Automatic Depth Pruning and Post-Processing for Efficient Deep Learning

Michal Kuchar

Delft University of Technology

TU Delft

Automatic Depth Pruning and Post-Processing for Efficient Deep Learning

by

Michal Kuchar

Thesis advisor:	Georgi Gaydadjiev
Daily co-supervisors:	Yunqiang Li, Mahmood Naderan-Tahan
External supervisors:	Henk Corporaal, Manil Dev Gomony
Project duration:	April, 2025 - November, 2025
Faculty:	Faculty of Electrical Engineering, Mathematics & Computer Science, Delft

Cover: Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
CNN	Convolutional Neural Network
DRAM	Dynamic Random Access Memory
FLOPS	Floating-Point Operations Per Second
GPU	Graphics Processing Unit
HBM	High-Bandwidth Memory
LLM	Large Language Model
MAC	Multiply-And-Accumulate (Operation)
MHA	Multi-Head Attention
MLP	Multi-Layer Perceptron
PPFT	Post-Pruning Fine-Tuning

1 Introduction

In recent years, deep learning models have proven highly effective for tasks that have been previously challenging for classical computer algorithms. For example, Convolutional Neural Networks (CNNs) have excelled at image classification [24], and Large Language Models (LLMs) have performed remarkably well at natural language processing [36]. As a result, deep learning models have seen widespread adoption, but this also comes with challenges. Their performance, in terms of prediction accuracy, tends to scale with parameter count, reaching trillions of floating-point parameters in the most recent advances [40]. This means that the cost of deploying these models has also grown rapidly, as this often requires expensive and power-hungry accelerator hardware. As a result, fields of study have emerged focusing on efficient deployment. In this work, we explore the landscape of efficiency techniques, identifying gaps in the literature, and contributing a novel technique that addresses some of the gaps.

1.1. Motivation

This high cost of deploying large deep learning models is due to several factors. For one, the models are typically deployed on accelerator hardware, e.g., graphics processing units (GPUs), where the high-bandwidth memory (HBM) can be orders of magnitude more expensive than standard dynamic random access memory (DRAM). Additionally, accelerator hardware is known to consume significant amounts of electricity, adding an operational cost component to the already high fixed cost of hardware. This power consumption can also be a concern from the standpoint of sustainability, especially in cases when the power does not come from renewable sources [20].

Naturally, this high cost can drastically limit deployment possibilities. On-premise deployment can be challenging not only because of direct hardware cost, but also due to the difficult maintenance required for the infrastructure around the hardware. As a consequence, the deployment of LLMs is often offloaded to data centers, which can raise concerns over control, privacy, and the cost charged to the user by the provider.

These difficulties have prompted a wide range of advancements in inference efficiency research, with the goals of reducing memory footprint and inference time. Memory footprint reduction saves on cost by reducing the amount of expensive, high-bandwidth memory required to deploy the model. Reducing inference time saves on cost by permitting more inference per unit time, thereby reducing the need to scale the system.

Starting from a pretrained model that one wishes to use for inference, existing efficiency techniques include quantization, knowledge distillation and pruning. Quantization involves converting weights and activations from high-precision floating-point datatypes to lower-precision datatypes. Quantization has been shown to be highly effective for reducing the memory footprint severalfold, but it creates only modest inference latency improvements compared to other techniques, at least in the absence of hardware-level quantization support [51]. Knowledge distillation involves training a smaller model with fewer parameters using outputs from the larger model. This has been shown to considerably improve both, memory footprint and inference latency, but the training process still requires significant computational resources [50]. Pruning involves the removal of parameters deemed unimportant or redundant. Many different types of pruning exist, with tradeoffs between granularity of parameter selection, memory footprint reduction, and inference latency reduction. This scope of tradeoffs creates the potential for balance between memory and latency, and thus this work focuses on pruning.

1.1.1. Pruning as an Efficiency Technique

The range of pruning starts at unstructured pruning, whereby parameters can be arbitrarily selected and removed, with no structural constraints. On the other end there is the structured pruning, which we subdivide into width pruning and depth pruning. Width pruning involves the removal of rows and columns in parameter matrices of linear layers, or the removal of channels and filters in parameter matrices of convolutional layers. Depth pruning involves the removal of full layers. [43]

Unstructured pruning has been shown to be effective for memory footprint reduction, with works like [14] achieving up to 50% parameter count reduction with minimal performance sacrifices on open-source LLMs. Similar cases exist for width pruning, with works like [2] achieving a 30% parameter reduction and 15% performance penalty tradeoff. However, these two types of pruning are usually not the best option for latency reduction on accelerator hardware. Unstructured sparsity is difficult for hardware to exploit because its pattern is unpredictable. Width pruning provides more structure, but authors generally argue that memory transfer contributes most to latency, and width pruning still does not alleviate this - it only reduces the amount of data that needs to be transferred. [23]

The most significant latency speedups are usually reported in depth pruning literature. For example, by pruning Transformer layers followed by post-pruning fine-tuning (PPFT), [23] reported a throughput speedup of 1.6x at a 67% parameter reduction while preserving the vast majority of benchmark performance on Llama LLMs. In [22], depth pruning of ResNet and MobileNet CNN models is reported, resulting in wall-clock time speedups between 1.25x and 2.49x. In addition to speedups, LLMs also demonstrate remarkable performance retention when depth pruned without PPFT. For example, [17] pruned up to 30% of Transformer blocks on Llama2 LLMs with a negligible performance penalty on certain benchmarks.

Despite its apparent advantages, depth pruning is not without limitations either. Depth pruning methods suffer from a lack of generality, in the sense that implementations are often dependent on the model architecture. Current LLMs are conveniently structured as residual chains of Transformer layers that preserve input and output dimensionality, and the Transformer layers themselves are composed of Multi-head Attention (MHA) and Multi-layer Perceptron (MLP) layers with the same property. This dimension preservation property allows for straightforward pruning because dimension conflicts do not arise ([17], [38]). CNNs often require the user to manually specify which layers can be pruned. Since convolutional layers change dimensionality by design, removing layers naively can render the underlying tensor multiplications ill-defined, rendering the model ill-defined by extension.

This dimensionality issue has limited layer selection in the CNN literature, because authors have resorted to excluding dimension-transforming convolutional layers from the search space of prunable layers ([22], [5]), and this can significantly limit the search space. Others have tried adding extra weights to re-transform activations to the correct dimensions ([12], [48]), but this goes against the parameter reduction goal and requires PPFT.

Perhaps for this reason, unstructured pruning and width pruning approaches are far more common in the CNN literature compared to depth pruning. For example, a review of YOLOv5 CNN model pruning [19] found only 2 depth pruning works out of a total of 30. Finding a way to efficiently prune these dimension-transforming layers could create better understanding of depth pruning on CNNs, enable us to map the "performance vs. parameter reduction" tradeoff curve like has already been done for LLMs by e.g., [17], and expand the search space of prunable layers.

In LLMs, the pruning of only MHA and MLP layers also limits the search space, because there is only one MHA and MLP layer per Transformer layer. In principle, depth pruning could be more granular by pruning the linear layers that make up MHA and MLP layers. The linear layers exist at the same level of abstraction as convolutional layers, however, due to their dimension-transforming nature, pruning them comes with the same limitations as with convolutional layers. Solving this issue could unlock a more fine-grained space of prunable layers in LLMs.

To depth-prune in an architecture-independent manner, we need to prune at the level of linear and convolutional layers. We call this *fine-grained depth pruning* to differentiate from other, more abstract, architecture-dependent methods. This work investigates this idea focusing on the following research questions:

Research Question 1: How do fine- and coarse-grained depth pruning differ in their accuracy - parameter reduction tradeoffs on Llama2 models?

Research Question 2: At the same parameter reduction level, how does fine-grained depth pruning compare to coarse-grained pruning in inference latency on Llama2 models?

Research Question 3: What is the complete accuracy - parameter reduction tradeoff curve for fine-grained depth pruning on the COCO dataset on YOLOv5 models?

Research Question 4: How does fine-grained depth pruning affect inference latency at the maximum parameter reduction that preserves acceptable accuracy with respect to the accuracy - parameter reduction tradeoff curve on the COCO dataset on YOLOv5 models?

1.2. Contributions

The aim of this work is to develop towards a general, architecture-agnostic depth pruning technique, and to test its performance on LLMs and CNNs. Our contributions are the following:

1. We mathematically formalize and implement a depth pruning methodology that operates on the level of linear and convolutional layers, i.e., *fine-grained depth pruning*.
2. We apply fine-grained depth pruning to the Llama2 [41] family of LLMs, subsuming and generalizing the existing Transformer/MHA/MLP pruning techniques. We demonstrate and analyze how it improves upon existing methods, achieving improved tradeoffs between parameter count and performance in certain contexts, and highlight properties that might be useful for future deep learning model compression efforts.
3. We apply fine-grained depth pruning to the YOLOv5 [21] family of CNN models. We highlight robustness under pruning and improvements in search space flexibility, all without the use of PPFT.

1.3. Organization

The rest of this manuscript is structured as follows:

- Chapter 2 provides additional context on pruning as an efficiency technique. Then it goes into detail on what efficiency results have been obtained using pruning in the literature to set a basis for comparison.
- Chapter 3 provides a mathematical and algorithmic formalization of our fine-grained depth pruning approach.
- Chapter 4 shows results of fine-grained depth pruning applied to the Llama2 7B and Llama2 13B LLMs, as well as YOLOv5X, YOLOv5L and YOLOv5M CNN models.
- Chapter 5 interprets the results through the lens of the related work in Chapter 2 to explain their significance, summarizes and concludes this report, and makes recommendations for future work.

2 Background and Related Work

In this Chapter, we first examine the space of techniques that can be used to enhance the inference efficiency of pretrained deep learning models in Section 2.1, highlighting pruning as an interesting technique for investigation. We start by focusing on techniques applied to LLMs and CNNs due to their widespread use and prohibitive resource requirements. In Section 2.2 we outline subcategories of pruning, comparing and contrasting their practical implementations in LLMs and CNNs. Then we observe how the architectural differences between LLMs and CNNs give rise to different pruning techniques. In Section 2.3 we highlight how the differences can be reconciled to develop a generalized, architecture-agnostic pruning methodology.

2.1. Efficiency Techniques

Prior work has mapped the space of deep learning efficiency techniques, including quantization, pruning, knowledge distillation, low-rank approximation, efficient fine-tuning, efficient architectures and more. Work by [51], [54] and [43] pertains to LLMs, and work by [8] describes a very similar set of techniques in the context of general machine learning, including many CNN cases. For enhancing inference efficiency on pretrained models, we consider quantization, pruning and knowledge distillation to be relevant, because they are the ones applicable to pretrained, inference-ready models. By comparing the ways in which they create efficiency gains, we highlight pruning as a candidate for research. Table 2.1 shows a summary of tradeoffs of the described techniques.

2.1.1. Quantization

Quantization is a ubiquitous technique, whereby weights and activations are converted from high-precision datatypes (e.g., 32-bit) to lower-precision datatypes (e.g., 4-bit) [54]. Quantization is often divided into Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ), but for the purpose of efficient inference on pretrained models, we focus on PTQ.

For efficiency gains on pretrained LLMs, [43] report PTQ achieving a 3.9x memory footprint reduction by quantizing 16-bit datatypes down to 4 bits, on a wide range of open-source models. As a result authors also report tripling throughput in tokens per second, while only incurring an average-task performance drop of 3 - 5%. In a survey on model compression techniques, [54] identified latency speedups ranging from 2x to 4.5x. [49] report similar memory footprint reductions due to quantizing from 16 bits to 4 bits, along with throughput speedups ranging from 1.7x to 3x.

For CNNs, [47] compared various PTQ methods on image classification models. On the GoogleNet model, the authors report only a 0.39% classification accuracy penalty using hybrid asymmetric quantization. Applying the same technique to VGG16, YOLOv1 and U-net models, the authors report penalties of 0.52%, 0.72% and 0.64% respectively.

The latency and throughput speedups resulting from PTQ can be explained in multiple ways. For one, PTQ reduces the total amount of data involved in arithmetic operations by reducing the effective number of bits [47]. Another important factor however is hardware support. As argued by [37], support for lower-precision arithmetic varies between accelerator devices, and therefore acceleration may not necessarily manifest itself in all cases.

2.1.2. Knowledge Distillation

Knowledge Distillation (KD) refers to the process of training a smaller, so called "student model" from a pretrained "teacher model". If the student model has a smaller architecture relative to the teacher, this process increases inference efficiency by encoding information from the teacher weights into a lower-footprint representation, saving on memory and compute. It is usually done by passing the same input into both models, and optimizing the weights of the student model such that the divergence between the teacher and student prediction probability distributions is minimized [50].

[3] showed that a 51B parameter model can be distilled from a 70B Llama [41] model, shrinking memory footprint by 27% and improving inference throughput by 2.17x with minimal performance degradation. This was achieved using a mixed-integer programming algorithm to design an optimal student model architecture. [32] used KD to post-train a pruned version of a pretrained model, creating the smaller architecture using pruning techniques. [33] compressed Nemotron-4 models by a factor of 2 - 4x, creating smaller versions that outperform other similarly-sized models on a range of benchmarks.

Similarly for CNNs, [10] distilled 26M parameter ResNet50 and 22M parameter ResNet34 models down to the 12M parameter ResNet18 architecture, effectively reducing model size by more than half. At the same time, this improved image classification performance by up to 2% relative to the ResNet18 model trained from scratch. In a broad performance comparison of CNN KD works, [16] reported improvements in CIFAR10 classification performance on the student models up to 2.91%. This included student-teacher combinations such as ResNet56-ResNet8, reducing memory footprint severalfold.

Although KD can be used to obtain a more lightweight version of a pretrained model, the clear tradeoff is that the training process is memory-intensive. The weights of the teacher and student models must both be stored in memory, along with the gradients of the student model for optimization. The more efficient student model becomes available only once this process is finished. Moreover, finding a good student model architecture could also be a resource-intensive process like in [3].

2.1.3. Pruning

Broadly speaking, pruning refers to the removal of weights deemed unimportant. It can be done on a range of scales balancing granularity and precision against benefits afforded by pruning memory-friendly structures. On the former end, unstructured pruning allows for the removal of individual parameters but only saves on memory or compute if the sparsity can be efficiently represented [27]. On the latter end, structured pruning can involve pruning all parameters in a given layer at a time, which can save on both memory and compute - discarding a matrix of parameters and thus not performing any computations involving that matrix [23].

2.1.4. Tradeoff Analysis

In summary, the mentioned inference efficiency techniques come with tradeoffs that might require application-specific consideration. PTQ effectively reduces the memory footprint, but speedups might be hardware dependent. KD can be used to obtain smaller and faster architectures but requires training. Structured pruning can provide up to layer-wise parameter reductions and speedups, but it introduces the risk of pruning in an overly coarse-grained fashion.

