Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Iterative blocked pruning of neural networks

Diploma Thesis

2025
Bc. Jana Viktória Kováčiková

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Iterative blocked pruning of neural networks

Diploma Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	FMFI.KAI - Department of Applied Informatics
Supervisor:	Mgr. Vladimír Boža, PhD.

Bratislava, 2025
Bc. Jana Viktória Kováčiková

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Jana Viktória Kováčiková
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Iterative blocked pruning of neural networks
Iteratívne blokové orezávanie neurónových sietí

Anotácia: V súčasnosti používaných neurónových sieťach sa najviac času venuje maticovému násobeniu medzi váhami a aktiváciami. Ukazuje sa, že je možné nahradiť veľkú časť váhových matíc (bloky) nulami bez veľkej straty presnosti. Napriek tomu sa praktické zrýchlenie nedarí dosiahnuť, keďže násobenie riedkych matíc je na bežnom hardvéri neefektívne.

Predchádzajúci výskum blokového orezávania sa zameral hlavne na orezávanie v jednom kroku po trénovaní. Na druhej strane výskum neštrukturovaného orezávania ukazuje, že lepšie výsledky sa dajú dosiahnuť orezávaním vo viacerých krokoch (a postupným zvyšovaním riedkosti siete). Našim cieľom je tieto techniky aplikovať v kontexte blokového orezávania.

Vedúci: Mgr. Vladimír Boža, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 05.12.2023

Dátum schválenia: 06.05.2025 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Jana Viktória Kováčiková
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Iterative blocked pruning of neural networks

Annotation: In currently used neural networks, a significant portion of computational time is dedicated to matrix multiplications involving weight matrices and activations. Research indicates the feasibility of pruning substantial portions (blocks) of weight matrices to zero without a significant sacrifice in accuracy. Despite this, practical speed-ups remain elusive, as the multiplication of sparse matrices is inefficient on current hardware.

Previous research of block pruning focused mostly on pruning in one shot after training. However, most of the research for unstructured pruning shows that pruning in multiple rounds (and increasing sparsity in each round) leads to higher accuracy. Our goal is to apply these iterative techniques in the context of block pruning.

Supervisor:	Mgr. Vladimír Boža, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. RNDr. Tatiana Jajcayová, PhD.
Assigned:	05.12.2023
Approved:	06.05.2025

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Čestné vyhlásenie: Čestne vyhlasujem, že som celú diplomovú prácu na tému *Iteratívne blokové orezávanie neurónových sietí*, vrátane všetkých jej príloh a obrázkov (pokiaľ nie je zdroj riadne vyznačený), vypracovala samostatne, a to s použitím literatúry uvedenej v priloženom zozname a nástrojov umelej inteligencie. Vyhlasujem, že nástroje umelej inteligencie som použila v súlade s príslušnými právnymi predpismi, akademickými právami a slobodami, etickými a morálnymi zásadami za súčasného dodržania akademickej integrity a že ich použitie je v práci vhodným spôsobom označené.

Pri príprave tejto práce boli použité nástroje UI: GitHub Copilot na úpravu Python kódu a ChatGPT na reformuláciu a gramatické úpravy textu, sumarizáciu niektorých článkov, ako podklad pre výrobu abstraktu, na vygenerovanie kostry práce (názvy kapitol a sekcií) a na úpravu LaTeX tabuliek. Za správnosť výslednej podoby textu zodpovedá autorka.

Acknowledgments: I would like to sincerely thank my supervisor, Mgr. Vladimír Boža, PhD., for his support throughout this thesis. His prompt responses, thoughtful suggestions, and patient guidance helped me stay on track. I am also thankful for the access to GPU resources he provided. I'm grateful not only for his expertise, but also for his approachable and encouraging attitude.

Abstrakt

V moderných neurónových sieťach väčšina výpočtového úsilia spočíva v násobení váhových matíc. Hoci metódy neštruktúrovaného orezávania dokážu odstrániť veľkú časť váhových parametrov bez veľkej straty presnosti, nepravidelná riedkosť, ktorú spôsobujú, má za následok potrebu ukladania množstva indexov a neefektívne využitie súčasného hardvéru, takže sa nedarí dosiahnuť praktické zrýchlenie. Štruktúrované orezávanie — najmä blokové orezávanie — sa javí ako sľubná alternatíva, ktorá dokáže lepšie využiť hardvérové možnosti na urýchlenie výpočtov, často však za cenu vyššej straty presnosti modelu. Naša práca sa zameriava na iteratívne blokové orezávanie, pričom skúma, či blokovo-riedke modely získané iteratívnym prístupom dosahujú vyššiu presnosť než blokovo-riedke modely orezané v jednom kroku, čo by zmierňovalo nutný kompromis medzi presnosťou a výpočtovou efektivitou. Na príklade neurónovej siete s architektúrou Wide Residual Network a datasete CIFAR-100 vyhodnocujeme rôzne stratégie blokového orezávania, ako orezávanie v jednom kroku, iteratívne orezávanie, postupné orezávanie, ale aj metódu s názvom AC/DC, využívajúc rôzne metriky dôležitosti blokov. S použitím iteratívneho a postupného blokového orezávania dosahujú v našich experimentoch riedke modely (pri vysokých úrovniach riedkosti, ako napríklad 90%) výrazne vyššiu presnosť, než s použitím blokového orezávania v jednom kroku. Naše zistenia potvrdzujú, že blokové orezávanie v kombinácii s vhodne zvolenou iteratívnou stratégiou dokáže zachovať vysokú presnosť modelu, čím naša práca prispieva k prepojeniu medzery medzi teoretickou možnosťou urýchlenia vďaka blokovo-riedkym štruktúram a praktickou využiteľnosťou blokovo-riedkych modelov.

Kľúčové slová: blokové orezávanie, iteratívne orezávanie, neurónové siete, štruktúrovaná riedkosť, Wide Residual Network

Abstract

In modern neural networks, a large portion of computational effort is consumed by matrix multiplications involving weight matrices and activations. While unstructured pruning techniques can remove a high percentage of individual weights with minimal loss in accuracy, they suffer from high indexing overhead and limited compatibility with existing hardware due to irregular sparsity patterns. In contrast, structured pruning — particularly block pruning — has emerged as a promising alternative that aligns better with hardware acceleration, though it often incurs greater accuracy degradation. This thesis investigates iterative block pruning strategies, particularly whether an iterative approach can improve the accuracy of block-sparse models, thereby narrowing the trade-off between performance and computational efficiency. Using a Wide Residual Network and the CIFAR-100 dataset, we evaluate global block pruning under various configurations: one-shot, iterative, and gradual pruning schedules, as well as a method called AC/DC, combined with different block importance metrics. Our results show that iterative and gradual block pruning significantly outperform one-shot approaches at high sparsity levels such as 90%. Overall, our findings demonstrate that block pruning, when combined with carefully designed iterative strategies, can preserve high model accuracy while inducing sparsity patterns more amenable to hardware acceleration — thereby bridging the gap between theoretical efficiency and practical usability.

Keywords: block pruning, iterative pruning, neural networks, structured sparsity, Wide Residual Network

Contents

Introduction

0.1 Motivation

Modern deep neural networks have emerged as powerful tools across various domains, largely due to their ability to model complex data relationships. However, their high performance often comes with substantial computational and memory demands. Modern architectures may contain millions or even billions of parameters, requiring vast numbers of arithmetic operations for both training and inference. For instance, the GPT-3 model required approximately 175 billion parameters to be evaluated, with its successors being even larger. Such large-scale models demand substantial resources, often necessitating training on supercomputers, with costs amounting to millions of dollars per training run ([1]).

According to the authors of paper [19], the rate of progress in computational capability delivered by Moore's law, Dennard scaling, and architectural specializations with GPUs and specialized machine learning accelerators is gradually slowing, eventually approaching its natural limits. Furthermore, the deployment of expansive and precise deep learning models into computing environments constrained by resources, such as mobile phones and smart cameras, presents several significant challenges. These challenges include limited on-device memory, as well as the substantial consumption of power due to high memory accesses and arithmetic operations, leading to heat dissipation and reduced battery life.

Efforts to address the above-mentioned issues have led to the development of model compression techniques aimed at improving computational and memory efficiency while preserving the model's accuracy. Common strategies include quantization, low-rank factorization, parameter sharing, knowledge distillation, and pruning. Our work focuses on neural network pruning, also known as sparsification.

Pruning reduces the size of a neural network by eliminating redundant or non-essential parameters. This offers several key benefits:

- **Lower memory usage:** Smaller models consume less storage, easing deployment on memory-constrained hardware.

- **Inference speed:** Pruned models require fewer operations, making them faster during inference.
- **Energy savings:** Reduced computation translates to lower power consumption.
- **Enhanced robustness:** Pruning can help prevent overfitting and enhance the model's generalization ability.

Additional potential benefits of pruning include better interpretability. Sparser models are often easier to analyze, and identifying which weights or connections are most important can provide valuable insights into model behavior. Furthermore, certain pruning strategies may also reduce training time.

Figure 1: Examples of pruning an 8×8 matrix

Unstructured pruning often results in sparse patterns that are difficult to exploit efficiently due to the overhead of storing and processing sparse indices. To address this, structured pruning methods have gained attention for their ability to produce hardware-friendly sparsity. Among them, block pruning strikes a balance between model compression and computational efficiency by enforcing sparsity in fixed-size blocks, making it more amenable to parallel processing and optimized hardware implementation.

NVIDIA has demonstrated speedups with block-sparse matrices. Examples from technical blogs [28, 50] show practical GPU speedups for block-sparse models. For instance, a noticeable GPU speedup is observed even at 25% sparsity — something not typically seen with unstructured sparsity ([50]). These applications highlight that block sparsity is not only theoretically appealing but also offers practical acceleration gains.

An example of both unstructured and block pruning is illustrated in Figure 1 using an 8×8 matrix with floating-point values. The elements are colored based on their magnitude — positive values appear in shades of purple, negative values in shades of green, with darker colors indicating higher magnitudes. The first image shows the original matrix. To its right is the result of unstructured pruning at 75% sparsity, retaining the 25% of weights with the highest magnitudes. The second row shows the results of block pruning at 75% sparsity using block sizes of 2×2 and 4×4, respectively, based on the *abs max* metric (as described later in Section 1.3.5). In practice, the matrices involved are typically much larger.

0.2 Objectives and Contributions of this Thesis

The primary objective of this thesis is to investigate the impact of various pruning strategies on the performance of a Wide Residual Network [52, 24] trained on the CIFAR-100 dataset [25], with a focus on Top-1 accuracy. To this end, we train a baseline dense model and a series of pruned models using different pruning techniques.

Specifically, we explore a global block pruning technique — using a variety of metrics for block elimination and a block size of 8×8 — and evaluate its performance against global unstructured pruning and the dense baseline. We evaluate the performance of block pruning methods under different pruning schedules: a non-iterative (one-shot) approach and an iterative approach. The primary goal is to determine whether the iterative strategy offers any advantages over one-shot pruning in the context of block pruning.

Additionally, this thesis investigates the AC/DC pruning method, introduced in the paper *AC/DC: Alternating Compressed/Decompressed Training* [43], particularly

in conjunction with block pruning, to assess its effectiveness in preserving model accuracy. Gradual pruning, as proposed by Zhu and Gupta [54], is also considered as an alternative strategy to be compared against the more abrupt pruning schedules.

0.3 Structure of the Thesis

Chapter 1, *Background and Related Work*, introduces the basic context of neural networks and deep learning, with a specific focus on the Wide Residual Network (WRN), which is used in our experiments. The chapter also gives a brief overview of model compression techniques and discusses the theory of pruning in greater depth, including structured and unstructured pruning, as well as iterative versus one-shot pruning strategies. Furthermore, it delves into blocked pruning, presenting an overview of related prior work. The AC/DC technique and gradual pruning are also described. Together, this chapter lays the groundwork for understanding the context of the experiments conducted in this thesis.

Chapter 2, *Methodology*, outlines the experimental setup, including the dataset used, details of the WRN architecture, and the evaluation metrics. It also presents the training and validation protocols, the workflow for training and pruning, and the hardware specifications. The chapter goes on to describe the implementation of the pruning strategies employed.

Chapter 3, *Experiments and Results*, presents the experimental findings of this thesis. It provides a detailed analysis of the impact of various pruning strategies on model accuracy, including a breakdown of results for each pruning technique evaluated.

Finally, Chapter 4, *Discussion*, summarizes the key findings and insights derived from the experiments, and discusses potential directions for future research and extensions of the current work.

Chapter 1

Background and Related Work

1.1 Neural Networks and Deep Learning Fundamentals

Neural networks and deep learning form the backbone of modern artificial intelligence and machine learning, drawing inspiration from the structure and function of the human brain. These methods are designed to tackle complex tasks such as image recognition, natural language processing, and game playing.

We do not provide a detailed explanation of how neural networks work, as we assume the reader is already familiar with the concept. For those seeking an introduction or further background, we recommend consulting relevant literature, such as books [10, 7].

Deep learning is a subset of machine learning in which the neural network consists of many layers — hence the term deep neural networks (DNNs). It addresses the challenge of extracting high-level, abstract features from raw data. The authors of the book [7] explain the concept of deep learning using the illustration in Figure 1.1: a deep learning system interprets complex inputs, such as an image of a person, by building up meaning through a series of simpler representations. Instead of trying to directly map raw pixel data to high-level concepts like object identity (a task far too complex to solve in one step), deep learning models break the problem down. Each layer in the model learns to extract increasingly abstract features: edges form the basis for contours and corners, which then compose object parts, ultimately leading to the recognition of whole objects. This layered structure transforms an otherwise intractable task into a sequence of manageable transformations, enabling machines to make sense of raw sensory input ([7]).

There are several fundamental types of neural networks, each suited to different types of tasks. A brief overview of the most common types is described in Table 1.1. For more detailed explanations, refer to the literature [53, 7].

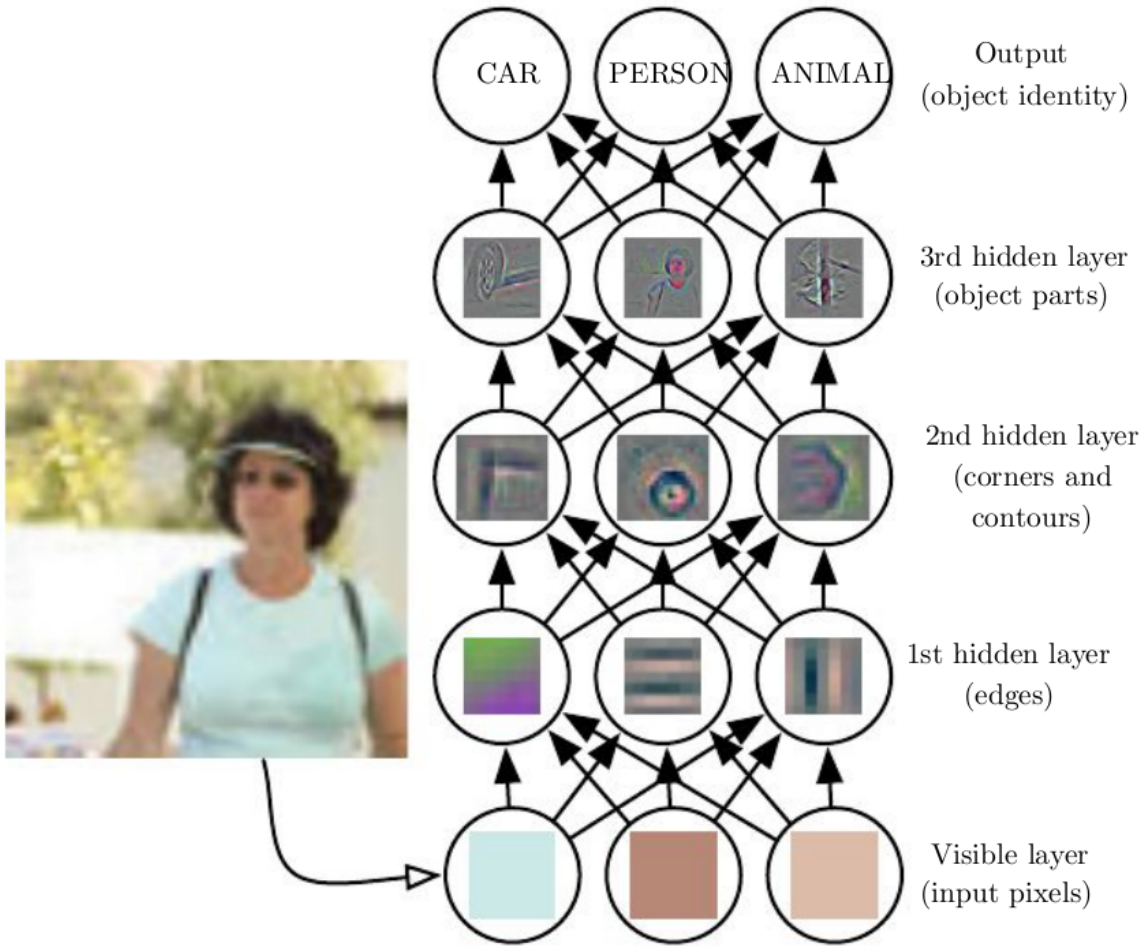

Figure 1.1: Illustration of a deep learning model, source: [7]

In our work, we employ a Wide Residual Network, which is a specific type of Convolutional Neural Network (CNN).

