Learned Token Pruning for Efficient Transformer Inference

Sehoon Kim
Sheng Shen
David Thorsley
Amir Gholami
Woosuk Kwon
Joseph Hassoun
Kurt Keutzer

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-119
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-119.html

May 11, 2023

Learned Token Pruning for Efficient Transformer Inference

by Sehoon Kim

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Kurt Keutzer
Research Advisor

05 / 09 / 2023

(Date)

* * * * * * *

Professor Sophia Shao
Second Reader

5/10/2023

(Date)

Learned Token Pruning for Efficient Transformer Inference

by

Sehoon Kim

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kurt Keutzer, Chair
Professor Sophia Shao

Spring 2023

Abstract

Learned Token Pruning for Efficient Transformer Inference

by

Sehoon Kim

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Kurt Keutzer, Chair

Efficient deployment of transformer models in practice is challenging due to their inference cost including memory footprint, latency, and power consumption, which scales quadratically with input sequence length. To address this, we present a novel token reduction method dubbed Learned Token Pruning (LTP) which adaptively removes unimportant tokens as an input sequence passes through transformer layers. In particular, LTP prunes tokens with an attention score below a threshold, whose value is learned for each layer during training. Our threshold-based method allows the length of the pruned sequence to vary adaptively based on the input sequence, and avoids algorithmically expensive operations such as top-k token selection. We extensively test the performance of LTP on GLUE and SQuAD tasks and show that our method outperforms the prior state-of-the-art token pruning methods by up to $\sim$2.5% higher accuracy with the same amount of FLOPs. In particular, LTP achieves up to 2.1$\times$ FLOPs reduction with less than 1% accuracy drop, which results in up to 1.9$\times$ and 2.0$\times$ throughput improvement on Intel Haswell CPUs and NVIDIA V100 GPUs. Furthermore, we demonstrate that LTP is more robust than prior methods to variations in input sequence lengths.