Interestingly, some recent work ([17], [38], [23]) suggests that pruning LLM layers in a coarse fashion is not as destructive than expected. Layer pruning in CNNs has also been shown to be feasible and effective, but the space of prunable layers tends to be restricted by architectural properties. In the following Sections, we sketch the pruning landscape, highlighting evidence that depth pruning (i.e., layer pruning) in particular appears to have a favorable tradeoff profile between accuracy, latency and memory footprint reduction. To leverage this fact, we focus on limitations that need to be addressed in order to develop a layer pruning method that is less dependent on model architecture, motivating our contribution.

Efficiency Technique	**Tradeoffs**
Quantization	Hardware support
Knowledge Distillation	Costly training Student architecture selection
Pruning	Granularity scale selection

Table 2.1: A summary of what issues and tradeoffs need to be taken into account when considering the described efficiency techniques for pretrained models.

2.2. The Pruning Landscape

To build up to our contributions, we start with a simplified taxonomy of pruning techniques by level of granularity. Afterwards, we emphasize how these differences lead to different implementations based on model architecture, and what implications they have on hardware acceleration. Finally, we propose the potential value of reconciling these differences through a unified methodology.

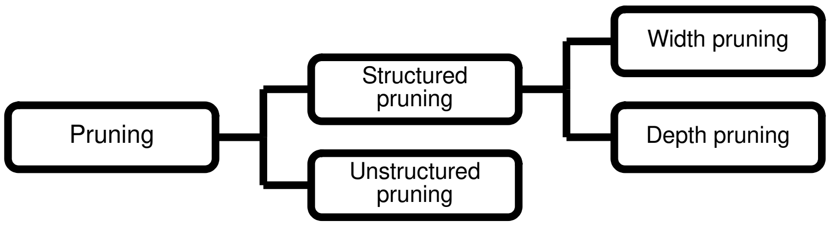

Figure 2.1: A visualization of the simplified pruning taxonomy.

2.2.1. Unstructured Pruning

Unstructured pruning can be thought of as being on the parameter level, lying on the most fine-grained end of the pruning granularity spectrum. Parameters are selected based on metrics usually referred to as "importance" or "salience" metrics (see Section 3.3). Selected parameters are removed by setting them to 0. Starting from a matrix $\mathbf{W}_{\text{orig}}$ collecting all the weights of an unpruned model, this process is usually formalized as finding weights $\mathbf{W}$ and a binary sparsity mask $\mathbf{M}$ such that the MSE between the pruned and unpruned model for an arbitrary input $\mathbf{X}$ is minimized:

$$E(\mathbf{W}) = \min_{\mathbf{M},\mathbf{W}} ||\mathbf{X}(\mathbf{M} \odot \mathbf{W}) - \mathbf{X}\mathbf{W}_{\text{orig}}||_2^2 \tag{2.1}$$

On LLMs, [14] achieved up to 50% unstructured sparsity with a negligible change in WikiText2 perplexity on OPT-175B and BLOOM-176B models without PPFT. Using information from activations for parameter selection, [39] also achieved a 50% sparsity. The authors compared their parameter selection technique against that of [14] on Llama and Llama2 models, achieving slightly improved perplexity. Building on top of this, [53] introduced small weight updates to minimize reconstruction error between dense and sparse models, improving further upon the perplexity results by [39] on sparsity levels from 50% to 70%.

Unstructured pruning also seems to be useful for domain specialization, as [52] achieved up to 50% unstructured sparsity using pruning to specialize Llama2 models for healthcare and legal question-answering.

On CNNs, [31] report a 30% sparsity, 40% FLOPS reduction with an accuracy loss of 0.02% on the ImageNet dataset by iteratively pruning small numbers of parameters and performing PPFT the model after each iteration to recover performance. By optimizing a sparsity mask during the training process, [6] report between 80.2% and 89.9% sparsity on ResNet and MobileNet variants, reducing MAC operations by an order of magnitude, while sacrificing 8% accuracy on ImageNet1K.

This suggests that unstructured pruning can compress the in-memory size of a model, but inference acceleration is a trickier problem. Even though authors report reduced theoretical arithmetic operation counts, this might not necessarily create speedups. On GPU hardware, support for unstructured sparsity is lacking, as vendors prioritize more structured sparsity patterns [35]. Non-GPU hardware seems to lend itself better to unstructured sparsity, but the fact remains that inference acceleration under unstructured pruning is highly hardware-dependent [27].

2.2.2. Structured Pruning

In general, structured pruning usually refers to imposing some kind of structural constraints on the sparsity mask $\mathbf{M}$. This can entail the removal of weights in patterns, the removal of row or column vectors in parameter matrices, removal of entire tensors along the feature dimension, and so on [29]. While all of these are possible in principle, structured LLM and CNN pruning literature tends to focus on sections of weight tensors along some dimension, e.g., rows or channels, which we refer to as "width pruning". Deviating from Equation 2.1, there is also work exploring layer pruning, which we consider an even higher order of structure. In the following Sections, we explore both of these.

Width Pruning

We define width pruning as removing rows or columns in a linear weight tensor $\mathbf{W}_{\text{lin}} \in \mathbb{R}^{o \times i}$, or removing sections along the feature dimension f or channel dimension c of a convolutional weight tensor $\mathbf{W}_{\text{conv}} \in \mathbb{R}^{f \times c \times x \times y}$. This can be done either by truncating the tensor or setting the relevant elements to 0.

In LLMs, this technique can be applied to e.g., the linear sub-layers within multi-head attention (MHA) or multi-layer perceptron (MLP) layers, while in CNNs, it can be applied to convolutional layers. In both cases, the operations performed by width-pruned components lead to a reduction in hidden dimensionality, making the affected parts of the network "slimmer" ([28], [2], [15]).

On CNNs, [13] developed a width pruning tool that removes convolution tensor channels by truncation, automatically grouping together interdependent input and output channels across layers. They report a 2.57x theoretical MAC speedup and a modest <1% increase in CIFAR10 accuracy after pruning and PPFT on ResNet56, along with a 8.92x MAC speedup with a 4% CIFAR100 accuracy penalty on VGG19. [42] also compared sparsity vs. ImageNet accuracy curves across a wide range of CNN architectures and CNN width pruning implementations without PPFT. It was shown that the vast majority of implementations drop accuracy to 0% by the time they reach 30% sparsity. In addition, a review of compression methods [19] for YOLOv5 identified 27 channel pruning papers, all of which use PPFT to achieve strong parameter reductions with minimal performance impacts.

On LLMs, [2] developed an architecture-specific width pruning method, applying it over a range of model sizes from the OPT family and the Llama 2 family, and reporting at most a 15% accuracy reduction on the largest model variants using the WikiText-2 and Alpaca datasets at a 30% parameter reduction without PPFT. The authors also claim a 1.55x throughput improvement and 11% - 17% end-to-end inference time speedups at a 25% parameter reduction, depending on the GPU used. Extending the automated, architecture-agnostic width pruning implementation of [13] to LLMs, [28] obtained a 20% parameter reduction on the Llama-7B, Llama-13B and Vicuna-7B models, incurring a 8% - 12% average performance penalty over a range of benchmarks after PPFT. They also report latency speedups of up to around 16% at the 20% parameter count reduction.

The reported speedups can be explained in a number of ways. Firstly, the authors report using Nvidia A100 GPUs in their experimental setups, which have built-in support for structured sparsity [35]. This is supported by the fact that [23] and [9] find contradictory results, reporting worsened throughput on width-pruned models when using hardware configurations without A100 GPUs. Secondly, [2] claim that the reduced memory footprint allowed for a larger batch size, increasing parallelism efficiency. Thirdly, as suggested by [44], data movement and memory access contributes significantly to latency, and thus reducing the tensor size might lower it.

Depth Pruning

Depth pruning refers to removing entire layers from a model. In LLMs, this means removing Transformer layers or their substructures; in CNNs, it typically means removing convolutional layers. In this work, we allow the term "depth pruning" to encompass multiple layers of granularity, since composite elements like Transformer blocks are referred to as layers in the literature, but so are the smaller elements that Transformer blocks are composed of.

[17] observed that in multiple families of open-source LLMs, up to 30% of Transformer blocks can be pruned with a negligible performance impact on certain benchmarks, without requiring PPFT (illustrated in part A of Figure 2.2). The authors used these results to claim that certain layers might be redundant, at least for certain tasks. Following [17], [30] report a 27% reduction in Transformer layers with even a slight MMLU score improvement, along with a 1.19x throughput speedup on the Llama2 7B model. Looking for redundancy on a smaller scale, [38] treated MLP and MHA layers within Transformer blocks as separate pruning units (illustrated in part B of Figure 2.2), achieving removal of up to 33% of MHA layers without incurring any performance degradation on the MMLU benchmark for Mistral 7B. Similar results were found by [23], reporting a throughput speedup of 1.6x at 67% parameter reduction. The authors also deliberately compare this technique to width pruning, claiming it offers superior speedups by reducing memory access and matrix-level operations. Corroborating this finding, rather than pruning layers statically, [11] trained Llama 2 models to dynamically skip layers in an input-dependent manner, reaching up to a 1.86x inference latency improvement.

On CNNs, [22] pruned convolutional layers along with their nonlinear activation functions on ResNet and MobileNet models using a latency-based importance score, achieving between 1.25x and 2.49x wall-clock time speedups, depending on the exported model format. [5] used layer pruning to create compressed versions of ResNet and VGG models. They report single-label classification accuracies on CIFAR and SVHN that are nearly identical to their unpruned counterparts, along with up to 52.2% reductions in FLOPs, despite removing up to nearly 88% percent of parameters on some networks. Both of the mentioned CNN layer pruning authors used PPFT for performance recovery, either directly or via KD.

In the depth pruning literature, we frequently see speedups reported in terms of wall-clock time and throughput rather than theoretical arithmetic operation counts. The improvements show up more consistently compared to unstructured or width pruning literature, with less dependency on model architecture or hardware. As argued by [44] and [23], preventing memory access entirely by removing full layers might reduce latency more compared to simply reducing the amount of data to access.

Category	Method	Parameter Reduction	Speed	Hardware Support
Width	SliceGPT [2]	25%	1.55x throughput	Yes
Width	LLM-Pruner [28]	25%	-16% latency	Yes
Width	LLM-Pruner [23]	20%	0.82x throughput	Yes
Depth	ShortGPT [30]	27%	1.19x throughput	No
Depth	Shortened LLaMA [23]	67%	1.67x throughput	No

Table 2.2: A comparison of methods where changes in inference speed are reported after pruning.

2.3. Observations

Depth pruning seems to have a good tradeoff profile We have seen that quantization can help us significantly reduce the memory footprint, but the latency improvements might depend on hardware-level low-precision support. We have also seen that KD can help improve both of those variables, but requires considerable resources. With pruning, it appears that as we scale up, we increase the potential for both memory and speed gains.

With unstructured pruning, authors tend to only report parameter count reductions and benchmark performance, and due to the lack of hardware support, there is reason to believe that it does not lend itself to meaningful inference speedups. In the width pruning literature, authors do report some speedups, but we have seen evidence that this hinges on hardware support to some extent. Depth pruning on the other hand seems to provide inference speedups that are less reliant on hardware support.

To emphasize the speedup differences, Table 2.2 highlights methods where changes in inference speed are reported. The column labeled "Speed" shows either a throughput ratio measured in tokens per second (above 1 is a speedup, below 1 is a slowdown) or a relative percent change in inference latency. The column labeled "Hardware Support" indicates whether the speed values might depend on hardware support for structured sparsity. Rows 2 and 3 demonstrate the aforementioned hardware-related contradiction, where [23] used the method by [28].

In terms of accuracy, research suggests that despite its relatively coarse-grained scale, depth pruning does not damage models beyond repair. Some LLMs remain robust after depth pruning even without PPFT, and CNN performance can seemingly be restored using PPFT irrespective of pruning granularity.

LLMs depth-prune easily, CNNs less so Current GPT-style architectures compose LLMs as a sequential, residual chain of Transformer layers. Within a Transformer layer, one usually finds a MHA layer followed by a residual connection and an MLP layer. Due to the residual connections, it is required that activations produced by these layers all have the same dimensionality. As a consequence, to prune a layer, one can simply remove it a re-route the residual connection to the following layer.

In CNNs, convolutional layers generally change the dimensions of the activations. By pruning an arbitrarily selected convolutional layer, one might break the model. If the given layer transforms the channel dimension, pruning it is problematic because the following layer in the sequence is set up to process an input with the transformed channel dimension. To the best of our knowledge, no generally accepted solution exists.

To address the issue, authors like [22] and [5] simply exclude the dimension-transforming layers from the pruning search space. Alternatively, [12] replace weights in affected layers with randomly initialized weights that are dimensionally compatible with the modified representation, while [48] train a lightweight downsampling layer. To prune a YOLOv5 model, [46] restrict the search space to broad parts of the network that preserve structural integrity. The aforementioned review of compressed YOLOv5 models [19] identifies only 2 depth pruning papers out of the total 30, suggesting that the issue is indeed prohibitive. Addressing this problem could significantly expand the search space of prunable layers in CNNs.

LLM depth pruning techniques might be leaving gains on the table We reviewed literature that depth-prunes LLMs by removing Transformer layers, MHA layers and MLP layers. However, we can consider a lower level of abstraction, because MHA and MLP layers contain linear sub-layers that still qualify as depth pruning units (illustrated in part C of Figure 2.2); They are colloquially referred to as layers, they contain their own parameter matrices, and they represent discrete units of memory transfer and computation. If redundancies exist on the Transformer layer level, we posit that redundancies might also exist on this lower level of layers.

Depth pruning LLMs on this level could unlock a more granular search space that still preserves the benefits of depth pruning. To the best of our knowledge, no such work exists, possibly because it presents the same kind of dimensionality issues as when depth pruning CNNs. At the same time, this indicates that there is a general depth pruning problem to solve, and the solution could be applicable to a wide range of architectures, in contrast to the analyzed architecture-specific literature. To address this, we develop *fine-grained depth pruning*.

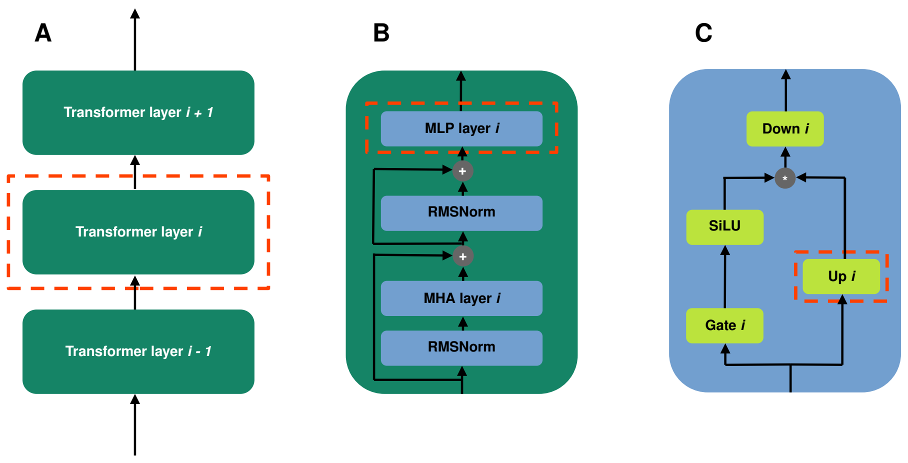

Figure 2.2: An illustration of depth pruning granularities on the Llama2 architecture. The red dashed line illustrates the unit to prune. **A** and **B** demonstrate coarse-grained depth pruning, and **B** demonstrates fine-grained depth pruning.

