Latency-aware structured pruning of pretrained transformer-based models

Alexander Hoffman, Department of Electrical and Computer Engineering
McGill University, Montreal
February, 2022

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Science Electrical Engineering

Abstract

The Transformer neural network architecture has grown popular in the field of natural language processing in recent years. However, these large networks remain difficult to deploy on edge devices where memory and compute resources are limited. Standard neural network compression techniques such as pruning and quantization can improve latency, but they are typically guided by hardware-independent metrics such as model size and number of floating point operations. These general statistics leave room for further latency optimization.

We propose Latency-Aware Pruned Neural Architecture Search (LAP-NAS) to enable latency-aware pruning for Transformer-based models on a target device. LAP-NAS prunes the model, then performs an efficient architecture search using pruning metrics and layer-wise latency measurements to reduce latency while maintaining accuracy. This technique combines the simplicity and speed of iterative pruning with the design space exploration of multi-objective neural architecture search. Linking these two techniques enables latency-aware optimization while keeping training time orders of magnitude lower than comparable compression methods. Our results show that LAP-NAS improves the latency of a pruned DistilBERT-base neural network by 2.3% while improving accuracy by 0.4% on a subset of the General Language Understanding Evaluation (GLUE) tasks. Our qualitative results promise even better speedup given more accurate latency modeling techniques.

Résumé

L'architecture de réseau neuronal Transformer s'est développée populaire dans le domain de traitement du langage naturel ces dernières années. Cependant, ces grands réseaux demeurent difficiles à déployer sur des appareils périphériques où les ressources de mémoire et de calcul sont limitées. Les techniques standard de compression de réseaux neuronaux telles que l'élagage et la quantification peuvent améliorer la latence, mais elles sont généralement guidées par des métriques indépendantes du matériel numérique comme la taille du modèle et le nombre d'opérations en virgule flottante. Ces statistiques générales laissent une marge potentielle pour une optimisation supplémentaire de la latence.

Nous proposons *Latency-Aware Pruned Neural Architecture Search* (LAP-NAS) pour permettre un élagage sensible à la latence pour les modèles basés sur Transformer sur un appareil cible. Le LAP-NAS élague le modèle, puis effectue une recherche d'architecture efficace à l'aide de métriques d'élagage et de mesures de latence par couche pour réduire la latence tout en maintenant la précision. Cette technique combine la simplicité et la rapidité de l'élagage itératif avec l'exploration de l'espace de conception du recherche automatique d'architecture neuronale multi-objectifs. La liaison de ces deux techniques permet une optimisation tenant compte de la latence tout en maintenant des ordres de grandeur de temps d'entraînement inférieurs à ceux des méthodes de compression comparables. Nos résultats montrent que LAP-NAS améliore la latence d'un réseau de neurones à base de DistilBERT élagué de 2,3% tout en améliorant la précision de 0,4% sur un sous-ensemble des tâches *General Language Understanding Evaluation* (GLUE). Nos résultats

qualitatifs promettent une meilleure accélération grâce à des techniques de modélisation de latence plus précises.

Acknowledgements

I would like to thank my supervisors Prof. James Clark and Prof. Warren Gross for their support and guidance throughout my Master's degree. I never failed to discover a new point of view or potential solution to a problem after a conversation with them. In addition, I would like to emphasize my appreciation for Prof. Brett Meyer's help in concluding my research and translating my ideas into academic writing. Thank you as well to Maryam Ziaeefard for her ideas and encouragement in our weekly online meetings in my early days of research.

I am grateful for the socially-distanced support of my labmates in the CIM lab and the McGill Edge Intelligence Lab throughout the pandemic, online board games and all.

I also express my endless gratitude to my family and friends for their unwavering support throughout my studies.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Subject Overview

Deep neural networks (DNNs) have revolutionized machine learning tasks in domains such as computer vision and natural language processing (NLP). However, state of the art networks have scaled quickly with GPU performance and cloud computing, which has left low power devices at the edge too weak to process them with adequate latency. As advances in compute efficiency slow, researchers have pivoted focus from developing the highest accuracy models to the models which deliver the best tradeoff between accuracy and a cost metric such as model size, memory accesses, or FLOPs [1]. A parallel field of research is DNN compression, which aims to make edge AI applications a reality by reducing the computational complexity of neural networks. Rather than designing new efficient architectures and training from scratch, DNN compression methods reduce the size of a trained network with techniques such as pruning, quantization, knowledge distillation, or neural architecture search (NAS).

The field of DNN compression can be further divided into hardware-agnostic and hardware-aware methods. While the methods for modifying networks generally remain the same across the two fields, the search process for the optimal compressed network differs. The objective function for a hardware-agnostic compression method uses metrics

such as Floating point Operations (FLOPs) or model size to characterize the efficiency of the network. These metrics are defined by the network itself rather than any specific use case involving a device and operating system. When implementing neural networks in real applications, these hardware-agnostic metrics deliver mixed results in identifying models that perform best. For a device with extremely limited on-device storage, model size may be the best metric to optimize. If the model's weights did not fit into the target device's storage, then the model would be unusable. Even for a more complex objective like inference latency, the size of the model could serve as an effective optimization objective if device throughput is limited by memory access time. Yet there is clear evidence that in many cases these objectives fail to deliver optimal models for metrics which ML practitioners truly care about [2].

Hardware-aware optimization methods consider the full hardware and software stack to optimize a model for an observable metric such as latency, power, or energy. Several previous works have demonstrated that observed inference latency does not always correspond to the number of operations or the number of weights in a network [2] [3] [4]. Furthermore, seemingly inconsequential design choices such as the activation function can have significant impact on latency while having almost no effect on FLOPs or model size [5].

Hardware-aware methods have mainly been applied using NAS. Given a set of network building blocks and their associated HW-aware costs, NAS methods can search for a network with the best tradeoff between accuracy and cost[6][4]. While this method may result in the most optimal architecture for a given dataset and target device, it requires training several candidate networks or a single supernet in order to generate trained weights for the optimal architecture. This additional training time is problematic for problems with very large datasets and ML practitioners with small budgets. Transformer-based natural language processing (NLP) models such as BERT [7] exemplify this problem. These models are trained on massive datasets built by mining text from the internet, which results in significant environmental and financial costs [1].

Motivation

This work attempts to fill gaps in the HW-Aware compression literature. Specifically, there is a lack of work applying HW-Aware optimization to BERT-based models for NLP tasks, and a lack of efficient HW-Aware compression methods. Compressing BERT-based models is a valuable line of research because they offer state-of-the-art performance on NLP tasks but are not yet widely adopted on edge devices due to their cumbersome size and runtime. Tailoring these models to specific devices could further improve the latency and make them more useful outside of the datacenter.

Proposed Method

Our method makes use of structured pruning to compress a BERT-based deep neural network. Structured pruning [8] is one simple and effective way to reduce model size and FLOPs. Hardware-agnostic iterative pruning algorithms [9] assign a sparsity to each layer of the network and prune until that sparsity is met, which does not directly optimize for a platform-specific metric like latency.

We propose Latency Aware Pruned Neural Architecture Search (LAP-NAS), a novel technique that considers latency during pruning. First, we construct latency lookup tables for a single BERT layer by pruning the two building blocks of BERT: the feed forward network (FFN) and multi-head self-attention layer. Next, we apply iterative structured pruning and fine-tuning to a pretrained model until reaching a target sparsity for both the FFN and self-attention blocks. At this point, the layer dimensions (# FFN neurons, # heads) are unoptimized for the target device. LAP-NAS then performs an efficient architecture search using pruning metrics and layer-wise latency measurements to discover a more optimal model architecture. This step makes relatively small changes to each layer's dimensions, but can enable large changes in latency where fine-grained architecture optimizations are needed. We use a genetic algorithm to perform the architecture search with the objective of minimizing latency while maintaining an adequate number of im-

portant model parameters to preserve accuracy. As a result, we reduce the latency of a BERT-based model by up to 2.8% compared to iterative pruning alone.

Contribution

Our contributions are as follows:

- We design and demonstrate a latency-aware NAS method which optimizes a pruned BERT model's FFN and self-attention layer dimensions to minimize latency while maintaining accuracy
- With our LAP-NAS method we return up to 2.8% faster latency when compared to an iterative pruning baseline, without further accuracy degradation
- We present a latency-aware iterative pruning method to monotonically reduce model latency while reducing model size and demonstrate its utility by comparing to the iterative pruning baseline

We evaluate our method using two pretrained BERT-based models on five GLUE tasks and the Nvidia Jetson Xavier NX low power GPU.

Chapter 2

Literature Review of Neural Network Compression

Previous works in neural network compression generally fall under the categories of pruning, knowledge distillation, neural architecture search, and quantization. These methods are applied with either hardware-agnostic or hardware-aware optimization to guide the compression process. The final section of this chapter focuses on evaluation methods for determining neural network performance.

2.1 Hardware-Agnostic Methods

Hardware-agnostic compression methods use objectives such as FLOPs or model size to characterize the efficiency of the network. These methods function without consideration of the hardware platform where the model will perform its task.

2.1.1 Pruning

Pruning is a method that reduces the number of parameters in a neural network. There are two general forms of pruning, i.e. Elementwise and Structured, which will be discussed further in the following subsections.

Elementwise Pruning

Elementwise pruning removes individual weights within the network according to an importance metric. Considering individual weights allows for more flexibility in network architecture at the expense of added overhead during inference for sparse matrix operations and storage. Custom hardware can offset some of the overhead costs by natively supporting sparse matrix storage and multiplication at the hardware level. Heuristics for selecting weights are diverse, but generally aim to quantify how much a weight affects the accuracy of the network in order to enable pruning of the least important weights. Han et al. [9] propose pruning weights of the lowest magnitude within each layer. Frankle et al. [10] also perform magnitude pruning, but at a global scale, which allows varying levels of sparsity at different layers. Gradient-based approaches include Taylor Pruning [11], movement pruning [12], and L_0 regularization [13]. These methods approximate the loss function with a first order Taylor polynomial, then rank weights according to their impact on this loss function.

With models trained via transfer learning, pruning can occur during the pretraining or fine-tuning phase of training. Several studies have attempted to compress a pretrained BERT model while fine-tuning on a downstream task such as question answering or sentence classification. Gordon et al. [14] study the effects of pruning BERT before fine-tuning and conclude that BERT's prunability does not improve when pruning on a downstream task, compared to pruning during pretraining. The work of Sanh et al. [12] contradicts this conclusion by pruning weights during fine tuning according to their relative importance on the downstream task. They find that the magnitude of weights does not change very much during the fine-tuning, so using magnitude pruning is ill-suited to take the task into account. Instead, they prune weights whose magnitudes decrease during fine-tuning, which intuitively captures the weights relevance to the task. Their results outperform magnitude pruning.

Structured Pruning

Structured pruning removes weights in groups for the benefit of making weight storage and access more efficient. The main drawback is that pruning is far less precise, making it difficult to only remove unimportant weights. In the computer vision domain, entire feature maps are often pruned [11].

In the NLP domain, previous work on structured pruning has mainly targeted the self-attention heads. These works identify redundancies in the self attention blocks but do not address model FLOPs or latency very well because they ignore the FFN layers. Voita et al. [15] apply L_0 regularization at the level of attention heads to induce sparsity and prune them. They prune a majority of heads with little loss in translation quality and show that the most important heads have clear linguistic roles while many heads have no clear role and are easily pruned. Michel et al. Michel et al. [16] also experiment with removing attention heads and find that pruning up to 40% of BERT heads results in no drop in performance. Wang et al. [17] propose a method for structured pruning of any weight matrix in the Transformer by performing low rank factorization and pruning individual columns of the matrices (neuron pruning). They are able to reach 65% sparsity on RoBERTa while only losing an average of 1% accuracy on several GLUE tasks after fine tuning. They achieve only 1.5x speedup at 80% sparsity of Transformer-XL, partially because matrix multiplication only contributes 40% of latency during inference.

Previous works have also pruned structures beyond the self-attention heads. Fan et al. [18] introduce LayerDrop, which stochastically removes layers during training as a regularization technique, then prunes them away at inference time. They found that their pruned models outperform models trained from scratch with the same final depth. McCarley et al. [19] apply structured pruning to a BERT model that is already fine-tuned on two question-answering tasks. Their method prunes attention heads, feed-forward network (FFN) neurons, and embedding dimensions. They begin with a fine-tuned bert-base model pretrained on SQUAD 2.0 and train their pruning gate variables for one epoch on the same dataset while holding weight parameters constant. The best sparsifying algo-

rithm was L_0 regularization of the gate variables [13]. Given a 5 point F1 accuracy loss threshold, the algorithm was able to prune 48% of attention heads or 70% of FFN activations, but could not prune entire dimensions from the embedding throughout the full network because of the varying dependencies in each layer. The results also show that the early and late layers of the network were most prunable. To test how transferable the pruned network is, they experiment with training the network on the Natural Questions dataset after pruning, and observe promising results which suggest the pruned network removes redundancies which are not needed in similar datasets.

SchuBERT [20] uses structured pruning and sparsity-inducing regularization to learn optimal layer dimensions of BERT on the pretraining text corpus. The method can prune the hidden layer width, number of attention heads, FFN width, and key-query matrix columns. Each of these dimensions is independent between layers, so the pruning algorithm simply seeks to minimize training loss and number of network parameters. The pruning parameters mask activations, which corresponds to masking columns of preceding weights and rows of following weights.