1.1.1 Wide Residual Networks (Wide ResNets)

Deep residual networks have achieved remarkable performance across various image recognition tasks. Despite their success, deeper networks face challenges such as vanishing/exploding gradients and slow training. Techniques like skip connections, knowledge transfer, better optimizers, and improved initialization strategies have been proposed to address these issues. However, very deep networks still suffer from diminishing feature reuse (where features learned in earlier layers are not effectively utilized by later ones), leading to inefficient training.

The paper [52] introduced Wide Residual Networks (WRNs) as an improvement over existing deep residual networks. WRNs address the limitations of deep networks by increasing the width of residual networks and decreasing the depth, showing significant improvements in performance and training efficiency. For example, their 16-layer WRN

Name	Description	Example use case
Multilayer Perceptrons (MLPs)	A class of feedforward artificial neural networks composed of multiple layers of neurons.	Classification problems, regression tasks.
Convolutional Neural Networks (CNNs)	Uses convolutional layers to detect features like edges and textures.	Image classification, object detection, video processing.
Recurrent Neural Networks (RNNs)	Networks with recurrent connections (loops) that allow them to maintain a form of memory over previous inputs.	Sequential data tasks such as time-series forecasting and language modeling.
Generative Adversarial Networks (GANs)	Consist of two networks (generator and discriminator) that compete against each other to generate realistic data.	Generation of photorealistic images, image reconstruction.
Transformer Networks	The core idea behind the Transformer model is the attention mechanism; modern LLMs (e.g., GPT) are transformer-based.	Natural language processing tasks such as translation, summarization, and text generation.

Table 1.1: Common types of neural networks and their typical use cases

achieved similar accuracy to a 1000-layer thin deep residual network, while being several times faster to train ([52]).

Standard Deep Residual Networks (ResNets) [13] are composed of sequentially stacked residual blocks. The architecture from paper [14] features two main types of blocks:

- Basic block B(3,3) — two sequential 3×3 convolutions, each preceded by batch normalization and ReLU.
- Bottleneck block — a 3×3 convolution sandwiched between two 1×1 convolutions for dimensionality reduction and expansion.

Since bottleneck blocks aim to make networks thinner, WRNs focus exclusively on the basic block structure. The widening of the residual blocks is achieved by increasing the number of feature planes (also known as feature maps or channels).

A WRN is typically denoted as WRN-N-k, where N is the depth and k is the widening factor (how many times wider each layer is compared to the original ResNet design [14], which corresponds to k=1). The general structure of wide residual networks

Group name	**Output size**	**Block type** = B(3, 3)
conv1	32×32	$[3 \times 3, 16]$
conv2	32×32	$\begin{bmatrix} 3 \times 3, 16 \times k \\ 3 \times 3, 16 \times k \end{bmatrix} \times n$
conv3	16×16	$\begin{bmatrix} 3 \times 3, 32 \times k \\ 3 \times 3, 32 \times k \end{bmatrix} \times n$
conv4	8×8	$\begin{bmatrix} 3 \times 3, 64 \times k \\ 3 \times 3, 64 \times k \end{bmatrix} \times n$
avg-pool	1×1	$[8 \times 8]$

Table 1.2: Structure of wide residual networks, source: [52]. Final classification layer is omitted for clarity and ResNet block of type B(3, 3) is used in this example.

is outlined in Table 1.2: it consists of an initial convolutional layer (conv1), followed by three groups (conv2, conv3, and conv4) of n residual blocks, then followed by average pooling and a final classification layer ([52]).

WRNs are computationally more efficient than their deeper counterparts because of the way GPU computations are handled. Thin and deep residual networks with small kernels are inherently sequential and not suitable for GPU parallelization. In contrast, widening the layers leverages the GPU's strength in processing large tensors in parallel.

1.2 Overview of Model Compression Techniques

Model compression techniques aim to reduce the size, memory footprint, and computational demands of machine learning models while preserving their accuracy as much as possible. Common techniques include pruning, quantization, operator factorization, and parameter sharing. Here, we provide a brief explanation behind each of the techniques mentioned above. As pruning is the primary focus of our work, we will go into more detail on pruning in the next section.

Although some definitions of model compression are limited to methods that reduce the size of an already trained model — without involving the training of a new one — the terminology in the literature is not consistent. In this section, we also mention model down-sizing approaches, such as Knowledge Distillation, which, although not universally classified as a compression method, shares similar objectives.

Quantization

Quantization [21] is a technique of reducing the numerical precision of values in a model, such as weights, activations, or gradients. Instead of relying on usual high-precision data types like 32-bit or 16-bit floating-point, lower-precision formats — such as 8-bit or even 4-bit integers — are used.

The core challenge in quantization is to effectively map a high-precision floating-point range to the limited representation space of a low-precision data type. For instance, an `int8` data type can represent only 256 discrete values, whereas `float32` covers a vastly larger continuous range. To address this, affine quantization scheme is often used ([20]).

Reducing the bit-width of values decreases the model's memory footprint and enables faster computation, particularly for operations like matrix multiplication that benefit from efficient integer arithmetic. Additionally, integer quantization enables deploying models on embedded devices that support only integer types ([21]).

Operator factorization

Operator factorization refers to techniques that reduce the computational cost and memory requirements of neural networks by decomposing complex mathematical operators into smaller, more efficient ones. Examples include Low-Rank Factorization and Tensor Train Decomposition.

Low-Rank Factorization approximates a large weight matrix $W \in \mathbb{R}^{m \times n}$ by decomposing it into the product $W \approx UV$ of two smaller matrices $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{k \times n}$, where $k \ll \min(m, n)$.

Parameter sharing

Parameter sharing is the practice of exploiting the same parameters across different parts of a model, rather than learning separate parameters for each part. Typically, weight sharing is used. It is commonly used in models like CNNs, where the same filter is applied to different regions of an image, and RNNs, where the same weights are shared across time steps.

Knowledge Distillation

Knowledge Distillation [18] is a technique in which a large, complex model (referred to as the "teacher") is first trained on a large dataset, and then the knowledge is transferred to a smaller model (the "student"). Instead of learning solely from the ground truth labels, the student model is trained using the soft targets (i.e., probability distributions) generated by the teacher. These soft targets allow the student to capture richer

information, such as the relative similarities between classes, that are not conveyed by the hard labels alone ([18]).

Pruning

Pruning is the concept of removing subsets of parameters from an existing model. More details will be provided in the following section.

1.3 Pruning of Neural Networks

Pruning, also known as sparsification, is a technique used to reduce the size and complexity of a neural network by eliminating certain neurons, weights, or connections. Typically, parameters that are deemed unnecessary or less important for the model's performance are pruned. The process is conceptually similar to synaptic pruning in the human brain, where less useful neural connections are removed to improve overall efficiency.

Deep learning models are traditionally dense and heavily over-parameterized. Research shows it is often possible to reduce the number of parameters significantly without sacrificing accuracy. In fact, sparse models tend to outperform dense models when provided with equivalent parameter budgets ([54]). According to Hoefler et al. [19], this may be linked to findings that over-parameterized models exhibit a "convexity-like property" that benefits the convergence of gradient descent methods used for training.

1.3.1 Targets of Pruning

Pruning techniques can be applied to various elements of a deep learning model. The authors of the work *Sparsity in Deep Learning* [19] distinguish between model and ephemeral sparsification. Model sparsification maintains a consistent sparsity pattern across multiple inference or training passes — commonly targeting weights or neurons. In contrast, ephemeral sparsification is applied dynamically during computation, affecting only individual examples. An overview of elements in a deep neural network that are commonly targeted for sparsification is shown in Figure 1.2.

Furthermore, weight pruning can be applied in two main ways:

(a) **Layer-wise**: Weights are pruned independently within each individual layer. This approach is useful when layer-wise importance cannot be reliably estimated.

(b) **Globally**: Weights are pruned across all pruned layers based on a global criterion. In this case, the target sparsity does not need to be achieved within each individual layer. Some layers may remain dense, while others exceed the target

sparsity. The key requirement is that the overall target sparsity is met when aggregated across all pruned layers.

In our work, we adopt model sparsity, specifically, we prune weights globally. We primarily explore block pruning, which is a structured form of pruning, and include unstructured pruning for comparison. Therefore, following passages present the theoretical background related to **model sparsification**, with a main focus on **weight pruning**.

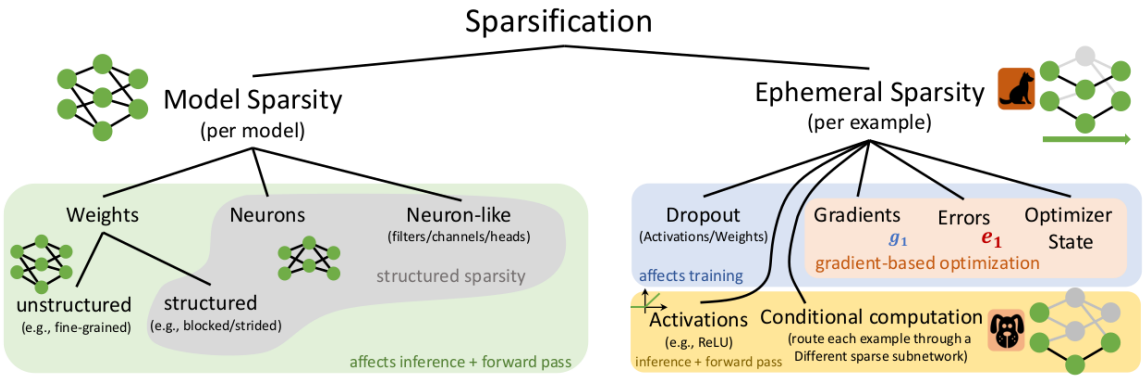

Figure 1.2: Overview of DNN elements to sparsify, source: [19]

1.3.2 Pruning as Regularization

Sparsification might serve as a form of regularization and thus improve generalization. Hoefler et al. [19] claim that certain sparsification schemes follow Occam's Hill, as illustrated in Figure 1.3, where initially, as the model becomes slightly sparse, accuracy tends to improve due to the elimination of redundant information. As sparsity increases further, accuracy plateaus or declines slightly, until at very high levels of sparsity, model performance drops sharply.

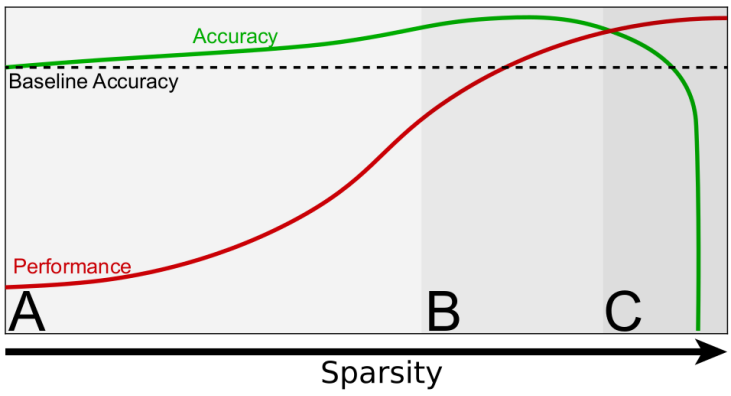

Figure 1.3: Sketch of test accuracy and performance vs. sparsity, source: [19]

1.3.3 Pruning Schedule

An important consideration in pruning is determining the point in the training cycle at which it should be applied. The authors of *Sparsity in Deep Learning* [19] identify three main classes of pruning schedules:

- Train-then-sparsify: A dense model is first trained to convergence. Pruning is then applied, followed by fine-tuning of the resulting sparse model. According to the paper [19], this is the most widely used approach. One of its main advantages is the ability to directly compare the accuracy of the final sparse model with its original dense counterpart.

- Sparsify-during-training: Pruning is introduced before the dense model has fully converged. This can lead to reduced training time and computational cost.

- Fully-sparse training: The model is initialized in a sparse state and remains sparse throughout training, with the possibility of dynamically adding or removing parameters during the process.

For further details, see the paper [19]. In our work, we mainly utilize the train-then-sparsify schedule (for our base experiments 3.1.2), although the AC/DC pruning (3.1.3) can be seen as a hybrid between the fully-sparse training and sparsify-during-training schedules.

In addition to the schedule type, pruning itself can be applied in various ways: one-shot, iterative, gradual, or other.

In **one-shot pruning**, the entire pruning process happens in a single step. While simple to implement, it can disrupt training abruptly and lead to performance degradation.

Iterative pruning removes weights progressively over several training steps. Typically, the sparsity level of the model is increased in each pruning step, followed by a fine-tuning period. Iterative pruning allows for a smoother transition to a sparse model. Recent research suggests it might be more effective at maintaining model accuracy compared to one-shot pruning.

Gradual pruning increases the sparsity level slowly over time. The distinction between iterative and gradual pruning is not sharply defined, but gradual pruning typically implies a more continuous pruning schedule. Nevertheless, it can also be seen as a highly granular form of iterative pruning.

Authors of the work [19] explain that "Even though many pruning schemes pick the least important elements, the degradation of model quality greatly varies." Therefore, fine-tuning (retraining) after pruning is generally crucial to restore or maintain performance.

1.3.4 Structured vs. Unstructured Pruning

Unstructured pruning is a type of pruning that operates at the level of individual elements, such as pruning of individual weights without considering their position or alignment within the tensor structure.

In theory, pruning reduces memory usage and computational cost. Pruning a fixed percentage of weights across all layers should reduce the computational cost proportionally. However, unstructured pruning rarely delivers this expected speed-up in practice. The irregular sparsity patterns induced by unstructured pruning cause inefficiencies in computation, "due to irregular memory access, storage overheads, as well as the inability to take advantage of array data-paths in modern processors", as noted by Narang et al. [40].

When elements of a weight matrix are removed without structural constraints, the positions of the remaining non-zero elements must be stored, which incurs significant overhead. Efficient storage of these elements requires specialized indexing formats. Depending on the sparsity level, schemes like bitmaps, coordinate offsets, and compressed sparse row/column formats, using additional metadata to encode the positions of non-zero elements, offer trade-offs in storage overhead and access efficiency ([19]).