3 Methodology

In this Chapter, we describe the methods used to develop our fine-grained depth pruning technique. As discussed in Section 2.2.2, depth pruning can occur on a variety of scales due to the varying levels of complexity of neural network components that are generally referred to as "layers".

To leverage the apparent memory footprint and compute benefits of depth pruning while maximizing granularity, we develop techniques for pruning on the smallest scale that the notion of a layer allows. This means pruning on the level of linear transformation components (e.g., linear layers and convolution layers), activation functions, and so on. As described in Section 2.3, this scale presents challenges related to dimensionality, so we also develop techniques for addressing those.

In certain ways, this work builds on top of [13], so some of the notation in the following sections closely resembles theirs. In Section 3.1, we modify and extend their notation to get the required level of expressivity for this work, and in some places restate their definitions in terms of our extended notation. In Sections 3.2 and 3.3, we use this notation to formally define fine-grained depth pruning. In Section 3.4, we aggregate everything from the former Sections to describe the approach to our experiments.

3.1. Notation Preliminaries

In this Section, we state notation that has been used by other pruning works and extend it to describe our depth pruning approach.

3.1.1. Width Pruning

Given a representation of a parametrized layer, e.g., a weight matrix $\mathbf{W} \in \mathbb{R}^{o \times i}$, where i is the input dimension (determined by the number columns) and o is the output dimension (determined by the number of rows), we denote a *width pruning scheme* as $\mathbf{W}[\mathcal{R}, :]$ on the input feature dimension (columns or input channels), and $\mathbf{W}[:, \mathcal{R}]$ on the output feature dimension (rows or output channels). We treat $\mathcal{R}$ as a set containing indices of sections to remove.

Example Let $\mathbf{W} \in \mathbb{R}^{5 \times 5}$ be a linear layer weight tensor from which we want to remove the second and third column. Let $\mathcal{R} = \{2, 3\}$ denote indices of columns to remove. Then $\mathbf{W}[\mathcal{R}, :] \in \mathbb{R}^{5 \times 3}$, i.e., $\mathbf{W}[\mathcal{R}, :]$ is a compressed version of $\mathbf{W}$, with the 2nd and 3rd columns removed.

3.1.2. Dependency Modelling and the Computation Graph

We can decompose a neural network into its components as $\mathcal{F} = \{f_1, f_2, \ldots, f_L\}$, where each component f can represent a(n)

- linear transformation component, i.e., a linear or convolutional layer,
- nonlinear activation function, e.g., ReLU, SiLU,
- normalization function, e.g., LayerNorm, RMSNorm,

- arithmetic operation, e.g., addition, multiplication
- concatenation operation

We then use f^- and f^+ to denote the input and output of f respectively, which lets us further decompose $\mathcal{F}$ into $\{(f_1^-, f_1^+), (f_2^-, f_2^+), ..., (f_L^-, f_L^+)\}$.

To preserve the functional representation, we treat each tuple as an input-output mapping $(f^-, f^+) : \mathbb{R}^i \to \mathbb{R}^o$. For f that represent parametrized layers we also allow width pruning schemes $(f^-, f^+)[\mathcal{R}, :]$ and $(f^-, f^+)[:, \mathcal{R}]$ along with one-dimensional analogs $f^-[\mathcal{R}]$ and $f^+[\mathcal{R}]$.

Now let us treat $\mathcal{F}$ as a set of nodes and define the *computation graph* $G = (\mathcal{F}, E)$. We denote a directed edge as $(f_i^-, f_i^+) \to (f_j^-, f_j^+)$, which exists if f_j requires f_i^+ as an operand during computation. Note that this is a notational convenience - $(f_i^-, f_i^+) \to (f_j^-, f_j^+)$ is a more visual way of saying $((f_i^-, f_i^+), (f_j^-, f_j^+)) \in E$.

For convenience, let us also partition the nodes $\mathcal{F}$ of the computation graph into the following subsets:

- $\mathcal{T}$: The subset of linear transformation component nodes.
- $\mathcal{A}$: The subset of nonlinear activation function nodes.
- $\mathcal{N}$: The subset of normalization function nodes.
- $\mathcal{M}$: The subset of arithmetic operation and concatenation nodes.

Dependency Modelling

To model how width pruning in one place propagates through the network, we restate the two types of dependencies in terms of the above notation. The propagation happens because changing the dimensions of a matrix influences its multiplicative properties. For example, let $\mathbf{x} \in \mathbb{R}^5$ be an input into the network, and let $\mathbf{W}_1 \in \mathbb{R}^{5\times5}$ and $\mathbf{W}_2 \in \mathbb{R}^{5\times5}$ be linear layer weight tensors. Suppose the following computation happens in the network:

$$\mathbf{W}_2 \mathsf{RMSNorm}(\mathbf{W}_1 \mathbf{x})$$

Inter-layer Dependency If we want to width-prune the output dimension on $\mathbf{W}_1$ as $\mathbf{W}_1[:, \{2, 3\}] \in \mathbb{R}^{3\times5}$, we also have to prune the input dimension on $\mathbf{W}_2$ as $\mathbf{W}_2[\{2, 3\}, :] \in \mathbb{R}^{5\times3}$ to maintain matrix multiplication compatibility.

This is an instance of *inter-layer dependency* - a dependency $f_i^- \Leftrightarrow f_j^+$ exists for all $(f_j^-, f_j^+) \to (f_i^-, f_i^+)$.

Intra-layer Dependency RMSNorm is a dimension-preserving, element-wise operation, but some implementations include learnable parameters for element-wise multiplication. Pruning such parameters requires symmetric pruning on both input and output dimensions.

This is an instance of inter-layer dependency - a dependency $f_i^- \Leftrightarrow f_i^+$ exists if f_i^- and f_i^+ require the same pruning scheme, denoted by $\mathsf{sch}(f_i^-) = \mathsf{sch}(f_i^+)$.

If either of these 2 dependency conditions hold, we say there is a **width pruning dependency** between f_i^- and f_j^+, formally

$$D(f_i^-, f_j^+) = \mathbb{1}[(f_j^-, f_j^+) \to (f_i^-, f_i^+)] \vee \mathbb{1}[i = j \wedge \mathsf{sch}(f_i^-) = \mathsf{sch}(f_j^+)]$$

3.2. Depth Pruning Mechanics

In this Section, we mathematically formalize our depth pruning technique.

3.2.1. Parameter Pruning

To depth-prune a layer $(f_\alpha^-, f_\alpha^+) : \mathbb{R}^i \to \mathbb{R}^o$, we first replace it with an identity mapping, denoted as $(\mathsf{id}_\alpha^-, \mathsf{id}_\alpha^+) : \mathbb{R}^i \to \mathbb{R}^i$. In-place updates will be used throughout this work, so we formalize it through this example using the following notation:

$$G \leftarrow G[(f_\alpha^-, f_\alpha^+) \Rightarrow (\mathsf{Id}_\alpha^-, \mathsf{Id}_\alpha^+)] \tag{3.1}$$

Now, let us address the dimensionality complications described in Section 2.3. Note the change in the codomain, i.e., the dimension of the output, resulting from Equation 3.1. If $i \neq o$, then (f_α^-, f_α^+) maps the input to a space of different dimensions. In contrast, $(\mathsf{id}_\alpha^-, \mathsf{id}_\alpha^+)$ maps back to its domain by simply passing the input through. To accommodate for this, we need to use width pruning to adjust the dimensions on nodes for which there exists a width pruning dependency. Formally, if $i \neq o$, then

$$\begin{aligned}
&\text{for all } (f_\beta^-, f_\beta^+) \text{ where } D(f_\alpha^+, f_\beta^-), \quad G \leftarrow G[(f_\beta^-, f_\beta^+) \Rightarrow (f_\beta^-, f_\beta^+)[\mathcal{R}, :]] \text{ with } |\mathcal{R}| = o - i \quad \text{if } i < o, \\
&\text{for all } (f_\beta^-, f_\beta^+) \text{ where } D(f_\beta^+, f_\alpha^-), \quad G \leftarrow G[(f_\beta^-, f_\beta^+) \Rightarrow (f_\beta^-, f_\beta^+)[:, \mathcal{R}]] \text{ with } |\mathcal{R}| = i - o \quad \text{if } i > o.
\end{aligned} \tag{3.2}$$

For convenience, let $\mathcal{W}$ be a set collecting all such (f_β^-, f_β^+), i.e., $\mathcal{W}$ is the set of all nodes that have a width pruning dependency on (f_α^-, f_α^+).

We discuss choosing $\mathcal{R}$ in Section 3.3. Through the mechanisms defined here, we fix dimension conflicts that might arise if a layer is pruned by replacement with an identity mapping. Note that after such a replacement, $\mathcal{T}$ will contain a subset $\mathcal{I}$ of identity mapping nodes.

3.2.2. Operation Pruning

For the goal of subsuming methods that prune composite block layers, e.g., Transformers, and maximizing latency gains, we need a way to also prune redundant activation and normalization functions. We treat the pruned node $(\mathsf{id}_\alpha^-, \mathsf{id}_\alpha^+) \in \mathcal{I}$ as a starting point for searching the graph, and we think of surrounding activation and normalization nodes as being coupled to $(\mathsf{id}_\alpha^-, \mathsf{id}_\alpha^+)$ and thus prunable along with it.

We think of a normalization node $n \in \mathcal{N}$ as occurring before transformation node $t \in \mathcal{T}$ in the computation order, normalizing the activations in preparation for the linear transformation. We treat n as coupled to t if there is no other transformation node $t' \neq t$ that immediately follows n. This is formalized in Algorithm 2.

The case for an activation node $a \in \mathcal{A}$ is analogous, except we treat it as coming after t in the computation order. This is formalized in Algorithm 1.

Using these auxiliary algorithms, we depth-prune activation nodes coupled to $(\mathsf{id}_\alpha^-, \mathsf{id}_\alpha^+)$ as follows:

$$\begin{gathered}
G \leftarrow G[(f_\beta^-, f_\beta^+) \Rightarrow (\mathsf{id}_\beta^-, \mathsf{id}_\beta^+)] \\
\text{for all } (f_\beta^-, f_\beta^+) \in \texttt{FindAct}((\mathsf{id}_\alpha^-, \mathsf{id}_\alpha^+), T)
\end{gathered} \tag{3.3}$$

And we depth-prune normalization nodes coupled to $(\mathsf{id}_\alpha^-, \mathsf{id}_\alpha^+)$ as follows:

$$\begin{gathered}
G \leftarrow G[(f_\beta^-, f_\beta^+) \Rightarrow (\mathsf{id}_\beta^-, \mathsf{id}_\beta^+)] \\
\text{for all } (f_\beta^-, f_\beta^+) \in \texttt{FindNorm}((\mathsf{id}_\alpha^-, \mathsf{id}_\alpha^+), T)
\end{gathered} \tag{3.4}$$

T is a set of transformation nodes that are expected to be coupled to the activation and normalization nodes that the algorithm finds. In most cases, simply $T = \{t\}$, i.e., containing only the origin transformation node. There is however an edge case for the MHA operation, when the following subgraph exists in G:

In this situation, line 7 in Algorithm 2 would find all 3 linear nodes, even if the search is initialized from only one of them, so we must set T to include all of them. We discuss pruning of MHA operations in-depth in Section 3.2.4.

Algorithm 1 `FindAct`

```
1: $t \in \mathcal{T}$: Starting transformation node
2: $T$: Set of expected coupled transformation nodes
3: $X \leftarrow$ ForwardBoundedSearch($t, \mathcal{T} \cup \mathcal{M}$)
4: $A \leftarrow X \cap \mathcal{A}$
5: $A' \leftarrow \emptyset$
6: for $a \in A$ do
7:     $X' \leftarrow$ BackwardBoundedSearch($a, \mathcal{T} \cup \mathcal{M}$)
8:     if $X' \cup \mathcal{T} = \mathrm{T}$ then
9:         $A' \leftarrow A' \cup \{a\}$
10:    end if
11: end for
12: return $A'$
```

Algorithm 2 `FindNorm`

```
1: $t \in \mathcal{T}$: Starting transformation node
2: $T$: Set of expected coupled transformation nodes
3: $X \leftarrow$ BackwardBoundedSearch($t, \mathcal{T} \cup \mathcal{M}$)
4: $A \leftarrow X \cap \mathcal{N}$
5: $N' \leftarrow \emptyset$
6: for $n \in N$ do
7:     $X' \leftarrow$ ForwardBoundedSearch($a, \mathcal{T} \cup \mathcal{M}$)
8:     if $X' \cup \mathcal{T} = \mathrm{T}$ then
9:         $N' \leftarrow N' \cup \{n\}$
10:    end if
11: end for
12: return $N'$
```

Algorithm 3 `BackwardBoundedSearch`

```
1: $n \in \mathcal{F}$: Starting node
2: $B$: Set of boundary nodes to terminate the search at
3: $X \leftarrow \emptyset$
4: Do a depth-first search from $n$ in $G$, following the directed edges backwards, collecting seen nodes
   in $X$, terminating if the last collected node is in $B$
5: return $X$
```

Algorithm 4 `ForwardBoundedSearch`

```
1: $n \in \mathcal{F}$: Starting node
2: $B$: Set of boundary nodes to terminate the search at
3: $X \leftarrow \emptyset$
4: Do a depth-first search from $n$ in $G$, following the directed edges forwards, collecting seen nodes in
   $X$, terminating if the last collected node is in $B$
5: return $X$
```

$$\begin{tikzcd}
 & (\mathsf{Linear}^-_Q, \mathsf{Linear}^+_Q) \\
(\mathsf{Norm}^-, \mathsf{Norm}^+) \arrow[ur] \arrow[r] \arrow[dr] & (\mathsf{Linear}^-_K, \mathsf{Linear}^+_K) \\
 & (\mathsf{Linear}^-_V, \mathsf{Linear}^+_V)
\end{tikzcd}$$

3.2.3. Identity Patching

To continue towards architecture-agnostic depth pruning while maximizing latency gains, it helps to deal with identity-induced redundancies around nodes in $\mathcal{M}$.

Arithmetic Nodes

If we apply the parameter pruning (Equations 3.1 and 3.2) and operation pruning (Equations 3.3 and 3.4) steps more than once, we might modify G such that there exists a binary arithmetic operation node $m \in M$ with both operands identity nodes, inducing a redundant arithmetic operation of an input with itself. This happens if G contains one or more instances of the following subgraph

$$\begin{tikzcd}
 & (\mathsf{Id}^-, \mathsf{Id}^+)^* \arrow[dr] & \\
(f^-, f^+) \arrow[ur] \arrow[dr] & & (m^-, m^+) \\
 & (\mathsf{Id}^-, \mathsf{Id}^+)^* \arrow[ur] &
\end{tikzcd}$$

where $(\mathsf{Id}^-, \mathsf{Id}^+)^*$ denotes a path of one or more identity nodes.