2.1.2 NAS

Inspired by the great success of AutoML in model optimization, neural architecture search (NAS) is introduced to design model architectures automatically. NAS is expected to reduce the effort of human experts in neural architecture design. The most accurate mathematical solution to NAS is to train each of the candidates within the search space from scratch to convergence, and compare their performance. However, when the search space is large, this kind of solution is impractical because of the high training cost. NAS methods offer design spaces and search algorithms to find optimal architectures without training for an unacceptable amount of time.

Problem description of NAS

Neural architecture search (NAS) offers another method for reducing the size and computation complexity of deep neural networks. There are several popular methods for finding an optimal architecture for a given dataset, such as differentiable architecture search [21], reinforcement learning [22], and evolutionary search [4]. The design spaces also vary in their use of weight sharing [23] [6] or repeated network cells [21]. Most NAS works have focused on CNN design spaces, but the search methods have also been applied to architectures such as recurrent neural networks and Transformers. Methods also vary from designing individual cells which are repeated to form a network [21], or learning optimal hyperparameters for each layer in the network [6]. The NAS problem is also often formulated as a multi-objective optimization problem with both accuracy and a hardware-related metric such as latency or energy. Hardware-aware NAS methods are described in section 2.2.3.

Pruning as NAS

Several studies have equated pruning to neural architecture search. Frankle et al. [10] proposed the Lottery Ticket Hypothesis, which claims that certain neural connections combined with their initialized weights can achieve comparable accuracy to the unpruned network. In contrast, Liu et al. [24] and Gale et al. [25] find that the pruned architecture alone is responsible for the network accuracy regardless of weight initialization, which reframes pruning as a form of architecture search. The aforementioned methods both studied convolutional neural networks, although Frankle et al. [10] used smaller datasets and lower learning rates, which altered their results. Pruning can be a less costly alternative to NAS for a model like BERT because pruning starts from a pretrained model.

2.1.3 Quantization

Quantization seeks to reduce the size and computational cost of a model by reducing the bit width of weights and activations. Quantization is well studied in the vision domain, but not as much in NLP. Shen et al. [26] propose a group-wise quantization scheme using second order Hessian information with the ability to mix precision across layers. The idea is that weight matrices with higher hessian eigenvectors are more sensitive to quantization, so they do not quantize those layers to ultra-low precision. They employ group-wise quantization by splitting each layer's weights into 128 subgroups, which are each quantized independently according to their range of values. The results show that quantizing down to as low as 2 bits in some parts of the network is possible without losing significant accuracy. They also show that the embedding layer is more sensitive to quantization. Zafrir et al. [27] quantize all weights and activations to 8 bits for a 4x reduction in model size and compatibility with 8 bit instructions on supported hardware. They train with quantization aware training and the straight-through estimator to allow for gradients to pass through the fake quantizer.

2.1.4 Efficient Architectures

Another method for improving NLP model performance on edge devices is to design more efficient architectures. Lan et al. [28] propose AlBERT, which reduces the memory footprint of BERT by sharing weights across layers and factorizing the embedding matrix. While sharing weights across all layers does not significantly reduce latency, it does reduce memory overhead, which would reduce energy use on edge devices. More importantly, the reduced footprint of the model permits larger batch sizes or a larger model during training. They decompose the large input embedding layer into two smaller matrixes at the beginning of the network to further reduce model size. SqueezeBERT [29] replaces the position-wise fully connected layers and parts of the self attention layers with 1D group convolutions, which significantly reduces latency on mobile devices. The

paper also provides a detailed breakdown of latency in the BERT model. Wu et al. [30] propose a long-short range attention module, which uses convolution to handle the local syntactic relationships, and traditional self-attention to handle the long range ones. The authors show that the self-attention block focuses on long-range relationships when placed in parallel with a convolution which is restricted to short-range relationships.

2.2 Hardware-Aware Methods

2.2.1 Hardware-aware Modeling and Optimization

Hardware-aware optimization differs from conventional methods in the choice of objective function. Rather than using a generic hardware metric such as FLOPs or model size, hardware-aware methods optimize device-specific metrics like latency, throughput, energy consumption, or operating cost. Model size can accurately represent the memory footprint of a model, but FLOPs has been shown to poorly predict latency on real hardware [2][3][31]. Sparse networks exemplify this observation due to the inability of parallel hardware to speed up sparse matrix operations despite a reduction in FLOPs.

Predicting Hardware-Dependent Metrics

Optimizing device-specific metrics requires a model which can predict the metrics given a neural network, or a method for rapidly evaluating the metrics with the hardware in the loop.

Analytical methods offer calculations of DNN energy consumption and latency given detailed knowledge about the device under test. Simulators such as Accelergy [32] and Paleo [33] consider the processing element architecture, memory hierarchy, and technology node of a chip in order to predict energy or latency for a given DNN. These methods require no training data to learn a model, but they may struggle to model the effects of the entire software stack for a particular use case.

NeuralPower [34] predicts CNN inference latency and power consumption by using sparse polynomial regression. In contrast to analytical methods, NeuralPower trains a performance model by sampling CNN layers of different dimensions and measuring the corresponding power and latency. The use of real data improves the quality of the power and latency estimation over Paleo, but requires physical testing of the hardware.

More recent approaches also treat the target device as a black box. Stamoulis et al. [6] sample each possible layer configuration, test the latency on the hardware, and store each value in a lookup table. Since each individual layer has a small number of hyperparameter configurations, the process is fast and relies on collected data rather than a regression model. They smooth the discrete architecture choices with a sigmoid function to make the lookup table differentiable for optimization purposes. Cai et al. [35] [4] follows this strategy for their hardware-aware NAS method as well. Both methods assume that the sum of predicted layer-wise latencies accurately predicts overall model latency.

Hardware-aware Optimization

Maximizing DNN performance while minimizing latency is a multi-objective optimization problem. Given more than one objective, it is common to present multiple solutions along the pareto curve which optimize the trade-off in objectives. Alternatively, one can set a fixed constraint on one objective and optimize the other.

Stamoulis et al. [6] use stochastic gradient descent (SGD) to optimize both neural network weights and architecture parameters during training. The loss function is a weighted sum of cross entropy loss and the predicted latency. Minimizing the loss function results in an efficient and accurate model, with the trade-off controlled by the ratio of the loss terms.

Cai et al. [4] use an accuracy prediction and latency prediction model to rapidly evaluate potential DNN architectures. The fast evaluation of architectures allows the algorithm to use evolutionary search to find pareto-optimal solutions. Their method requires a large

one-time cost of training a supernet and evaluating its subnets to train an accuracy predictor. Searching for hardware-specific architectures comes at very little cost afterwards.

HyperPower [36] uses Bayesian optimization to search for DNN architectures subject to power and memory constraints. Using a power model, the algorithm samples potential DNN architectures and trains them to convergence to test for accuracy. Each pair of architecture and test accuracy is added to a Gaussian process model which predicts accuracy given an architecture. Hyperpower samples architectures with the highest likelihood of lying on the pareto curve, and updates the model after each sampled architecture to focus sampling on more promising architectures. This method is costly because it requires training many models from scratch to convergence.

2.2.2 Pruning

Hardware-aware pruning methods consider a device-specific information when choosing which weights to remove from a model. Lu et al. [37] propose to group weights according to the SIMD parallelism of the target processor. The groups of pruned weights can be more efficiently stored, and the sparse matrix operations more efficiently computed when the weights remain in SIMD-sized groupd. This method was most effective for low parallelism hardware such as a microcontroller and a CPU.

Turner et al. [38] propose a latency-aware pruning technique which uses knowledge distillation to train the lightweight network. First, they profile the target hardware to discover the staircase pattern of latency vs. number of channels in a CNN layer. Next, they prune a model to the desired latency, and modify the channel widths in each layer to match the nearest optimal widths found in the latency model. This modification allows the model to increase width where it makes little impact to latency, and reduce width when it greatly improves latency. Another paper [39] from the research group gives an analysis of neural network latency on mobile GPUs. The work shows how small variations in CNN channel dimensions can cause latency variations of +/- 50% due to kernel optimizations in the deep learning stack.

Netadapt [40] uses device latency as a target while shrinking a CNN. At each pruning iteration, the prunes each layer individually to the desired latency threshold and evaluates the new network. It then prunes the layer which maintained the highest accuracy and leaves the others untouched. The iterative layer-wise pruning approach makes it possible to prune all layers of the network according to their sensitivities to network accuracy, while also considering the desired model latency. Yang et al. [41] proposed a similar method, but uses inference energy as the objective. They order layers according to their energy consumption at inference time and first prune layers which consume the most energy.

2.2.3 NAS

Single Path NAS [6] addresses the problem of efficient hardware-aware neural architecture search. The authors' approach shares weights across architectures by using a "Single-Path" search space. Their search space spans each layer of the network and consists of superkernels, from which the NAS algorithm attempts to find the optimal subset of weights for accuracy and latency. The latency loss function is differentiable with respect to architecture parameters, so the architecture is co-optimized along with model weights during training. The hardware latency model considers each DNN layer independently, and shows that summing the latency of each layer is a reasonable approximation for the latency of the full network. Single-Path NAS achieves similar accuracy to comparable multi-path methods such as ProxylessNAS [35] when training on the ImageNet dataset with mobile latency constraints, but only uses 4% of the training time, which is faster than any comparable method.

Cai et al. [4] perform neural architecture search (NAS) for multiple hardware platforms with low training cost. Their method shrinks a deep neural network (DNN) in the depth, width, resolution, and kernel size dimensions to produce more efficient networks without retraining. By sharing weights across network scales and training across these scales, the learned weights for the smaller networks do not degrade the accuracy of the

larger network, which is a major concern for methods that alter network architecture during training. The paper also proposes a hardware-aware evolutionary search method for finding an optimal network architecture according to real performance constraints. They develop an accuracy predictor, and a latency lookup table to quickly guide their search. Their method results in state-of-the-art latency and accuracy on ImageNet [42] in the mobile setting in 2020, with high initial training cost that does not scale with the number of devices under test. The progressive shrinking method could prove useful in developing models that are resilient to architecture alterations for various hardware platforms.

Wang et al. [43] propose a NAS method for Transformers which uses a single-path supertransformer from which they derive smaller networks similar to the supernets of Cai et al. [4] and Stamoulis et al. [6]. They use latency and accuracy prediction models to efficiently sample architectures for an evolutionary search algorithm which seeks to optimize latency and accuracy on a specific hardware platform. They expand the architecture space by allowing encoder-decoder attention to access lower levels of the encoder instead of limiting access to the encoder output layer. The algorithm uniformly samples subnetworks during training of the supertransformer.

2.3 DNN Performance Evaluation

Deep neural network performance is characterized with a variety of metrics, both hardware-agnostic and hardware-aware. This section focuses on metrics related to speed and efficiency rather on task-specific metrics like accuracy or F1 score.

2.3.1 Hardware-agnostic Evaluation

Model Size

Model size is a measure of the memory footprint of the parameters of a network. It is a function of the number of parameters and their precision, where a FP32 model would

be four times the size of an int8 model. At runtime, the model may require much larger memory footprint due to the size of the activations, but model size ignores this fact and instead focuses on the footprint of the model parameters alone. For devices with extremely limited memory, the model size can have a high impact on latency

FLOPs

FLoating point OPerations (FLOPs) is the number of operations required to perform inference with a model using floating point weights and activations. A floating point operation is an arithmetic operation with floating point precision operands, typically 32 bits in the case of deep neural networks. FLOPs can serve as an excellent proxy for latency in some cases, particularly where a neural network operation throughput is limited by computation bandwidth rather than memory bandwidth [44].

2.3.2 Hardware-aware Evaluation

Throughput

Throughput is the number of inferences per second. It is most relevant for cloud hardware with large batch sizes. This metric is equivalent to the inverse of latency when the batch size is one.

Latency

Latency is the amount of time it takes to perform inference. It is most applicable to edge devices where real-time results are required, so a batch size of one is appropriate.

Energy and Power

For battery-powered devices, the energy consumed by one inference is a useful metric. It is difficult to measure the impact of software on instantaneous power at any given time, but the average power is simpler to derive as $\overline{P} = energy/latency$.

Hardware cost

Hardware cost is the cost of manufacturing a device and maintaining it's operation. Factors include the fab process and die size for the silicon. Hardware systems in the cloud have additional costs related to power delivery, cooling, and electricity usage.

2.4 Conclusion

DNN model compression is a well studied topic, yet it offers several research directions which remain unexplored. Compression of the heavily overparameterized BERT model is an active area of research. There is no fundamental consensus of which individual compression methods, or combination of methods, is optimal for reducing model size and inference time on Transformer-based models.

Chapter 3

Methods

This work proposes LAP-NAS, a model compression algorithm which applies several techniques from the deep learning and hardware modeling fields to minimize on-device latency of BERT-based deep neural networks via structured pruning. We describe each technique in the following sections to enable full reproducibility of our results and to share the techniques for future work. Our work combines pruning metrics, latency modeling, and multi-objective optimization to solve the proposed problem. First we describe in mathematical terms the optimization problem. Next, we explain the first order taylor approximation importance metric and iterative pruning algorithm used in our work. In addition, we explain how our genetic algorithm search method integrates pruning and latency modeling to enable latency minimization while preserving accuracy. Finally, we outline our layer-wise latency modeling method for pruned BERT-based networks.