Contents

Chapter 1

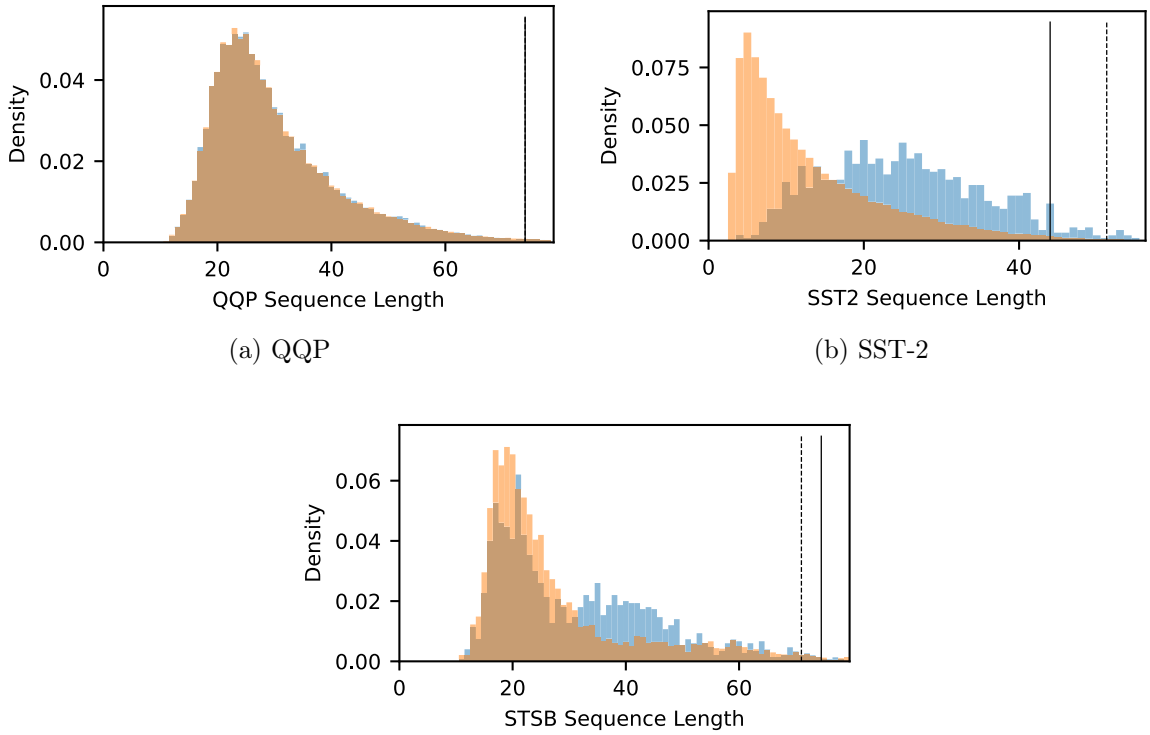
Introduction

Transformer-based deep neural network architectures [45], such as BERT [7] and RoBERTa [28], achieve state-of-the-art results in Natural Language Processing (NLP) tasks such as sentence classification and question answering. However, efficiently deploying these models is increasingly challenging due to their large size, the need for real-time inference, and the limited energy, compute, and memory resources available. The heart of a transformer layer is the multi-head self-attention mechanism, where each token in the input sequence attends to every other token to compute a new representation of the sequence. Because all tokens attend to each others, the computation complexity of the self-attention mechanism is quadratic with respect to the input sequence length; thus the ability to apply transformer models to long input sequences becomes limited.

Pruning is a popular technique to reduce the size of neural networks and the amount of computation required. *Unstructured* pruning allows arbitrary patterns of sparsification for parameters and feature maps and can, in theory, produce significant computational savings while preserving accuracy. However, commodity DNN accelerators cannot efficiently exploit unstructured sparsity patterns. Thus, *structured* pruning methods are typically preferred in practice due to their relative ease of deployment to hardware.

Multi-head self-attention provides several possibilities for structured pruning; for example, head pruning [30, 46] decreases the size of the model by removing unneeded heads in each transformer layer. Another orthogonal approach that we consider in this thesis is *token pruning*, which reduces computation by progressively removing unimportant tokens in the sequence during inference. For NLP tasks such as sentence classification, token pruning is an attractive approach to consider as it exploits the intuitive observation that not all tokens (i.e., words) in an input sentence are necessarily required to make a successful inference.

There are two main classes of token pruning methods. In the first class, methods like PoWER-BERT [13] and Length-Adaptive Transformer (LAT) [21] search for a single token pruning configuration (i.e., sequence length for each layer) for an entire dataset. In other words, they prune all input sequences to the same length. However, input sequence lengths can vary greatly within tasks and between training and validation sets as in Figure 1.1, and thus applying a single pruning configuration to all input sequences can potentially under-prune



(a) QQP

(b) SST-2

(c) STS-B

Figure 1.1: Distributions of processed input sequence lengths from datasets for representative tasks in the GLUE benchmark: (a) QQP (b) SST-2; (c) STS-B. The training set is in orange and the validation set is in blue. The dashed and solid vertical lines indicate the 99th percentile value for the training and validation sets, respectively.

shorter sequences or over-prune longer sequences.

In the other class, the token pruning method adjusts the configuration based on the input sequence. SpAtten [49] uses a pruning configuration proportional to input sentence length; however, it does not adjust the proportion of pruned tokens based on the content of the input sequence. The recently published TR-BERT [54] uses reinforcement learning (RL) to find a policy network that dynamically reduces the number of tokens based on the length and content of the input sequence; however, it requires additional costly training for convergence of the RL-based method. Additionally, all of these prior methods rely in part on selecting the k most significant tokens during inference or training. This selection can be computationally expensive without the development of specialized hardware, such as the top-k engine introduced in SpAtten [49].