When such a situation occurs, it can be an opportunity to remove leftover operations when pruning composite computations (e.g., residual architecture components or gated MLP [26] components) to remove their effects on the activations completely. This allows us to recreate the effects of pruning these composite components on a block level.

As an example, consider the following representation of the gated MLP block used in Llama models:

$$\begin{tikzcd}
 & (\mathsf{Linear}^-_{\text{Gate}}, \mathsf{Linear}^+_{\text{Gate}}) \arrow[rr] & & (\mathsf{SiLU}^-, \mathsf{SiLU}^+) \arrow[d] \\
(\mathsf{Norm}^-, \mathsf{Norm}^+) \arrow[ur] \arrow[rr] & & (\mathsf{Linear}^-_{\text{Up}}, \mathsf{Linear}^+_{\text{Up}}) \arrow[r] & (\mathsf{Mult}^-, \mathsf{Mult}^+)
\end{tikzcd}$$

Applying the parameter pruning steps and operation pruning steps starting from $(\mathsf{Linear}^-_{\text{Gate}}, \mathsf{Linear}^+_{\text{Gate}})$ and $(\mathsf{Linear}^-_{\text{Up}}, \mathsf{Linear}^+_{\text{Up}})$, all nodes in the diagram above would become identity nodes except for $(\mathsf{Mult}^-, \mathsf{Mult}^+)$.

Intuitively, the idea is to follow the identity operands of m backwards in G and check if there exists a path of identity nodes originating from a shared transformation node. If this is the case, the operands of m are both copies of the same input, and we can replace one of the copies with the identity element of the arithmetic operation. For example, if m represents $f^+ + f^+$, then the goal is to obtain $f^+ + 0$.

Recalling that $\mathcal{I}$ is the set of identity nodes created as in Equation 3.1, the search process is formalized in Algorithm 5. Using this, let $(\mathsf{Id}(m)^-, \mathsf{Id}^+(m))$ be a node that returns the identity element corresponding to the arithmetic operation of m. We perform identity patching on m as follows:

$$G \leftarrow G[(p^-, p^+) \Rightarrow (\mathsf{Id}(m)^-, \mathsf{Id}^+(m))] \qquad (3.5)$$
$$\text{for one arbitrarily chosen } (p^-, p^+) \in \texttt{GetRedundantIDs}(m)$$

Concatenation Nodes

In applying pruning steps, we might also modify G such that there exists a concatenation node $c \in \mathcal{M}$ with all operands identity nodes, inducing a concatenation operation resulting in a concatenation of repeating copies. This happens if for fixed (f^-, f^+) and (c^-, c^+), G contains two or more paths of the form

$$(f^-, f^+) \rightarrow (\mathsf{Id}^-, \mathsf{Id}^+)^* \rightarrow (c^-, c^+),$$

i.e., we concatenate two or more copies of f^+ via pruned identity paths when all relevant information is encoded in only one.

Let $(\mathsf{Emp}^-, \mathsf{Emp}^+)$ be a node that returns an empty tensor. We use the empty tensor to prevent all but one copy of f^+ from being inputted into the concatenation operation, performing identity patching on c as follows:

$$G \leftarrow G[(p^-, p^+) \Rightarrow (\mathsf{Emp}^-, \mathsf{Emp}^+)] \qquad (3.6)$$
$$\text{for all except one } (p^-, p^+) \in \texttt{GetRedundantIDs}(c)$$

$$G \leftarrow G[(s^-, s^+) \Rightarrow (s^-, s^+)[\mathcal{R}, :]] \text{ with } |\mathcal{R}| = \dim(s^-) - \dim(f^+) \qquad (3.7)$$
$$\text{for all } (s^-, s^+) \text{ where } (c^-, c^+) \rightarrow (s^-, s^+)$$

In Equation 3.6 we modify G very similarly to arithmetic node patching in Equation 3.5, except we replace all but one node with the empty tensor node. For concatenation nodes, we need to perform an additional step described in 3.7. By passing empty tensors to the concatenation node, we are changing its output dimension, so we need to width-prune the input dimension on all successor nodes accordingly.

Algorithm 5 `GetRedundantIDs`

```
1:  $m \in \mathcal{M}$: Starting arithmetic operation or concatenation node
2:  operands $\leftarrow [(p^-, p^+) | \forall (p^-, p^+) : (p^-, p^+) \rightarrow (c^-, c^+)]$
3:  $P \leftarrow \emptyset$
4:  for op $\in$ operands do
5:      if op $\in \mathcal{I}$ then
6:          $P \leftarrow P \cup$ FollowIdOperand(op, $\mathcal{I}$)
7:      else
8:          return $\emptyset$
9:      end if
10: end for
11: if $|P \cap \mathcal{T}| = 1$ then
12:     return operands
13: else
14:     return $\emptyset$
15: end if
```

3.2.4. Attention Pruning

The depth pruning procedures defined thus far work for transformation nodes that exist outside of MHA blocks. The transformation nodes within MHA blocks cannot be pruned in the same way because together, they make up the overall attention head logic. Instead, following [38], we simply treat the entire MHA operation as a prunable unit.

Algorithm 6 `FollowIDOperand`

```
1: $n \in \mathcal{I}$: Starting identity operand node
2: $X \leftarrow \emptyset$
3: Do a breadth-first search from $n$ in $G$, following the directed edges backwards, collecting seen nodes
   in $X$, terminating if the last collected node is not in $\mathcal{I}$
4: return $X$
```

We prune the MHA block by first pruning activation and normalization nodes coupled to it using the procedures defined in Equations 3.3 and 3.4 respectively. Then we replace the constituent MHA nodes with a single identity node and route the MHA input and output edges through it. This is formalized in Algorithm 7.

Note the different choices of T in lines 1 and 3. This solves the problem described in Section 3.2.2. Returning to this in more detail, we restate the subgraph that exists in G if it contains an MHA operation

$$\begin{tikzcd}
 & (\mathsf{Linear}^-_Q, \mathsf{Linear}^+_Q) \\
(\mathsf{Norm}^-, \mathsf{Norm}^+) \arrow[ur] \arrow[r] \arrow[dr] & (\mathsf{Linear}^-_K, \mathsf{Linear}^+_K) \\
 & (\mathsf{Linear}^-_V, \mathsf{Linear}^+_V)
\end{tikzcd}$$

and consider the following formulation of the MHA operation, noting that the query, key and value projection matrices $\mathbf{W}^Q$, $\mathbf{W}^K$ and $\mathbf{W}^V$ are represented by $(\mathsf{Linear}^-_Q, \mathsf{Linear}^+_Q,)$, $(\mathsf{Linear}^-_K, \mathsf{Linear}^+_K,)$ and $(\mathsf{Linear}^-_V, \mathsf{Linear}^+_V,)$ respectively:

$$\begin{aligned} \mathrm{MultiHead}(\mathbf{X}) &= \mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_h)\mathbf{W}^O \\ \text{where } \mathrm{head}_i &= \mathrm{Attention}(\mathbf{X}\mathbf{W}^Q_i, \mathbf{X}\mathbf{W}^K_i, \mathbf{X}\mathbf{W}^V_i) \end{aligned}$$

The head computation involves a multiplication of one input $\mathbf{X}$ by 3 different projection matrices independently. As a consequence, nodes $(\mathsf{Linear}^-_Q, \mathsf{Linear}^+_Q,)$, $(\mathsf{Linear}^-_K, \mathsf{Linear}^+_K,)$ and $(\mathsf{Linear}^-_V, \mathsf{Linear}^+_V,)$ are immediate successors of $(\mathsf{Norm}^-, \mathsf{Norm}^+)$, and therefore, a search using Algorithm 4 should encounter at least those 3 nodes. This is configured on line 3.

For line 1, we observe that the multiplication by $\mathbf{W}^O$ is the final computation that occurs in the MHA operation (not depicted in the graph above), and thus it is the only node to look for when searching backwards with Algorithm 3.

3.2.5. Process Definitions

Now that we defined the steps for depth pruning, width pruning for dimension reconciliation and operation pruning, we can define shorthand syntax for the full process, starting from a trigger layer f.

Definition We write $\mathsf{Prune}(f)$ to denote the following modifications to G:

- If f is a non-MHA transformation layer,
  1. Pruning f by replacement with an identity mapping Id per (3.1).
  2. If applicable, width pruning for dimension reconciliation per (3.2).
  3. Pruning coupled activation and normalization nodes starting from Id.
- If f is an MHA operation, applying Algorithm 7.

Definition We refer to the set of nodes replaced and modified by $\mathsf{Prune}(f)$ as the **pruning unit** induced by f. We refer to f as the **trigger**.

Algorithm 7 `PruneMHABlock`

```
1: $T_1 \leftarrow \{(\mathsf{Linear}_O^-, \mathsf{Linear}_O^-)\}$
2: Prune coupled activation nodes using Equation 3.3 using $T = T_1$
3: $T_2 \leftarrow \{(\mathsf{Linear}_Q^-, \mathsf{Linear}_Q^-), (\mathsf{Linear}_K^-, \mathsf{Linear}_K^-), (\mathsf{Linear}_V^-, \mathsf{Linear}_V^-)\}$
4: Prune coupled activation nodes using Equation 3.4 using $T = T_2$
5: Cut $G$ into partitions $G_H$ that contains all nodes of the given MHA operation, and $G'$ that contains all other nodes, and let $H$ be the cut-set
6: Add a new node $(\mathsf{Id}_H^-, \mathsf{Id}_H^+)$ into $G$
7: for $(f_\alpha^-, f_\alpha^+) \rightarrow (f_\beta^-, f_\beta^+) \in H$ where $(f_\alpha^-, f_\alpha^+)$ is in $G_H$ do
8:     Add an edge $(\mathsf{Id}_H^-, \mathsf{Id}_H^+) \rightarrow (f_\beta^-, f_\beta^+)$ into $G$
9: end for
10: for $(f_\alpha^-, f_\alpha^+) \rightarrow (f_\beta^-, f_\beta^+) \in H$ where $(f_\beta^-, f_\beta^+)$ is in $G_H$ do
11:     Add an edge $(f_\alpha^-, f_\alpha^+) \rightarrow (\mathsf{Id}_H^-, \mathsf{Id}_H^+)$ into $G$
12: end for
13: return
```

3.3. Importance Estimation

Using our method requires an importance metric for solving two problems:

1. Selecting nodes to depth prune.
2. Selecting indices $\mathcal{R}$ for width pruning if dimension reconciliation is required.

First, to obtain a per-parameter importance estimate, we utilize the widely-used Taylor importance score ([25], [31], [23], [28]). Given the network weights, $\mathbf{W}$, using a calibration dataset $\mathcal{D}$ assumed to be i.i.d. (see Table 3.3), we estimate the prediction error caused by setting parameter w_m to 0 (our method of truncating tensor sections is semantically equivalent to setting the sections to 0)

$$\mathcal{I}_m = (E(\mathcal{D}, \mathbf{W}) - E(\mathcal{D}, \mathbf{W} | w_m = 0))^2$$

using a first-order Taylor expansion around $\mathbf{W}$

$$\mathcal{I}_m \approx \left(\frac{\partial E}{\partial w_m} w_m\right)^2.$$

Following [31], we measure the group importance of a set of parameters $\mathcal{S}$ as a sum of individual parameter contributions

$$\mathcal{I}_{\mathcal{S}} = \sum_{m \in \mathcal{S}} \mathcal{I}_m. \tag{3.8}$$

Using this, to measure the importance of a transformation layer f with weights $\mathbf{W}_f$, let $\mathcal{S}_{\mathbf{W}}$ collect all parameters in $\mathbf{W}_f$ and let

$$\mathcal{I}_f = \sum_{m \in \mathcal{S}_{\mathbf{W}}} \mathcal{I}_m. \tag{3.9}$$

Selecting Width Pruning Indices If depth pruning f requires width pruning for dimension reconciliation, recall that 3.2 implicitly defines a set of nodes $\mathcal{W}$ on which width pruning is applied. Since all nodes in $\mathcal{W}$ share an axis on which width pruning is applied (e.g., input/output channels, rows/columns), we can obtain a sequence $(\mathcal{I}_1, ..., \mathcal{I}_k)$ where $\mathcal{I}_j$ is the total Taylor importance score of the axis section at index j, aggregated over all nodes in $\mathcal{W}$ (see [13] for details).

Then, to select n axis sections to width-prune for dimension reconciliation, we set $\mathcal{R}$ to contain indices of the n lowest elements in $(\mathcal{I}_1, ...\mathcal{I}_k)$.

With Equation 3.9 defining the importance score of a single transformation node f in isolation, we can now use the definition of $\mathcal{R}$ to define the importance score $\mathcal{I}_f^*$ of the full pruning unit induced by f:

$$\mathcal{I}_f^* = \mathcal{I}_f + \sum_{j \in \mathcal{R}} \mathcal{I}_j \tag{3.10}$$

LLM-Specific Importance Estimation We found the Taylor-based transformation node importance score in Equation 3.9 ineffective for the large sums of parameters in LLM transformation layer weights (see Section 4.3). Instead, to define $\mathcal{I}_f^*$ in LLMs, we follow [17] in using *angular distance*.

$$\mathcal{I}_f^* = E\left[\frac{1}{\pi}\arccos\left(\frac{y_T^G \cdot y_T^{G[\mathsf{Prune}(f)]}}{\|y_T^G\|\|y_T^{G[\mathsf{Prune}(f)]}\|}\right)\right], \tag{3.11}$$

Since we defined a separate process for pruning MHA operations, we take f to be either a non-MHA transformation layer, or an MHA operation, within a given Transformer block T. Moreover y_T^G is the output of T in the default, unpruned state, and $y_T^{G[\mathsf{Prune}(f)]}$ is the output of T with $\mathsf{Prune}(f)$ applied. If f is a non-MHA transformation layer and applying $\mathsf{Prune}(f)$ requires width pruning for dimension reconciliation, the sections are selected as described in Section 3.3. The expectation is computed by simply taking the mean over the calibration dataset $\mathcal{D}$.

3.4. Experimental Setup

In this Section, we describe our implementation of the methods described above, our choice of models for testing, the hardware setup and the experimental workflow.

3.4.1. Implementation Details

Data structures for the computation graph formalized in Section 3.1.2 along with the dependency width pruning technique for dimension reconciliation described in Equations 3.2 are supported by the DepGraph tool developed by [13].

All other techniques described in Section 3.2 are realized in our custom implementations, leveraging PyTorch's [34] Module system and the DepGraph implementation, treating them as two parallel representations of the model. The code is available at this URL. All nodes in our computation graph directly correspond to nodes in the DepGraph data structure, some of them map to a PyTorch Module defining the given layer, and pruning-related modifications are made on the PyTorch Module level.

The DepGraph implementation is derived from the PyTorch AutoGrad mechanism. As a consequence, what we call the "computation graph" in this work is closely related to the AutoGrad computational graph, but is not intended to be a complete mathematical formalization of it. The purpose of the general mathematical formalization is to allow for potential future implementations using frameworks other than PyTorch.