An overview of the LAP-NAS algorithm is shown in Figure 3.1. In short, we use structured pruning to remove self-attention heads and feed forward network neurons from each layer in the BERT model until reaching a target sparsity. Next, we apply the latency-aware architecture search method over the pruneable model parameters to discover an architecture which runs faster on a target device without sacrificing accuracy.

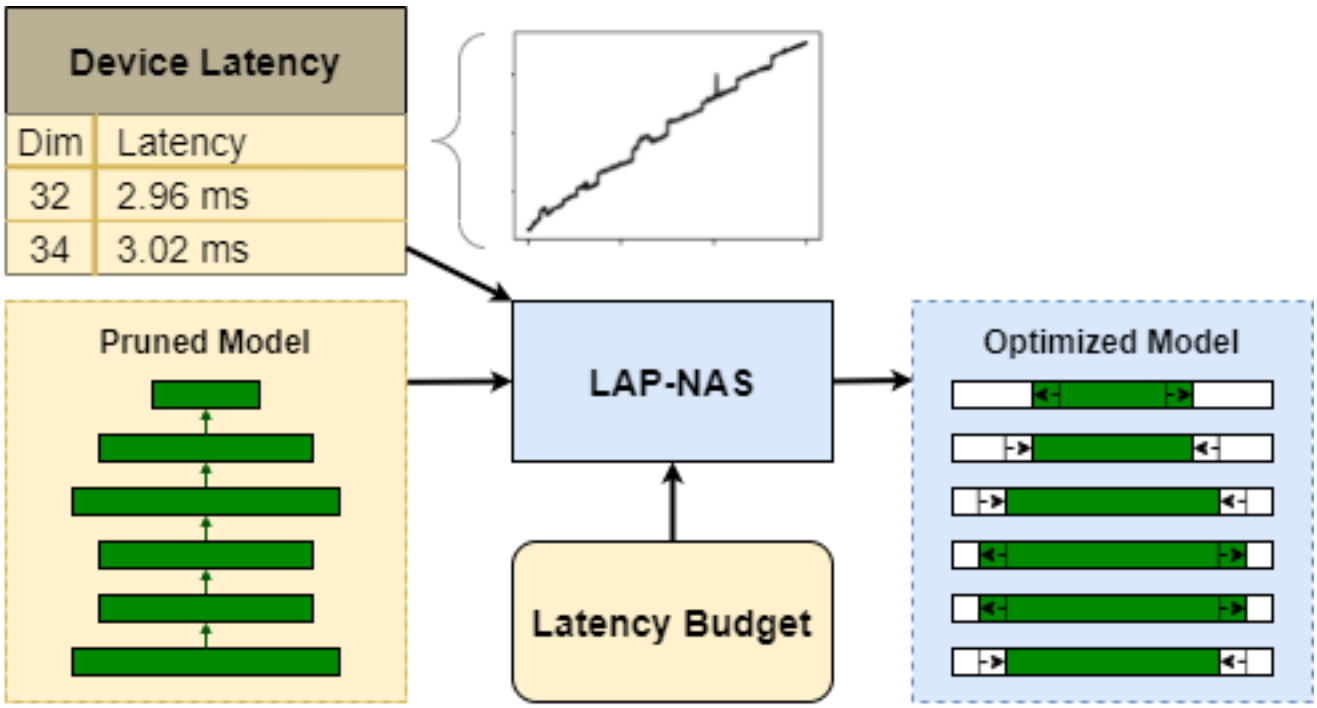

Figure 3.1: LAP-NAS Overview.

3.1 Optimization Problem

Optimization of a neural network with multiple objectives introduces significant challenges. When there is a single objective being optimized, one can compare any two solutions and decide which is better by simply comparing the magnitude of the objective for each solution. With multiple objectives, we may find solutions which are better in some aspects while worse in others, and it no longer becomes clear which solution is superior. There exists a set of solutions, the pareto front, which contains all dominant solutions where one cannot improve a single objective without worsening another. In this work, we constrain the search space according to a fixed sparsity budget, from which we aim to find the pruned network with minimal latency without degrading accuracy. Therefore we search for only one solution along the pareto front.

Our optimization problem is shown in Equation (3.1). We minimize the latency $L(\mathrm{W})$ of the pruned model with pruned weights W subject to the constraint that accuracy on the

test dataset D_{test} does not drop below its starting value. We aim to preserve the starting accuracy of the pruned model $acc_{\text{start}}(D_{\text{test}})$.

$$\min_{W} \; L(\mathrm{W}) \;\; \text{s.t.} \;\; \mathrm{acc}(D_{\text{test}}|\mathrm{W}) \geq acc_{\text{start}}(D_{\text{test}}) \tag{3.1}$$

Accuracy on the test set is not available during training, so we approximate $acc(D_{\text{test}}|\mathrm{W})$ with the training loss $\mathcal{L}$ on training set D_{train}. The pruned model latency on the target device must also be approximated during training with $\hat{L}(\mathrm{W})$, which gives us the practical objective in Equation (3.2).

$$\min_{\mathrm{W}} \hat{L}(\mathrm{W}) \;\; \text{s.t.} \;\; \mathcal{L}(D_{\text{train}}|\mathrm{W}) \leq \mathcal{L}_{\text{start}} \tag{3.2}$$

3.2 Structured Pruning

To compress the pretrained Transformer-based model, we choose to use structured pruning. We rank structures of weights from the multi-head self-attention and feedforward network (FFN) sublayers of each BERT layer with a first order data dependent importance metric. The choice of structured pruning as a compression method rests on the following requirements for LAP-NAS:

1. Low cost: The compression method must not require retraining with the pretraining dataset

2. Flexible: The method must work for a variety of mobile hardware platforms

The requirement of low cost for a pretrained model makes it difficult to add parameters or change operators in the network, because the modified network would need significant retraining with the original dataset to overcome the drastic changes. Therefore, we limit the architecture search space to all pruned subnetworks within the pretrained model. While this search space is still extremely large, we can quickly evaluate the sub-

networks by simply removing weights and fine tuning the network with the downstream task dataset.

Structured pruning enables speedup on a variety of target devices, because it reduces both model size and FLOPs at inference time without requiring any changes to the runtime in use. The degree of speedup seen on the Nvidia Xavier NX with structured pruning of Distilbert-base is shown in Table 3.1.

Table 3.1: Latency reduction on DistilBERT FP32 model with structured pruning of self-attention heads and FFN neurons.

Sparsity	Latency Improvement
10%	6%
30%	23%
50%	38%
70%	55%
90%	65%

The choice of which elements to prune from BERT is guided by our goals and the design of BERT. Due to the residual connections within BERT layers, the hidden dimension h is constrained to remain constant throughout the original BERT layer. In our design, we maintain this rule in order to prevent the insertion of extra operations to realign the pruned activations at each residual connection. Our conclusion is to prune the inner dimension of the feed forward network and the individual heads of the multi-head self-attention layer. Previous work has investigated pruning more parameters, but only during the pretraining step [20]. Figure 3.2 shows the architecture of a single BERT layer. When pruning the inner dimension of the FFN layer (d_f), we remove rows of weights from the first fully connected (FC) layer and the columns from the second FC layer. When pruning attention heads, we remove each of the weight matrices Q,K,V associated with that head, as well as the corresponding columns of the output linear layer after self-attention which returns the multi-head outputs to the hidden dimension. The effect of pruning one of n heads is to remove $\frac{1}{n}$ of the parameters and FLOPs from multi-head self-attention layer. Figure 3.3 shows the multi-head self-attention operation, where each

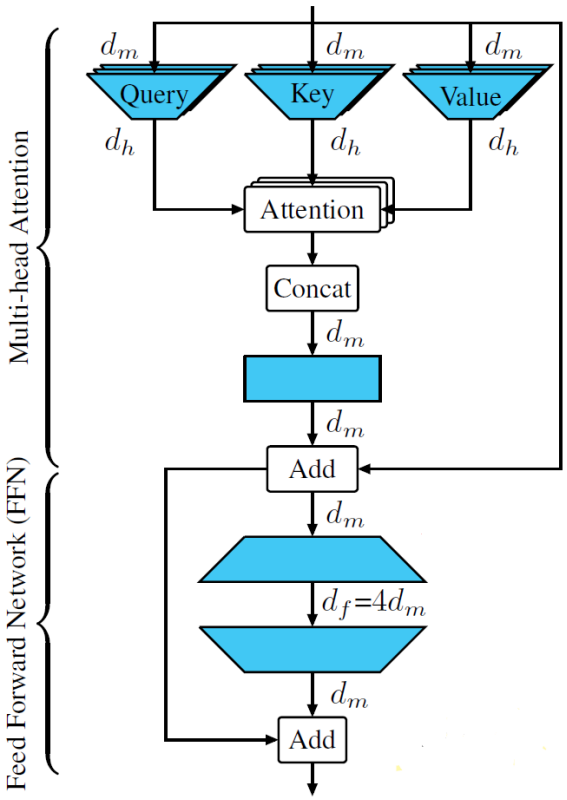

Figure 3.2: An encoder layer of the Transformer architecture [45] [46].

row of Q,K,V matrices belongs to one head. Pruning the head removes those matrices and the corresponding weights from W^O.

Multiple structured pruning methods were explored to choose the best for this work. The simplest method explored was magnitude pruning, which ranks groups of weights by the L2 norm of their weight values. While this method works well for unstructured pruning [48] we observe far worse performance with large structure sizes compared to data-dependent approaches. The work of Sanh et al. [12] showed that L0 regularization and first order Taylor approximation approaches worked well when pruning BERT during fine-tuning. We also observed promising results with the first order Taylor approximation importance metric. A positive demonstration of the pruning method is shown in Table 3.2.

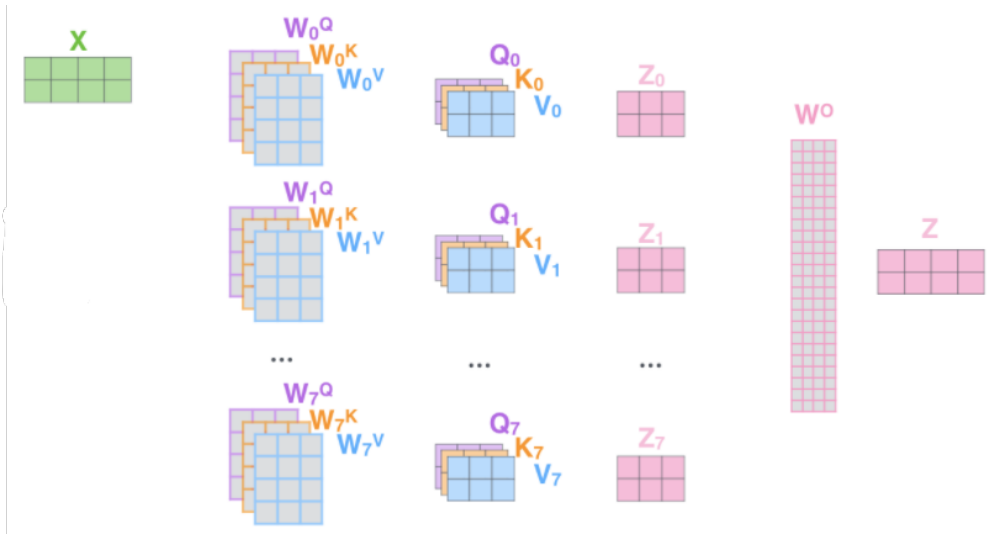

Figure 3.3: Visualized multi-head self-attention operation of the encoder layer in the Transformer architecture [47].

Table 3.2: Accuracy of BERT-base model when jointly pruned and fine-tuned on the SST-2 dataset at a sparsity of 90% in the FFN layer alone.

Pruning Method	SST-2 Accuracy
Random	0.87
Taylor	0.91

3.2.1 First Order Taylor Series Importance Metric

We use structured pruning to reduce the latency of the BERT model. We rank weight structures with the first order Taylor approximation importance metric and prune those structures with the least importance.

The self-attention and feed forward network sublayers contribute a majority of latency in the BERT model [49], so we choose to prune from these parts of the network. The weights pruned are not tied to other weights via residual connections, further simplifying the approach. While only pruning self-attention heads and feed forward network neurons, we observe an inference latency reduction of 65% on the Nvidia Jetson Xavier NX GPU platform by pruning 90% of weights in these two layers, as shown in Table 3.1. We use iterative pruning to progressively reduce sparsity and fine-tune the network until a target sparsity budget is reached [9].

We choose the first order Taylor expansion-based metric to measure parameter importance and globally rank parameters across layers [11]. Previous works [50] [19] have already applied this metric to BERT models to remove the self-attention heads and FFN neurons. The intuition behind this metric is that we wish to prune weights which have a small effect on the model output when removed. Therefore, we aim to minimize the change in loss with respect to removing a weight. A first order approximation of this change in loss with respect to weight magnitude performs well in practice, and is easily calculated with gradient information backpropagated during training. Note that LAP-NAS is compatible with any importance metric which can rank parameters across all layers of the model, but we choose the Taylor approximation method because we observe strong pruning results.