To this end, we propose a learned *threshold*-based token pruning method which adapts to the length and content of individual examples and avoids the use of top-k operations. In

particular, our contributions are as follows:

- We propose Learned Token Pruning (LTP), a threshold-based token pruning method, which only needs a simple threshold operation to detect unimportant tokens. In addition, LTP fully automates the search for optimal pruning configurations by introducing a differentiable soft binarized mask that allows training the correct thresholds for different layers and tasks. (Section 3.3)

- We apply LTP to RoBERTa and evaluate its performance on GLUE and SQuAD tasks. We show LTP achieves up to 2.10× FLOPs reduction with less than 1% accuracy degradation, which results in up to 1.93× and 1.97× throughput improvement on NVIDIA V100 GPU and Intel Haswell CPU, respectively, as compared to the unpruned FP16 baseline. We also show that LTP outperforms SpAtten and LAT in most cases, achieving additional FLOPs reduction for the same drop in accuracy. (Section 4.2 and 4.5)

- We show that LTP is highly robust against sentence length variations. LTP exhibits consistently better accuracy over different sentence length distributions, achieving up to 16.4% accuracy gap from LAT. (Section 4.3)

Chapter 2

Related Work

2.1 Efficient Transformers

Multiple different approaches have been proposed to improve speed and diminish memory footprint of transformers. These can be broadly categorized as follows: (i) efficient architecture design [25, 5, 23, 50, 16, 47, 44, 19, 57, 35]; (ii) knowledge distillation [42, 18, 43, 38, 41]; (iii) quantization [2, 56, 39, 10, 55, 58, 1, 22]; and (iv) pruning. Here, we focus only on pruning and briefly discuss the related work.

2.2 Transformer Pruning

Pruning methods can be categorized into unstructured pruning and structured pruning. For unstructured pruning, the lottery-ticket hypothesis [11] has been explored for transformers in [31, 4]. Recently, [59] leverages pruning as an effective way to fine-tune transformers on downstream tasks. [37] proposes movement pruning, which achieves significant performance improvements versus prior magnitude-based methods by considering the weights modification during fine-tuning. However, it is often quite difficult to efficiently deploy unstructured sparsity on commodity neural accelerators for meaningful speedup.

For this reason, a number of structured pruning methods have been introduced to remove structured sets of parameters. [30, 46] drop attention heads in multi-head attention layers, and [36, 9] prunes entire transformer layers. [51] structurally prunes weight matrices via low-rank factorization, and [20, 27] attempt to jointly prune attention heads and filters of weight matrices. [29, 15] dynamically determines structured pruning ratios during inference. Recent block pruning schemes chunk weight matrices into multiple blocks and prune them based on group Lasso optimization [26], adaptive regularization [53], and movement pruning [24]. All of these methods correspond to *weight pruning*, where model parameters (i.e., weights) are pruned.

Recently, there has been work on pruning input sentences to transformers, rather than model parameters. This is referred to as *token pruning*, where less important tokens are

progressively removed during inference. PoWER-BERT [13], one of the earliest works, proposes to directly learn token pruning configurations. LAT [21] extends this idea by introducing LengthDrop, a procedure in which a model is trained with different token pruning configurations, followed by an evolutionary search. This method has an advantage that the former training procedure need not be repeated for different pruning ratios of the same model. While these methods have shown a large computation reduction on various NLP downstream tasks, they fix a single token pruning configuration for the entire dataset. That is, they prune all input sequences to the same length. However, as shown in Figure 1.1, input sequence lengths vary greatly within a task. As a consequence, fixing a single pruning configuration can under-prune shorter sequences so as to retain sufficient tokens for processing longer sequences or, conversely, over-prune longer sequences to remove sufficient tokens to efficiently process shorter sequences. More importantly, a single pruning configuration lacks robustness against input sequence length variations, where input sentences at inference time are longer than those in the training dataset [32].

In contrast, SpAtten [49] circumvents this issue by assigning a pruning configuration proportional to the input sequence length. While this allows pruning more tokens from longer sequences and fewer tokens from shorter ones, it is not adaptive to individual input sequences as it assigns the same configuration to all sequences with the same length regardless of their contents. In addition, the pruning configurations are determined heuristically and thus can result in a suboptimal solution. Recently, TR-BERT [54] proposes to learn a RL policy network to apply adaptive pruning configurations for each input sequence. However, as noted by the authors, the problem has a large search spaces which can be hard for RL to solve. This issue is mitigated by heuristics involving imitation learning and sampling of action sequences, which significantly increases the cost of training. Importantly, all of the aforementioned token pruning methods depend partially or entirely on top-k operation for selecting the k most important tokens during inference or training. This operation can be costly without specialized hardware support, as discussed in [49]. LTP, on the other hand, is based on a fully learnable threshold-based pruning strategy. Therefore, it is (i) adaptive to both input length and content, (ii) robust to sentence length variations, (iii) computationally efficient, and (iv) easy to deploy.

