to perform better, as unmasked gradients from inactive weights may destabilize training. These findings for CIFAR-10 suggest that achieving significant memory access reductions while maintaining performance and stability involves careful co-design of the masking strategy, block granularity, and backward pass handling.

The preliminary results from the ImageNet experiments further inform the implications regarding scalability. The contrasting performance on ImageNet, where a tested configuration performed poorly compared to its dense counterpart, suggests that masking strategies and their optimal parameters are likely dataset-dependent and not directly transferable across datasets of vastly different scales and complexities. It implies that realizing benefits from dynamic structured masking on large-scale tasks such as ImageNet necessitates careful, dataset-specific adjustments. This could involve modifications to the base model architecture, dedicated optimization of masking parameters, or different training regimes, an exploration which was outside the scope of this thesis's ImageNet investigation.

Collectively, these findings suggest that while dynamic structured masking shows promise for improving training efficiency, achieving significant memory access reductions, particularly on diverse and large-scale tasks, will likely require co-design strategies that integrate hardware-friendly sparse algorithms with dedicated software or hardware support, alongside dataset-aware tuning of the masking approach.

7.5 LIMITATIONS OF THE STUDY

This study has several limitations that must be acknowledged. Firstly, the evaluation relies on theoretical estimates of FLOPs and memory movement (computed using the `ptflops` library [17]), which do not fully capture real-world hardware behaviors such as cache effects, memory bandwidth limitations, and kernel launch overheads. Consequently, the reported computational and memory savings represent theoretical potentials rather than empirically measured performance gains. Moreover, the current implementation within the PyTorch framework applies masking via element-wise multiplication, which zeroes out weights but does not prevent them from being loaded into memory or from participating in computations within dense linear layers.

Additionally, while the primary experiments were conducted on a single model architecture (ResMLP) and dataset (CIFAR-10), the preliminary exploration on ImageNet also had specific constraints.

The investigation on ImageNet was intended as an initial scalability check and was limited to 90 training epochs and a single, fixed masking configuration (32×32 blocks, 80% mask ratio). A dedicated optimization of the masking approach for ImageNet, which would involve a search for suitable masking parameters, model adaptations, or more extensive training, was not conducted as the primary research objective was not to find optimal masking parameters for diverse network-dataset combinations. Consequently, the observed accuracy of 16.51% for the masked model on ImageNet (compared to a 40.60% dense baseline) reflects the performance of this specific, untuned setup rather than an optimized application of the masking technique on this dataset. It is plausible that with further, dataset-specific adaptations, performance could be improved. These factors limit the generalizability of the specific quantitative findings from ImageNet, though they highlight important scalability considerations.

Finally, for all experiments, only the Frobenius norm (based on L2 calculations) was employed as the criterion for top-k block selection, and the target mask ratio was held fixed after an initial warmup phase; alternative selection criteria, such as utilizing the L1 norm of block weights or incorporating gradient magnitudes to assess block importance, or adaptive mask ratio strategies might yield different performance trade-offs.

8 CONCLUSION AND OUTLOOK

This concluding chapter summarizes the primary findings of the thesis regarding the reduction of global memory accesses in deep neural network (DNN) training through dynamic structured weight masking applied to the ResMLP architecture. It reiterates the key contributions and outlines promising directions for future research stemming from this work.

8.1 CONCLUSION

The central objective of this thesis was to evaluate the efficacy of dynamic structured weight masking as a means to mitigate the bottleneck imposed by global memory accesses during DNN training, with a specific focus on the ResMLP model. A block-wise masking mechanism, based on Frobenius norm magnitude (derived from L2 calculations) and top-k selection, was implemented. Its impact on model accuracy, theoretical computational cost (FLOPs), and estimated memory movement was thoroughly assessed across various block sizes and mask ratios, primarily using the CIFAR-10 dataset.

Empirical results from the CIFAR-10 experiments indicate that dynamic structured weight masking significantly reduces the computational load, as evidenced by a near-linear decrease in the number of active (loaded) weight elements and theoretical FLOPs with increasing mask ratio (see Table 6.2 and Figure 6.7). Regarding model accuracy, experiments employing masked backward passes demonstrated that accuracy could be maintained for mask ratios up to approximately 50%, beyond which performance degradation typically occurred; this behavior was strongly influenced by the chosen block granularity, with larger blocks often proving beneficial under these masked conditions (Figure 6.1). This 50% threshold for robust accuracy was not observed in experiments using unmasked backward passes; these instead exhibited a lower overall tolerance to masking, with accuracy declining at lower mask ratios, where the advantages of finer, smaller blocks became more apparent (Figure 6.2).