Equation (3.3) derives the importance of d_{ff} FFN neurons with m inputs to the feed-forward network and a bias term vector b accumulating to pre-activation sum u. Importance is defined as $I(\mathrm{w}_{\mathrm{l}}^{\mathrm{k}})$ for the weights of neuron k in layer l. Importance approximates the absolute change in loss $\mid \Delta\mathcal{L} \mid$ when removing a group of weights, where a larger change corresponds to higher sensitivity to pruning. Importance is accumulated during the training, making use of the entire training set. For each training batch the backpropagated gradients are used to update the importance scores, as shown in Algorithm 1. The normalization of importance from each layer allows for a global ranking of parameters in the network [11]. Equation (3.4) shows the L2 normalization of the importance within a layer.

$$
\begin{aligned}
I(\mathrm{w_l}) =& \mid \Delta\mathcal{L}(\mathrm{w_l} = 0) \mid \\
=& \mid \mathcal{L} - (\mathcal{L} - \frac{\partial \mathcal{L}}{\partial u_l} u_l) \mid \\
=& \mid \frac{\partial \mathcal{L}}{\partial z} z \mid \\
=& \sum_{i=1}^{m} \mid \frac{\partial \mathcal{L}}{\partial z} \frac{\partial z}{\partial w_i} z \mid \\
=& \sum_{i=1}^{m} \mid \frac{\partial \mathcal{L}}{\partial w_i} w_i \mid + \mid \frac{\partial \mathcal{L}}{\partial b} b \mid
\end{aligned}
\tag{3.3}
$$

$$
I_{norm}(\mathrm{w_l}) = \frac{I(\mathrm{w_l})}{\|I(\mathrm{w_l})\|_2} \tag{3.4}
$$

Algorithm 1: LAP-NAS Iterative Pruning Training Step

```
for batch in train dataset do
    Perform inference, compute loss, backpropagate gradients
    for layer l in model do
        Update head importance
        Update FFN neuron importance
    end
    Update model weights with Adam Optimizer
    Zero gradients
end
```

Calculating the importance for FFN layers and multi-head self-attention layers is fairly similar. As shown above, for the FFN neurons we perform a weighted sum of gradients flowing through the neuron's connected weights at both the input and output of the neuron. For the heads, we simply sum the absolute value of the gradient propagated through a binary mask layer applied to the output of each head, as in [16]. The formulation is shown in Equation (3.5), where ξ_h is the mask variable for head h. Note that this formulation is mathematically equivalent to the neuron importance equations (3.3) [16]. The gradient in the equation measures the change in loss with respect to the binary mask variable ξ_h. A large gradient suggests that the self-attention head's output has a large effect on the model's output and is therefore important to model performance. A small gradi-

ent suggests that the head can be removed from the model with minimal impact on the model output by zeroing the binary mask.

$$I_h = \mathbb{E}_{x \sim X} \left| \frac{\partial \mathcal{L}(x)}{\partial \xi_h} \right| \tag{3.5}$$

3.2.2 Iterative Pruning

LAP-NAS uses iterative pruning to gradually reduce model sparsity with a small degradation in accuracy. Our iterative pruning algorithm is adopted from Han et al [51] which showed that alternating between pruning and fine-tuning a model enables higher levels of sparsity than pruning in one shot and fine-tuning.

The sparsity schedule controls how much to prune between fine-tuning iterations. This work uses a cubic sparsity decay schedule adopted from Zhu and Gupta [52]. While they continuously prune parameters during training, we alternate between pruning and fine tuning. Algorithm 2 shows our iterative pruning procedure. For each step in the pruning schedule, we independently prune the FFN parameters and the self-attention heads. When pruning one of these sublayers, we rank all of the accumulated importance scores of active parameters groups in the model, then prune the least important parameters until we reach the step's scheduled sparsity. After pruning both the FFN and self-attention sublayers we fine-tune the network for two epochs, repeating until we complete all pruning iterations.

After the model has been pruned to the final sparsity s_f, the model is ready for the latency-aware optimization step described in section 3.3.1

3.2.3 Code Implementation

We implemented pruning by writing modified BERT-based models in Pytorch and fine-tuning them with a modified training script. Our pruneable models are derived from models found in the Hugging Face Transformers library [53], an open source Python li-

Algorithm 2: LAP-NAS Iterative Pruning

Input = [pretrained BERT-based Model]
Output= [Pruned model at sparsity s_f]
for *i in [0,1..,n]* **do**
  s_i = s$_f + (1 - s_f)\left(1 - \frac{i}{n}\right)^3$
  for *Sublayer in (heads, FFN)* **do**
    $k_i = \lfloor n\_parameters * s_i \rfloor$
    $num\_to\_prune = k_i - k_{i-1}$
    Get k_{i-1} importance scores I of active parameters from each layer
    sort I
    Prune num_to_prune lowest scoring parameters
  end
  Fine-tune 2 epochs
end

brary for machine learning with Pytorch and Tensorflow. It's model zoo provides the pretrained BERT-base and DistilBERT-base models used in our experiments. Our implementation of structured pruning involves introducing mask layers in the FFN and multihead self-attention layers. During inference, the activations of these layers are each multiplied by a mask variable, where a mask value of 0 effectively removes the preceding weights from the network and a value of 1 retains them. For the FFN layer, we insert a mask of length equal to the inner dimension d_{ff}. For the Bert-base model d_{ff} is 3072, a factor of four times greater than the model's hidden dimension of 768. The Hugging Face model already has a head mask layer implemented for inference, so we use that mask to prune the heads of the multi-head self-attention layer. The gradients through this mask are also used to calculate head importance scores.

3.3 Latency Optimization

Structured pruning alone is a hardware-agnostic tool for DNN compression. In this section we present the latency-aware architecture search aspect of LAP-NAS. The search method optimizes a model after pruning to a desired sparsity. We also present an al-

ternative latency-aware pruning method, which introduces latency information into the iterative pruning algorithm to avoid the search process altogether.

3.3.1 Pruned Neural Architecture Search

We apply neural architecture search (NAS) to maximize accuracy while keeping pruned model latency under a threshold. LAP-NAS applies this search after pruning to optimize layer dimensions within a limited search space starting from the pruned model dimensions. This design space consists of models reachable by pruning/unpruning the model by a limited amount. The candidate architectures are evaluated using a custom fitness function drawn from pruning importance metrics and latency lookup tables. By searching for a model architecture with minimal latency and enforcing an accuracy budget, we can efficiently optimize the pruned model for latency on the target device.

Starting from a pruned model, we initialize a NAS search space consisting of all pruned architectures in the model's neighborhood. For each layer in the network, we consider candidate dimensions within +/- 64 FFN neurons and +/- 2 attention heads from the pruned model. The design space has a granularity of two neurons and one attention head. Evaluating this large design space of $5^{12} * 65^{12} \approx 10^{30}$ candidates for the 12-layer BERT model by training each one would be computationally infeasible. To speed up the search process, we construct a fitness function which uses the pruning importance metrics as a proxy for model accuracy, Eq. (3.6). For each target architecture in the search space, we determine the which parameters must be pruned or unpruned from the network. We then add the importance of unpruned parameters and subtract the importance of pruned parameters to generate a change in importance ΔI for the target architecture. To compute the change in latency when pruning to the candidate architecture, ΔL, we sum the expected change in latency for each layer. We then add a penalty for architectures with $\Delta I < 0$, as we expect them to degrade accuracy.

We use a genetic algorithm [54] to search for the best architecture. This algorithm initializes a set of candidate solutions and iteratively improves the population quality

with an evolution-inspired procedure. First, all candidates are evaluated using a fitness function. Next, the strongest solutions are chosen as parents for new solutions. The new solutions are generated by mixing characteristics of two parents with a crossover operation, followed by randomly perturbing the resulting offspring with a mutation operation. Finally, the new candidates are evaluated and we return the population back to its original size by removing the weakest solutions. The algorithm repeats until its strongest solution converges or until reaching the maximum number of generations.

Our genetic algorithm optimizes the dimensions of each layer in the model. Each candidate architecture (chromosome) consists of the number of attention heads and the number of FFN neurons for each layer (genes). The 12 layer BERT model therefore has 24 genes to optimize. We setup the genetic algorithm with population size 100, mutation probability 0.1, elite ratio 0.01, crossover probability 0.5, and a convergence threshold of 30 generations without improvement.

The LAP-NAS search algorithm is shown in Algorithm 3.

$$\begin{aligned} \Delta L &= \sum^{l} LUT(candidate) - LUT(model) \\ penalty &= \begin{cases} 0 & \Delta I \geq 0 \\ 1 - 1000 * \Delta I & \Delta I < 0 \end{cases} \\ f &= \Delta L + penalty \end{aligned} \tag{3.6}$$

3.3.2 Latency-aware Pruning

The standard LAP-NAS algorithm in this work does not use latency information during the initial pruning phase. This leaves the architecture search method with a potentially ill-suited initial architecture for latency optimization. In response to this problem, we propose a latency-aware pruning technique to encourage the model to prune only where it improves the predicted on-device latency. This simple method ensures monotonic latency reduction throughout pruning despite potential noise and abormalities in the la-

Algorithm 3: Latency-Aware Pruned NAS

```
Input = [Fine-tuned, pruned BERT-based model, Latency LUT]
Output = [Latency-Optimized model]
Collect latencies, parameter importance
for layer in network do
    for search space in (heads,FFN) do
        for dim in search space do
            Compute Δlatency
            Compute ΔI
        end
    end
end
for search space in (heads,FFN) do
    Initialize population of architectures
    while not converged do
        update population of pruned architectures
    end
end
```

tency lookup tables which could cause a model with higher sparsity to have a higher latency.

This method prunes as many weights at a time as is needed to reduce the latency of a layer. In the standard algorithm, we globally rank parameters by importance from each layer in the network and prune the least important ones until reaching a desired sparsity. The latency-aware pruning method takes several iterations to reach the target sparsity. For each layer l, we determine the number of parameters p_l that must be pruned in order to reach the next lowest latency in the latency lookup table. Then we average the importance of each group of p_l parameters and prune the group with lowest average importance, repeating until the desired sparsity is reached.

For example, if a BERT FFN layer has 10 active heads, and the LUT shows an increase in latency when pruning to 9 heads, but a decrease in latency when pruning to 8 heads, the pruning algorithm will propose to prune 2 heads from that layer. It will then compare the average importance of head 9 and 10 of the layer to the importance of heads in other layers to determine where to prune next.

Algorithm 4: Latency-Aware Iterative Pruning

Input = [pretrained BERT-based Model]
Output= [Pruned model at sparsity s_f]
for *i in [0,1..,n]* **do**
  s_i = $s_f + (1 - s_f)\left(1 - \frac{i}{n}\right)^3$
  for *Sublayer in (heads, FFN)* **do**
    while $s(\mathrm{W}_{\mathrm{sublayer}}) \leq s_i$ **do**
      for *layer in model* **do**
        Find next sublayer dimension d_{next} which lowers latency
        p = d - d_{next}
        Average importance of p least important parameters in sublayer
      end
      Prune from layer with lowest average importance
    end
  end
  Fine-tune 2 epochs
end

3.4 Latency Modeling

3.4.1 layer-wise Latency Model

Optimizing a model for on-device latency requires knowledge of the relationship between model architecture and latency on the target device. We gather this knowledge by generating lookup tables of layer latency with respect to the number of attention heads or FFN neurons in the model.

The latency of a deep neural network can be characterized analytically via study of the operations in the network or empirically via direct measurements. Latency depends on all elements of the hardware/software stack and does not always scale linearly with FLOPs [3]. These elements include the device, math kernels (e.g. CUDNN, MKL), model graph optimization (e.g. Torch JIT, TensorRT), deep learning libraries (Tensorflow, PyTorch), and model architecture. Accurately modeling all of these elements is difficult due to the depth of the software stack. Therefore, we choose to directly measure latency and treat the HW/SW stack as a black box.