3.4.2. Experiment Details

Experiments are conducted on Llama2 [41] and YOLOv5 [21] models. The experiments involve pruning the models to various levels of parameter reduction to observe tradeoffs between parameter reduction, task performance, latency reduction and prunable layer search space.

The Llama2 models were chosen to represent LLMs in our experiments because they are free and open-source, and therefore widely used in the literature. One such example is [17], which to the best of our knowledge contains state-of-the-art, peer-reviewed results for coarse-grained Transformer layer pruning on Llama2. We treat this work as a baseline, comparing our fine-grained depth pruning method against our reproduction of their results. The models used in the experiments are outlined in Table 3.1.

YOLOv5 was chosen to represent CNNs due to its widespread mainstream adoption and architectural complexity. Due to the architectural complexity, YOLOv5 depth pruning work is limited to pruning

Model	Parameter Count (B)
Llama2 13B	13.016
Llama2 7B	6.738

Table 3.1: Overview of Llama2 models used in experiments.

specific parts of the models where dimension conflicts do not occur, and this tends to be the case for general CNN depth pruning work too. For this reason, we do not compare parameter reduction and task performance tradeoffs against any baseline like we do for Llama2, because to the best of our knowledge, no such baseline exists. Instead, we treat this as an exploratory problem. Since our method is intended to expand the search space of prunable CNN layers, we do compare our search space against methods with more restricted search spaces. The models used in the experiments are outlined in Table 3.2.

Table 3.2: Overview of Llama2 models used in experiments.

Model	Parameter Count (M)
YOLOv5X	8.671
YOLOv5L	4.653
YOLOv5M	2.117

Hardware Llama2 experiments were conducted on the DelftBlue HPC cluster [1] at the Delft University of Technology. The cluster is equipped with Nvidia A100 80GB GPUs. YOLOv5 experiments were conducted on a personal computer equipped with an Nvidia RTX 4060 8GB GPU to avoid long queue waiting times on DelftBlue.

Evaluation Llama2 benchmarking was carried carried using the Language Model Evaluation Harness by Eleuther AI, only setting the benchmark name and batch size, leaving all other settings default. The benchmarks used are MMLU [18] for multi-domain general knowledge, PIQA [4] for commonsense reasoning, and BoolQ [7] for reading comprehension. YOLOv5 models were evaluated using the validation script provided in the YOLOv5 Github repository, using the validation subset of the COCO dataset, since this is the default dataset for evaluating YOLOv5 models. All used datasets are listed in Table 3.3. Batch sizes are listed in Table 3.4.

Purpose	**Dataset**	**Subset**	**Num. Samples Used**
Llama2 angular distance estimation	`arcee-ai/sec-data-mini`	train	2048
Llama2 cross-entropy estimation	`arcee-ai/sec-data-mini`	train	2048
Llama2 perplexity experiment	`SamuelYang/bookcorpus`	train	2048
Llama2 Taylor gradient estimation	`SamuelYang/bookcorpus`	train	20
YOLOv5 Taylor gradient estimation	COCO128	val	All
YOLOv5 evaluation	COCO	val	All

Table 3.3: Datasets used for various tasks. Datasets denoted with `this font` also serve as HuggingFace locators.

Llama2 Experiment Workflow

Algorithm 8 describes the pruning procedure for Llama2 models. The importance of each pruning unit is computed using the angular distance score, but non-MHA and MHA pruning units are collected and sorted separately. This was decided because angular distance produced good orderings (w.r.t results) only when non-MHA and MHA pruning units were ranked separately, and we were unable to find an importance metric that produced a good global, shared ordering (see Section 4.3). After that, we prune 2 non-MHA units for each MHA unit. This was selected as a heuristic for picking from the split orderings because 2 non-MHA units contain roughly as many parameters as one MHA unit in the Llama2 models.

Model Type	Batch Size
Llama2	8
YOLOv5	50

Table 3.4: Batch sizes used for evaluation.

Note that for the YOLOv5 models, importance was computed in an iterative fashion (see Section 3.4.2), whereas for the Llama2 models, importance was computed in one shot. This was due to the higher resource requirements of obtaining accurate angular distance (3.11) measurements. See Table 3.3 for more details.

Algorithm 8 `PruneLlama`

```
1:  $\tau \leftarrow$ predefined parameter reduction threshold
2:  $n_t \leftarrow$ total unpruned parameter count
3:  $n_p \leftarrow n_t$
4:  transform_layers $\leftarrow$ mapping from each transformation layer to its importance score $\mathcal{I}_f^*$ per (3.11)
5:  transform_layers $\leftarrow$ transform_layers sorted by importance scores in ascending order
6:  mha_layers $\leftarrow$ mapping from each MHA layer to its importance score $\mathcal{I}_f^*$ per (3.11)
7:  mha_layers $\leftarrow$ transform_layers sorted by importance scores in ascending order
8:  transform_index $\leftarrow 0$
9:  mha_index $\leftarrow 0$
10: total_index $\leftarrow 0$
11: while $n_p/n_t < \tau$ do
12:     if total_index $\bmod 3 = 1$ then
13:         $f \leftarrow$ mha_layers[mha_index]
14:         mha_index $\leftarrow$ mha_index $+ 1$
15:     else
16:         $f \leftarrow$ transform_layers[transform_index]
17:         transform_index $\leftarrow$ transform_index $+ 1$
18:     end if
19:     Prune($f$)
20:     $n_p \leftarrow$ updated parameter count
21:     total_index $\leftarrow$ total_index $+ 1$
22: end while
23: for all $m \in \mathcal{M}$ do
24:     Patch $m$ using (3.5) or (3.6)
25: end for
```

YOLOv5 Experiment Workflow

Algorithm 9 describes the pruning procedure for YOLOv5 models. The importance of each pruning unit is computed using the Taylor importance score. The trigger layer for each pruning unit is associated with a score, and the nodes are sorted by ascending importance. The layers and their total pruning units are pruned in ascending order of importance, until a parameter reduction threshold is reached.

Algorithm 9 `PruneYOLO`

```
1: $\tau \leftarrow$ predefined parameter reduction threshold
2: $n_t \leftarrow$ total unpruned parameter count
3: $n_p \leftarrow n_t$
4: while $n_p/n_t < \tau$ do
5:     layers $\leftarrow$ mapping from each convolutional layer $f$ to its importance score $\mathcal{I}_f^*$ per (3.10)
6:     layers $\leftarrow$ layers sorted by importance scores in ascending order
7:     $f \leftarrow$ lowest importance layer in layers
8:     Prune($f$)
9:     $n_p \leftarrow$ updated parameter count
10: end while
11: for all $m \in \mathcal{M}$ do
12:     Patch $m$ using (3.5) or (3.6)
13: end for
```

4
Results

In this Chapter, we present the results of our experiments. Section 4.1 pertains to Llama2 LLMs and Section 4.2 pertains to YOLOv5 CNNs. Additionally, Section 4.3 presents ablation study results that provide evidence for our choices of importance metrics (see Section 3.3, and Section 4.4 presents results of experiments that attempt to explain the differences between hypothesized and realized results from the Llama2 experiments.

4.1. Llama2 Results

In this Section, we use the work by [17] as the baseline for comparison. To the best of our knowledge, it contains state-of-the-art, peer-reviewed results for coarse-grained Transformer layer pruning on Llama2. We compare results obtained by our fine-grained depth pruning method against our reproduction of their results, as the authors did not make their raw data available, and we considered reading it off of their figures too imprecise.

4.1.1. MMLU Performance

Figure 4.1 compares scores on the MMLU benchmark [18] on the Llama2 13B model. We observe that the baseline preserves performance up to a parameter reduction of approximately 34%, before a rapid phase change where performances collapses to the level of random guessing, making our reproduction consistent with the baseline paper. Our fine-grained method preserves performance up to 41%, albeit with a dip at 27% and subsequent recovery over the next 2 steps. We propose an explanation for this effect in Section 4.4.

Figure 4.2 compares MMLU scores on the Llama2 7B model. The baseline once again appears consistently reproduced with large fluctuations, including a dip to random with recovery, before a final collapse to random at 48%. Our method outperforms the baseline up to 24% but collapses to random earlier, at 30%. For this model, we observe dip-recovery behavior in both methods, with ours being less dramatic, as it does not dip to random. While neither method has the robustness of the 13B model, our method appears to trade per-parameter-performance for reduced fluctuation.

In Table 4.1 we highlight points in Figures 4.1 and 4.2 where our method outperforms the baseline on MMLU. These are the 41% reduction ("labeled Llama2 13B 0.41") and 24% reduction ("labeled Llama2 13B 0.24") points respectively. We include detailed parameter counts, the validation latency, i.e., time taken to perform all inference passes on the benchmarks, along with MMLU scores.

We observe that with both methods, parameter count reduction is approximately proportional to latency reduction. However, only our method allows us to achieve a new operator point - a 41% parameter and latency reduction that preserves 94% of the MMLU score on the Llama2 13B model. On the Llama2 7B model, the enhanced robustness of our method allows us to outperform the baseline at a 24% parameter reduction in a similar fashion. However, the baseline method outperforms ours at greater parameter reductions, as its larger fluctuations stabilize.

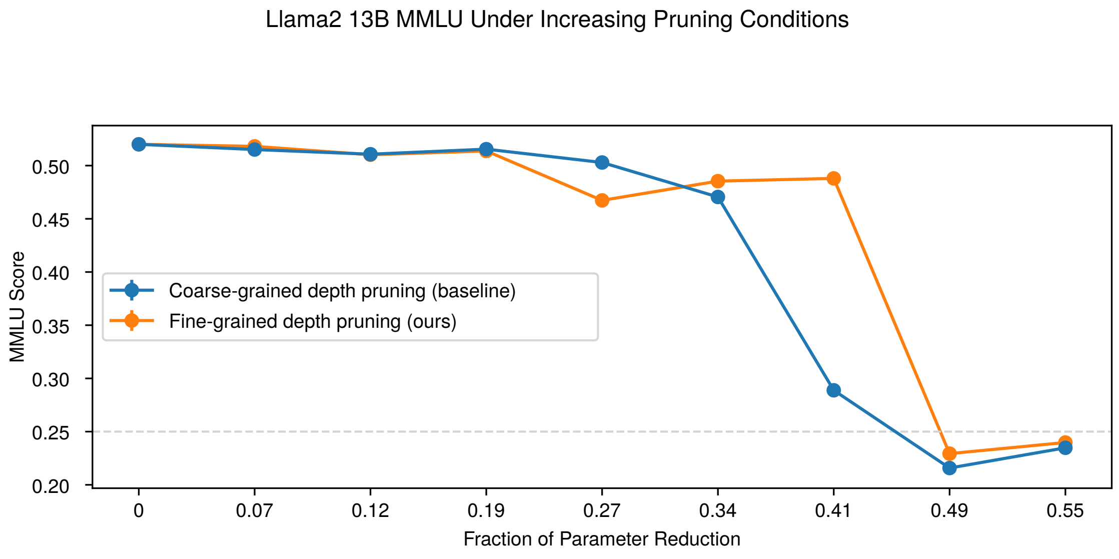

Figure 4.1: A comparison of MMLU scores under various amounts of parameter reduction on the Llama2 13B model. The baseline for comparison (blue) is reproduced from Figure 2 in [17]. The gray dashed line represents the random guessing score on MMLU.

Model	Parameter Count (B)	Total Val Latency (s)	MMLU Score
Llama2 13B	13.016	1113	0.520
Llama2 13B 0.41 Coarse-grained	7.679	652	0.289
Llama2 13B 0.41 Fine-grained	7.679	656	0.488
Llama2 7B	6.738	588	0.420
Llama2 7B 0.24 Coarse-grained	5.121	460	0.236
Llama2 7B 0.24 Fine-grained	5.121	455	0.372

Table 4.1: A comparison of points in Figures 4.1 and 4.2 where our method outperforms the baseline on MMLU.

Figure 4.3 shows an illustration of the state of the pruned Llama2 13B model at the 41% parameter reduction point using our fine-grained depth pruning method. The figure represents Transformer layers in the model and the modifications that have been made to the layers by the pruning process. In the illustration, components labeled "Zero" represent an additive identity, and components labeled "One" represent a multiplicative identity. The behavior of layers 23 - 25 is equivalent to what it would be if they were pruned in a coarse-grained fashion - the full Transformer layers are identity mappings that make no changes to the input.

4.1.2. Search Space Comparison

A deliberate feature of our method is the increased granularity of pruning unit search space. To the best of our knowledge, the most fine-grained SOTA methods treat MHA layers and MLP layers as smallest pruning units, while our method enables pruning inside the MLP layer. This is quantified in Table 4.2.

Technique	Llama2 7B		Llama2 13B	
	Pruning Units	**Total**	**Pruning Units**	**Total**
[17]	32 Transformer layers	32	40 Transformer layers	40
[38]	32 MHA layers, 32 MLP layers	64	40 MHA layers, 40 MLP layers	80
Ours	32 MHA layers, 96 MLP sub-layers	**128**	40 MHA layers, 120 MLP sub-layers	**160**

Table 4.2: A comparison of the compositions of prunable Llama2 layers when using our method compared to others.

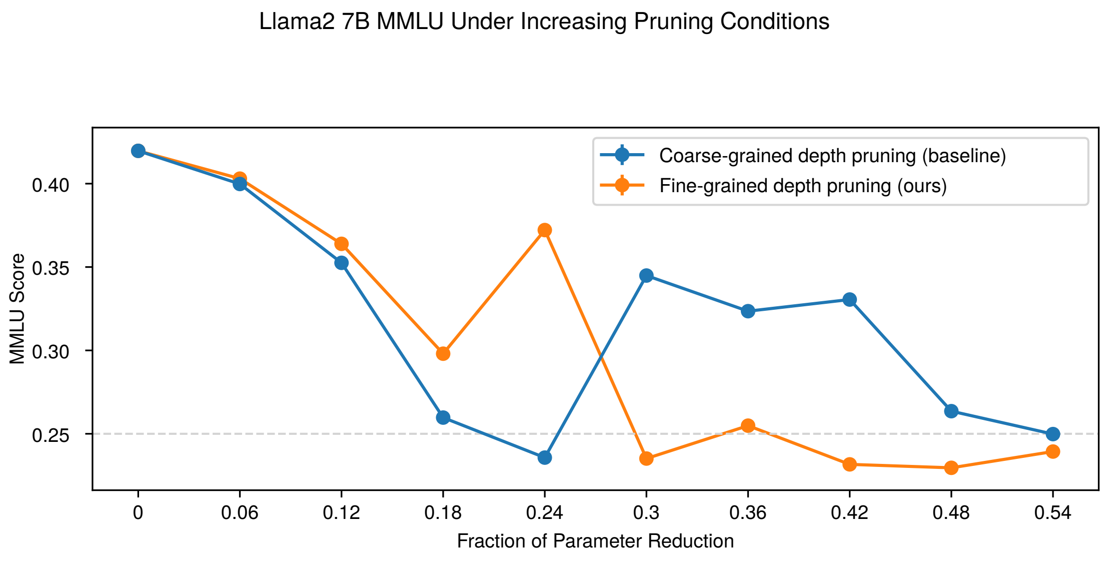

Figure 4.2: A comparison of MMLU scores under various amounts of parameter reduction on the Llama2 7B model. The baseline for comparison (blue) is reproduced from Figure 2 in [17]. The gray dashed line represents the random guessing score on MMLU.