A key observation on CIFAR-10 was the discrepancy between the reduction in computationally active elements and the limited decrease in estimated theoretical memory movement. This disparity likely results from the assumptions made regarding cache line granularity, combined with a conservative estimate of worst-case alignment, suggesting that significant memory savings may only be realized under

specific configurations involving large block sizes and high mask ratios. Furthermore, the study underscores the critical importance of applying the mask during the backward pass. Without backward pass masking, gradients are computed for inactive weights, leading to unstable training dynamics and poor convergence—particularly at high mask ratios (e.g., 80%) (Figures 6.4 and 6.5).

Preliminary application of the approach to the ImageNet dataset (90 epochs, 32×32 blocks, 80% mask ratio) resulted in a substantial accuracy reduction for the masked model (16.51%) compared to its dense counterpart (40.60%). This indicates that while the core mechanisms of dynamic structured masking are in place, achieving effective performance on larger, more complex datasets requires further investigation into appropriate model and masking adaptations, an aspect not fully explored in this thesis.

In summary, this thesis provides a detailed empirical evaluation of dynamic structured weight masking for training ResMLP networks, primarily on CIFAR-10. It quantifies the trade-offs between accuracy, mask ratio, and block size, highlights the limitations of theoretical memory movement estimates in standard frameworks, and demonstrates the necessity of backward pass masking for stable high-mask-ratio training. This work contributes valuable insights into the application of structured sparsity during DNN training, identifies practical challenges, particularly concerning scalability to larger datasets, and outlines potential benefits.

8.2 OUTLOOK AND FUTURE WORK

The findings and limitations of this study open several promising avenues for future research aimed at further enhancing the efficiency of DNN training through sparsity:

- **Implementation of True Sparse Operations:** Future work should move beyond masking within dense operations by implementing true sparse tensor representations and computations. Leveraging libraries such as `PyTorch Sparse` or developing custom CUDA kernels would ensure that masked elements are entirely excluded from computation, thereby preventing unnecessary loading of data from global memory. This approach is essential for validating the theoretical memory savings in practice.
- **Hardware-Specific Performance Evaluation:** Empirical evaluation on target hardware is critical. Future studies should profile performance on various GPUs (e.g., the NVIDIA A30 and others) using tools such as NVIDIA Nsight Systems or `nvprof`. Measuring real-world metrics—including wall-clock time, memory bandwidth utilization, cache hit/miss rates, and energy con-

sumption—under different mask ratio configurations would provide a clearer picture of the practical benefits and limitations.

- **Evaluation and Adaptation for Diverse Models and Datasets:** To assess and improve the generalizability of dynamic structured weight masking, future research should extend the evaluation to other architectures, such as Convolutional Neural Networks (CNNs) and Transformers. Critically, further work is needed for larger datasets like ImageNet. Given the preliminary performance on ImageNet with the tested configuration, future efforts should focus on strategies for adapting and optimizing structured masking for such large-scale tasks. This includes investigating the interplay with model architecture scaling, appropriate training regimes, and dedicated exploration of masking parameters (e.g., block sizes and mask ratios) suited for these complex datasets, moving beyond the initial fixed-configuration tests conducted in this thesis's exploratory phase. It is conceivable that with such dataset-specific adjustments, improved performance could be achieved.
- **Adaptive Masking Strategies:** Exploring adaptive masking techniques—where the mask ratio and block size are dynamically adjusted during training based on performance metrics or computational constraints—could lead to more optimal trade-offs between accuracy and efficiency, potentially offering a more robust solution across different datasets and training stages.

By addressing these areas, future research can build upon the foundation laid by this thesis, paving the way for more resource-efficient, scalable, and practically applicable methods for sparse neural network training tailored to modern hardware architectures.

BIBLIOGRAPHY

[1] Daniel Barley. "Reducing the State of Large-Scale MLPs by Compressing the Backward Pass." Supervised by Holger Fröning. Master's thesis. Heidelberg University, Faculty of Engineering Sciences, 2023.

[2] Daniel Barley and Holger Fröning. "Compressing the Backward Pass of Large-Scale Neural Architectures by Structured Activation Pruning." In: *Proceedings of the Computing Systems Group, Institute of Computer Engineering, Heidelberg University*. arXiv preprint arXiv:2311.16883. 2023.

[3] Daniel Barley and Holger Fröning. "Less Memory Means Smaller GPUs: Backpropagation with Compressed Activations." In: *Computing Systems Group, ZITI, Heidelberg University*. arXiv preprint arXiv:2409.11902. 2024.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[5] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. "Rigging the Lottery: Making All Tickets Winners." In: *arXiv preprint arXiv:1911.11134* (2019). Version v3, Jul 23, 2021.