Thus, although unstructured pruning reduces parameter count and theoretical FLOPs, it often fails to deliver proportional memory savings and runtime benefits on typical hardware.

To address these limitations, structured pruning was introduced. Rather than removing individual elements, structured pruning removes entire components such as neurons, channels, filters, attention heads, or groups of weights (e.g., blocks or strides). This results in regular, hardware-friendlier sparsity patterns that align more closely with the memory and compute architecture. By enforcing sparsity patterns, structured sparsity reduces storage overhead and simplifies processing. However, the structured approach may limit the flexibility of the sparsification process, often leading to reduced accuracy due to fewer degrees of freedom compared to unstructured pruning. Furthermore, to fully benefit from the speedup and memory savings of structured pruning, specialized implementations are required.

In our work, we focus on blocked pruning, a type of structured weight pruning. Existing benchmarks and reports [28, 50] show that block-sparse models can achieve substantial GPU speedups in practice.

1.3.5 Blocked Pruning

Blocked pruning (also called *block pruning*) is a technique where entire **groups** of weights ("blocks") are pruned together, rather than removing individual weights. Blocks are an intuitive concept and their definition may vary. Examples of blocks are illustrated in Figure 1.4. In a two-dimensional matrix, blocks typically correspond to rectangular submatrices, often square in shape. In the one-dimensional case, a block consists of a group of n consecutive elements. For higher-dimensional tensors, the notion of a block becomes more flexible and less clear; however, the key requirement is that the pruned groups adhere to a well-defined and easily describable structure. Our approach to block pruning of four-dimensional weight tensors is described later in Section 2.2.2. Traditional block pruning applies blocks of *uniform* fixed size either to individual layers or across the entire network. In our work, we use blocks of a fixed size 8×8, as also described later in Section 2.2.2.

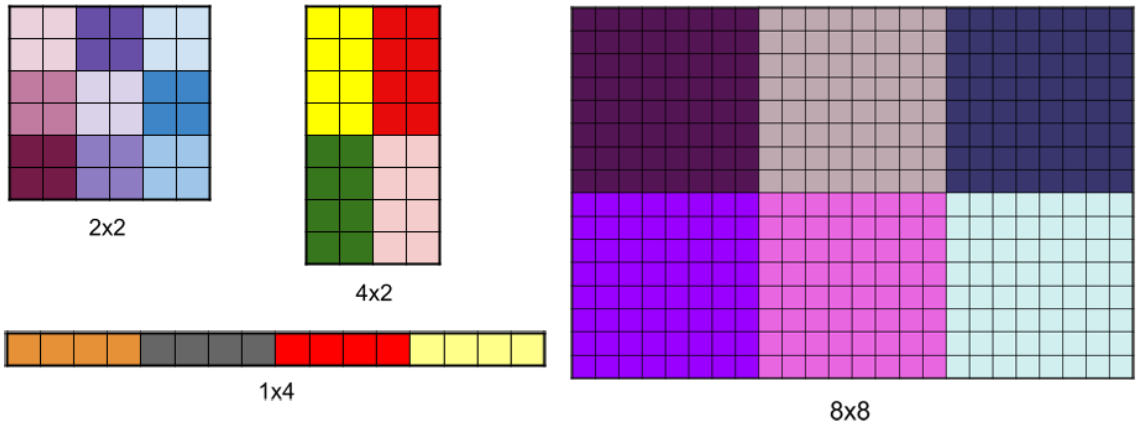

Figure 1.4: Examples of blocks in two-dimensional and one-dimensional cases

Traditionally, blocks are pruned based on some importance criteria. If a block is deemed unimportant, all weights within that block are set to zero. A key question, then, is how to determine which blocks are less important. In global unstructured pruning, individual weights are usually removed based on their magnitude — typically their absolute value (or equivalently, their L_1 norm). In contrast, block pruning removes entire blocks from the weight matrix/tensor, which requires defining a measure of **block magnitude** (also referred to as block significance, block importance, block representative). Popular approaches are:

- **abs max**: Selects the weight with the maximum absolute value within the block.
- **abs min**: Selects the weight with the minimum absolute value within the block.
- **L1**: Uses the L_1 norm (the sum of absolute values) of all weights in the block.

- **L2**: Uses the L_2 norm (the square root of the sum of squares) of all weights in the block.

The result is a single scalar value considered the block magnitude.

A very simplified version of block pruning using the *abs max* block criterion is illustrated in Figure 1.5.

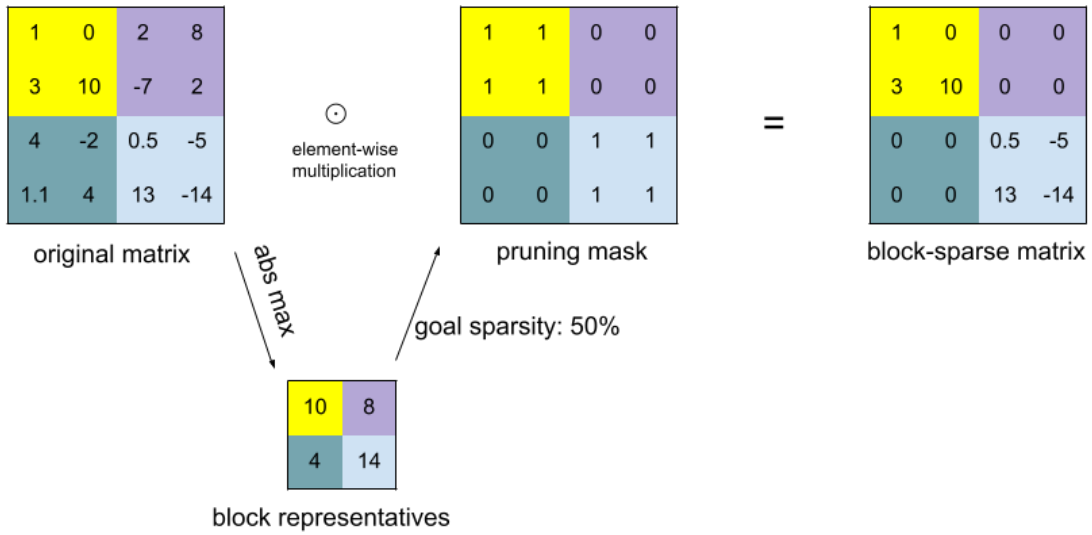

Figure 1.5: Example of block pruning a simple 4x4 matrix with a block size of 2x2, block criterion *abs max*, and a goal sparsity of 50%

For example, Narang et al. [40] explain they use the following block pruning procedure: "In order to prune blocks, we pick the weight with the maximum magnitude as a representative for the entire block. If the maximum magnitude of a block is below the current threshold, we set all the weights in that block to zeros."

The pruning **threshold** is predetermined: it can be manually selected by the user, defined as part of a pruning schedule, or computed based on a target sparsity level. For example, if we aim to achieve 50% sparsity in the matrix shown in Figure 1.5, the threshold would be set such that 50% of the block magnitudes fall below it. In this case, a threshold of 10 would result in pruning all blocks with representatives less than 10, yielding a matrix with 50% sparsity. However, the exact implementation may vary — for instance, in a variant where all blocks with values *less than or equal* to the threshold are pruned, the threshold would instead be 8.

The process of zeroing out elements in a weight matrix (or tensor) is typically carried out using a so-called **pruning mask**. A pruning mask is a matrix or tensor of the same shape as the original, containing only zeros and ones. During pruning, instead of directly setting weights to zero, zeros are placed in the pruning mask at the indices corresponding to the weights that should be pruned (based on the result

of a comparison with the threshold). The remaining elements of the mask are ones, indicating which weights should be retained. The actual pruning effect is achieved by performing element-wise multiplication between the original weight matrix (or tensor) and the pruning mask. The purpose of the pruning mask is to maintain sparsity across multiple training epochs. It ensures that once a weight has been pruned, it remains zero and is not updated again during fine-tuning, thereby preserving the sparsity pattern.

To summarize, the typical global block pruning procedure consists of the following steps:

Algorithm 1 Global block pruning in pseudocode

```
Select a suitable block size
Choose a block importance metric
Determine the global pruning threshold (manually or based on target sparsity)
for each pruned layer in the neural network do
    Initialize the pruning mask with all elements set to 1
    for each block in the weight matrix/tensor do
        Compute block magnitude using the chosen metric
        if magnitude < threshold then
            Set all entries in the pruning mask corresponding to this block to 0
        end if
    end for
end for
Apply pruning: element-wise multiply the pruning masks with the original weights
```

1.4 Related Works

Pruning of neural networks has its roots in early research from the 1980s and 1990s. Papers such as *Optimal Brain Damage* [27] and *Optimal Brain Surgeon* [12] demonstrated the effectiveness of weight pruning in reducing model complexity. In these works, pruning was performed using second-order Taylor approximations of the increase in the loss function when a weight is set to zero. These are considered early forms of **gradient-based pruning**, where weights that contribute least to the loss function are removed.

Subsequent research [5, 11, 6] refined pruning techniques, with some works aiming to reduce computational overhead and improve scalability ([4]). Other works [9, 31] emphasized dynamic pruning and recovery mechanisms. **Magnitude-based** pruning has gained significant popularity ([11, 44, 39, 54]) due to good scalability and computational efficiency. In this approach, weights with the smallest magnitudes are pruned

under the assumption that they contribute least to the model's output. Approaches such as gradient-based pruning or Bayesian methods ([38]) might yield better results, but they tend to have higher computational and implementation costs.[1]

Some recent research has focused on the adversarial robustness of compressed models ([8]), while other works [49, 32] explore techniques for direct sparsity learning — methods that automatically identify which parts of a neural network can be pruned during training without manual intervention. Additionally, reinforcement-learning-based approaches have been introduced to dynamically determine pruning ratios ([16]).

Structured pruning has attracted significant attention due to its potential for improving computational efficiency and reducing memory usage. A large body of work has focused on filter pruning, which removes entire filters from convolutional layers to compress Convolutional Neural Networks effectively. Notable examples include studies [49, 33, 15, 29]. Similarly, channel pruning — the removal of entire feature map channels — has been investigated ([17]). Further works have explored structured pruning in the context of CNNs, for instance, Yu et al. [51] propose the *Scalpel* framework, and Mao et al. [35] use a simple threshold-based technique to induce structured sparsity in CNNs. Beyond CNNs, other architectures have been considered as well: Wen et al. [48] modify the structure of Long Short-Term Memory Networks (LSTM) to reduce memory footprint, while Wang et al. [47] apply structured pruning techniques to Large Language Models.

Block pruning has also been explored as a means to induce structured sparsity. For example, Narang et al. [40] apply block pruning to Recurrent Neural Networks using a monotonically growing threshold during training. Block pruning has also been investigated in the context of Transformer models, as demonstrated in work [26].

Several studies have addressed the trade-off between sparsity structure and model quality by proposing innovative approaches. For instance, *Hierarchical Block Sparse Neural Networks (HBsNN)* [46] combine both unstructured and structured sparsity patterns. The *Tetris* method [22] clusters the irregularly distributed low-magnitude weights into structured groups by reordering the input/output dimensions, resembling the tile-matching game. Dao et al. [3] introduce a new class of structured matrices called *Monarch* matrices. Mishra et al. [37] present *Sparse Tensor Cores* and propose a 2:4 structured sparsity pattern designed to better utilize GPU architectures. Meanwhile, Lin et al. [30] propose the *1×N pruning pattern* for CNNs, targeting consecutive sets of N output kernels with shared input channel indices.

Iterative pruning strategies have also gained traction. For example, *Iterative Clustering Pruning for Convolutional Neural Networks* [2] improves pruning effectiveness by refining pruning decisions over multiple iterations.

[1] Additionally, **random pruning** is sometimes used as a baseline. In this technique, weights are pruned without regard to their contribution or importance.

However, despite these advances, iterative block pruning, specifically, remains an under-explored area. This work aims to fill this gap and provide benchmarks to support future research.

1.4.1 AC/DC: Alternating Compressed/DeCompressed Training

The paper titled *AC/DC: Alternating Compressed/DeCompressed Training of Deep Neural Networks* [43] presented at NeurIPS 2021 by Peste et al., introduces a novel approach for efficiently training deep sparse neural networks. The AC/DC method alternates between compressed (sparse) and decompressed (dense) versions of the model during training. This technique enables the simultaneous co-training of sparse and dense models, resulting in sparse–dense model pairs by the end of the training process. In contrast to traditional pruning schedules, which typically involve training a dense model followed by pruning and a fine-tuning phase, AC/DC reduces the overall training time and cost.

The authors provide a theoretical foundation for their method and demonstrate that AC/DC outperforms existing sparse training techniques in terms of accuracy when operating within similar computational budgets. Remarkably, at high sparsity levels, their findings suggest that AC/DC can even exceed the performance of methods that rely on accurate pre-trained dense models.

The AC/DC method is formally described in pseudocode 2, as presented in the original paper [43]. Here, the Top-k operator T_k is a function that retains the k largest-magnitude entries of a weight vector θ, and sets the rest to zero. While the authors assume weights are represented as vectors $\theta \in \mathbb{R}^N$ in the algorithm, based on the conceptual explanation and applications in the paper [43], we believe the approach can be generalized to weight matrices and higher-dimensional tensors, and a different pruning function can be used instead of the Top-k operator.

As outlined in the algorithm and further elaborated in the paper, the authors primarily experiment with global unstructured magnitude-based pruning. Additionally, they also explore semi-structured pruning, using the 2:4 sparsity pattern [37].

The training procedure for AC/DC can be summarized as follows: Training begins with a dense warm-up period lasting Δ_w epochs using standard dense training. Then, the method alternates multiple times between compressed (sparse) and decompressed (dense) training phases of fixed durations Δ_c, Δ_d epochs, respectively. At the start of each sparse phase, new pruning masks are computed and applied. At the end of the sparse phase, these masks are reset to all ones (i.e., no pruning), enabling the following dense phase to optimize over the full dense support. The training process concludes with a compressed fine-tuning phase of Δ_f epochs, resulting in the final sparse model.

The authors evaluate AC/DC across various network architectures, including ResNet and MobileNetV1, and datasets such as CIFAR-100 [25] and ImageNet, covering both image classification and language modeling tasks. They report validation accuracy and compute the number of floating-point operations (FLOPs) required for training and inference, comparing their approach with typical post-training pruning methods.

The appendix of the paper includes convergence proofs, extended experiments, and implementation specifics. Notably, they evaluate AC/DC on Wide ResNet-28-10 with CIFAR-100, which allows for a direct comparison with our own experimental results.

For more detailed information, refer to the full paper [43].

Algorithm 2 Alternating Compressed/Decompressed (AC/DC) Training, source: [43]

Require: Weights $\theta \in \mathbb{R}^N$, data S, sparsity k, compression phases C, decompression phases D

Train the weights θ for Δ_w epochs ▷ Warm-up phase
while epoch $\leq$ max epochs **do**
  if entered a compression phase **then**
    $\theta \leftarrow T_k(\theta, k)$ ▷ Apply compression (Top-k) operator on weights
    $m \leftarrow \mathbf{1}[\theta_i \neq 0]$ ▷ Create masks
  end if
  if entered a decompression phase **then**
    $m \leftarrow \mathbf{1}_N$ ▷ Reset all masks
  end if
  $\theta \leftarrow \theta \odot m$ ▷ Apply the masks (ensure sparsity for compression phases)
  $\tilde{\theta} \leftarrow \{\theta_i \mid m_i \neq 0, 1 \leq i \leq N\}$ ▷ Get the support for the gradients
  for x mini-batch in S **do**
    $\theta \leftarrow \theta - \eta \nabla_{\tilde{\theta}} f(\theta; x)$ ▷ Optimize the active weights
  end for
  epoch $\leftarrow$ epoch + 1
end while
return θ

1.4.2 Gradual Pruning

The paper titled *To prune, or not to prune: exploring the efficacy of pruning for model compression* [54] investigates the effectiveness of model pruning for compressing deep neural networks. The authors aim to determine whether pruning large neural networks to induce sparsity (resulting in large-sparse models) is more effective than designing smaller, densely connected networks (small-dense models) when both have the same memory footprint. They find that large-sparse models consistently outperform small-

dense models with equivalent memory usage across various architectures, including CNNs and LSTMs.