Chapter 3

Methodology

3.1 Background

BERT [7] consists of multiple transformer encoder layers [45] stacked up together. A basic transformer encoder layer consists of a multi-head attention (MHA) block followed by a point-wise feed-forward (FFN) block, with residual connections around each. Specifically, an MHA consists of N_h independently parameterized heads. An attention head h in layer l is parameterized by $\mathbf{W}_k^{(h,l)}$, $\mathbf{W}_q^{(h,l)}$, $\mathbf{W}_v^{(h,l)} \in \mathbb{R}^{d_h \times d}$, $\mathbf{W}_o^{(h,l)} \in \mathbb{R}^{d \times d_h}$, where d_h is typically set to d/N_h and d is the feature dimension. We drop the superscript l for simplicity in the following formula. The MHA measures the pairwise importance of each token on every other token in the input:

$$\mathrm{MHA}(\mathrm{x}) = \sum_{h=1}^{N_h} \mathrm{Att}_{\mathbf{W}_{k,q,v,o}^{(h)}}(\mathrm{x}), \tag{3.1}$$

where $\mathrm{x} \in \mathbb{R}^{d \times n}$ is the input sequence with the sequence length n, and $\mathrm{Att}_{\mathbf{W}_{k,q,v,o}}$ is:

$$\mathrm{Att}_{\mathbf{W}_{k,q,v,o}}(\mathrm{x}) = \mathbf{W}_o \sum_{i=1}^{n} \mathbf{W}_v \mathrm{x}_i \mathrm{softmax}(\frac{\mathrm{x}^T \mathbf{W}_q^T \mathbf{W}_k \mathrm{x}_i}{\sqrt{d}}), \tag{3.2}$$

$$\mathbf{x}_{\mathrm{MHA}} = \mathrm{LN}\big(\mathrm{Att}_{\mathbf{W}_{k,q,v,o}}(\mathrm{x}) + \mathrm{x}\big), \tag{3.3}$$

where Eq. 3.3 is the residual connection and the follow up LayerNorm (LN). The output of the MHA is then fed into the FFN block which applies two feed-forward layers to this input:

$$\mathrm{FFN}(\mathrm{x}_{\mathrm{MHA}}) = \sigma\big(\mathbf{W}_2(\mathbf{W}_1 \mathrm{x}_{\mathrm{MHA}} + b_1)\big) + b_2, \tag{3.4}$$

$$\mathbf{x}_{\mathrm{out}} = \mathrm{LN}\big(\mathrm{FFN}(\mathrm{x}_{\mathrm{MHA}}) + \mathrm{x}_{\mathrm{MHA}}\big), \tag{3.5}$$

where $\mathbf{W}_1, \mathbf{W}_2, b_1$ and b_2 are the FFN parameters, and σ is the activation function (typically GELU for BERT).