Directly measuring latency of every possible pruned BERT architecture would be infeasible given a pruned search space of up to $3072^{12} * 12^{12} \approx 10^{55}$ architectures BERT-base. Assuming that network latency can be approximated by the sum of individual layer latencies greatly reduces the space of layer architectures which must be measured to a more manageable $3072 * 12 = 36864$, for model with 3072 FFN neurons and 12 attention heads. This assumption has been employed in previous HW-NAS work [6] [35]. Further separating the FFN and multi-head self-attention sublayers reduces the space of architectures to measure down to just 3084. We outline the derivation of model latency from layer-wise measurements in Equation (3.7), where the latency of a model with pruned weights W is computed by summing over each layer's pruned latency. The pruned latency for a layer is computed by subtracting the speedup obtained from pruning the FFN sublayer $\mathrm{LUT}_{\mathrm{FFN}}(\mathrm{W}_{\mathrm{FFN}})$ and the self-attention sublayer $\mathrm{LUT}_{\mathrm{Heads}}(\mathrm{W}_{\mathrm{Heads}})$ from the unpruned layer latency $\mathrm{LUT}_{\mathrm{W}^*}$. Our lookup tables measure latency of an entire BERT layer, with respect to the number of FFN neurons or number of active self-attention heads in the layer.

$$
\begin{aligned}
\hat{L}(\mathrm{W}) &= \sum_{l=1}^{n} LUT_{\mathrm{W}^*} - \Delta L_{\text{FFN Pruned}} - \Delta L_{\text{Heads Pruned}} \\
\Delta L_{\text{FFN Pruned}} &= \mathrm{LUT}_{\mathrm{W}^*} - \mathrm{LUT}_{\mathrm{FFN}}(\mathrm{W}_{\mathrm{FFN}}) \\
\Delta L_{\text{Heads Pruned}} &= \mathrm{LUT}_{\mathrm{W}^*} - \mathrm{LUT}_{\mathrm{Heads}}(\mathrm{W}_{\mathrm{Heads}})
\end{aligned}
\tag{3.7}
$$

3.4.2 Latency Measurement Technique

We measure a layer's latency by isolating it from the rest of the BERT model and initializing it with the dimensions under test. A latency figure is obtained by feeding random data as the input to the layer and computing the output. We time 100 inference iterations of the layer and store the mean in our lookup table. To improve measurement consistency, the board is warmed up with 30 seconds of inference prior to data collection. We synchronize data between GPU and CPU at the end of each inference and use a batch size of one. If high variance is seen during the 100 inferences, the measurement is repeated, which helps remove outliers from the measurement process. We find that warming up

Table 3.3: Standard deviation of measured latency vs. measurement technique. Statistics computed over 100 inference latency measurements. Relative deviation calculated using mean measured latency of 3.2 ms.

Method	Latency Std. Dev. (ms)	Relative Std. Dev. (%)
No Warmup	2.5	78.1
30 sec Warmup	0.04	0.0125
30 sec Warmup + Max Clk	0.02	0.00625

the board before measuring latencies reduces variance, leading to a more reliable latency measurement. Setting the board to its maximum clock speed and fan setting also allows for more consistent results, as shown in Table 3.3. We do not measure latency of the input embedding or classifier layers because our pruning algorithm does not alter them.

3.4.3 Latency Lookup Table

In general, latency has a linear relationship with the number of neurons in a FFN layer, as shown in Figure 3.4 for the full precision layer on Nvidia Jetson Xavier NX GPU. However, there are layer dimensions for which latency changes non-linearly in a staircase pattern. For the 16-bit floating point quantized BERT layer on the same hardware, we observe an oscillating latency pattern with large variation every four neurons pruned, shown in Figure 3.4. To uncover the cause of the variation, we trace the CUDA function calls of the forward pass for each BERT layer and record the top 4 most called CUDA general matrix multiply (GEMM) functions in Table 3.4. The tracing revealed that despite differing in FFN width by only 4 neurons, the lower latency architecture of dimension 1024 made use of far more GEMM ldg8 CUDA function calls, which interweave 8 multiply-accumulate operations while loading data from global memory.

Table 3.4: GEMM function calls for FP16 BERT layer inference with FFN layer pruned to 1024 and 1028.

d_{ff}=1024			
Function Name	Time (%)	Time (ms)	Calls
volta_fp16_s884gemm_fp16_256x64_ldg8_f2f_tn	34.505112	8.159971	120
volta_fp16_s884gemm_fp16_64x64_ldg8_f2f_tn	13.127884	3.104559	30
volta_fp16_s884gemm_fp16_128x64_ldg8_f2f_tn	8.02185	1.897054	30
volta_fp16_s884gemm_fp16_64x64_ldg8_f2f_nn	5.307518	1.255153	60
d_{ff}=1028			
Function Name	Time(%)	Time (ms)	Calls
volta_fp16_sgemm_fp16_128x64_tn	24.579553	8.558379	30
volta_fp16_s884gemm_fp16_256x64_ldg8_f2f_tn	22.977374	8.000515	120
volta_fp16_sgemm_fp16_32x128_tn	22.008518	7.663168	30
volta_fp16_s884gemm_fp16_64x64_ldg8_f2f_nn	3.607457	1.256084	60

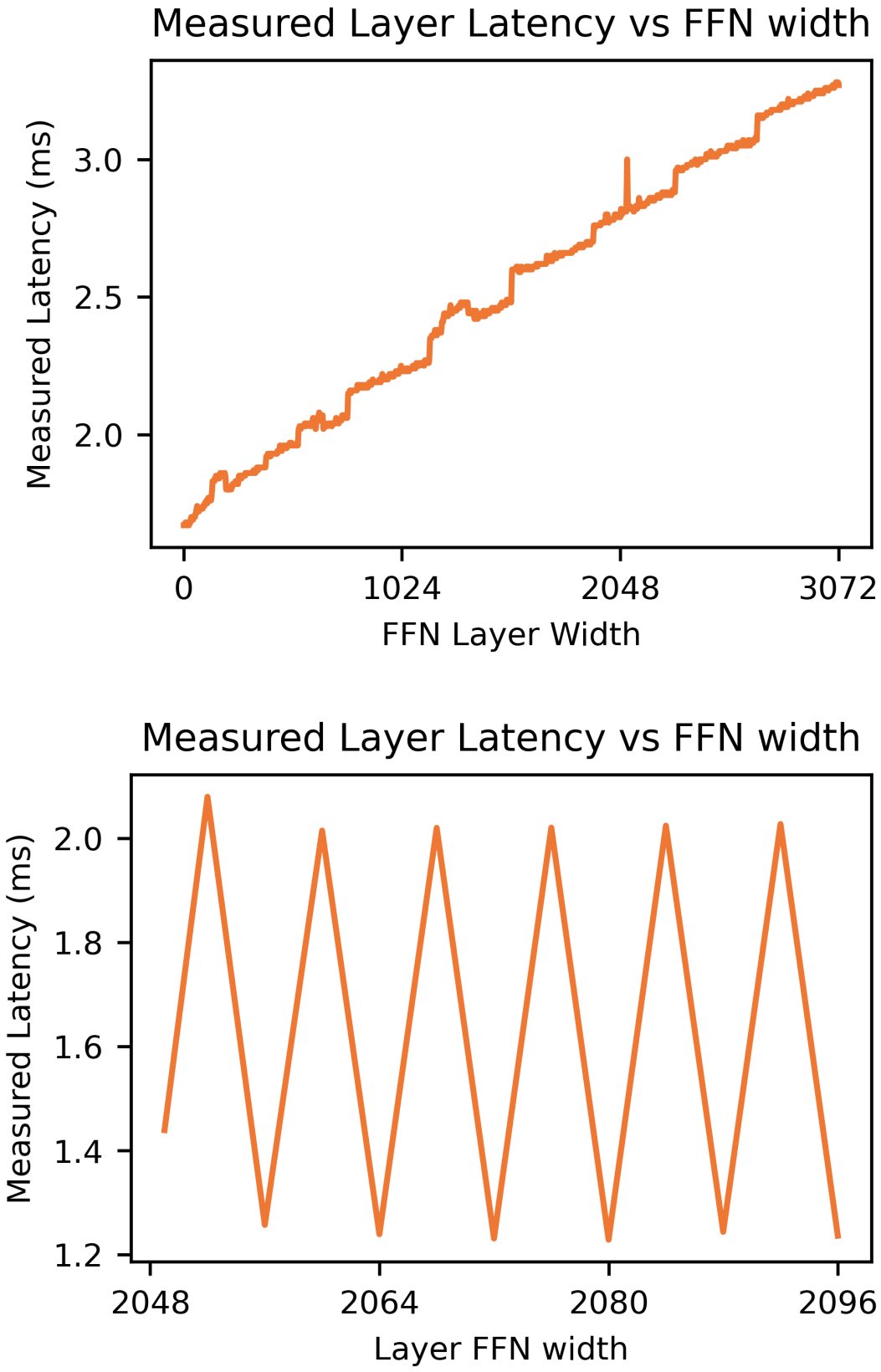

Figure 3.4: Latency measurements of a single BERT layer. Nvidia Jetson Xavier NX GPU. *Top*: FP32, *Bottom*: FP16.

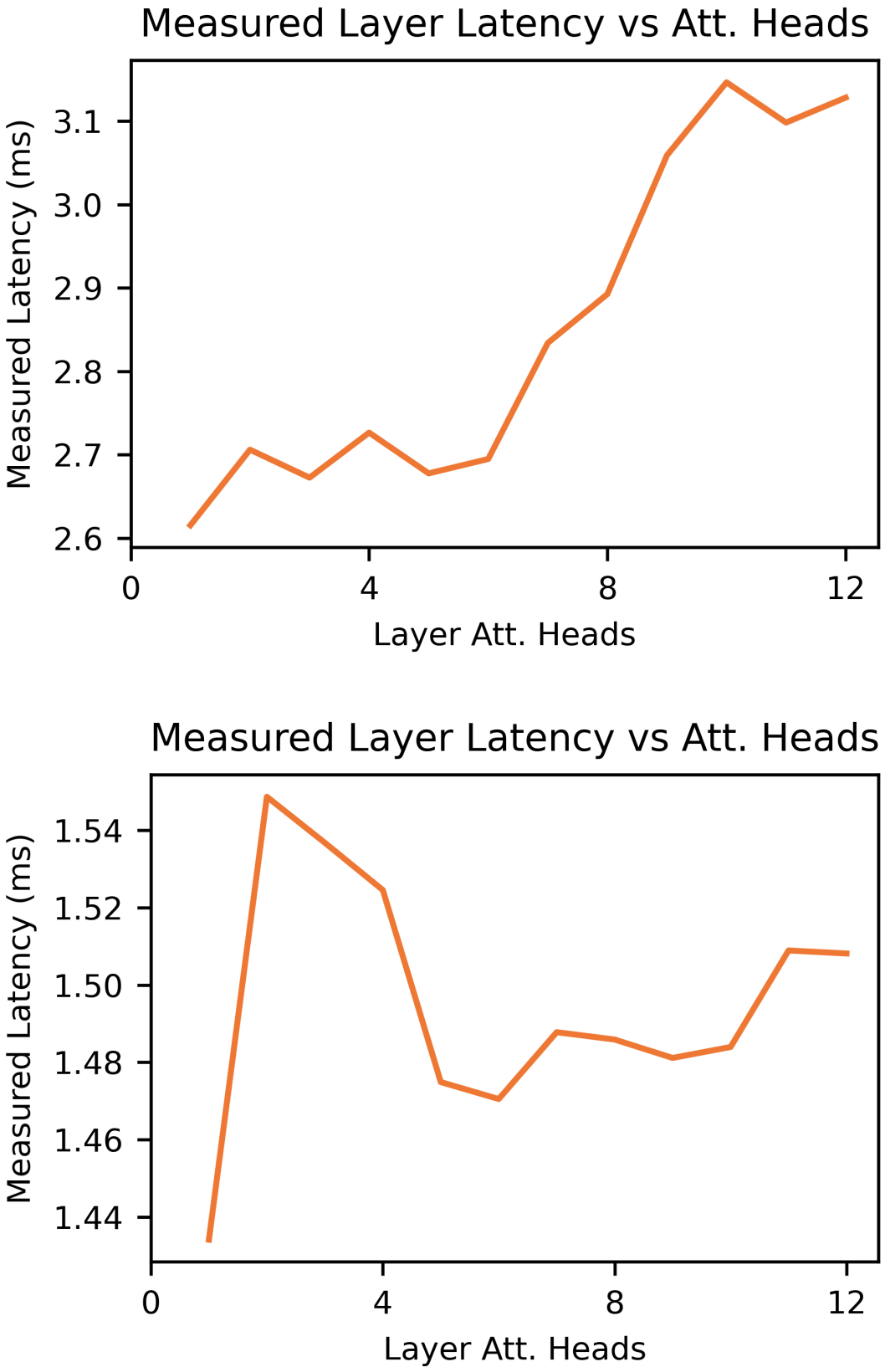

Figure 3.5: BERT layer latency vs. Number of active self-attention heads, Nvidia Jetson Xavier NX GPU. *Top*: FP32, *Bottom*: FP16.

Chapter 4

Experiments

We conducted a series of experiments to evaluate how well LAP-NAS improves latency over iterative pruning without any latency optimization. To support reproducibility of our results, we explain our experimental procedures, hyperparameters, and hardware configuration. First, we apply LAP-NAS to pruned models and record the latency and accuracy of the fine-tuned models to observe the utility of the NAS method in minimizing latency. Next, we evaluate the accuracy of the BERT latency lookup tables to determine whether it is useful as a proxy for on-device latency during NAS. In addition, we observe whether the optimized model dimensions line up with visually identifiable optimal dimensions along the lookup table latency curves to give a qualitative review of our method. Finally, we explore pruned BERT latencies and pruning importance scores with experiments performed during development of our final methods.

4.1 Experimental Setup

Model Settings We prune BERT-base [7] and DistilBERT-base [55], sourced from the Huggingface transformers model zoo [56]. Models are optimized for inference with TorchScript [57]. These two model architectures are identical to each other except that DistilBERT does not use token type id inputs and has only six layers rather than 12. We use

a learning rate of 3e-05 with a linear learning rate decay from the start of training to finish. We warm up the learning rate for one epoch and apply linear learning rate decay for the duration of pruning and fine-tuning. The models are trained with the AdamW optimizer with an epsilon of 1e-8. Although we tried applying learning rate rewinding [58] between pruning steps to reduce accuracy degradation during pruning, we found it to have a negative effect on fine-tuned performance.