Moreover, the authors introduce a simple, gradual pruning method that incrementally increases sparsity during training. This method is governed by a monotonically increasing sparsity function $s(t)$, defined as:

$$s(t) = s_f + (s_i - s_f)\left(1 - \frac{t - t_0}{n\Delta t}\right)^3 \quad \text{for} \quad t \in \{t_0, t_0 + \Delta t, \ldots, t_0 + n\Delta t\}. \tag{1.1}$$

Here, $s(t)$ represents the sparsity at training step t. Pruning begins at training step t_0, starting from an initial sparsity value s_i (typically 0). Sparsity is increased at regular intervals of Δt training steps, over the course of n pruning steps, until the final target sparsity s_f is reached. This technique is easy to implement and requires minimal tuning, making it applicable across a wide range of models and datasets ([54]).

We adopt the gradual sparsity function described above in our experiments. The combination of gradual pruning with block pruning is largely unexplored; to our knowledge, only one prior work [23] has applied this combination.

Chapter 2

Methodology

This chapter outlines the methodology used to apply the selected pruning strategies and investigate their impact on the performance of the network. It is organized into two main sections: an overview of the experimental setup and the implementation details. The Overview of Experimental Setup provides a description of the datasets used, the architectural details of the Wide Residual Network model, the training process, the train/prune workflow, evaluation metrics, and the hardware equipment. The Implementation section contains details about the existing libraries and codebases utilized in our experiments, along with a comprehensive description of our custom implementation of the selected pruning strategies. Together, the chapter provides the necessary context for the experimental setup and establishes the foundation for reproducibility.

2.1 Overview of Experimental Setup

2.1.1 Datasets Used

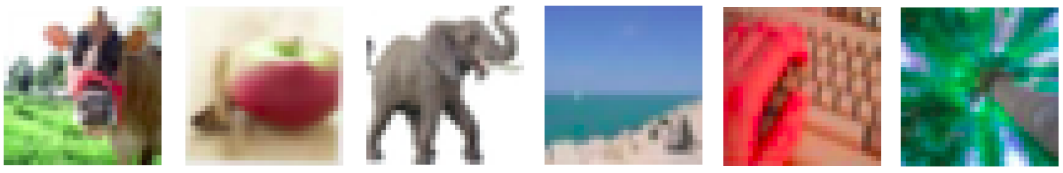

Figure 2.1: CIFAR-100 examples, source: [25]

We use **CIFAR-100** dataset [25], which consists of tiny 32x32 color images. It is a classic benchmark in computer vision and machine learning, particularly for tasks like image classification and deep learning model evaluation. The authors of the dataset are Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The dataset has 100 classes, each containing 600 images. There are 500 training images and 100 testing images per class. Each class further belongs to one of 20 superclasses. Every image has a 'fine'

label (class) and a 'coarse' label (superclass) ([25]). The classes and superclasses of the CIFAR-100 dataset are listed in Table 2.1. Examples of images from the CIFAR-100 dataset are shown in Figure 2.1.

Superclass	**Classes**
Aquatic mammals	beaver, dolphin, otter, seal, whale
Fish	aquarium fish, flatfish, ray, shark, trout
Flowers	orchids, poppies, roses, sunflowers, tulips
Food containers	bottles, bowls, cans, cups, plates
Fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
Household electrical devices	clock, computer keyboard, lamp, telephone, television
Household furniture	bed, chair, couch, table, wardrobe
Insects	bee, beetle, butterfly, caterpillar, cockroach
Large carnivores	bear, leopard, lion, tiger, wolf
Large man-made outdoor things	bridge, castle, house, road, skyscraper
Large natural outdoor scenes	cloud, forest, mountain, plain, sea
Large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
Medium-sized mammals	fox, porcupine, possum, raccoon, skunk
Non-insect invertebrates	crab, lobster, snail, spider, worm
People	baby, boy, girl, man, woman
Reptiles	crocodile, dinosaur, lizard, snake, turtle
Small mammals	hamster, mouse, rabbit, shrew, squirrel
Trees	maple, oak, palm, pine, willow
Vehicles 1	bicycle, bus, motorcycle, pickup truck, train
Vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

Table 2.1: CIFAR-100 superclasses and classes, source: [25]

In the Python version of the dataset, the data is stored in six batch files - `data_batch_1, data_batch_2, ..., data_batch_5`, and `test_batch` - which are Python 'pickled' objects, along with a file named `batches.meta`. Each batch file contains a dictionary with the following keys:

- `data`: A numpy array in which each row stores a 32x32 image with 3 RGB channels. Each row has a length of 3072: the first 1024 entries store the red channel values, the next 1024 store the green, and the final 1024 store the blue, while the image pixels are flattened row-wise into the one-dimensional array.

- `labels`: A list of integers, where each integer is a class label ranging from 0 to 99.

The file `batches.meta` contains a dictionary that maps label indices to their corresponding class names ([25]).

In our code, we load the CIFAR-100 dataset using the `torchvision.datasets` module [34]. To improve model robustness, the training dataset is augmented as follows:

- `torchvision.transforms.RandomCrop(32, padding=4)`: Simulates slight translations by randomly cropping the image with padding.
- `torchvision.transforms.RandomHorizontalFlip()`: Randomly flips the image horizontally with a 50% probability.

For both training and test datasets, the following transformations are applied:

- `torchvision.transforms.ToTensor()`: Converts the image to a PyTorch tensor and scales pixel values from [0, 255] to [0.0, 1.0].
- `torchvision.transforms.Normalize(mean=(0.5071, 0.4867, 0.4408), std=(0.2675, 0.2565, 0.2761))`: Normalizes each channel of the image tensor with the specified mean and standard deviation values. These values are taken from the configuration file in the GitHub project [24].

2.1.2 Wide ResNet Architecture Details

In our work, we use a PyTorch implementation of Wide ResNet from the GitHub repository [24] by Bumsoo Kim. Specifically, we use a **WRN-28-10** model without dropout. The architecture of the 28x10 WRN is structured as follows:

- `conv1` (first convolutional layer),
- `block1` $\times$ n,
- `block2` $\times$ n,
- `block3` $\times$ n,
- `batchnorm` (batch normalization layer),
- `linear` (final fully connected layer).

Here, $n = \frac{\text{depth}-4}{6} = \frac{28-4}{6} = 4$. The basic building block is a residual block containing two 3×3 convolutional layers and two batch normalization layers. Thus, the total number of convolutional layers in the network is: $1 + 2 \times 4 \times 3 = 25$. Similarly, the total number of batch normalization layers is: $2 \times 4 \times 3 + 1 = 25$. Each of the three groups of blocks contains one shortcut convolutional layer between the first and second building blocks. Finally, there is one fully connected (linear) layer at the end. All layers and their shapes are summarized in Table 2.2.

Layer Name	**Dim**	**Shape**
conv1	4	16 × 3 × 3 × 3
layer1.0.bn1	1	16
layer1.0.conv1	4	160 × 16 × 3 × 3
layer1.0.bn2	1	160
layer1.0.conv2	4	160 × 160 × 3 × 3
layer1.0.shortcut.0	4	160 × 16 × 1 × 1
layer1.1.bn1	1	160
layer1.1.conv1	4	160 × 160 × 3 × 3
layer1.1.bn2	1	160
layer1.1.conv2	4	160 × 160 × 3 × 3
layer1.2.bn1	1	160
layer1.2.conv1	4	160 × 160 × 3 × 3
layer1.2.bn2	1	160
layer1.2.conv2	4	160 × 160 × 3 × 3
layer1.3.bn1	1	160
layer1.3.conv1	4	160 × 160 × 3 × 3
layer1.3.bn2	1	160
layer1.3.conv2	4	160 × 160 × 3 × 3
layer2.0.bn1	1	160
layer2.0.conv1	4	320 × 160 × 3 × 3
layer2.0.bn2	1	320
layer2.0.conv2	4	320 × 320 × 3 × 3
layer2.0.shortcut.0	4	320 × 160 × 1 × 1
layer2.1.bn1	1	320
layer2.1.conv1	4	320 × 320 × 3 × 3
layer2.1.bn2	1	320
layer2.1.conv2	4	320 × 320 × 3 × 3

Layer Name	**Dim**	**Shape**
layer2.2.bn1	1	320
layer2.2.conv1	4	320 × 320 × 3 × 3
layer2.2.bn2	1	320
layer2.2.conv2	4	320 × 320 × 3 × 3
layer2.3.bn1	1	320
layer2.3.conv1	4	320 × 320 × 3 × 3
layer2.3.bn2	1	320
layer2.3.conv2	4	320 × 320 × 3 × 3
layer3.0.bn1	1	320
layer3.0.conv1	4	640 × 320 × 3 × 3
layer3.0.bn2	1	640
layer3.0.conv2	4	640 × 640 × 3 × 3
layer3.0.shortcut.0	4	640 × 320 × 1 × 1
layer3.1.bn1	1	640
layer3.1.conv1	4	640 × 640 × 3 × 3
layer3.1.bn2	1	640
layer3.1.conv2	4	640 × 640 × 3 × 3
layer3.2.bn1	1	640
layer3.2.conv1	4	640 × 640 × 3 × 3
layer3.2.bn2	1	640
layer3.2.conv2	4	640 × 640 × 3 × 3
layer3.3.bn1	1	640
layer3.3.conv1	4	640 × 640 × 3 × 3
layer3.3.bn2	1	640
layer3.3.conv2	4	640 × 640 × 3 × 3
bn1	1	640
linear	2	100 × 640

Table 2.2: WRN-28-10 layer shapes and dimensions

2.1.3 Evaluation Metrics

TOP-1 Accuracy

TOP-1 Accuracy is a commonly used metric for evaluating the quality of classifiers. It measures the percentage of times the model's highest-confidence prediction ('top prediction') matches the true label.

$$\text{Top-1 Accuracy [\%]} = \frac{\text{Number of correct top predictions}}{\text{Total number of samples}} \times 100 \tag{2.1}$$

For example, consider a classification model that predicts the label of an image among three possible classes: Dog, Rat, and Horse. As input to the model, we use two

samples: a picture of a horse and a picture of a Chihuahua dog.
Sample 1: Image of a horse
The model assigns the following probabilities:

Horse 0.85 (Highest confidence)
Dog 0.10
Rat 0.05

The Top-1 prediction is 'Horse', which is also the correct label.
Sample 2: Image of a Chihuahua dog
The model assigns the following probabilities:

Rat 0.51 (Highest confidence)
Dog 0.48
Horse 0.01

The Top-1 prediction is 'Rat', but the correct label is 'Dog'.
So, there is only one correct top prediction (the model correctly classified the horse), and the total number of samples is two. Therefore, the TOP-1 accuracy of the model in this example is:

$$\text{Top-1 Accuracy} = \frac{1}{2} \times 100\% = 50\%.$$

Cross-Entropy Loss

Cross-Entropy Loss is one of the most common loss functions in machine learning, especially for classification tasks. It measures the difference between two probability distributions: the true distribution (the actual labels) and the predicted distribution (output of the model, typically from a softmax). It evaluates the extent to which the probabilities predicted by the model correspond with the actual labels ([41]). The concept of cross-entropy originates from information theory, where the idea of information entropy — also known as Shannon entropy — was introduced by Claude Shannon in his 1948 paper, 'A Mathematical Theory of Communication' [45].

For a single example with class label l, and predicted probabilities p, the cross-entropy loss is defined as:

$$\text{Loss} = -\sum_{i=1}^{C} y_i \log(p_i), \tag{2.2}$$

where C is the number of classes, y_i is the true label encoded one-hot (i.e., $y_i = 1$ if $i = l$, indicating the correct class, and $y_i = 0$ otherwise), and p_i is the predicted probability for class i. Since only the correct class y_l has a value of 1, the equation simplifies to:

$$\text{Loss} = -\log(p_{\text{true class}}). \tag{2.3}$$

The lower the value, the better — a perfect prediction (i.e., probability = 1 for the correct class) results in a loss of 0.

Let's provide a simple example. Suppose we have 3 classes, and our model predicts $p = [0.7, 0.2, 0.1]$ for a given sample, while the true label is class 1. The corresponding one-hot encoded labels are $y = [1, 0, 0]$. The cross-entropy loss for this sample is: $\text{Loss} = -\log(0.7)$.

A useful property of the cross-entropy is that it penalizes confident but incorrect predictions more heavily than less confident mistakes. Additionally, the cross-entropy is differentiable, making it well-suited for gradient-based optimization methods commonly used in training neural networks ([41]).

In our experiments, the TOP-1 accuracy and cross-entropy loss are evaluated on the validation dataset in each epoch. We consider TOP-1 accuracy as the main performance metric of the network in our comparisons.

2.1.4 Training and Validation Details

We train the 28x10 WRN on the CIFAR-100 training dataset in batches of 128 samples. The optimization objective is cross-entropy loss, and the model uses stochastic gradient descent (SGD) with a momentum of 0.9 and a weight decay of 5×10^{-4} for error backpropagation.

The initial learning rate is set to 0.1 and is adaptively lowered throughout the training process. We experiment with three different types of learning rate schedules. Schedule (a) corresponds to the original learning rate policy defined in the `config.py` file of the GitHub project [24]. While this schedule performs well for dense models trained over 200 epochs, we noticed the learning rate might be too low for our pruning experiments, which typically take place between epochs 201 and 400. To address this, we also experiment with alternative learning rate schedules in which the learning rate temporarily increases following certain pruning steps, and later decreases again. All learning rate schedules used in our experiments are summarized in Figure 2.2, and will be referenced throughout the remainder of this work. Note that the learning rate (LR) during the first 200 epochs is the same across all schedules.

For validation, the CIFAR-100 testing dataset is used in batches of 100 samples.

In each training run, two versions of the model's weights are saved:

- Last epoch: Captures the final state of training.
- Best epoch: Saves the weights from the epoch with the highest validation accuracy.

Epoch Range	Learning Rate
epoch ≤ 60	0.1
$60 <$ epoch ≤ 120	0.02
$120 <$ epoch ≤ 160	0.004
epoch > 160	0.0008

Learning rate schedule (a)

Epoch Range	Learning Rate
epoch ≤ 60	0.1
$60 <$ epoch ≤ 120	0.02
$120 <$ epoch ≤ 160	0.004
$160 <$ epoch ≤ 200	0.0008
$200 <$ epoch < 260	0.02
$260 \leq$ epoch < 320	0.004
epoch ≥ 320	0.008

Learning rate schedule (b)

Epoch Range	Learning Rate
epoch ≤ 60	0.1
$60 <$ epoch ≤ 120	0.02
$120 <$ epoch ≤ 160	0.004
$160 <$ epoch ≤ 200	0.0008
epoch $> 200 \wedge 1 <$ epoch mod $40 \leq 10$	0.02
epoch $> 200 \wedge 10 <$ epoch mod $40 \leq 20$	0.004
epoch $> 200 \wedge 20 <$ epoch mod $40 \leq 39$	0.0008
epoch $> 200 \wedge$ epoch mod $40 == 0$	0.0008

Learning rate schedule (c)

Figure 2.2: Learning rate schedules used in our experiments

We assess the model's performance at the final epoch, rather than selecting the best result from all epochs, to avoid the effect of validation overfitting.