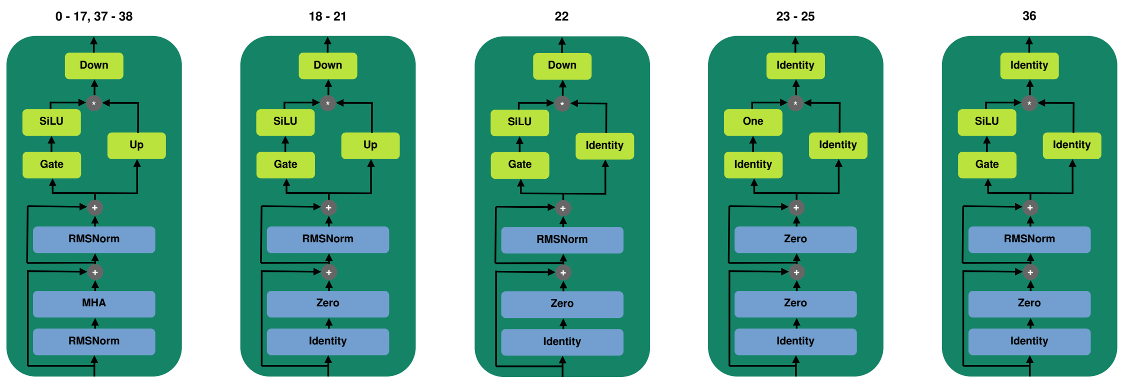

Figure 4.3: The state of Llama2 13B after being pruned to a 41% parameter reduction using fine-grained depth pruning. The layers are 0-indexed.

4.1.3. Additional Benchmarks

To further evaluate our method, we additionally present results on the BoolQ [7] and PIQA [4] benchmarks. We relegate this to a separate Section because the baseline does not provide results beyond MMLU on the Llama2 7B and Llama2 13B models. In contrast to the MMLU results, we do not have external validation data for the benchmarks in this Section.

Figure 4.4 shows the results on the Llama2 7B model. For PIQA we see the two methods behaving quite similarly, with our fine-grained method slightly underperforming up to a 30% parameter reduction, and slightly outperforming beyond that point. For BoolQ, we see a long plateau at an almost random 0.62 BoolQ score. With the coarse-grained baseline method, we reach this plateau at a 18% parameter reduction. With our fine-grained method, we achieve an additional operator point at 18%, reaching the plateau at 24%.

Figure 4.5 shows the results on the Llama2 13B model. For PIQA, we again see the methods performing similarly, with coarse-grained method having a slight advantage on most points in the parameter reduction domain. For BoolQ, we again observe the plateau behavior for the coarse-grained method, while our fine-grained method collapses to random instead of plateauing. Our method seems to provide improved robustness before the plateau range, preserving a nearly full BoolQ score up to a 12% parameter reduction.

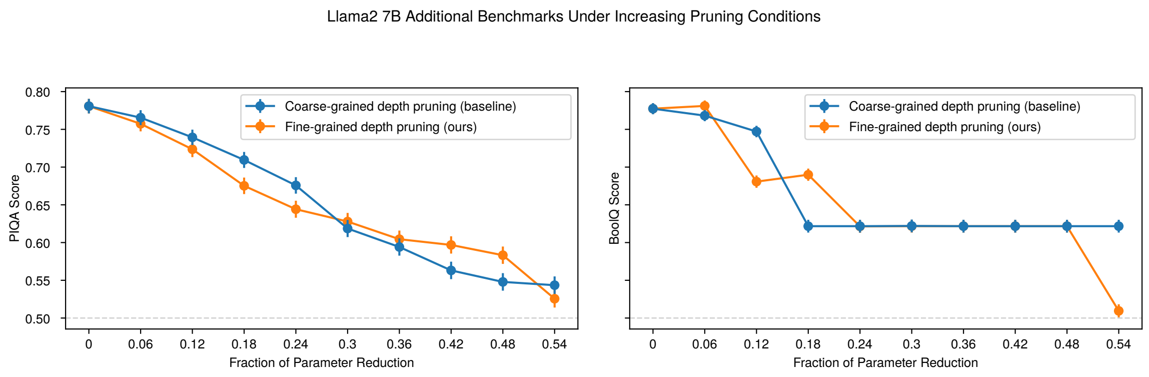

Figure 4.4: A comparison of PIQA and BoolQ benchmark scores on the Llama2 7B model. The gray dashed line represents the random guessing score on the given benchmark.

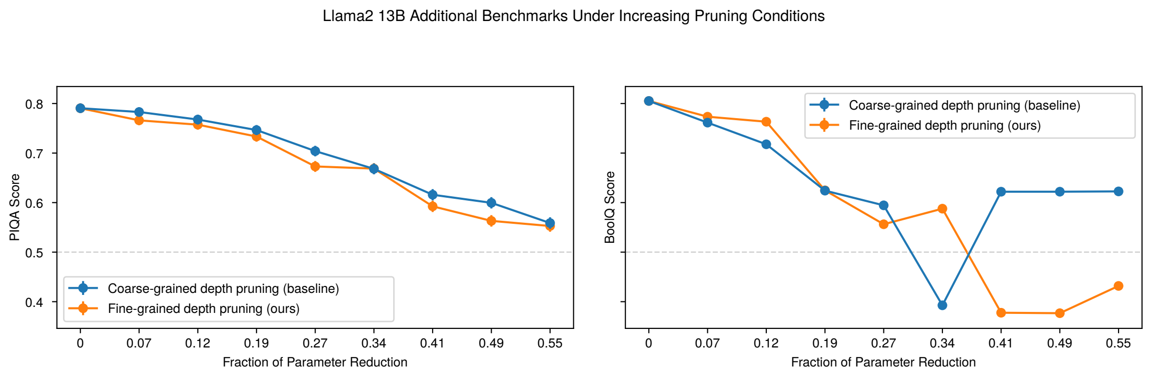

Figure 4.5: A comparison of PIQA and BoolQ benchmark scores on the Llama2 13B model. The gray dashed line represents the random guessing score on the given benchmark.

For both models, the PIQA curves decrease in a more linear fashion compared to the sharp drops in MMLU and BoolQ.

4.2. YOLOv5 Results

In this Section, we explore the performance of YOLOv5 CNN models using our depth pruning method. As noted in Section 2, existing methods select layers to prune from a relatively restricted search space since most exclude layers that change the channel dimension. Additionally, CNN depth pruning is far less studied than width pruning, and authors tend to only report results on pruned models after PPFT was applied. To address this gap, we take a similar approach to the one in Section 4.1: We study "parameter reduction vs. performance" curves to demonstrate the applicability of our method to CNNs even in the absence of PPFT, highlight the extended search space, and demonstrate effects on latency.

4.2.1. COCO Dataset Performance

Figure 4.6 shows two types of Mean Average Precision (mAP) scores on the COCO validation dataset under increasing parameter reductions on pre-trained YOLOv5X, YOLOvL and YOLOv5M models. In all 3 cases, we observe a slight drop in mAP as soon as any pruning is applied, followed by a plateau that remains robust until a transition point where mAP drops to 0. The transition point appears to shift to a lower parameter reduction threshold as model size decreases. In the case of the largest model YOLOv5X, this allows us to prune 46% of total parameters while preserving approximately 80% of mAP.

We also demonstrate the impact of our depth pruning method on latency in Table 4.3. The table compares the 46%, 28% and 26% points (labeled "YOLOv5X 0.46", "YOLOv5L 0.28" and "YOLOv5M 0.26" respectively), as these are the best "parameter reduction vs. performance" cases, occurring right

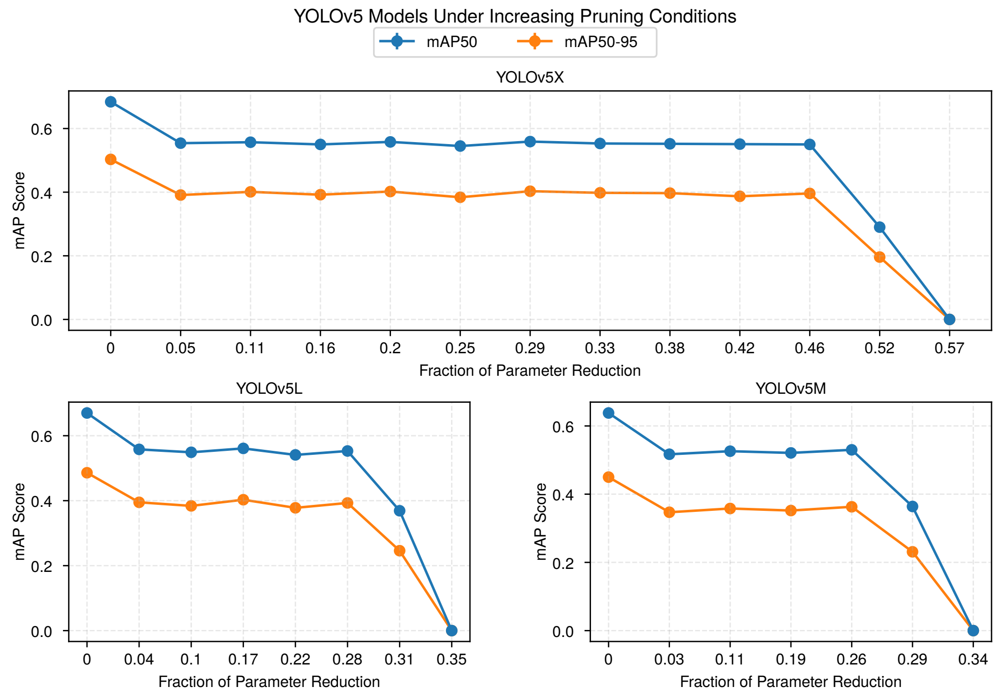

Figure 4.6: A comparison of mAP scores under various amounts of parameter reduction on YOLOv5 models.

before mAP begins to drop to 0. The table shows total validation latency, which is the time taken to perform all inference passes on the COCO validation dataset, along with per-batch latency, which is the mean time to perform an inference pass on one batch. We see latency reductions between 4% and 15%, increasing with model size.

Model	Parameter Count (M)	Total Val Latency (s)	Per-batch Latency (ms)
YOLOv5X	8.671	169	27.8
YOLOv5X 0.46	4.675	148	23.6
YOLOv5L	4.653	110	16.6
YOLOv5L 0.28	3.384	107	15.4
YOLOv5M	2.117	76	9.9
YOLOv5M 0.26	1.563	77	9.5

Table 4.3: A comparison of the final plateau points in Figure 4.6, before mAP begins to drop to 0.

4.2.2. Search Space Comparison

Table 4.4 compares the quantity of convolutional layers treated as prunable by our method to that of the existing methods. The row labeled "Spatial dim. exclusion" quantifies the number of layers that are excluded because they modify the feature map spatial dimensions. This happens in our method because we were not able to apply width pruning on this axis, and also in other methods in the literature. The row labeled "Channel dim. exclusion" quantifies the number of layers excluded because they modify the channel dimension. This happens in the works mentioned in Chapter 2, while our method solves this problem by combining width pruning with depth pruning.

4.3. Ablation Studies

As mentioned in Section 3.3, we used different importance metrics for pruning units depending on the type of model. Starting with the YOLOv5 models, we found the Taylor importance score ranked in

	YOLOv5		YOLOv5L		YOLOv5M	
	Ours	**Reduced**	**Ours**	**Reduced**	**Ours**	**Reduced**
Candidate layers	123	123	101	101	79	79
Spatial dim. exclusion	7	7	7	7	7	7
Channel dim. exclusion	0	18	0	20	0	20
Remaining prunable	116	98	94	74	72	52
% extension	**18%**		**27%**		**38%**	

Table 4.4: A comparison of the compositions of prunable layers when using our method compared to more restrictive methods in the literature. The percentage by which our method extends the reduced search spaces is highlighted in bold.

ascending order to be sufficient. We validate that this strategy indeed generates a meaningful ranking by comparing it to a descending ordering (highest importance scores first) and a randomized ordering. This can be thought of as modifying line 7 in Algorithm 9 to select the highest importance layer or a random layer, respectively. As shown in Table 4.5, in ascending order, we are able to reach a parameter reduction of 11% while preserving most of the unpruned mAP score. Note that this corresponds to the 11% point in Figure 4.6. In descending and random orders, the mAP scores collapse to near-zero at the same level of compression. This serves as evidence that the Taylor importance score is meaningful, since a reasonable amount of performance is only preserved when pruning the lowest-scoring units.

	Importance Score Ordering - YOLOv5X		
	Ascending	Descending	Random
mAP50 @ 11%	0.557	0.000256	0.00879
mAP50-90 @ 11%	0.401	0.00113	0.00245

Table 4.5: A comparison of YOLOv5X mAP scores at an 11% parameter reduction achieved using different orderings of Taylor importance scores.

To validate the choice of angular distance as an importance score for the Llama2 models, we compare it against the Taylor importance score and against importance measured using cross-entropy. In the field of machine learning, cross-entropy is a commonly-used measure of divergence between probability distributions [45], and so we use it to quantify the impact that pruning a given unit has on the next-token distribution.

Formally, let p^G be a next-token distribution for some input on the unpruned model and let $p^G[\text{Prune}(f)]$ be the same with layer f pruned. Then, $\mathcal{I}_f^* = E[CE(p^G, p^G[\text{Prune}(f)])]$ where CE is the cross-entropy function, with the expectation computed over a calibration dataset (see Table 3.3).

Table 4.6 shows this comparison. When using angular distance to achieve a reduction of 12%, performance is retained. Using the cross-entropy and Taylor importance scores, performance collapses to random at the same compression level.

	Importance Score Type - Llama2 7B		
	Angular	Cross-Entropy	Taylor
MMLU @ 12%	0.353	0.25	0.239

Table 4.6: A comparison of Llama2 7B MMLU scores at a 12% parameter reduction using different importance scores.

4.4. Explanation Studies

In Section 4.1, we observe that the MMLU and BoolQ scores do not decrease monotonically with increasing compression levels. Intuition would suggest that the benchmark scores would not increase as more parameters are removed. Instead, it appears that pruning certain units creates "damage" that worsens benchmark scores, and pruning some additional units corrects the damage.

To start, we look at how individually pruning each Transformer layer on the Llama2 7B model impacts next-token perplexity on a dataset independent from the benchmarks. In Figure 4.7, we observe a counter-intuitive effect: There are a number of layers where pruning them in isolation decreases perplexity, with the most pronounced decrease effect in layer 18. Note that the first two layers (indices 0 and 1) and the last layer (index 31) are excluded because they cause a very large perplexity increase that makes the other points unreadable.