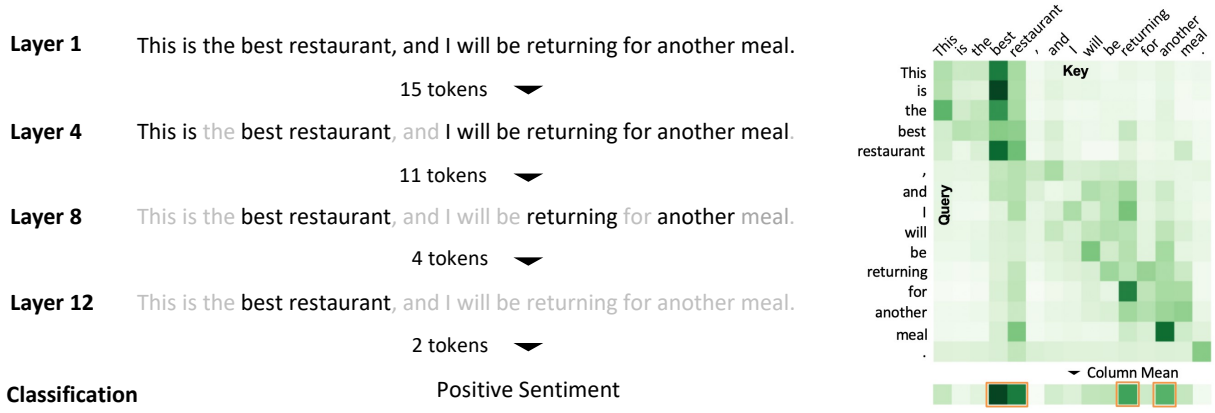


Figure 3.1: (Left) Schematic of token pruning for a sentiment analysis task. Unimportant tokens are pruned as the input sequence passes through the layers. (Right) An example of attention probability in a single head where a more important token receives more attention from other tokens. Thus each token's importance score is computed by taking the average attention probability it receives, which is computed by taking the column mean of the attention probability.

3.2 Threshold Token Pruning

Let us denote the *attention probability* of head h between token x_i and x_j as $\mathbf{A}^{(h,l)}$:

$$\mathbf{A}^{(h,l)}(\mathrm{x}_i, \mathrm{x}_j) = \mathrm{softmax}(\frac{\mathrm{x}^T \mathbf{W}_q^T \mathbf{W}_k \mathrm{x}}{\sqrt{d}})_{(i,j)} \in \mathbb{R}. \tag{3.6}$$

The cost of computational complexity for computing the attention matrix is $\mathcal{O}(d^2 n + n^2 d)$, which quadratically scales with sequence length. As such, the attention operation becomes a bottleneck when applied to long sequences. To address this, we apply *token pruning* which removes unimportant tokens as the input passes through the transformer layers to reduce the sequence length n for later blocks. This is schematically shown in Figure 3.1 (Left).

For token pruning, we must define a metric to determine unimportant tokens. Following [13, 49, 21], we define the *importance score* of token x_i in layer l as:

$$s^{(l)}(\mathrm{x}_i) = \frac{1}{N_h} \frac{1}{n} \sum_{h=1}^{N_h} \sum_{j=1}^{n} \mathbf{A}^{(h,l)}(\mathrm{x}_i, \mathrm{x}_j). \tag{3.7}$$

Intuitively, the attention probability $\mathbf{A}^{(h,l)}(\mathrm{x}_i, \mathrm{x}_j)$ is interpreted as the normalized amount that all the other tokens x_j attend to token x_i. Token x_i is thus considered *important* if it receives more attention from all tokens across all heads, which directly leads us to equation 3.7. The procedure for computing importance scores from attention probabilities is illustrated in Figure 3.1 (Right).

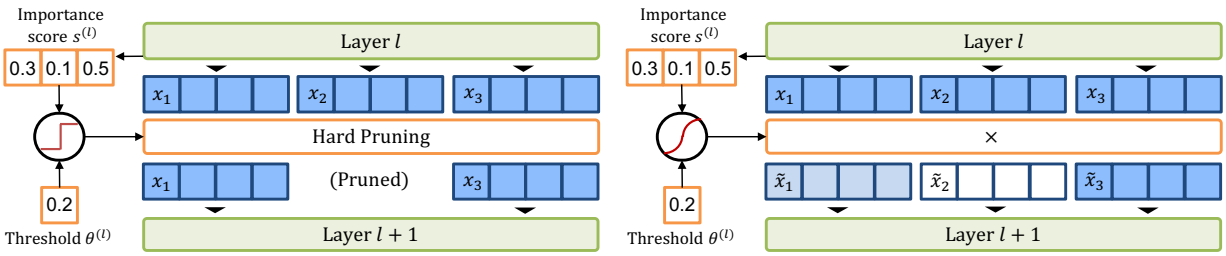


Figure 3.2: Different pruning strategies for threshold-based token pruning methods. (Left) Hard pruning uses a binary hard mask to select tokens to be pruned. (Right) Soft pruning replaces the binary mask with a differentiable soft mask.