2.1.5 Train/Prune Workflow

First, we train the full network without sparsification, which we consider as a baseline model. Then, we apply different weight pruning strategies to sparsify the trained full network, such as global unstructured pruning, block pruning in one step, and iterative block pruning, and further fine-tune the network. The training and pruning workflow schema for our base experiments is illustrated in Figure 2.3.

We also try the AC/DC pruning strategy, which requires training the network from scratch without using the trained dense network. Instead, the AC/DC algorithm is applied from the beginning.

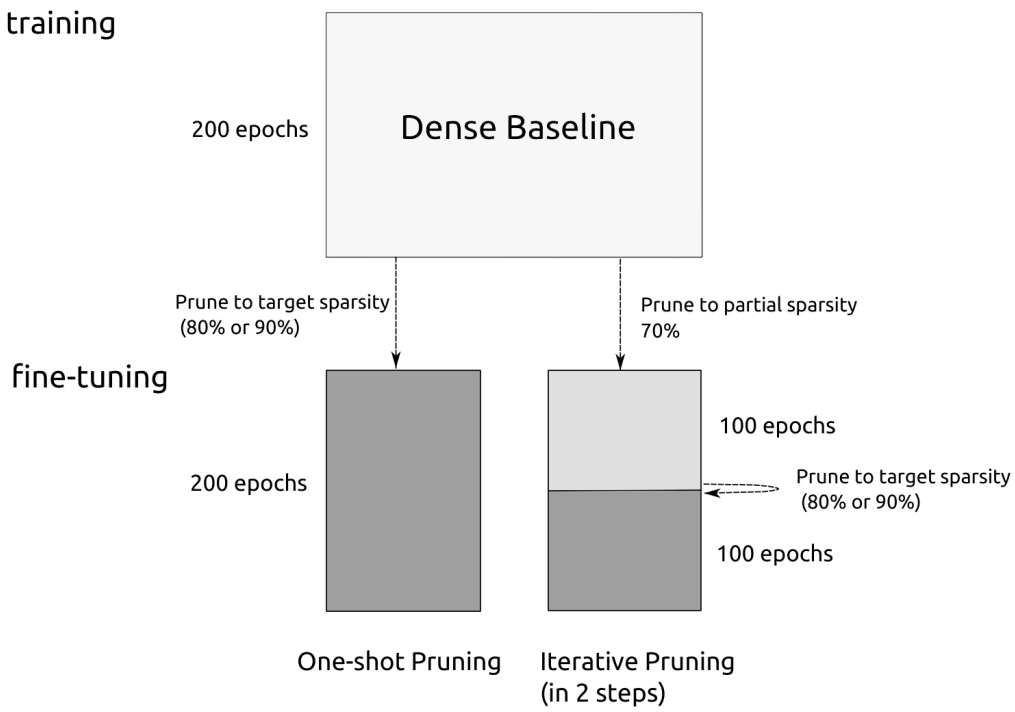

Figure 2.3: Training and pruning workflow schema for base experiments

Pruned Layers

During pruning, we remove weights across selected layers. Biases, BatchNorm layers, and other one-dimensional layers are not pruned, nor are the first convolutional layer or the final linear layer — this is a common convention. The pruning is applied only to the remaining layers of the WRN, all 27 of which are convolutional — comprising 24 standard convolutional layers and 3 shortcut convolutional layers.

Let's calculate the total number of elements within pruned layers:

Standard convolutional layers

- one layer of shape $160 \times 16 \times 3 \times 3$,
- seven layers of shape $160 \times 160 \times 3 \times 3$,
- one layer of shape $320 \times 160 \times 3 \times 3$,
- seven layers of shape $320 \times 320 \times 3 \times 3$,
- one layer of shape $640 \times 320 \times 3 \times 3$,
- seven layers of shape $640 \times 640 \times 3 \times 3$.

Shortcut convolutional layers

- one layer of shape $160 \times 16 \times 1 \times 1$,
- one layer of shape $320 \times 160 \times 1 \times 1$,
- one layer of shape $640 \times 320 \times 1 \times 1$.

Thus, the total number of elements across all pruned layers is

$$\begin{aligned} \text{n\_elements} = 9\Big(&160 \cdot 16 + 7 \cdot 160 \cdot 160 + 320 \cdot 160 + 7 \cdot 320 \cdot 320 + \\ &+ 640 \cdot 320 + 7 \cdot 640 \cdot 640\Big) + \\ &+ 160 \cdot 16 + 320 \cdot 160 + 640 \cdot 320 = \\ &= 36\,454\,400. \end{aligned}$$

Random Seeds

All experiments are conducted using three different random seeds to account for variability introduced by stochastic processes during training. Setting a specific random seed ensures reproducibility in data augmentations, network weight initialization, and the ordering of samples within each epoch.

However, it's important to note that this does not guarantee fully deterministic results. GPU computations — particularly those involving cuDNN — can exhibit non-deterministic behavior unless explicitly constrained by setting `torch.backends.cudnn.deterministic = True`. While this setting enforces determinism, it can degrade computational speed, which is why we choose not to enable it.

Running experiments with multiple seeds adds variation to the training process. This diversity is important, as differences in initialization, augmentation, and data ordering can result in performance fluctuations in validation accuracy. By averaging results across seeds, we obtain a more reliable estimate of model performance and reduce the risk of drawing conclusions from outlier runs.

2.1.6 Hardware

Neural networks are typically trained on graphics processing units (GPUs) because GPUs perform parallel computations efficiently. For our experiments, we used NVIDIA GeForce RTX 2070, NVIDIA GeForce RTX 2080 Ti, NVIDIA GeForce RTX 3060 Ti, and NVIDIA GeForce 4070 GPUs. On this kind of hardware, a single training run of the Wide Residual Network over 200 epochs can take anywhere from 5 to 15 hours.

2.2 Implementation

We use the Python library *PyTorch* [42] and its modules. As the foundation for our implementation, we build upon the code from the GitHub repository by Bumsoo Kim [24]. Their code is published under the MIT licence, which guarantees there is no limitation on using, copying, modifying, or distributing copies of the software. This allowed us to confidently reuse portions of their codebase, such as WRN and the training process, and we modified it according to our unique needs.

Specifically, we refactored the code for improved modularity, removed components unrelated to our experiments, and extended it to support various pruning strategies. Additionally, we implemented logging of the training process, which records information such as training configuration (including pruning method and goal sparsity), per-epoch validation accuracy and loss, learning rate, epoch duration, overall pruning time, total training time, and the final sparsity into a JSON file. We verify the correctness of parts of our custom code through a set of unit tests.

It is crucial that the sparsity of the weights (after pruning) is preserved throughout the entire training process. To achieve this, we use the `nn.utils.prune` submodule of PyTorch [42]. PyTorch does not simply delete the weights during pruning. Instead, it removes the original parameter such as `weight` (or `bias`[1]) from the model's parameters and replaces it with a new parameter `weight_orig`, which stores the original unpruned version of the tensor. The pruning mask generated by the selected pruning technique is stored as a module buffer named `weight_mask`, which is a binary tensor indicating which weights to retain. To ensure compatibility with the existing forward pass, the weight attribute needs to exist. PyTorch computes the pruned version of the weight by element-wise multiplying the original tensor with the mask:

$$\text{weight} = \text{weight\_orig} \odot \text{weight\_mask}.$$

This computed tensor is stored as a regular *attribute* named `weight` (in contrast to `weight` being stored as a *parameter* of the module before pruning). The pruning operation is applied automatically before each forward pass using PyTorch's `forward_pre_hooks` mechanism. As a result, during both training and inference, only the unpruned (active) weights contribute to the model's behavior, and the desired sparsity pattern is consistently preserved across the training lifecycle ([36]).

[1]While PyTorch supports pruning other parameters like `bias`, we only prune weights, so the explanation here focuses on weight pruning. However, the same principles apply when pruning biases, with `bias_orig` and `bias_mask`.

2.2.1 Baseline Model Training and Global Unstructured Pruning

We train the baseline model for 200 epochs, using the procedure from the GitHub project [24]. This serves as a reference point for evaluating the effects of pruning on model performance.

For global unstructured pruning, we adopt the `global_unstructured` function from PyTorch's `torch.nn.utils.prune` submodule ([42]). Specifically, we use the `prune.L1Unstructured` pruning method (by setting it as a value of the parameter `pruning_method`), which prunes (currently unpruned) elements in a tensor by zeroing out the ones with the lowest L_1-norm. In other words, it removes the elements with the smallest absolute values across all weight tensors selected for pruning. By applying pruning globally rather than layer-wise, sparsity is allowed to be distributed flexibly, removing the least important weights regardless of which layer they belong to.

2.2.2 Blocked and Iterative Blocked Pruning

To perform global blocked pruning of weights, we define our own custom pruning function, as no suitable implementation for use with WRN exists in commonly used Python libraries. We extended the `nn.utils.prune` module by subclassing the `BasePruningMethod` base class to create our block pruning function. To perform iterative blocked pruning, it is sufficient to call the implemented block pruning approach multiple times at different stages of training, each time with an increased goal sparsity.

Definition of Blocks in Multidimensional Weight Tensors

We defined the block pruning method for four-dimensional layers. While defining blocks would be straightforward for the two-dimensional case (where blocks of a matrix can be easily imagined), this is not the case for the four-dimensional tensors. Four-dimensional convolutional layers have the shape (c_{out}, c_{in}, $kernel_1$, $kernel_2$). Typically, the kernel size 3x3 is used, so the resulting layer shape is (c_{out}, c_{in}, 3, 3). In our custom block pruning function, we permute the original dimensions (c_{out}, c_{in}, $kernel_1$, $kernel_2$) into ($kernel_1$, $kernel_2$, c_{out}, c_{in}) and then use the matrices from the last two dimensions and iterate their blocks. This means we practically ignore the first two dimensions $kernel_1$, $kernel_2$. For example, if the original layer shape is (c_{out}, c_{in}, 3, 3), we create 9 separate two-dimensional matrices with the shape of $c_{out} \times c_{in}$. We describe this in Algorithm 3.

We iterate the two-dimensional matrices block by block (with a stride equal to the block size), using no overlap.

Block size

Since we apply the pruning in the context of image data, choosing a suitable block size allows the convolutional layer weight matrices to be divided into blocks completely, leaving no overhead on the borderline. In our experiments, we use square blocks of **8×8** (as shown in Table 2.2, the convolutional layers we prune have shapes where the original first two dimensions are divisible by eight). However, our implementation also supports the use of different square block sizes.

Algorithm 3 Permute 4-D Weights, Slice 2-D Matrices, and Iterate Blocks

Input: $\mathbf{W} \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K_1 \times K_2}$
$\mathbf{W}_{\text{perm}} \leftarrow \texttt{permute}(\mathbf{W}, (2, 3, 0, 1))$ ▷ $(K_1, K_2, C_{\text{out}}, C_{\text{in}})$
for i in range(K_1) **do**
    for j in range(K_2) **do**
        matrix $\leftarrow \mathbf{W}_{\text{perm}}[\, i,\, j\,]$ ▷ $\mathbb{R}^{C_{\text{out}} \times C_{\text{in}}}$
        for u in range(0, C_{out}, step=block_size) **do**
            for v in range(0, C_{in}, step=block_size) **do**
                block $\leftarrow$ matrix$[\, u : u + \textit{block_size},\ v : v + \textit{block_size}\,]$
                $r \leftarrow \textit{calculate\_block\_magnitude}(\textbf{block})$
                /* use r as needed (either store it when computing the threshold,
                or set the mask values for this block to 0 or 1 based on comparison
                with the threshold) */
            end for
        end for
    end for
end for

Pruning Criteria Selection

In our implementation, a parameter called the *block_criterion* determines how blocks are selected for elimination. This criterion defines how to choose a representative value for each block, which is then used to assess its importance. We implement four approaches:

- **abs max**:

$$\operatorname{abs}\max(B) = \max_{i,j} |B_{i,j}|\,,$$

- **abs min**:

$$\operatorname{abs}\min(B) = \min_{i,j} |B_{i,j}|\,,$$

- **L1**:

$$\|B\|_1 = \sum_{i,j} |B_{i,j}|,$$

- **L2**:

$$\|B\|_2 = \sqrt{\sum_{i,j} B_{i,j}^2},$$

where B denotes the two-dimensional block matrix.

2.2.3 Implementation of AC/DC and Gradual Pruning

Implementing the AC/DC pruning strategy, as well as the gradual pruning strategy, into our codebase based on the respective scientific papers [43, 54] was relatively straightforward. We only needed to modify the training loop to trigger pruning or remove it at specific points during training, in accordance with the desired strategy.

For AC/DC pruning, a new pruning mask is generated at the start of each sparse phase using the selected pruning method (e.g., global unstructured or blocked pruning). At the beginning of each dense phase, we invoke `torch.nn.utils.prune.remove()` on the weights of each previously pruned module. This operation maintains the current sparsity pattern in the weights but lifts the pruning constraints, allowing the pruned weights to be updated again during the dense (non-pruned) phase. This alternating dense-sparse cycle allows the network to recover some of the pruned weights.

In the case of gradual pruning, the solution is based on the equation (1.1) that specifies the sparsity schedule.

Chapter 3

Experiments and Results

The aim of this chapter is to provide a detailed description of our experiments, present the results, analyze the impact of selected pruning strategies on model performance, and compare the experiments with each other. By systematically varying pruning strategies and levels of sparsity, we aim to understand how different methods affect the network's ability to retain performance while reducing the number of active parameters.

All experiments were conducted using the Wide Residual Network described in Section 2.1.2, in combination with the CIFAR-100 dataset [25]. The experiments follow the methodology outlined in Chapter 2. Each experiment was conducted for three different random seeds: 31, 40, and 71. In all block pruning experiments, blocks of size 8×8 were used. Pruning was applied only to selected layers of the network, as detailed in Section 2.1.5.

Throughout this work, when referring to model sparsity or sparse models, the reported sparsity values correspond specifically to the sparsity levels within the pruned layers, rather than the overall sparsity of the entire model. Although the two values differ slightly, the discrepancy is minimal. For instance, pruning 80% of the weights in the pruned layers (which contain 36 454 400 parameters) results in approximately 29 163 520 parameters being zeroed out. This corresponds to an overall model sparsity of 79.84% when considering all 36 527 808 parameters in the full network. Similarly, a sparsity level of 90% in the pruned layers translates to an overall sparsity of ca. 89.82%.

3.1 Pruning Impact of Selected Strategies on Accuracy

3.1.1 Dense Baseline

We trained a dense model (i.e., without any sparsity) over 200 epochs using the learning rate schedule specified in Table 2.2. The validation accuracy at the final epoch was 81.15%, 81.04%, and 80.69%, respectively, resulting in a mean accuracy of **80.96%** with a standard deviation of 0.24. The result we obtained is consistent with the reported accuracy of 81.02% by the author of the GitHub repository [24], which they claim is the average of five runs for the same setup. We use the dense model as the baseline for evaluating the impact of pruning on model accuracy.

3.1.2 Base Experiments

In our base set of experiments, we included three pruning approaches: global unstructured pruning, global block pruning in one step (one-shot), and iterative global block pruning in two steps. For block pruning, we explored four different selection criteria for block elimination: absolute maximum (*abs max*), absolute minimum (*abs min*), L_1 norm, and L_2 norm, as described in Sections 1.3.5 and 2.2.2, to identify which one performed best in this context.

Pruning was applied to the trained dense models obtained in Section 3.1.1, followed by fine-tuning for 200 additional epochs (up to a total of 400 epochs). All fine-tuning runs used the same random seeds as those in the initial training phase.

For one-shot block pruning, the target sparsity level (either 80% or 90%) was applied immediately on the dense model prior to fine-tuning, and the sparsity remained fixed throughout the entire 200-epoch fine-tuning period. On the other hand, the iterative block pruning approach involved an initial pruning to 70% sparsity, followed by 100 epochs of fine-tuning, then a second pruning step to reach the target sparsity (80% or 90%), followed by another 100 epochs of fine-tuning. This workflow is also illustrated in Figure 2.3.