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

[7] Yongchang Hao, Yanshuai Cao, and Lili Mou. "NeuZip: Memory-Efficient Training and Inference with Dynamic Compression of Neural Networks." In: *arXiv preprint arXiv:2410.20650* (2024).

[8] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[9] Dan Hendrycks and Kevin Gimpel. "Gaussian Error Linear Units (GELUs)." In: *arXiv preprint arXiv:1606.08415* (2016).

[10] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. "Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks." In: *Journal of Machine Learning Research* 23 (2021), pp. 1–124.

[11] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[12] Yoshimasa Kubo, Michael Traynor, Thomas Trappenberg, and Sageev Oore. "Learning Adaptive Weight Masking for Adversarial Examples." In: *International Joint Conference on Neural Networks (IJCNN)*. 2019.

[13] Mike Lasby, Anna Golubeva, Utku Evci, Mihai Nica, and Yani A. Ioannou. "Dynamic Sparse Training with Structured Sparsity." In: *International Conference on Learning Representations (ICLR)*. 2024.

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539.

[15] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. "Scalable Training of Artificial Neural Networks with Adaptive Sparse Connectivity Inspired by Network Science." In: *Nature Communications* 9.1 (2018), pp. 1–12.

[16] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[17] Dmitry Nikolaev. *ptflops: FLOPs counter for PyTorch models*. https://github.com/sovrasov/flops-counter.pytorch. Accessed: 2024-11-18. 2020.

[18] NVIDIA Corporation. *NVIDIA AMPERE GA102 GPU ARCHITECTURE: Second-Generation RTX*. White Paper. V2.0, Updated with NVIDIA RTX A6000 and NVIDIA A40 Information. Available at: https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V2.pdf [Accessed: YYYY-MM-DD]. NVIDIA Corporation, Jan. 2021. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V2.pdf.

[19] Tejas Pote, Muhammad Athar Ganaie, Atif Hassan, and Swanand Khare. "Dynamic Forward and Backward Sparse Training (DFBST): Accelerated Deep Learning Through Completely Sparse Training Schedule." In: *Proceedings of Machine Learning Research*. Vol. 189. PMLR, 2022, pp. 1027–1041.

[20] Md Aamir Raihan and Tor M. Aamodt. "Sparse Weight Activation Training." In: *arXiv preprint arXiv:2001.01969* (2020).

[21] Sourjya Roy, Priyadarshini Panda, Gopalakrishnan Srinivasan, and Anand Raghunathan. "Pruning Filters While Training for Efficiently Optimizing Deep Learning Networks." In: *arXiv preprint arXiv:2003.02800* (2020).

[22] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[23] Ralf C. Staudemeyer and Eric Rothstein Morris. "Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks." In: *arXiv preprint arXiv:1909.09586* (2019).

[24] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey." In: *Proceedings of the IEEE* 105.12 (Dec. 2017). arXiv preprint arXiv:1703.09039 [cs.CV], version v2 dated Aug 13, 2017, pp. 2295–2329. DOI: 10.1109/JPROC.2017.2761740.

[25] Hugo Touvron et al. "ResMLP: Feedforward networks for image classification with data-efficient training." In: *arXiv preprint arXiv:2105.03404* (2021).

[26] Jiali Wang, Hongxia Bie, Zhao Jing, Yichen Zhi, and Yongkai Fan. "Weight Masking in Image Classification Networks: Class-Specific Machine Unlearning." In: *Knowledge and Information Systems* (2025), pp. 1–23.

[27] Yuyang Xue, Junyu Yan, Raman Dutt, Fasih Haider, Jingshuai Liu, Steven McDonagh, and Sotirios A. Tsaftaris. "BMFT: Achieving Fairness via Bias-Based Weight Masking Fine-Tuning." In: *arXiv preprint arXiv:2408.06890* (2024).

[28] Ping Yu, Mikel Artetxe, Myle Ott, Sam Shleifer, Hongyu Gong, Ves Stoyanov, and Xian Li. "Efficient Language Modeling with Sparse All-MLP." In: *arXiv preprint arXiv:2203.06850* (2022).

[29] Yongchen Zhou and Richard Jiang. "Advancing Explainable AI Toward Human-Like Intelligence: Forging the Path to Artificial Brain." In: *arXiv preprint arXiv:2402.06673* (2024).

ERKLÄRUNG

Ich versichere, dass ich diese Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 02/06/2025

_____

Your Name Here