Datasets We evaluate our model on a subset of the GLUE [59] tasks including CoLA [60], SST-2 [61], QNLI [59], QQP [62], and MRPC [63]. The variety of dataset size and task types present in the subset form a representative set of the GLUE tasks while keeping computation costs reasonable.

Hardware and Software Our experiments measure latency on the Nvidia Jetson Xavier NX development board, which has a six core ARMv8.2 64-bit CPU, 8GB memory, and 384-core Nvidia Volta GPU. We run inference using the full CPU and GPU with locked maximum clock speed and a board power limit of 15W. The board was used with Nvidia Jetpack 4.4.1 and the python packages torch 1.7.0 and transformers 3.2.0.

Pruning We use the cubic pruning sparsity schedule from [12] to set the pruning threshold at each iteration. We prune the model four times, each followed by two epochs of fine-tuning. This pruned model is then used for the baseline iterative pruning experiment and the LAP-NAS experiment For the iterative pruning results, we further fine-tune this pruned model for six epochs. For LAP-NAS results, we apply our NAS algorithm to the pruned model before fine-tuning for six epochs.

4.2 Experiment Descriptions

4.2.1 LAP-NAS Latency Optimization

In this experiment we minimize latency of the pruned model while aiming to maintain accuracy. First we prune the FFN and self attention sub layers to a global sparsity of 60% with iterative pruning. Next, we apply LAP-NAS to optimize the widths of all FFN

layers, then all self-attention layers of the same model. Following optimization we report the accuracy of the model on the GLUE task and the latency of the pruned model. Latency is presented using either the estimate of the lookup table or by directly measuring the pruned model on the device.

4.2.2 Evaluating Latency Lookup Table

In order to validate our use of a layer-wise latency lookup table, we perform several experiments comparing measured and predicted model latency on the Nvidia Jetson Xavier NX GPU. Our algorithm requires a good measure of relative latency of pruned models, so we construct experiments which measure how well the LUT predicts the difference in latency between two models. Previous works simply measure the accuracy of the latency model to determine its utility, but in our case we do not need absolute accuracy when searching for the fastest model from a limited set of models.

Chapter 5

Results

We present the latency and accuracy of our LAP-NAS pruned models in this section. In addition, the run time of the optimization algorithm is compared to previous work, and the latency lookup table is evaluated independently of LAP-NAS. Finally, we share latency lookup tables for diverse hardware platforms and discuss how our results contribute to the field. Our algorithm manages to improve model latency up to 2.3% when measured on the target device and a more promising 18.5% when using the latency LUT's predicted value.

5.1 LAP-NAS Latency Optimization

The goal of LAP-NAS is to minimize latency while maintaining accuracy of the pruned model. To measure its ability, we compare the LAP-NAS optimized model to its iteratively pruned baseline model in terms of LUT optimized latency, measured on-device latency, and accuracy. The baseline model has 60% of its attention heads and FFN neurons pruned using the Taylor Series importance metric. We apply LAP-NAS to this pruned model, and fine-tune both the original pruned model and the LAP-NAS optimized model before evaluating for accuracy.

Table 5.1 shows that LAP-NAS improves DistilBERT-base latency while also maintaining or improving accuracy. Latency (ms) values in Table 5.1 are drawn from the latency lookup table with respective bit precision. These are the latencies that the LAP-NAS algorithm believes it has reached when optimizing the model, but which rely on the simplifying assumptions made in generating the LUT. Using these latency calculations, LAP-NAS reduces latency by 18.5% and 2.3% on DistilBERT-base at 16 bit and 32 bit floating point precision, respectively. Accuracy results are less consistent, with the 16 bit experiments slightly improving accuracy and the 32 bit experiments slightly degrading accuracy. With the BERT-base model, we observe similar results, with a 14.1% and 2.2% latency reduction for the 16 and 32 bit models. The advantage to the FP16 optimization case is made clear by the latency measurements of Figure 3.4. The FP16 BERT layer exhibits large variations in latency with small changes in model dimensions, making it an ideal target for latency-aware optimization techniques. In contrast, the FP32 BERT layer exhibits a far more linear relationship between model dimensions and latency, so the difference between an optimal and sub-optimal architecture is smaller.

Table 5.2 shows the measured latency on the Nvidia Jetson Xavier NX board for each of the pruned models from Table 5.1. Here the results are less impressive, with LAP-NAS averaging a 0.8% latency improvement across all models.

5.1.1 LAP-NAS Runtime Comparison

One principal feature of our method is its relative speed compared to other NAS methods. Our task-specific compression method fine-tunes the model for a total of 14 epochs, with a NAS step which takes on the order of 10 seconds to complete. A comparable hardware-aware NAS method, HAT [43], optimizes the Transformer architecture [45] for translation tasks. This paradigm does not make use of a pretrained model, which makes it hard to directly compare the training time, but we can measure their training time relative to training the model from scratch without their NAS approach. They train an uninitialized supernet architecture on a translation task for 40-50K steps. They then perform a genetic

Table 5.1: LAP-NAS and Iterative Pruning results for accuracy and lookup table predicted latency on selected GLUE tasks. $\Delta\%$ shows average improvement of LAP-NAS over Iterative Pruning (IP).

DistilBERT-base														
Bits	Method	**CoLA**		**SST-2**		**MRPC**		**QNLI**		**QQP**		**Avg**		**Avg Δ %**
		Lat	Mcc	Lat	Acc	Lat	Acc	Lat	AccF1	Lat	Acc	Lat	Acc	Lat
16	IP	8.1	48.3	8.02	89.8	8.1	82.6	8.43	85.9	11.7	**86.1**	8.3	79.0	
	LAPNAS	**6.96**	**50.7**	**7.00**	**90.1**	**6.83**	**83.7**	**6.48**	**86.6**	**11.3**	85.8	**6.8**	**80.0**	18.5
32	IP	**11.32**	**48.3**	11.62	**89.8**	11.71	82.6	11.48	**85.8**	11.7	**86.1**	11.6	**79.2**	
	LAPNAS	11.36	45.0	**11.45**	89.3	**11.29**	**82.8**	**11.25**	85.0	**11.3**	85.8	**11.3**	78.1	2.3
BERT-base														
16	IP	15.50	52.4	16.3	91.3	16.47	**87.8**	17.08	**90.2**	15.86	88.9	16.2	**82.1**	
	LAPNAS	**14**	**52.9**	**14.06**	**91.4**	**14.02**	87.6	**13.84**	89.7	**13.87**	**89.0**	**14.0**	**82.1**	14.1
32	IP	23.22	52.4	23.38	91.3	23.23	**87.8**	23.26	**90.2**	23.4	88.9	23.3	**82.1**	
	LAPNAS	**22.75**	**53.0**	**22.72**	**91.6**	**22.76**	87.4	**22.83**	88.6	**22.91**	**89.0**	**22.8**	81.9	2.2

Table 5.2: LAP-NAS and Iterative Pruning on-device latency on selected GLUE tasks for models from Table 5.1. $\Delta\%$ shows average improvement of LAP-NAS over Iterative Pruning (IP).

DistilBERT-base								
Bits	Method	**CoLA**	**SST-2**	**MRPC**	**QNLI**	**QQP**	**Avg**	**Avg Δ %**
16	IP	8.23	8.43	8.26	8.19	8.79	8.38	
	LAPNAS	**8.15**	**8.38**	**8.24**	**8.09**	**8.06**	**8.18**	2.3
32	IP	10	10.31	**9.82**	**9.77**	**9.97**	**9.97**	
	LAPNAS	**9.99**	**10.2**	9.97	9.85	10.07	10.02	-0.4
BERT-base								
16	IP	**17.53**	**17.41**	19.04	17.86	17.87	17.942	
	LAPNAS	17.56	17.61	**17.76**	**17.77**	**17.78**	**17.70**	1.4
32	IP	19.96	**19.75**	20.27	19.93	**20.1**	20.00	
	LAPNAS	**19.87**	19.88	**20.24**	**19.7**	20.27	**19.99**	0.0

search to choose an architecture for a specific device. Finally, they fine tune the chosen network to maximize performance. Their total training time is 0.9-2x the training time of the Transformer model, depending on the translation task.

Applying the HAT method to the GLUE tasks would require pretraining a supernet on the masked language modeling task, then fine tuning the chosen architecture on each downstream GLUE task. Training the BERT model following the original authors' method

Table 5.3: Average GLUE performance and on-device latency for each method under test. LA signifies method using latency-aware iterative pruning.

DistilBERT-base					
Bits	Method	**Avg**		**% Improvement**	
		Lat	Acc	Lat	Acc
16	IP	8.38	79.0	0.0	0.0
	LAPNAS	8.18	80.0	**2.3**	**1.3**
	IP-LA	8.62	78.0	-2.9	-1.2
	LAPNAS-LA	8.63	78.4	-3.0	-0.7
32	IP	9.97	79.2	**0.0**	0.0
	LAPNAS	10.02	78.1	-0.4	-1.3
	IP-LA	10.24	79.3	-2.6	**0.2**
	LAPNAS-LA	10.08	78.5	-1.1	-0.9
BERT-base					
16	IP	17.94	82.1	0.0	**0.0**
	LAPNAS	17.70	82.1	1.4	**0.0**
	IP-LA	17.44	81.6	**2.8**	-0.6
	LAPNAS-LA	17.45	80.9	**2.8**	-1.5
32	IP	20.00	82.1	0.0	**0.0**
	LAPNAS	19.99	81.9	0.0	-0.2
	IP-LA	20.37	81.8	-1.8	-0.4
	LAPNAS-LA	19.89	81.7	**1.0**	-1.0

Table 5.4: Training time in GPU hours of LAP-NAS for each of the selected GLUE Tasks on Nvidia V100 GPUs. Models were trained with distributed training across two GPUs. HAT numbers predicted based on reported results their reported results and the increased complexity of BERT training [1].

	GPU hrs per task						
Method	Model	CoLA	SST-2	MRPC	QNLI	QQP	total
LAP-NAS	BERT	.4	2.11	.2	3.5	11.1	**17.3**
HAT	BERT	n/a	n/a	n/a	n/a	n/a	5000

requires 5000 GPU hours. Therefore, we expect the supernet training time for pretraining of BERT to be at least 5000 hours. Table 5.4 shows a comparison of the runtime for our method compared to the predicted runtime for HAT.

5.1.2 Latency-Aware Pruning

LAP-NAS makes no use of the latency information during the initial pruning phase, but integrating this information into the training phase could improve performance. To test this uncertainty, we train using our latency-aware pruning method (see Chapter 3.3.2) with and without NAS to see if a faster model can be pruned at the same sparsity.

Table 5.5 presents the latency-aware pruned and latency-aware pruned + NAS results. We compare these models to the baseline iterative pruning models in the rightmost column. In addition, we compare on-device measured latency in Table 5.3. We observe that the latency-aware pruning method provides an even larger latency improvement than the latency-unaware LAP-NAS method. For FP16 inference the LUT latency is improved by 27% over the iterative pruning baseline with DistilBERT-base and 25% with BERT-base. Interestingly, the latency-aware pruning method appears to reach these numbers even without any architecture search step. However, the FP32 results are largely the same with and without latency-aware pruning.

Unfortunately, the optimized models do not provide a speedup on par with the predictions of the lookup table. Instead we see inconsistent latency improvements and degradation near 0 across all models under test.

Table 5.5: Pruned model results selected GLUE tasks with latency-aware pruning instead of the baseline iterative pruning approach. Bold percentages emphasize an improvement over the baseline.

DistilBERT-base															
Bits	Method	**CoLA**		**SST-2**		**MRPC**		**QNLI**		**QQP**		**Avg**		**% Imp.**	
		Lat	Mcc	Lat	Acc	Lat	Acc	Lat	AccF1	Lat	Acc	Lat	Acc	Lat	Acc
16	IP	6.03	42.2	6.01	88.9	6.02	86.2	6.00	85.8	6.06	87.1	6.02	78.0	**27**	-1
	LAPNAS	6.03	46.2	6.03	88.2	6.04	85.1	6.01	85.3	6.07	87.5	6.03	78.4	**27**	-1
32	IP	11.42	48.6	11.53	89.9	11.61	84.7	11.37	86.2	11.38	87.3	11.46	79.3	**1**	0
	LAPNAS	11.32	46.1	11.34	89.8	11.24	83.2	11.30	86.1	11.33	87.2	11.30	78.5	**2**	-1
BERT-base															
16	IP	12.07	50.9	12.08	91.1	12.06	87.6	12.07	89.8	12.08	88.7	12.07	81.6	**26**	-1
	LAPNAS	12.10	52.3	12.15	89.1	12.14	85.2	12.14	89.4	12.10	88.5	12.13	80.9	**25**	-1
32	IP	23.04	52.5	22.85	91.5	23.13	85.7	22.94	90.4	23.19	89.1	23.03	81.8	**1**	0
	LAPNAS	22.59	51.9	22.61	91.5	22.63	86.9	22.71	89.3	22.73	88.8	22.65	81.7	**3**	-1

5.2 Latency Lookup Table Evaluation

Our method for optimizing model architecture from the latency lookup table (LUT) relies on the assumption that the latency of each pruned layer is independent. To evaluate this assumption, we initialize a model with random layer dimensions and prune from a single randomly selected BERT layer. If our layer-wise latency assumption is correct, we expect to see an equal change in latency predicted from the lookup table and measured from the full model after pruning. The results in Figure 5.1 show that the assumption holds some value, but is not universally valid. We report a mean squared percent error between the predicted and measured latency change of 64%, which is not a very compelling result. The Spearman rank correlation coefficient of the changes in latency is 0.50, showing that if we were to rank the choices of pruning based on the lookup table, our ranking would still correlate positively with the measured ranking of those pruning choices. LAP-NAS aims to compare pruned model dimensions according to their relative latencies, so if we can still choose the best architecture with the LUT then we can accept a relatively high error in the magnitude of the latency improvement.