An example training curve — showing validation accuracy over epochs for both types of block pruning — is presented in Figure 3.1. Each pruning event is followed by a noticeable drop in accuracy, with the drop being more significant when a larger proportion of weights is removed. The importance of fine-tuning becomes evident, as it allows the model to gradually recover from the degradation induced by sparsification and regain part of its lost accuracy.

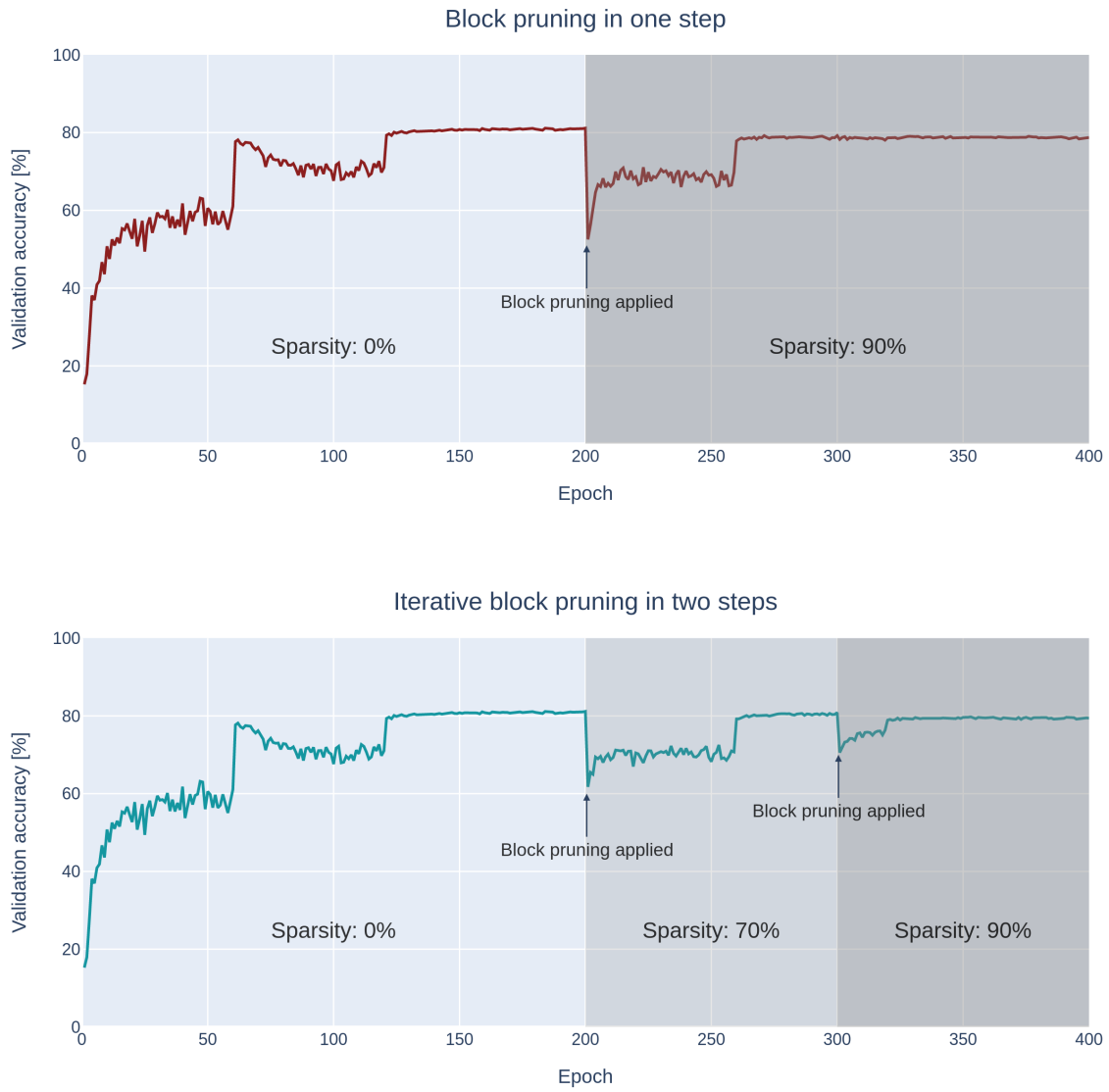

Figure 3.1: Examples of training:
Block pruning in one step vs. Iterative block pruning in two steps
(seed: 31, LR schedule (b), metric: *abs max*)

We used the learning rate schedule (a) (as defined in Table 2.2), where the learning rate remains low and constant throughout the whole fine-tuning phase (epochs 201–400). The resulting validation accuracies at the final epoch of these experiments, reported as the mean and standard deviation (std) across three random seeds, are summarized in Table 3.1.

However, we observed that better results can be achieved using learning rate schedule (b), where the learning rate is temporarily increased after pruning at epoch 201 and then decreased later. The corresponding results are presented in Table 3.2. Notably, the mean validation accuracy for each pruning approach under schedule (b) exceeds the corresponding accuracy achieved with learning rate schedule (a), indicating that

the modified schedule offers a more suitable training dynamic for the pruned models.

Pruning Approach	**Validation TOP-1 Accuracy [%]**	
	Sparsity: 80%	**Sparsity: 90%**
Unstructured	80.40 ± 0.12	79.98 ± 0.04
Block, Metric: *abs max*	79.21 ± 0.02	76.96 ± 0.80
Block Iterative, Metric: *abs max*	**79.57 ± 0.05**	**77.62 ± 0.13**
Block, Metric: *abs min*	78.32 ± 0.07	74.98 ± 0.77
Block Iterative, Metric: *abs min*	78.49 ± 0.04	76.62 ± 0.49
Block, Metric: L_1 norm	76.53 ± 1.40	72.91 ± 1.19
Block Iterative, Metric: L_1 norm	78.08 ± 0.10	76.27 ± 0.36
Block, Metric: L_2 norm	77.94 ± 0.59	74.78 ± 0.73
Block Iterative, Metric: L_2 norm	79.03 ± 0.62	76.69 ± 0.29
Dense Baseline	80.96 ± 0.24	

Table 3.1: Validation accuracy (mean ± std) of base experiments, variant with **LR schedule (a)**

Pruning Approach	**Validation TOP-1 Accuracy [%]**	
	Sparsity: 80%	**Sparsity: 90%**
Unstructured	80.87 ± 0.06	80.72 ± 0.10
Block, Metric: *abs max*	80.34 ± 0.05	78.71 ± 0.06
Block Iterative, Metric: *abs max*	**80.50 ± 0.10**	**79.59 ± 0.15**
Block, Metric: *abs min*	78.74 ± 0.05	76.26 ± 0.35
Block Iterative, Metric: *abs min*	79.14 ± 0.36	78.10 ± 0.58
Block, Metric: L_1 norm	79.29 ± 0.18	76.57 ± 0.48
Block Iterative, Metric: L_1 norm	79.45 ± 0.16	78.59 ± 0.39
Block, Metric: L_2 norm	79.70 ± 0.39	77.87 ± 0.20
Block Iterative, Metric: L_2 norm	80.08 ± 0.16	78.98 ± 0.30
Dense Baseline	80.96 ± 0.24	

Table 3.2: Validation accuracy (mean ± std) of base experiments, variant with **LR schedule (b)**

It can be observed that for both LR schedules, unstructured sparsity yielded the best results. This outcome is somewhat expected, as imposing constraints on structure limits the flexibility in selecting the least important elements for pruning and consequently, more critical elements may inadvertently be removed, while some less

important ones may be retained. For unstructured sparsity under LR schedule (a), the mean validation accuracy at a sparsity level of 80% differed by 0.56% compared to the dense model. With LR schedule (b), this gap was further reduced to just 0.09%. At 90% sparsity, the differences were 0.98% and 0.24%, respectively.

Among all block pruning approaches, the highest validation accuracy — across both LR schedules — was achieved by the **iterative** block pruning method using the ***abs max*** metric. Under LR schedule (b), this approach resulted in only a 0.46% drop in mean validation accuracy at 80% sparsity and a 1.37% drop at 90% sparsity, compared to the original dense model. The second-best performer at 80% sparsity was its non-iterative counterpart — block pruning in one step with the *abs max* metric — followed closely by iterative block pruning based on the L_2 norm. At 90% sparsity, the ranking between these two methods reversed under LR schedule (b).

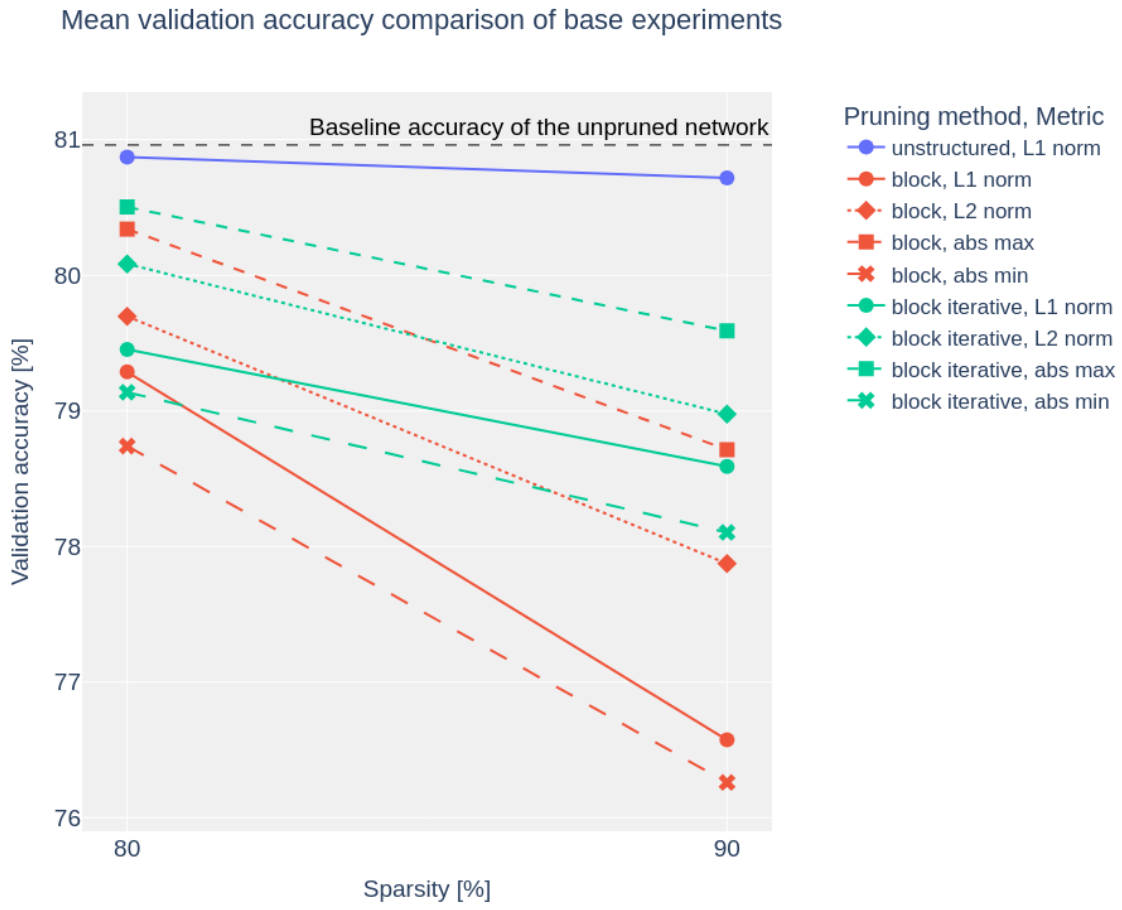

Figure 3.2: Base experiments: Comparison of mean validation accuracy for selected pruning strategies at two sparsity levels, variant with LR schedule (b)

Under LR schedule (a), the lowest mean validation accuracy for both sparsity levels was observed with one-step block pruning based on the L_1 norm. However, its performance variability was high, as indicated by a large standard deviation, which was

due to poor training results for one of the random seeds. With LR schedule (b), the worst-performing method was block pruning in one step with the *abs min* metric.

Overall, the iterative block pruning approaches outperformed their non-iterative counterparts, as illustrated in Figure 3.2, where the validation accuracies (mean of all three seeds) for all selected pruning methods at both sparsity levels are shown for LR schedule (b). We can observe that each iterative approach achieved higher validation accuracy than its non-iterative version across both sparsity levels, with the difference becoming even more noticeable at the higher sparsity level of 90%. Furthermore, Table 3.1 confirms that this trend holds across both LR schedules. The choice of the metric for block elimination also proved to be very important, with the *abs max* metric performing the best, followed by the L_2 norm.

3.1.3 AC/DC Experiments

We applied the AC/DC pruning schedule proposed in the paper [43], combining it with both global unstructured pruning and global block pruning with various block pruning metrics, again for three random seeds. Global unstructured pruning was included to directly compare our results with those reported in their work, while the combination of block pruning with the AC/DC schedule represents our novel contribution, as this approach has not been previously explored.

To closely follow the training conditions described in their Section B.1 (CIFAR-100 Experiments, page 27 of the Appendix), we mimicked their procedure: "For our main experiment, after a warm-up period of ten epochs, we alternated sparse and dense phases of 20 epochs until epoch 170, at which point we allowed the sparse model to train to convergence for 30 more epochs" [43]. A momentum of 0.9 was used throughout the entire training process. An example of the training process for a selected random seed at 80% sparsity is illustrated in Figure 3.3.

Our results are presented in Table 3.3.