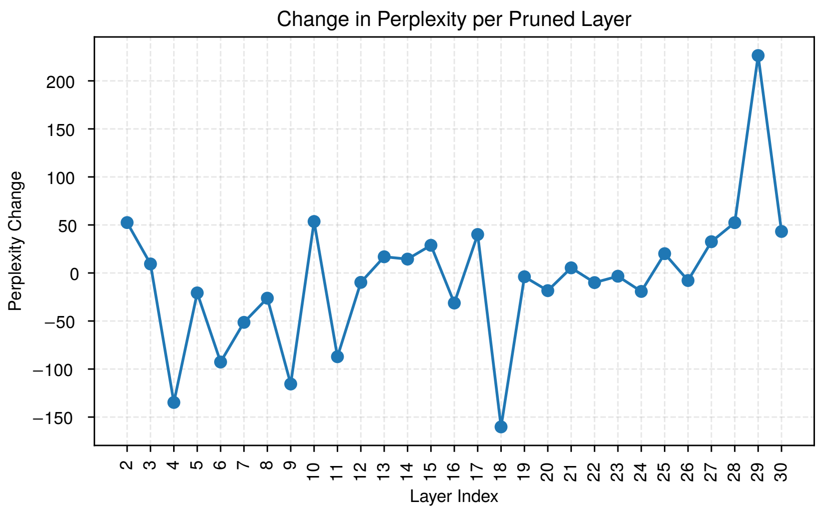

Figure 4.7: Changes in perplexity induced by pruning Llama2 7B Transformer layers.

To understand how the damage correction effect manifests itself, we zoom in on the 12%, 18%, 24% and 30% points in Figure 4.2. For the coarse grained case, this is shown in the left subplot of Figure 4.8. In the leftmost column, the unpruned score is shown. The second column shows the 12% point, where Transformer layers 25 through 28 are pruned. By additionally pruning layer 29 (third column) performance collapses to random, resembling the 24% point. From there, if we additionally prune layer 21 (fourth column), performance is restored back to the level of the 12% point.

Moreover, pruning only layer 21 (fifth column) does not increase MMLU score above the unpruned level, invalidating the possibility that pruning layer 21 in isolation improves MMLU performance. Pruning layer 29 alone (sixth column) causes only a modest decrease in MMLU score, invalidating the possibility that pruning layer 29 in isolation causes MMLU score to collapse to random. This indicates that layer 21 needs to pruned together with layer 29 to preserve MMLU performance. Also, contrary to Figure 4.7, pruning only layer 18 (seventh column) causes no significant change to MMLU performance, indicating some kind of dataset dependence.

Next, we zoom in on the 12%, 18% and 24% points in the fine-grained case in Figure 4.2. To describe these points, we write `i.up` to denote MLP up projection sublayer in Transformer layer `i`. The right subplot in Figure 4.8 shows the MMLU Score at the 12% point (second column). Pruning `22.up` on top of this decreases performance to the level of the 18% point (third column), and pruning `21.up` in addition to that restores performance to the level of the 24% point (fourth column). This suggest that `21.up` and `22.up` need to be pruned together to preserve MMLU performance.

We posit that a similar effect occurs in the Llama2 13B model in Figure 4.1, causing the small dip at 27%.

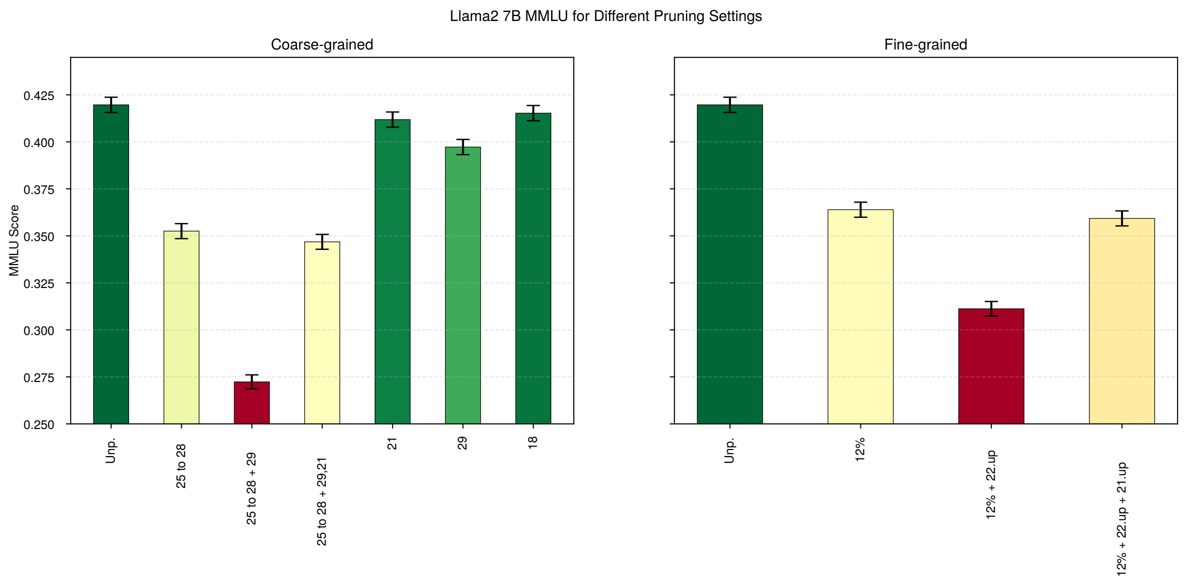

Figure 4.8: An explanation of which Transformer layers and linear layers cause the damage correction effect.

5 Discussion and Conclusion

In this concluding Chapter, Section 5.1 discusses our findings. Then, Section 5.2 concludes this report, makes recommendations for further work, and acknowledges its limitations.

5.1. Discussion

In this Section, we discuss the results of our Llama2 and YOLOv5 experiments.

5.1.1. Discussion of Llama2 Results

We obtained several findings in which fine-grained depth pruning outperformed coarse-grained depth pruning approaches. For Llama2 13B, our fine-grained method is able to create a 41% parameter reduction while preserving MMLU performance almost entirely, whereas the coarse-grained method could only reach 34% (Figure 4.1). On BoolQ, we get a slower rate of deterioration in the 0 - 12% parameter reduction range (Figure 4.5). For Llama2 7B, our method keeps the BoolQ score above the low-performance plateau up to an 18% parameter reduction, while the coarse-grained method does so only up to 12%. On PIQA, our method slightly outperforms the coarse-grained method at 42% and 48% (Figure 4.4).

Other findings, however, paint a different picture. We have seen cases where our method caused a collapse to random at a compression level where the coarse-grained method retrained a non-trivial level of performance. In other results, the coarse-grained method overtook later after dropping to random earlier. There were also situations where the winner varies based on compression level, and one case where the fine-grained method consistently lags behind.

These observations bring into question the assumption stated in Section 2.3: if prunable redundancies exist on the Transformer layer level, then prunable redundancies might exist on the fine-grained level. In Section 4.4 we have seen that to preserve MMLU performance under pruning, certain units might need to be pruned simultaneously, at least in a context where some pruning was already applied at an earlier step (Figure 4.8). Additionally, this effect seems to occur in a task-dependent manner, given that perplexity on a general, non-benchmark dataset seems to benefit from pruning layers that have no significant impact on MMLU score (Figure 4.7). Based on this evidence, we posit that there exist *semantic dependencies* between pruning units both on the coarse-grained and fine-grained scale.

If this is indeed the case, it challenges the implicit assumptions made in many pruning works: the treatment of parameters, layers and other structures as independent units, for which importance can be estimated in a context-independent manner. For any given layer of granularity, one might be able to make better pruning decisions by considering semantic dependencies between the pruning units. This could be addressed by iterative pruning where importance is reevaluated at each step, as this was computationally prohibitive for us on the Llama2 models. The strength of our method is in the granularity of its search space, and this idea could be further extended to identify semantic dependencies with greater resolution.

Regarding latency, the results in Table 4.1 on the Llama2 models are strongly consistent with expectations set by related work. The latency gains obtained by our method resemble that of the coarse-grained method very closely. Both techniques are shown to reduce latency by approximately the same ratio as the parameter reduction, while our method subsumes and generalizes the coarse-grained method (Table 4.2). These findings are consistent with arguments made by [44] and [23] - pruning layers improves latency more than pruning other types of structures because memory access and computation associated with the layer is avoided entirely.

5.1.2. Discussion of YOLOv5 Results

Contrary to the findings of [42], we observed remarkable robustness on the YOLOv5 models when pruning with our method. For all tested YOLOv5 models, we observed a modest dip in mAP as soon as pruning began, but then the score remained approximately constant as more units were pruned, until it eventually rapidly fell to 0. This allowed us to obtain favorable tradeoffs between parameter reduction and performance. To the best of our knowledge, no CNN pruning technique exhibits this level of robustness, and no CNN depth pruning work explores the parameter reduction - accuracy tradeoff curve. In our literature review, CNN width pruning papers significantly outnumbered depth pruning papers, but our findings point towards depth pruning being more feasible and effective than this imbalance would suggest.

The latency gains on the YOLOv5 models are less pronounced compared to Llama2 models, as shown in Table 4.3. Latency still improves measurably, but only by at most 15%, at the largest tested parameter reduction of 46% on the biggest, YOLOv5X model. We still consider this a remarkable result, because while many other CNN pruning works report theoretical changes in MACs or FLOPS, we report real wall-clock time.

A possible explanation for this latency scaling discrepancy between the Llama2 and YOLOv5 models is data locality. A convolutional layer in YOLOv5 contains far fewer parameters than even a layer on the fine-grained scale in the smallest Llama2 model. If memory transfer overhead dominates the time complexity, then eliminating more, smaller memory transfers saves less time than eliminating fewer, larger memory transfers.

Additionally, Table 4.4 demonstrates the flexibility of our method, expanding the search space of prunable layers by 18% - 38%, compared to architecture-dependent methods that must exclude channel-dimension-transforming layers from the search space. Moreover, on YOLOv5, layers that change the channel dimension remain roughly constant in models of different sizes, and thus become a greater portion of the total layers in smaller models. Our method still requires the exclusion of layers that modify the feature map spatial dimensions, and fortunately, these make up less than 10% of all layers in the models we tested. This advantage afforded by our method scales with the proportion of layers that modify the channel dimension.

Another remarkable advantage of our method is the ability to address these dimension conflicts efficiently. We have seen work that does not exclude channel-dimension-transforming layers from the search space, instead performing some form of layer replacement and using PPFT to optimize the new weights ([12], [48]). Our method fixes dimension conflicts by width pruning layers that have a dependency on the layer being pruned, which only requires making a good selection of tensor sections to prune. When using the Taylor importance score (3.10), we are required to approximate the gradients of the parameters involved, but we accomplished this with a calibration dataset (see Table 3.3) that is much smaller than what is likely to be required for PPFT. One constraint of note is that more memory is required for storing gradients than for storing only weights for inference.

5.2. Conclusion

In this work, we developed a methodology for depth pruning of deep learning models in a way that aims to be architecture-agnostic. We did this by identifying the lowest level of abstraction on which layer pruning can occur, recognizing its applicability to a wide range of architectures, and designing it to subsume pruning techniques developed for higher levels of abstraction. Our goal was to maximize the search space of prunable units to enhance granularity of selection, while leveraging the latency improvements afforded by pruning full layers.

We posited that the finer-grained nature of our method would enable better selection of layers to prune on Llama2 models. The results showed that our method can indeed achieve finer-grained layer selection, competing with state-of-the-art coarse-grained depth pruning. Our method does lead to improvements in tradeoffs between parameter reduction and performance, but the improvements are task-specific. We also found that latency reduction scales well with parameter reduction with both fine- and coarse-grained depth pruning. Importantly, we found evidence suggesting that treating layers as independent pruning units might be counterproductive, questioning this approach which is ubiquitous in the literature. We used this to describe the notion of semantic dependency between pruning units, suggesting that pruning as a whole could be done better by considering task-specific dependencies between units.

We also found that YOLOv5 can be depth pruned using our technique, and performance remains remarkably robust up to significant compression levels. We identified operator points where the relative percent performance penalty is less than the relative percent parameter reduction. We achieved this without PPFT, which to the best of our knowledge is a novel result, since other pruning techniques require PPFT for performance retention. At the operator points, we also obtained modest but non-negligible latency reductions, measured in real wall-clock time in contrast to theoretical MACs and FLOPS often found in the literature. The latency reductions do not scale as aggressively as they do in LLMs, and we attribute this to the relatively lower per-layer parameter count and its impact on data locality. We successfully expanded the search space of prunable layers compared to methods that exclude channel-dimension-transforming layers, and resolved dimension conflicts in a manner that is significantly less computationally intensive than PPFT.

5.2.1. Future Work

Having seen evidence of the semantic dependency effect, we recommend that future pruning work considers this perspective. In our case, the semantic dependency seemed to surface only once some parts of the model have already been pruned, so semantic dependency might need to be measured iteratively, and we advise against one-shot measurement. We acknowledge the challenging nature of this, since a naive implementation would involve an exponential-time subset selection problem, but we believe it could be a path towards improved pruning strategies.

The semantic dependency effect with its data-dependent nature also suggests that pruning could be an effective tool for creating small, task specific language models. While this is also achievable with knowledge distillation, our pruning method might achieve it without training.

5.2.2. Limitations

We acknowledge the following limitations of our work:

1. Since the Llama2 models required a computationally expensive importance estimation method, the importance of their pruning units was measured in one shot. This might have led to suboptimal choices of which units to prune, compromising the quality of our method.
2. Algorithm 8 uses a heuristic selection strategy - pruning two MLP sublayers for each MHA layer. As explained, this was done because we were unable to come up with an importance score where MLP sublayers and MHA layers could share the same ranking. This impacts the ultimate order in which the units are pruned, which might also compromise the quality of our method.
3. Due to time constraints, we opted to only study YOLOv5 out of all available CNN models. We chose it due to its widespread popularity and commercial use. This means that our findings might not necessarily generalize to other types of CNNs.

References

[1] Delft High Performance Computing Centre (DHPC). *DelftBlue Supercomputer (Phase 2)*. https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2. 2024.

[2] Saleh Ashkboos et al. *SliceGPT: Compress Large Language Models by Deleting Rows and Columns*. Feb. 9, 2024. DOI: 10.48550/arXiv.2401.15024. arXiv: 2401.15024[cs]. URL: http://arxiv.org/abs/2401.15024 (visited on 07/23/2025).

[3] Akhiad Bercovich et al. *Puzzle: Distillation-Based NAS for Inference-Optimized LLMs*. June 3, 2025. DOI: 10.48550/arXiv.2411.19146. arXiv: 2411.19146[cs]. URL: http://arxiv.org/abs/2411.19146 (visited on 07/23/2025).

[4] Yonatan Bisk et al. *PIQA: Reasoning about Physical Commonsense in Natural Language*. Nov. 26, 2019. DOI: 10.48550/arXiv.1911.11641. arXiv: 1911.11641[cs]. URL: http://arxiv.org/abs/1911.11641 (visited on 11/06/2025).