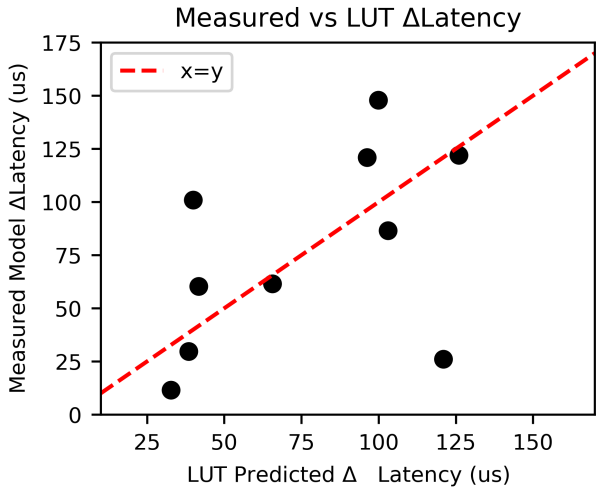

Figure 5.1: Expected vs measured latency change when randomly pruning 10 full-precision DistilBERT models (FP32) on Xavier NX.

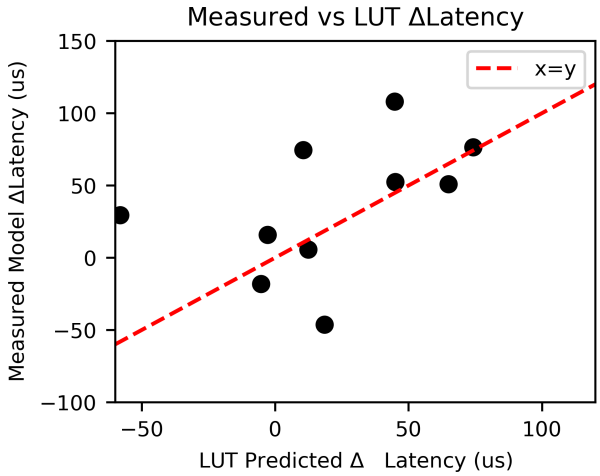

Figure 5.2: Expected vs measured latency change when randomly pruning 10 full-precision DistilBERT models (FP16) on Xavier NX.

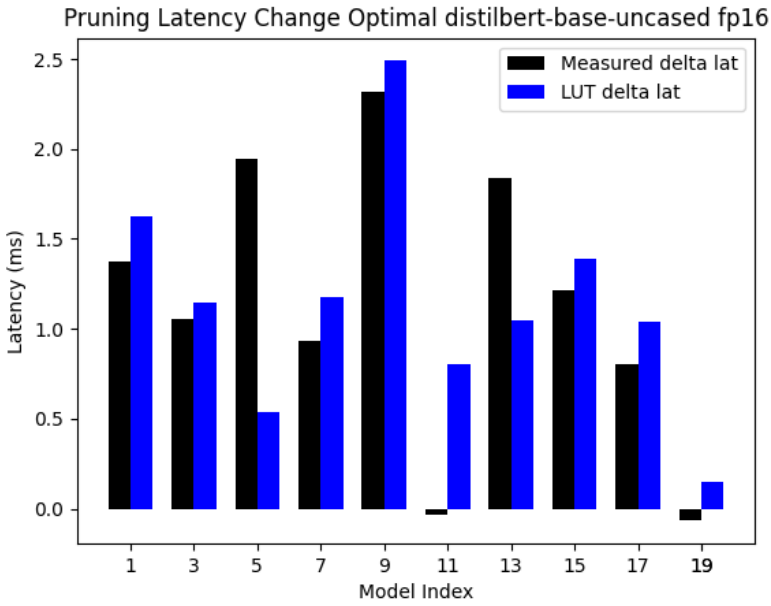

Figure 5.3: Change in latency after optimizing the widths of a randomly pruned network. Pairs of bars which are closer together show accuracy in the latency LUT.

We perform further experiments to observe whether the latency improvement from optimizing widths according to the layer-wise LUT corresponds to a latency improvement when measuring latency of the entire model. In this experiment, we randomly initialize 10 models, and apply LAP-NAS with a goal of minimizing latency while maintaining the number of parameters in the network. We plot the change in latency after NAS predicted by the LUT and the observed change in Figure 5.3. We also report a spearman rank correlation coefficient of 0.87 for the 20 measured models, which shows that there is high correlation between the ranking of models by the lookup table and the actual latency measurement.

5.3 Survey of Inference Platforms

While we only report LAP-NAS results on only the Nvidia Xavier NX with two software configurations, we have also measured BERT layer latencies on several other configurations which we present to provide insight into how latency scales with layer dimensions

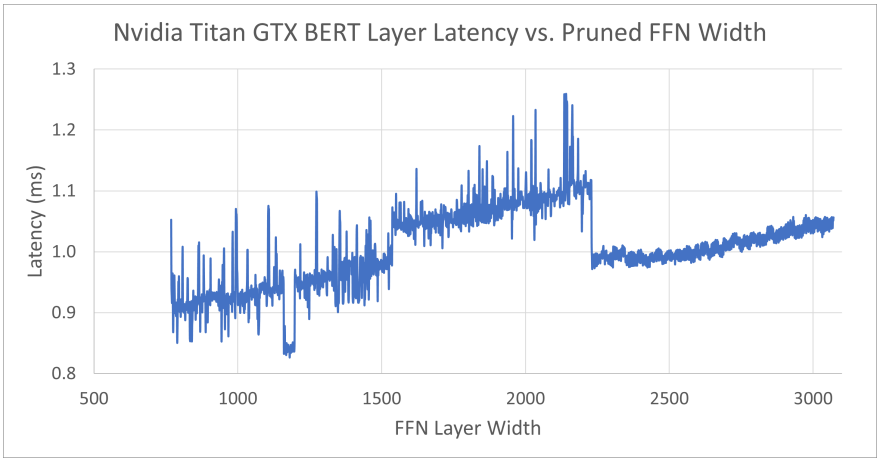

Figure 5.4: BERT layer latency vs. FFN widths on Nvidia GTX Titan GPU.

on different platforms. We specifically report layer latencies on an Intel Xeon CPU (Figure 5.4) and Nvidia GTX Titan GPU (Figure 5.5).

In general, we observe linear scaling between layer width and latency for the FFN layer. For the head layers, as shown in Figure 3.5, the relationship is not a linear one in either precision. While one would expect nonlinear latency scaling to offer a good use case for latency-aware NAS, the head latencies vary so little that even an optimally pruned layer does not achieve a large latency improvement relative to the improvement from removing FFN neurons.

On both the Intel Xeon CPU and the Nvidia GTX Titan GPU, we observe a roughly linear scaling of latency and FFN layer width. These hardware platforms do not offer a good use case for latency-aware NAS via pruning, because sparsity serves as a good proxy metric for latency. In these cases, iterative pruning is sufficient to reach the best expected latency gains. While previous works [39] [64] observed steep staircase patterns in their latency lookup tables when pruning CNNs, we did not observe the same magnitude of steps with BERT models. Our attempt to replicate the results of [39] were unsuccessful despite using the same hardware and CNN architecture as well. This observation supports our choice to use a search-based pruning method which is not tailored to a specific latency lookup table pattern.

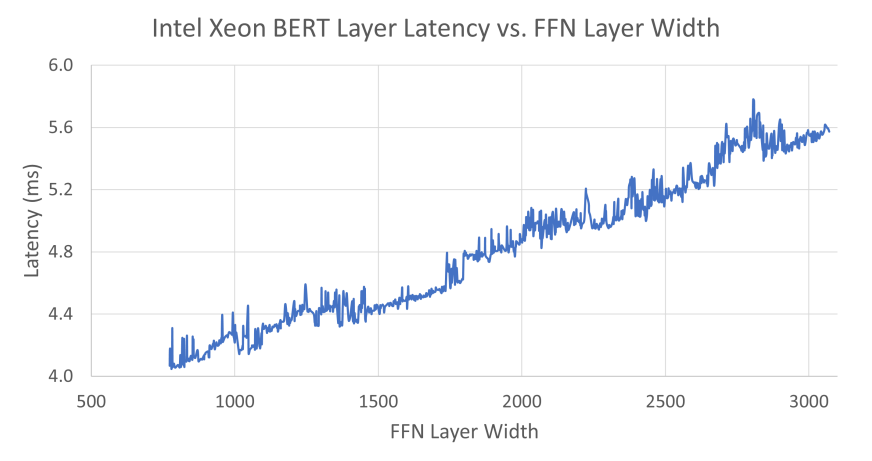

Figure 5.5: BERT layer latency vs. FFN widths on Intel E5-2683 v4 Broadwell @ 2.1Ghz with 16 threads in use.

5.4 Discussion

The reported results show somewhat disappointing performance when latency is measured on the target device. In order to understand why such a gap exists between the latency lookup table values and the measured latencies, we analyze our results another way. The quality of the LAP-NAS compression scheme can also be assessed with manual inspection of layer widths before and after pruning. We refer the reader to Figure 5.6, which shows how optimized widths align with the latency lookup table used for optimization. This analysis provides a sanity check of the neural architecture search method alone by eliminating error from the latency measurements.

Figure 5.6 plots the number of active FFN neurons before the NAS optimization (Green) and after (Blue). Given the LAP-NAS goal of minimizing latency and maintaining accuracy, we expect optimized widths to converge where there is an optimal tradeoff between model size and latency. The optimal tradeoffs occur after pruning across the steep drops in latency seen at widths of ~ 2300, ~ 2700. At these widths, further pruning of a layer returns a modest latency improvement while unpruning the layer results in a sharp increase in latency.

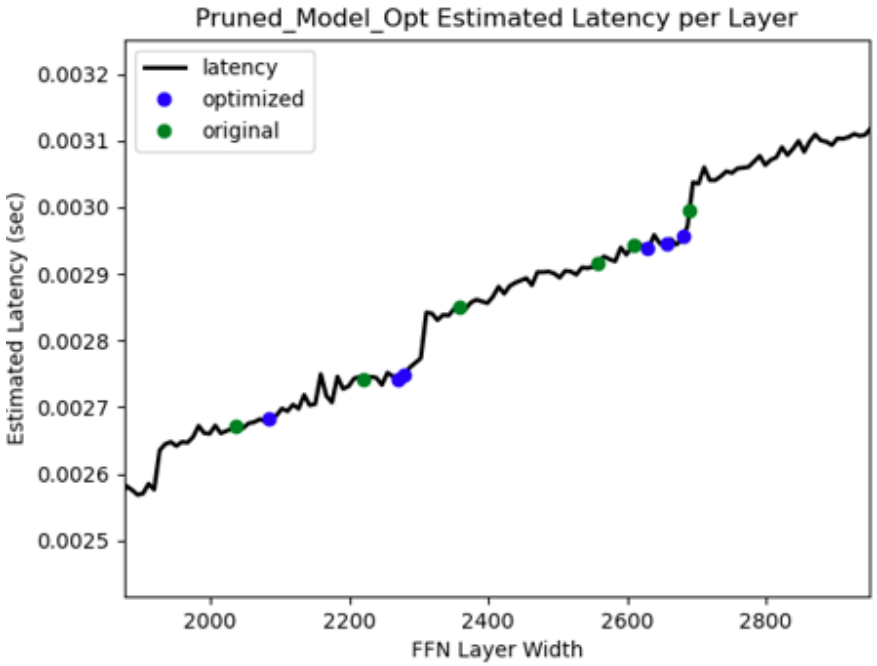

Figure 5.6: Model widths before and after evolutionary architecture search.

We observe that the LAP-NAS optimized widths cluster around the optimal widths. The two unoptimized layer widths that sat close to a steep drop in latency were pruned to the bottom of the drop, as we would expect to see. This qualitative result supports the numbers in Figure 5.1, which show that the LAP-NAS algorithm returns considerable LUT latency improvements over the baseline pruning method. However, when we measure the latency of the pruned model to verify the improvements predicted by the lookup table, the benefit of the NAS step appear quite small. Therefore, we conclude that LAP-NAS is an effective optimization method, but further work is needed in developing accurate latency modeling methods to close the gap between expected and measured performance. This flaw can also be framed as an issue with the search method, since it requires a fast latency model to complete its search. A more efficient NAS method could potentially have avoided the use of the simplistic lookup table latency model.

Chapter 6

Conclusions

This work presents Latency-Aware Pruned Neural Architecture Search (LAP-NAS), a latency-aware DNN compression method for BERT-based models. Our methodology demonstrates how structured pruning and NAS can be applied in sequential fashion to optimize for accuracy early in the compression process and latency later on when it is most useful. The experimental results demonstrate that LAP-NAS can effectively optimize a model given two simple latency lookup tables in a relatively short amount of time. However, the latency lookup table itself presents a significant bottleneck to the realized performance on a target device.