Pruning Approach (combined with AC/DC)	**Validation TOP-1 Accuracy [%]**	
	Sparsity: 80%	**Sparsity: 90%**
Unstructured	80.29 ± 0.13	80.10 ± 0.30
Block, Metric: *abs max*	79.27 ± 0.35	77.79 ± 0.48
Block, Metric: *abs min*	78.08 ± 0.13	76.49 ± 0.52
Block, Metric: L_1 norm	77.37 ± 0.22	75.93 ± 0.34
Block, Metric: L_2 norm	77.61 ± 0.03	75.24 ± 0.15
Dense Baseline	80.96 ± 0.24	

Table 3.3: Validation accuracy (mean ± std) for AC/DC experiments

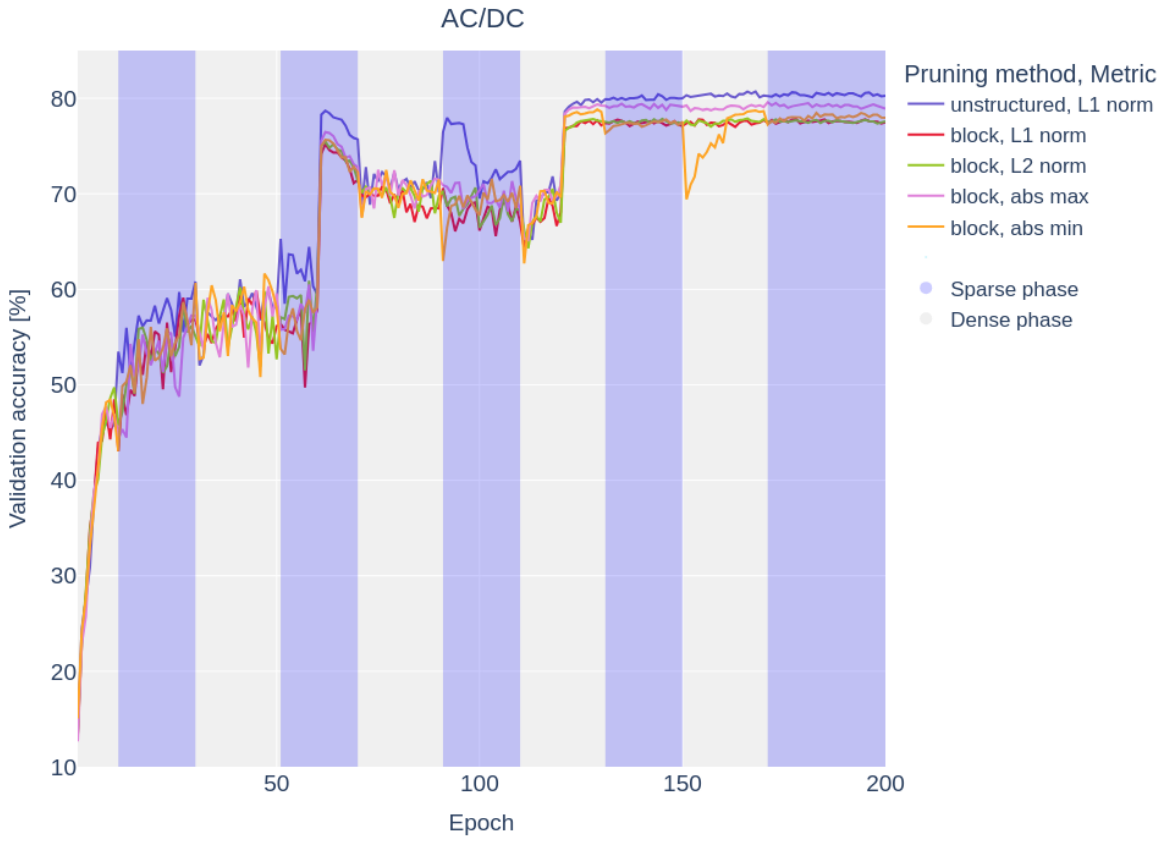

Figure 3.3: Example of AC/DC training (seed: 71, sparsity: 80%)

Under our experimental settings, we successfully trained a better-performing AC/DC-20 Wide-ResNet-28-10 model on the CIFAR-100 dataset at 90% sparsity compared to the model reported by the authors of the paper [43]. Our unstructured pruned model achieved a validation accuracy that was approximately 1% higher — 80.1% versus their 79.1% (as shown in their Table 5). Moreover, our unstructured pruned model outperformed all the models presented in their work for this use case (in their Table 5), including those with lower sparsity levels and their dense baseline.

However, we did not observe the same phenomenon reported in their work [43] under our setup, namely that "the models pruned at 90% at least match the accuracy of the dense baseline". In our case, the dense baseline achieved a validation accuracy of 80.96% (which is, incidentally, almost 2% higher than their dense baseline accuracy of 79.0%), while our AC/DC unstructured pruned model at 90% sparsity reached 80.10%.

Furthermore, the results obtained using the AC/DC pruning schedule did not surpass those from our base experiments using the learning rate schedule (b) (Table 3.2). When comparing AC/DC with the non-iterative pruning experiments under LR schedule (b), AC/DC yielded lower validation accuracy across all pruning methods — both block-based and unstructured — and at both sparsity levels, except for the block pruning with *abs min* at 90% sparsity. Moreover, AC/DC combined with block pruning

resulted in significantly worse validation accuracy than all *iterative* block pruning variants under LR schedule (b). This performance gap was so pronounced that even under the less effective LR schedule (a), iterative block pruning still achieved better results than AC/DC in most cases.

We also experimented with extending the AC/DC schedule to 400 epochs to match the total training duration used in Section 3.1.2 for a fairer comparison (although one of the main advantages of AC/DC is supposed to be its reduced computational cost). However, this led to a degradation in validation accuracy. A closer inspection of the training loss suggested that the model was overfitting during the prolonged training, and therefore, these results are omitted from further comparisons.

AC/DC with alternating LR

Furthermore, we experimented with alternating the learning rate during training. Specifically, we reduced the learning rate by a factor of three during the dense phases relative to the scheduled value, based on the intuition that the sparse phases need to be more explorative. We observed a slight improvement in the mean validation accuracy in three out of the four cases tested, as shown in Table 3.4, although the improvement was so small that it could easily be due to random fluctuations.

Pruning Approach	**Sparsity 80%**		**Sparsity 90%**	
(combined with AC/DC)	**Stable LR**	**Altern. LR**	**Stable LR**	**Altern. LR**
Global Unstructured	80.29 ± 0.13	80.50 ± 0.38	80.10 ± 0.30	80.10 ± 0.14
Block, Metric: max	79.27 ± 0.35	79.37 ± 0.27	77.79 ± 0.48	77.29 ± 0.18

Table 3.4: AC/DC: Comparison of validation accuracy (mean ± std) between the standard approach and the alternating LR approach, where the LR is 3× lower during dense phases

3.1.4 Iterative Pruning from Unstructured to Block Sparsity

We explored a modified variant of the two-step iterative pruning approach used in our base experiments in Section 3.1.2. In this variant, instead of applying block pruning in both steps, we performed unstructured pruning in the first step and block pruning in the second. Specifically, we applied global unstructured pruning to the dense models from Section 3.1.1 obtained at epoch 200, removing 70% of the weights across the pruned layers. The model was then fine-tuned until epoch 300. Afterward, the pruning masks were reset to eliminate previous sparsity constraints, and block pruning using the *abs max* metric was applied to reach the target sparsity levels of 80% or 90%. Here,

we used only the LR schedule (b), as it had proven more suitable in previous similar experiments.

This method achieved validation accuracies of **80.24 ± 0.19%** at 80% sparsity and **78.91 ± 0.51%** at 90% sparsity. Compared to the original iterative block pruning method — where block pruning was applied in both stages — these results were lower by 0.26% and 0.68% in validation accuracy, respectively.

One might question whether the weights initially pruned in an unstructured manner were able to recover after the mask reset, or whether the final model exhibited a higher sparsity than intended. The resulting models displayed sparsity levels of 80.6% (target 80%) and 90.4% (target 90%) across the pruned layers. Although these values are slightly above the desired targets, the deviation does not seem to suggest significant issues with the training process. Nevertheless, the heightened sparsity levels could have impacted the final accuracy.

3.1.5 Iterative Block Pruning in Multiple Steps

We applied the iterative block pruning strategy (using the *abs max* metric) in multiple steps. Starting from the dense model at epoch 200, we initially pruned it to a block sparsity of 50%. The model was then fine-tuned until epoch 400, with sparsity progressively increased at set intervals: to 60% at epoch 240, 70% at epoch 280, 80% at epoch 320, and finally 90% at epoch 360.

Using the learning rate schedule (b), the final validation accuracy at epoch 400 for the 90% sparse model was 78.23 ± 0.17%. Interestingly, this multi-step pruning approach did not outperform the simpler two-step iterative pruning method described in Section 3.1.2. We noticed that the learning rate during later stages might have been too low to allow the model to recover effectively from the accuracy degradation caused by pruning to high sparsity levels.

To address this, we also experimented with the learning rate schedule (c), which increases the LR slightly after each of the five pruning steps, and it resulted in improved performance: the final 90% sparse model achieved a validation accuracy of 79.40 ± 0.19%. This mean validation accuracy was 0.7% higher than the best achieved using the non-iterative approach; however, it still fell slightly short of the 79.59% mean accuracy obtained with the two-step iterative pruning approach.

We hypothesized that extending the fine-tuning period after reaching the final sparsity level of 90% could give the model more time to adapt and stabilize, potentially improving performance. Therefore, we adjusted the pruning schedule under the learning rate schedule (b) to maintain the same total number of epochs, but condensed the pruning steps into the first half, reserving the second half for fine-tuning at the final sparsity level. The updated pruning schedule was as follows: pruning to 50% spar-

sity at epoch 200, then increasing it to 60% at epoch 225, 70% at epoch 250, 80% at epoch 275, and finally 90% at epoch 300. This left a dedicated fine-tuning period of 100 epochs, from epoch 301 to 400. This revised schedule yielded improved results, achieving a validation accuracy of **79.70 ± 0.22%** at the final epoch — marking the highest mean validation accuracy we have observed for a 90% block-sparse model so far.

3.1.6 Gradual Pruning

We explored two gradual block pruning strategies in combination with block pruning using the *abs max* metric, under LR schedule (b):

1. **Continuous Pruning:** Starting from 0% sparsity at epoch 201, pruning was gradually increased to 90% by epoch 400, following the schedule defined by the equation (1.1), with parameters $n = 199$ and $\Delta t = 1$.

2. **Stepwise Gradual Pruning:** Sparsity was increased in stages: first to 50% at epoch 201, then gradually to 60% by epoch 250, 70% by epoch 300, 80% by epoch 350, and finally from 80% to 90% during epochs 351–400. The same pruning schedule was applied for each gradual step using $n = 49$ and $\Delta t = 1$.

Both approaches resulted in significantly lower validation accuracies compared to the results reported in previous sections. The first method achieved an accuracy of $74.39 \pm 0.38\%$, and the second reached $75.67 \pm 0.25\%$.

As observed in Subsection 3.1.5, introducing a longer fine-tuning period after reaching the target sparsity of 90% led to improved performance. Therefore, we also experimented here with modified versions of both strategies, in which gradual pruning was applied between epochs 201 and 300, followed by a 100-epoch fine-tuning phase from epoch 301 to 400. In the first case, $n = 99$ and $\Delta t = 1$ are used; in the second, $n = 24$ and $\Delta t = 1$ were applied for each gradual step. The validation accuracies at the final epoch were **79.78 ± 0.27**% and **79.83 ± 0.18**%, respectively. The results obtained here are even slightly better than those obtained in Section 3.1.5 *Iterative Block Pruning in Multiple Steps*.

One might question whether it is beneficial to use the *Stepwise Gradual Pruning* approach rather than the simpler *Continuous Pruning*. To evaluate this, we conducted an experiment using a modified version of continuous pruning in which the model was sparsified to 50% at the beginning of epoch 201 (to match the initial sparsity used in Stepwise Gradual Pruning). From that point, we applied gradual pruning according to Equation (1.1), with an initial sparsity value $s_i = 0.5$, over $n = 99$ epochs using a step size of $\Delta t = 1$ epoch. After reaching the final sparsity of 90% at epoch 300, we continued fine-tuning at fixed sparsity for an additional 100 epochs. This resulted

in a validation accuracy of 79.68 ± 0.32% (compared to 79.83 ± 0.18% in stepwise gradual pruning). These results suggest there might be a potential slight benefit of using the stepwise approach; however, the observed difference could also be due to random variation, and thus the conclusion is not definitive.

3.2 Comparison of Pruning Experiments

To compare the results from the various experiments described in Section 3.1, we select the best-performing model from each experiment type. For experiments that spanned 400 training epochs, we observed that the best results were achieved using the learning rate schedule (b). Since all learning rate schedules are identical for the first 200 epochs, this distinction is irrelevant for experiments that were trained only for 200 epochs—such as the AC/DC method and the dense baseline.

In summary, for the comparison at the 80% sparsity level, we include the following:

- the dense baseline model from Section 3.1.1,
- the base experiments using LR schedule (b) from Section 3.1.2,
- all AC/DC experiments trained over 200 epochs from Section 3.1.3, including the variant that used the alternating learning rate reduced by a factor of three during dense phases,
- the mixed iterative approach combining unstructured pruning followed by block pruning from Section 3.1.4.

For the 90% sparsity level comparison, we include all the experiments listed above, and additionally:

- the iterative block pruning approach with multiple steps from Section 3.1.5,
- the stepwise gradual pruning method from Section 3.1.6,

both using the LR schedule (b), with pruning steps occurring between epochs 200 and 300, followed by a fine-tuning phase from epoch 301 to 400.

The results of the experiments discussed above are presented in Figures 3.4 and 3.5. As shown, none of the pruning experiments achieved or surpassed the validation accuracy of the dense baseline. When comparing the mean validation accuracy, the second-best performance was observed with unstructured pruning within the base experiments group, for both sparsity levels. In third place at the 80% sparsity level, we observed block iterative pruning using the *abs max* metric, alongside AC/DC unstructured pruning with alternating LR. At the 90% sparsity level, AC/DC unstructured

Figure 3.4: Comparison of pruning experiments at 80% sparsity level. Individual points for all three seeds are shown. The boxes are centered at the mean, with heights representing the mean ± standard deviation. For each group of experiments, only the best results are included.

pruning (with both fixed and alternating LR) again secured third place, followed closely by gradual stepwise block pruning with the *abs max* metric.

In general, unstructured pruning preserved higher accuracy than block pruning within the same experimental group. This trend also holds when comparing across all experiments, with one exception: at 80% sparsity, the AC/DC unstructured pruning model performed worse than the block-sparse model using the *abs max* metric from the base experiments group. Nonetheless, for both sparsity levels, the difference in mean validation accuracy between the overall best unstructured and best block-pruned models was under 1%.

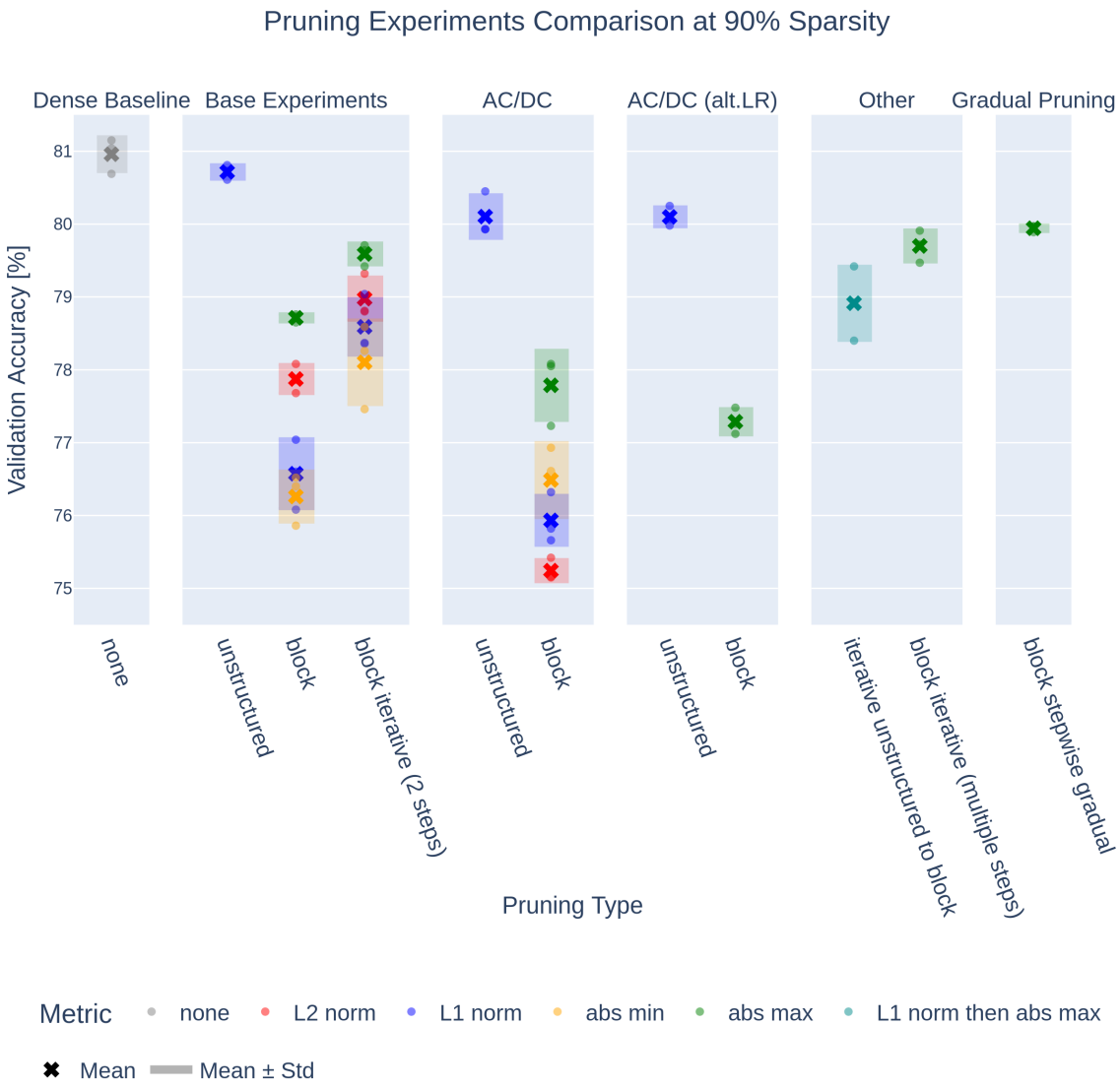

Figure 3.5: Comparison of pruning experiments at 90% sparsity level. Individual points for all three seeds are shown. The boxes are centered at the mean, with heights representing the mean ± standard deviation. For each group of experiments, only the best results are included.