[5] Shi Chen and Qi Zhao. "Shallowing Deep Networks: Layer-Wise Pruning Based on Feature Representations". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41.12 (Dec. 2019), pp. 3048–3056. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2018.2874634. URL: https://ieeexplore.ieee.org/document/8485719 (visited on 07/03/2025).

[6] Minsik Cho, Saurabh Adya, and Devang Naik. *PDP: Parameter-free Differentiable Pruning is All You Need*. Nov. 17, 2023. DOI: 10.48550/arXiv.2305.11203. arXiv: 2305.11203[cs]. URL: http://arxiv.org/abs/2305.11203 (visited on 10/29/2025).

[7] Christopher Clark et al. *BoolQ: Exploring the Surprising Difficulty of Natural Yes/No Questions*. May 24, 2019. DOI: 10.48550/arXiv.1905.10044. arXiv: 1905.10044[cs]. URL: http://arxiv.org/abs/1905.10044 (visited on 11/06/2025).

[8] Pierre Vilar Dantas et al. "A comprehensive review of model compression techniques in machine learning". In: *Applied Intelligence* 54.22 (Nov. 1, 2024), pp. 11804–11844. ISSN: 1573-7497. DOI: 10.1007/s10489-024-05747-w. URL: https://doi.org/10.1007/s10489-024-05747-w (visited on 10/29/2025).

[9] Xuan Ding et al. *A Sliding Layer Merging Method for Efficient Depth-Wise Pruning in LLMs*. May 15, 2025. DOI: 10.48550/arXiv.2502.19159. arXiv: 2502.19159[cs]. URL: http://arxiv.org/abs/2502.19159 (visited on 07/03/2025).

[10] Peijie Dong, Lujun Li, and Zimian Wei. *DisWOT: Student Architecture Search for Distillation WithOut Training*. Mar. 28, 2023. DOI: 10.48550/arXiv.2303.15678. arXiv: 2303.15678[cs]. URL: http://arxiv.org/abs/2303.15678 (visited on 10/29/2025).

[11] Mostafa Elhoushi et al. "LayerSkip: Enabling Early Exit Inference and Self-Speculative Decoding". In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2024, pp. 12622–12642. DOI: 10.18653/v1/2024.acl-long.681. arXiv: 2404.16710[cs]. URL: http://arxiv.org/abs/2404.16710 (visited on 07/03/2025).

[12] Sara Elkerdawy et al. *To Filter Prune, or to Layer Prune, That Is The Question*. Nov. 8, 2020. DOI: 10.48550/arXiv.2007.05667. arXiv: 2007.05667[cs]. URL: http://arxiv.org/abs/2007.05667 (visited on 07/03/2025).

[13] Gongfan Fang et al. *DepGraph: Towards Any Structural Pruning*. Mar. 23, 2023. DOI: 10.48550/arXiv.2301.12900. arXiv: 2301.12900[cs]. URL: http://arxiv.org/abs/2301.12900 (visited on 09/24/2025).

[14] Elias Frantar and Dan Alistarh. *SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot*. Mar. 22, 2023. DOI: 10.48550/arXiv.2301.00774. arXiv: 2301.00774[cs]. URL: http://arxiv.org/abs/2301.00774 (visited on 09/24/2025).

[15] Shangqian Gao et al. *DISP-LLM: Dimension-Independent Structural Pruning for Large Language Models*. Nov. 4, 2024. DOI: 10.48550/arXiv.2410.11988. arXiv: 2410.11988[cs]. URL: http://arxiv.org/abs/2410.11988 (visited on 07/03/2025).

[16] Jianping Gou et al. "Knowledge Distillation: A Survey". In: *International Journal of Computer Vision* 129.6 (June 1, 2021), pp. 1789–1819. ISSN: 1573-1405. DOI: 10.1007/s11263-021-01453-z. URL: https://doi.org/10.1007/s11263-021-01453-z (visited on 11/13/2025).

[17] Andrey Gromov et al. *The Unreasonable Ineffectiveness of the Deeper Layers*. Mar. 3, 2025. DOI: 10.48550/arXiv.2403.17887. arXiv: 2403.17887[cs]. URL: http://arxiv.org/abs/2403.17887 (visited on 07/03/2025).

[18] Dan Hendrycks et al. *Measuring Massive Multitask Language Understanding*. Version Number: 3. 2020. DOI: 10.48550/ARXIV.2009.03300. URL: https://arxiv.org/abs/2009.03300 (visited on 11/13/2025).

[19] Mohammad Jani et al. *Model Compression Methods for YOLOv5: A Review*. July 21, 2023. DOI: 10.48550/arXiv.2307.11904. arXiv: 2307.11904[cs]. URL: http://arxiv.org/abs/2307.11904 (visited on 10/29/2025).

[20] Nidhal Jegham et al. *How Hungry is AI? Benchmarking Energy, Water, and Carbon Footprint of LLM Inference*. Nov. 12, 2025. DOI: 10.48550/arXiv.2505.09598. arXiv: 2505.09598[cs]. URL: http://arxiv.org/abs/2505.09598 (visited on 11/12/2025).

[21] Rahima Khanam and Muhammad Hussain. *What is YOLOv5: A deep look into the internal features of the popular object detector*. version: 1. July 31, 2024. DOI: 10.48550/arXiv.2407.20892. arXiv: 2407.20892[cs]. URL: http://arxiv.org/abs/2407.20892 (visited on 11/13/2025).

[22] Jinuk Kim et al. *LayerMerge: Neural Network Depth Compression through Layer Pruning and Merging*. July 8, 2024. DOI: 10.48550/arXiv.2406.12837. arXiv: 2406.12837[cs]. URL: http://arxiv.org/abs/2406.12837 (visited on 07/03/2025).

[23] Bo-Kyeong Kim et al. *Shortened LLaMA: Depth Pruning for Large Language Models with Comparison of Retraining Methods*. June 23, 2024. DOI: 10.48550/arXiv.2402.02834. arXiv: 2402.02834[cs]. URL: http://arxiv.org/abs/2402.02834 (visited on 07/06/2025).

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Commun. ACM* 60.6 (May 24, 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://dl.acm.org/doi/10.1145/3065386 (visited on 11/12/2025).

[25] Yann LeCun, John Denker, and Sara Solla. "Optimal Brain Damage". In: *Advances in Neural Information Processing Systems*. Vol. 2. Morgan-Kaufmann, 1989. URL: https://proceedings.neurips.cc/paper/1989/hash/6c9882bbac1c7093bd25041881277658-Abstract.html (visited on 07/23/2025).

[26] Hanxiao Liu et al. *Pay Attention to MLPs*. June 1, 2021. DOI: 10.48550/arXiv.2105.08050. arXiv: 2105.08050[cs]. URL: http://arxiv.org/abs/2105.08050 (visited on 09/27/2025).

[27] Shiwei Liu and Zhangyang Wang. *Ten Lessons We Have Learned in the New "Sparseland": A Short Handbook for Sparse Neural Network Researchers*. June 24, 2023. DOI: 10.48550/arXiv.2302.02596. arXiv: 2302.02596[cs]. URL: http://arxiv.org/abs/2302.02596 (visited on 07/23/2025).

[28] Xinyin Ma, Gongfan Fang, and Xinchao Wang. *LLM-Pruner: On the Structural Pruning of Large Language Models*. Version Number: 3. 2023. DOI: 10.48550/ARXIV.2305.11627. URL: https://arxiv.org/abs/2305.11627 (visited on 07/03/2025).

[29] Huizi Mao et al. "Exploring the Regularity of Sparse Structure in Convolutional Neural Networks". In: *CoRR* abs/1705.08922 (2017). arXiv: 1705.08922. URL: http://arxiv.org/abs/1705.08922.

[30] Xin Men et al. *ShortGPT: Layers in Large Language Models are More Redundant Than You Expect*. Oct. 11, 2024. DOI: 10.48550/arXiv.2403.03853. arXiv: 2403.03853[cs]. URL: http://arxiv.org/abs/2403.03853 (visited on 07/03/2025).

[31] Pavlo Molchanov et al. *Importance Estimation for Neural Network Pruning*. Version Number: 1. 2019. DOI: 10.48550/ARXIV.1906.10771. URL: https://arxiv.org/abs/1906.10771 (visited on 10/31/2025).

[32] Saurav Muralidharan et al. *Compact Language Models via Pruning and Knowledge Distillation*. version: 1. July 19, 2024. DOI: 10.48550/arXiv.2407.14679. arXiv: 2407.14679[cs]. URL: http://arxiv.org/abs/2407.14679 (visited on 09/23/2025).

[33] Nvidia et al. *Nemotron-4 340B Technical Report*. Aug. 6, 2024. DOI: 10.48550/arXiv.2406.11704. arXiv: 2406.11704[cs]. URL: http://arxiv.org/abs/2406.11704 (visited on 09/23/2025).

[34] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Dec. 3, 2019. DOI: 10.48550/arXiv.1912.01703. arXiv: 1912.01703[cs]. URL: http://arxiv.org/abs/1912.01703 (visited on 09/29/2025).

[35] Jeff Pool, Abhishek Sawarkar, and Jay Rodge. *Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT*. NVIDIA Technical Blog. July 20, 2021. URL: https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/ (visited on 09/24/2025).

[36] Alec Radford et al. *Language Models are Unsupervised Multitask Learners*. Tech. rep. OpenAI, 2019. URL: https://api.semanticscholar.org/CorpusID:160025533.

[37] Tianyao Shi and Yi Ding. *Systematic Characterization of LLM Quantization: A Performance, Energy, and Quality Perspective*. Aug. 26, 2025. DOI: 10.48550/arXiv.2508.16712. arXiv: 2508.16712[cs]. URL: http://arxiv.org/abs/2508.16712 (visited on 11/13/2025).

[38] Shoaib Ahmed Siddiqui et al. *A deeper look at depth pruning of LLMs*. July 23, 2024. DOI: 10.48550/arXiv.2407.16286. arXiv: 2407.16286[cs]. URL: http://arxiv.org/abs/2407.16286 (visited on 07/03/2025).

[39] Mingjie Sun et al. *A Simple and Effective Pruning Approach for Large Language Models*. May 7, 2024. DOI: 10.48550/arXiv.2306.11695. arXiv: 2306.11695[cs]. URL: http://arxiv.org/abs/2306.11695 (visited on 11/13/2025).

[40] Kimi Team et al. *Kimi K2: Open Agentic Intelligence*. July 29, 2025. DOI: 10.48550/arXiv.2507.20534. arXiv: 2507.20534[cs]. URL: http://arxiv.org/abs/2507.20534 (visited on 11/12/2025).

[41] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. Feb. 27, 2023. DOI: 10.48550/arXiv.2302.13971. arXiv: 2302.13971[cs]. URL: http://arxiv.org/abs/2302.13971 (visited on 07/29/2025).

[42] Alvin Wan et al. *UPSCALE: Unconstrained Channel Pruning*. July 17, 2023. DOI: 10.48550/arXiv.2307.08771. arXiv: 2307.08771[cs]. URL: http://arxiv.org/abs/2307.08771 (visited on 10/31/2025).

[43] Zhongwei Wan et al. *Efficient Large Language Models: A Survey*. May 23, 2024. DOI: 10.48550/arXiv.2312.03863. arXiv: 2312.03863[cs]. URL: http://arxiv.org/abs/2312.03863 (visited on 07/03/2025).

[44] Hanrui Wang, Zhekai Zhang, and Song Han. "SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). ISSN: 2378-203X. Feb. 2021, pp. 97–110. DOI: 10.1109/HPCA51647.2021.00018. URL: https://ieeexplore.ieee.org/abstract/document/9407232 (visited on 07/03/2025).

[45] Qi Wang et al. "A Comprehensive Survey of Loss Functions in Machine Learning". In: *Annals of Data Science* 9.2 (Apr. 2022), pp. 187–212. ISSN: 2198-5804, 2198-5812. DOI: 10.1007/s40745-020-00253-5. URL: https://link.springer.com/10.1007/s40745-020-00253-5 (visited on 11/14/2025).

[46] Zhipeng Wang et al. "Apple stem/calyx real-time recognition using YOLO-v5 algorithm for fruit automatic loading system". In: *Postharvest Biology and Technology* 185 (Mar. 1, 2022), p. 111808. ISSN: 0925-5214. DOI: 10.1016/j.postharvbio.2021.111808. URL: https://www.sciencedirect.com/science/article/pii/S0925521421003471 (visited on 10/29/2025).

[47] Lu Wei et al. "Advances in the Neural Network Quantization: A Comprehensive Review". In: *Applied Sciences* 14.17 (Jan. 2024). Number: 17 Publisher: Multidisciplinary Digital Publishing Institute, p. 7445. ISSN: 2076-3417. DOI: 10.3390/app14177445. URL: https://www.mdpi.com/2076-3417/14/17/7445 (visited on 07/23/2025).

[48] Wei Wen et al. *Learning Structured Sparsity in Deep Neural Networks*. Oct. 18, 2016. DOI: 10.48550/arXiv.1608.03665. arXiv: 1608.03665[cs]. URL: http://arxiv.org/abs/1608.03665 (visited on 07/03/2025).

[49] Xiaoxia Wu et al. "Understanding Int4 Quantization for Language Models: Latency Speedup, Composability, and Failure Cases". In: *Proceedings of the 40th International Conference on Machine Learning*. Ed. by Andreas Krause et al. Vol. 202. Proceedings of Machine Learning Research. PMLR, 23–29 Jul 2023, pp. 37524–37539. URL: https://proceedings.mlr.press/v202/wu23k.html.

[50] Xiaohan Xu et al. *A Survey on Knowledge Distillation of Large Language Models*. Oct. 21, 2024. DOI: 10.48550/arXiv.2402.13116. arXiv: 2402.13116[cs]. URL: http://arxiv.org/abs/2402.13116 (visited on 07/23/2025).

[51] Zhengqing Yuan et al. *EfficientLLM: Efficiency in Large Language Models*. version: 1. May 20, 2025. DOI: 10.48550/arXiv.2505.13840. arXiv: 2505.13840[cs]. URL: http://arxiv.org/abs/2505.13840 (visited on 07/03/2025).

[52] Nan Zhang et al. *Pruning as a Domain-specific LLM Extractor*. May 10, 2024. DOI: 10.48550/arXiv.2405.06275. arXiv: 2405.06275[cs]. URL: http://arxiv.org/abs/2405.06275 (visited on 07/03/2025).

[53] Yuxin Zhang et al. *Dynamic Sparse No Training: Training-Free Fine-tuning for Sparse LLMs*. Version Number: 3. 2023. DOI: 10.48550/ARXIV.2310.08915. URL: https://arxiv.org/abs/2310.08915 (visited on 11/13/2025).

[54] Xunyu Zhu et al. *A Survey on Model Compression for Large Language Models*. July 30, 2024. DOI: 10.48550/arXiv.2308.07633. arXiv: 2308.07633[cs]. URL: http://arxiv.org/abs/2308.07633 (visited on 07/23/2025).