LAP-NAS represents a novel approach to a unique problem in the field of hardware-aware model compression. To the best of our knowledge, there are no previous works which perform NAS to prune a model using a combination of latency metrics and pruning metrics. Our choice to combine these metrics was motivated by the observation that latency and model dimensions do not always have a linear relationship, so pruning to a sparsity may not result in an optimal trade-off between latency and accuracy. In situations where this observation is true, our algorithm obtains a latency improvement of up to 27% without a loss of accuracy. However, in our experiments we were not able to confirm these advantageous latency patterns and were left with latency improvements of 1-2%, making the latency-aware aspect of our pruning method less valuable. We also observed

that a small search space around a pruned model is sufficient to enable latency improvements via pruning, which allowed our algorithm to be compatible with a simple and fast iterative pruning algorithm. Our NAS-free approach, which integrates latency-awareness into the pruning process, is even faster and provides a similar level of performance without additional hyperparameters.

The speed of our compression algorithm is also notable in comparison to previous works. We enable fast latency optimization by relying on simple lookup tables and importance metrics which add negligible run time to existing iterative pruning methods. Our method optimized our selected GLUE tasks in 17 hours, while competing latency-aware methods would have taken thousands of hours. Making compression methods lightweight makes them far more accessible to researchers hoping to apply them in their own work.

The latency modeling aspect of this work remains as the main challenge to improving the latency minimization capabilities of LAP-NAS. When predicting the difference in latency between two models, the latency model offered a relatively poor 64% mean squared error. In addition, the ability of the latency model to accurately rank networks by latency did not fully meet expectations with a Spearman rank correlation of 0.5. The precision and accuracy of the latency prediction method contributed to the large difference in expected and measured latency improvement of LAP-NAS.

This work opens several avenues of future research. The principal challenge is to enable better improvements in on-device latency. One avenue is to improve layer-wise latency measurement techniques to make the layer-wise model more accurate in predicting pruned model latency. Another would be to modify LAP-NAS to consider a much smaller design space, so a more accurate latency measurement method could be applied without prohibitive cost. In the more general area of latency-aware DNN optimization, LAP-NAS could be modified to prune smaller weight structures, such as blocks. It could also adopt a different compression method entirely by applying decomposition or mixed precision quantization to the model layers.

In conclusion, this work proposes a novel compression method with promising experimental results and clear areas of concern for future work.

Bibliography

[1] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: *CoRR* abs/1906.02243 (2019). arXiv: 1906.02243. URL: http://arxiv.org/abs/1906.02243.

[2] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. "How to Evaluate Deep Neural Network Processors: TOPS/W (Alone) Considered Harmful". In: *IEEE Solid-State Circuits Magazine* 12.3 (2020), pp. 28–41.

[3] Jack Turner, José Cano, Valentim Radu, Elliot J. Crowley, Michaep O'Boyle, and Amos Storkey. "Characterising Across-Stack Optimisations for Deep Convolutional Neural Networks". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 2018, pp. 101–110. DOI: 10.1109/IISWC.2018.8573503.

[4] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. *Once-for-All: Train One Network and Specialize it for Efficient Deployment*. 2019. arXiv: 1908.09791 [cs.LG].

[5] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. *MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices*. 2020. arXiv: 2004.02984 [cs.CL].

[6] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. "Single-Path NAS: Designing Hardware-Efficient ConvNets in less than 4 Hours". In: *CoRR* abs/1904.02877 (2019). arXiv: 1904.02877. URL: http://arxiv.org/abs/1904.02877.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[8] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. "Structured Pruning of Deep Convolutional Neural Networks". In: *CoRR* abs/1512.08571 (2015). arXiv: 1512.08571. URL: http://arxiv.org/abs/1512.08571.

[9] Song Han, Jeff Pool, John Tran, and William J. Dally. "Learning both Weights and Connections for Efficient Neural Networks". In: *CoRR* abs/1506.02626 (2015). arXiv: 1506.02626. URL: http://arxiv.org/abs/1506.02626.

[10] Jonathan Frankle and Michael Carbin. "The Lottery Ticket Hypothesis: Training Pruned Neural Networks". In: *CoRR* abs/1803.03635 (2018). arXiv: 1803.03635. URL: http://arxiv.org/abs/1803.03635.

[11] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. *Pruning Convolutional Neural Networks for Resource Efficient Inference*. 2017. arXiv: 1611.06440 [cs.LG].

[12] Victor Sanh, Thomas Wolf, and Alexander M. Rush. *Movement Pruning: Adaptive Sparsity by Fine-Tuning*. 2020. arXiv: 2005.07683 [cs.CL].

[13] Christos Louizos, Max Welling, and Diederik P. Kingma. *Learning Sparse Neural Networks through L_0 Regularization*. 2018. arXiv: 1712.01312 [stat.ML].

[14] Mitchell A. Gordon, Kevin Duh, and Nicholas Andrews. *Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning*. 2020. arXiv: 2002.08307 [cs.CL].

[15] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. "Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned". In: *CoRR* abs/1905.09418 (2019). arXiv: 1905.09418. URL: http://arxiv.org/abs/1905.09418.

[16] Paul Michel, Omer Levy, and Graham Neubig. "Are Sixteen Heads Really Better than One?" In: *CoRR* abs/1905.10650 (2019). arXiv: 1905.10650. URL: http://arxiv.org/abs/1905.10650.

[17] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. *Structured Pruning of Large Language Models*. 2019. arXiv: 1910.04732 [cs.CL].

[18] Angela Fan, Edouard Grave, and Armand Joulin. *Reducing Transformer Depth on Demand with Structured Dropout*. 2019. arXiv: 1909.11556 [cs.LG].

[19] J. S. McCarley, Rishav Chakravarti, and Avirup Sil. *Structured Pruning of a BERT-based Question Answering Model*. 2020. arXiv: 1910.06360 [cs.CL].

[20] Ashish Khetan and Zohar Karnin. *schuBERT: Optimizing Elements of BERT*. 2020. arXiv: 2005.06628 [cs.CL].

[21] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "DARTS: Differentiable Architecture Search". In: *CoRR* abs/1806.09055 (2018). arXiv: 1806.09055. URL: http://arxiv.org/abs/1806.09055.

[22] Barret Zoph and Quoc V. Le. "Neural Architecture Search with Reinforcement Learning". In: *CoRR* abs/1611.01578 (2016). arXiv: 1611.01578. URL: http://arxiv.org/abs/1611.01578.

[23] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. *Efficient Neural Architecture Search via Parameter Sharing*. 2018. arXiv: 1802.03268 [cs.LG].

[24] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. "Rethinking the Value of Network Pruning". In: *CoRR* abs/1810.05270 (2018). arXiv: 1810.05270. URL: http://arxiv.org/abs/1810.05270.

[25] Trevor Gale, Erich Elsen, and Sara Hooker. "The State of Sparsity in Deep Neural Networks". In: *CoRR* abs/1902.09574 (2019). arXiv: 1902.09574. URL: http://arxiv.org/abs/1902.09574.

[26] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. *Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT*. 2019. arXiv: 1909.05840 [cs.CL].

[27] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. *Q8BERT: Quantized 8Bit BERT*. 2019. arXiv: 1910.06188 [cs.CL].

[28] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2020. arXiv: 1909.11942 [cs.CL].

[29] Forrest N. Iandola, Albert E. Shaw, Ravi Krishna, and Kurt W. Keutzer. *SqueezeBERT: What can computer vision teach NLP about efficient neural networks?* 2020. arXiv: 2006.11316 [cs.CL].

[30] Zhanghao Wu, Zhijian Liu, Ji Lin, Yujun Lin, and Song Han. *Lite Transformer with Long-Short Range Attention*. 2020. arXiv: 2004.11886 [cs.CL].

[31] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, Peter Vajda, Matt Uyttendaele, and Niraj K. Jha. *ChamNet: Towards Efficient Network Design through Platform-Aware Model Adaptation*. 2018. arXiv: 1812.08934 [cs.CV].

[32] Y. N. Wu, J. S. Emer, and V. Sze. "Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–8.

[33] Hang Qi, E. R. Sparks, and Ameet Talwalkar. "Paleo: A Performance Model for Deep Neural Networks". In: *ICLR*. 2017.

[34] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. *NeuralPower: Predict and Deploy Energy-Efficient Convolutional Neural Networks*. 2017. arXiv: 1710.05420 [cs.LG].

[35] Han Cai, Ligeng Zhu, and Song Han. "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware". In: *CoRR* abs/1812.00332 (2018). arXiv: 1812.00332. URL: http://arxiv.org/abs/1812.00332.

[36] Dimitrios Stamoulis, Ermao Cai, Da-Cheng Juan, and Diana Marculescu. "HyperPower: Power- and Memory-Constrained Hyper-Parameter Optimization for Neural Networks". In: *CoRR* abs/1712.02446 (2017). arXiv: 1712.02446. URL: http://arxiv.org/abs/1712.02446.

[37] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. "Scalpel: Customizing DNN pruning to the underlying hardware parallelism". In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, pp. 548–560.

[38] Jack Turner, Elliot Crowley, Valentin Radu, José Cano, Amos Storkey, and Michael O'Boyle. "Distilling with Performance Enhanced Students". In: *ArXiv* abs/1810.10460 (2018).

[39] Valentin Radu, Kuba Kaszyk, Yuan Wen, Jack Turner, José Cano, Elliot J. Crowley, Bjorn Franke, Amos Storkey, and Michael O'Boyle. "Performance Aware Convolutional Neural Network Channel Pruning for Embedded GPUs". In: *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 2019, pp. 24–34.

[40] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. "NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018.

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

[42] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[43] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. *HAT: Hardware-Aware Transformers for Efficient Natural Language Processing*. 2020. arXiv: 2005.14187 [cs.CL].

[44] Xiaohu Tang, Shihao Han, Li Lyna Zhang, Ting Cao, and Yunxin Liu. "To Bridge Neural Network Design and Real-World Performance: A Behaviour Study for Neural Networks". In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 21–37. URL: https://proceedings.mlsys.org/paper/2021/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[46] Sachin Mehta, Marjan Ghazvininejad, Srinivasan Iyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. "DeLighT: Very Deep and Light-weight Transformer". In: *CoRR* abs/2008.00623 (2020). arXiv: 2008.00623. URL: https://arxiv.org/abs/2008.00623.

[47] Jay Alammar. *The Illustrated Transformer*. https://jalammar.github.io/illustrated-transformer, June 2018.

[48] Trevor Gale, Erich Elsen, and Sara Hooker. "The State of Sparsity in Deep Neural Networks". In: *CoRR* abs/1902.09574 (2019). arXiv: 1902.09574. URL: http://arxiv.org/abs/1902.09574.

[49] Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Deming Chen, Marianne Winslett, Hassan Sajjad, and Preslav Nakov. *Compressing Large-Scale Transformer-Based Models: A Case Study on BERT*. 2020. arXiv: 2002.11985 [cs.LG].

[50] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. *DynaBERT: Dynamic BERT with Adaptive Width and Depth*. 2020. arXiv: 2004.04037 [cs.CL].

[51] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1510.00149.

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

[53] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. "Transformers: State-of-the-Art Natural Language Processing". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. URL: https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[54] Kim F. Man, Kit Sang Tang, and Sam Kwong. "Genetic algorithms: concepts and applications [in engineering design]". In: *IEEE Transactions on Industrial Electronics* 43.5 (1996), pp. 519–534. DOI: 10.1109/41.538609.

[55] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2020. arXiv: 1910.01108 [cs.CL].

[56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. "HuggingFace's Transformers: State-of-the-art Natural Language Process-

ing". In: *CoRR* abs/1910.03771 (2019). eprint: 1910.03771. URL: http://arxiv.org/abs/1910.03771.

[57] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[58] Alex Renda, Jonathan Frankle, and Michael Carbin. "Comparing Rewinding and Fine-tuning in Neural Network Pruning". In: *CoRR* abs/2003.02389 (2020). arXiv: 2003.02389. URL: https://arxiv.org/abs/2003.02389.

[59] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding*. 2019. arXiv: 1804.07461 [cs.CL].

[60] Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. *Neural Network Acceptability Judgments*. 2019. arXiv: 1805.12471 [cs.CL].

[61] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1631–1642. URL: https://aclanthology.org/D13-1170.

[62] Kaggle.com. *Quora Question Pairs — Kaggle*. Online, Apr. 2017.

[63] William B. Dolan and Chris Brockett. "Automatically Constructing a Corpus of Sentential Paraphrases". In: *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*. 2005. URL: https://aclanthology.org/I05-5002.

[64] Fuxun Yu, Zirui Xu, Tong Shen, Dimitrios Stamoulis, Longfei Shangguan, Di Wang, Rishi Madhok, Chunshui Zhao, Xin Li, Nikolaos Karianakis, Dimitrios Lymberopoulos, Ang Li, Chenchen Liu, Yiran Chen, and Xiang Chen. "Towards Latency-aware DNN Optimization with GPU Runtime Analysis and Tail Effect Elimination". In: *CoRR* abs/2011.03897 (2020). arXiv: 2011.03897. URL: https://arxiv.org/abs/2011.03897.