When comparing one-shot block pruning with the two-step iterative approach, we can see in both Figures 3.4 and 3.5 that, for each metric, the iterative version achieved higher mean validation accuracy than its one-shot counterpart, with the performance gap widening at the higher sparsity level. Figure 3.4 shows that, at 80% sparsity, the iterative block pruning method using the *abs max* metric achieved the highest validation accuracy among all block-sparse models. At 90% sparsity, we additionally applied a five-step pruning approach, which achieved an even higher mean validation accuracy than the two-step iterative approach. Nevertheless, the best block-sparse model at the sparsity level of 90% was obtained using the stepwise gradual pruning

approach, as shown in Figure 3.5.

The mixed iterative approach, which first applied unstructured pruning followed by block pruning, did not yield competitive results. At both sparsity levels, it underperformed compared to the standard iterative approach that used block pruning in both steps.

The AC/DC pruning method consistently performed worse than methods based on the dense baseline followed by post-training pruning. This observation also holds for AC/DC experiments using the alternating learning rate schedule. In fact, every AC/DC experiment produced a lower mean validation accuracy compared to its corresponding base experiment using the same block importance metric, with the sole exception being one-shot block pruning with the *abs min* metric at 90% sparsity. The difference in mean validation accuracy between AC/DC and the base experiments ranged from 0.37% to 0.62% for unstructured pruning, from 0.66% to 2.09% for one-shot block pruning at the 80% sparsity level, and from -0.23% to 2.63% for one-shot block pruning at the 90% sparsity level. When compared to iterative block pruning in two steps, this accuracy gap became even more pronounced, ranging from 1.06% to 2.47% at 80% sparsity, and even from 1.61% to 3.74% at 90% sparsity, which is a substantial difference.

When comparing block-elimination metrics, Figures 3.4 and 3.5 show that the *abs max* metric for block importance consistently achieved the best performance in all experiments where multiple metrics were evaluated. In the base experiments, the L_2 metric ranked second; however, in the AC/DC context, it performed poorly. Similarly, while the L_1 norm outperformed the *abs min* metric in the base experiments, their relative performance reversed in the AC/DC setting.

Chapter 4

Discussion

This chapter reflects on the experimental results presented earlier to draw conclusions and identify opportunities for potential improvement.

4.1 Key Findings and Insights

The Price of Blocked Sparsity

The experiments yielded several important insights regarding the trade-off between sparsity and model accuracy. None of the pruning experiments matched or exceeded the validation accuracy of the dense baseline, which was 80.96 ± 0.24%. The best-performing block-sparse model at 80% sparsity — the two-step iterative approach with the *abs max* metric — achieved a validation accuracy of 80.50 ± 0.10%. At 90% sparsity, the top block-sparse result — obtained by the stepwise gradual block pruning method with *abs max* — reached 79.83 ± 0.18%. As expected, unstructured pruning generally preserved accuracy better than block pruning. However, at both sparsity levels, the difference in mean validation accuracy between the best unstructured and best block-pruned models was less than 1%. These findings indicate that although there is a trade-off between accuracy and computational efficiency, block pruning can preserve most of the model's performance even at high sparsity levels, such as 80% or 90%, with accuracy drops of only 0.46% and 1.13%, respectively, compared to the dense baseline.

Slow and Steady Wins the Prune

When comparing one-shot block pruning with the two-step iterative approach under fair conditions — such as identical training epochs, learning rate schedules, and block importance metrics — the iterative method consistently outperformed the one-shot approach. This performance gap became more pronounced at higher sparsity levels.

At 90% sparsity, we also experimented with a five-step pruning schedule and a gradual pruning approach, both of which yielded even better performance. Notably, the stepwise gradual pruning method produced the best results. These results suggest that pruning less abruptly — in multiple, smooth, iterative steps — can be beneficial, especially at higher sparsity levels.

The Battle of Block Importance Metrics

In addition to the pruning schedule (iterative vs. one-shot), the choice of block importance metric had a significant impact on performance. As illustrated in Figure 3.2, in some cases, one-shot pruning with a strong metric outperformed iterative pruning with a weaker one — for example, one-shot pruning using the *abs max* metric achieved higher mean validation accuracy than iterative pruning with either the *abs min* or L_1 norm metrics. Across all experiments that evaluated multiple metrics, *abs max* consistently delivered the best performance. However, the relative effectiveness of the remaining metrics was less consistent, making it difficult to establish a clear ranking among them.

AC/DC: High Voltage but Low Performance

The AC/DC pruning method consistently underperformed compared to approaches that prune a well-trained dense baseline, with accuracy gaps reaching up to 3.74% relative to two-step iterative block pruning. Incorporating an alternating learning rate — where the dense phase uses a LR three times lower than the sparse phase — did not yield meaningful improvements.

Contrary to the claims in paper [43], we did not observe the reported effect that "at high sparsity levels, AC/DC even outperforms existing methods that rely on accurate pre-trained dense models". At 90% sparsity, pruning methods based on pre-trained dense models achieved higher validation accuracy. Additionally, we were unable to replicate their result on WRN-28-10 with CIFAR-100 that "pruned models at 90% can match the dense baseline". It is important to note that our experimental setup differed in several ways — such as hyperparameters (e.g., momentum, learning rate schedule) and potentially data augmentation strategies — and replicating their exact conditions was not our goal. Instead, we aimed to assess AC/DC within our own setup in context with block pruning methods. Moreover, we noticed their reported dense baseline accuracy (79.0 ± 0.25) was nearly 2% lower than ours (80.96 ± 0.24), suggesting that their baseline may not have been optimally trained.

That said, the AC/DC approach still offers practical advantages. It required only 200 training epochs, compared to the 400 total epochs used by post-training pruning methods. Thus, in contexts where reducing training time and computational cost is

critical, AC/DC remains a viable and efficient alternative, still providing reasonable accuracy.

Timing is Everything

Additionally, we found that the choice of learning rate schedule was critical, though the optimal LR schedule varied depending on the pruning method and pruning schedule, making fair comparisons between pruning methods challenging. Generally, we observed that a sufficiently high LR was necessary after pruning. When the LR was too low during the post-pruning fine-tuning phase, the model struggled to recover from the accuracy loss induced by pruning. Furthermore, it was essential to allow ample time for the model to recover from the pruning process through an extended fine-tuning phase, even when using less abrupt pruning schedules like gradual pruning.

4.2 Potential Extensions and Future Research Directions

While the findings of this work provide valuable insights into block pruning methods and iterative blocked pruning, several directions remain open for further investigation:

- **Exploring alternative block shapes:** In this work, we utilized a block size of 8×8 due to its balance between sparsity structure and potential hardware acceleration benefits. However, larger block size such as 16×16 is also a viable option, as suggested by the shapes of pruned layers in Table 2.2.

- **Benchmarking across learning rate schedules:** Our results showed that the learning rate schedule plays a critical role in post-pruning recovery. A systematic comparison of different learning rate strategies for post-pruning period could reveal important relationships.

- **Refining pruning schedules:** Future research could explore additional pruning schedules. For example, in our gradual pruning experiments, sparsity was increased after each epoch. It could be beneficial to investigate whether updating sparsity less frequently — e.g., every few epochs — can yield comparable accuracy while reducing the computational overhead of frequent pruning.

- **Extending to diverse datasets and architectures:** Most importantly, we suggest evaluating iterative block pruning strategies across various neural network models. Due to limited computational resources and time constraints, this work focused primarily on a single dataset and architecture. A broader evaluation

spanning multiple datasets and network types would be valuable for assessing the generalizability of the conclusions presented here.

- **Use specialized implementation to utilize hardware benefits:** While this work focused on evaluating the impact of pruning on model accuracy, future research could explore implementations that leverage hardware accelerators optimized for structured sparsity to unlock practical performance gains.

Conclusion

Through experimentation with a Wide Residual Network on the CIFAR-100 dataset, this work systematically examined block pruning strategies and their impact on model performance, with a particular focus on pruning schedules (one-shot vs. iterative) and the trade-offs between sparsity and accuracy. In addition to the basic iterative approach, we also explored a gradual pruning schedule and the AC/DC pruning method in combination with blocked pruning — an approach that has not been thoroughly examined previously.

We demonstrated that while block pruning introduces a modest degradation in accuracy compared to the dense model and unstructured pruning approaches, it preserves a substantial portion of model accuracy even at high sparsity levels, such as 80% and 90%.

In the context of blocked pruning, we showed that iterative and gradual pruning strategies significantly outperform one-shot pruning, particularly as sparsity increases, highlighting the importance of pruning schedules in maintaining model accuracy. By allowing the network to adapt progressively, iterative pruning seems to mitigate the accuracy loss that would otherwise occur with more aggressive, one-shot approaches. Among the four employed metrics for block elimination, the *abs max* block importance metric consistently delivered the best results.

While the AC/DC pruning method did not outperform post-training pruning approaches in our setting, it offers considerable benefits in terms of training efficiency, making it a practical alternative when computational resources are constrained.

Overall, the findings of this work suggest that iterative blocked pruning can strike a valuable balance between computational efficiency and model performance. Future work should investigate a broader range of models and datasets to further explore the potential of block-based sparsity guided by iterative schedules in practical deep learning applications. Additionally, extending this research to other architectures and datasets will enable more generalizable conclusions beyond those derived from the Wide Residual Network on the CIFAR-100 dataset.

Bibliography

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

[2] Jingfei Chang, Yang Lu, Ping Xue, Yiqun Xu, and Zhen Wei. Iterative clustering pruning for convolutional neural networks. *Knowledge-Based Systems*, 265:110386, 2023.

[3] Tri Dao, Beidi Chen, Nimit S Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*, pages 4690–4721. PMLR, 2022.

[4] Nolan Dey, Shane Bergsma, and Joel Hestness. Sparse maximal update parameterization: A holistic approach to sparse training dynamics. *arXiv preprint arXiv:2405.15743*, 2024.

[5] Xin Dong, Shangyu Chen, and Sinno Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. *Advances in neural information processing systems*, 30, 2017.

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

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[8] Shupeng Gui, Haotao Wang, Haichuan Yang, Chen Yu, Zhangyang Wang, and Ji Liu. Model compression with adversarial robustness: A unified optimization framework. *Advances in Neural Information Processing Systems*, 32, 2019.

[9] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. *Advances in neural information processing systems*, 29, 2016.

[10] Kevin Gurney. *An introduction to neural networks*. CRC press, 2018.

[11] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[12] Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, 5, 1992.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, pages 630–645. Springer, 2016.

[15] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1808.06866*, 2018.

[16] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*, pages 784–800, 2018.

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

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

[19] 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. *The Journal of Machine Learning Research*, 22(1):10882–11005, 2021.

[20] Hugging Face. Quantization. `https://huggingface.co/docs/optimum/en/concept_guides/quantization`, 2025. Accessed: 2025-05-03.

[21] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[22] Yu Ji, Ling Liang, Lei Deng, Youyang Zhang, Youhui Zhang, and Yuan Xie. Tetris: Tile-matching the tremendous irregular sparsity. *Advances in neural information processing systems*, 31, 2018.

[23] Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aaron Oord, Sander Dieleman, and Koray Kavukcuoglu. Efficient neural audio synthesis. In *International Conference on Machine Learning*, pages 2410–2419. PMLR, 2018.

[24] Bumsoo Kim. Pytorch implementation of sergey zagoruyko's wide residual networks. GitHub repository, `https://github.com/meliketoy/wide-resnet.pytorch`, 2017. Accessed: 2024-06-26.

[25] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. URL `http://www.cs.toronto.edu/~kriz/cifar.html`, 2009.

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

[27] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[28] SandyResearch ML Systems Lab led by Dan Fu. Chipmunk: Deep dive on gpu kernel optimizations and systems (part iii). `https://sandyresearch.github.io/chipmunk-part-III/`, 2025. Accessed: 2025-05-08.

[29] Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. Hrank: Filter pruning using high-rank feature map. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1529–1538, 2020.

[30] Mingbao Lin, Yuxin Zhang, Yuchao Li, Bohong Chen, Fei Chao, Mengdi Wang, Shen Li, Yonghong Tian, and Rongrong Ji. 1xn pattern for pruning convolutional neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(4):3999–4008, 2023.

[31] Tao Lin, Sebastian U Stich, Luis Barba, Daniil Dmitriev, and Martin Jaggi. Dynamic model pruning with feedback. *arXiv preprint arXiv:2006.07253*, 2020.

[32] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.

[33] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.

[34] TorchVision maintainers and contributors. Torchvision: Pytorch's computer vision library. `https://github.com/pytorch/vision`, 2016.

[35] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[36] Michela Paganini. Pruning tutorial — pytorch tutorials documentation. `https://pytorch.org/tutorials/intermediate/pruning_tutorial.html`, 2019. Accessed: 2025-04-21.

[37] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

[38] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *International conference on machine learning*, pages 2498–2507. PMLR, 2017.

[39] Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*, 2017.

[40] Sharan Narang, Eric Undersander, and Gregory Diamos. Block-sparse recurrent neural networks. *arXiv preprint arXiv:1711.02782*, 2017.

[41] OpenAI. Chatgpt (mar 14 version). `https://chat.openai.com`, 2023. Prompt: "Explain cross-entropy loss". Accessed: 2025-04-19.

[42] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. `https://openreview.net/forum?id=BJJsrmfCZ`, 2017.

[43] Alexandra Peste, Eugenia Iofinova, Adrian Vladu, and Dan Alistarh. Ac/dc: Alternating compressed/decompressed training of deep neural networks. *Advances in neural information processing systems*, 34:8557–8570, 2021.

[44] Abigail See, Minh-Thang Luong, and Christopher D Manning. Compression of neural machine translation models via pruning. *arXiv preprint arXiv:1606.09274*, 2016.

[45] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[46] Dharma Teja Vooturi, Dheevatsa Mudigere, and Sasikanth Avancha. Hierarchical block sparse neural networks. *arXiv preprint arXiv:1808.03420*, 2018.

[47] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. *arXiv preprint arXiv:1910.04732*, 2019.

[48] Wei Wen, Yuxiong He, Samyam Rajbhandari, Minjia Zhang, Wenhan Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning intrinsic sparse structures within long short-term memory. *arXiv preprint arXiv:1709.05027*, 2017.

[49] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

[50] Takuma Yamaguchi and Federico Busato. Accelerating matrix multiplication with block sparse format and nvidia tensor cores. `https://developer.nvidia.com`, 2021. Accessed: 2025-05-08.

[51] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News*, 45(2):548–560, 2017.

[52] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[53] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. *Dive into deep learning*. Cambridge University Press, 2023.

[54] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

Appendix A: Contents of the Electronic Supplement

The electronic supplement attached to this thesis contains the source code of the program and files with the results of the experiments. The source code is also available at `https://github.com/vikica/iterative-blocked-pruning-of-neural-networks`.