In [13, 49, 21], tokens are ranked by importance score and pruned using a top-k selection strategy. Specially, token x_i is pruned at layer l if its important score $s^{(l)}(\mathrm{x}_i)$ is smaller than the k-largest values of the important score from all the tokens. However, finding the k-largest values of the importance score is computationally inefficient without specialized hardware [49]; we provide empirical results showing this in Section 4.7. Instead, we introduce a new *threshold-based* token pruning approach in which a token is pruned only if its importance score is below a threshold denoted by $\theta^{(l)} \in \mathbb{R}$. Specifically, we define a pruning strategy by imposing a binary mask $M^{(l)}(\cdot) : \{1, \ldots, n\} \to \{0, 1\}$ which indicates whether a token should be kept or pruned:

$$M^{(l)}(\mathrm{x}_i) = \begin{cases} 1 & \text{if } s^{(l)}(\mathrm{x}_i) > \theta^{(l)}, \\ 0 & \text{otherwise.} \end{cases} \tag{3.8}$$

Note that this operation only requires a simple comparison operator without any expensive top-k calculation. Once a token is pruned, it is excluded from calculations in all succeeding layers, thereby gradually reducing computation complexity towards the output layers.

3.3 Learnable Threshold for Token Pruning

A key concern with the method above is how to determine the threshold values for each layer. Not only do threshold values change for different layers, they also vary between different tasks. We address this by making the thresholds (i.e., θ in Eq. 3.8) learnable. However, there are several challenges to consider. First, due to the binary nature of M there is no gradient flow for pruned tokens. Second, the M operator is non-differentiable which prevents gradient flow into the thresholds. To address these challenges, we use a *soft* pruning scheme that simulates the original *hard* pruning while still propagating gradients to the thresholds as shown in Figure 3.2.

Algorithm 1 Three-step Training Procedure for Learnable Threshold Token Pruning

Input: $\mathcal{M}$: model finetuned on target downstream task

Step 1: Apply soft mask to $\mathcal{M}$
and train both the thresholds and model parameters ▷ Soft Pruning
Step 2: Binarize the mask and fix the thresholds
Step 3: Finetune the model parameters ▷ Hard Pruning

Soft Pruning Scheme. In the soft pruning scheme, we replace the non-differentiable mask $M^{(l)}$ with a differentiable soft mask using the sigmoid operation σ:

$$\tilde{M}^{(l)}(\mathrm{x}_i) = \sigma\left(\frac{s^{(l)}(\mathrm{x}_i) - \theta^{(l)}}{T}\right), \tag{3.9}$$

where T is temperature, and $\theta^{(l)}$ is the learnable threshold value for layer l. With sufficiently small temperature T, $\tilde{M}^{(l)}(\mathrm{x}_i)$ will closely approximate the hard masking $M^{(l)}(\mathrm{x}_i)$ in Eq. 3.8. In addition, instead of selecting tokens to be pruned or kept based on the hard mask of Eq. 3.8, we multiply the soft mask to the output activation of layer l. That is,

$$\tilde{\mathrm{x}}^{(l)}_{\text{out}} = \tilde{M}^{(l)}(\mathrm{x}^{(l)}) \cdot \mathrm{x}^{(l)}_{\text{out}} \tag{3.10}$$

$$= \tilde{M}^{(l)}(\mathrm{x}^{(l)}) \cdot \mathrm{LN}(\mathrm{FFN}(\mathrm{x}^{(l)}_{\text{MHA}}) + \mathrm{x}^{(l)}_{\text{MHA}}), \tag{3.11}$$

where $\mathrm{x}^{(l)}_{\text{MHA}}$ is the output activation of MHA in layer l. If the importance score of token x_i is below the threshold by a large margin, its layer output activation nears zero and thus it has little impact on the succeeding layer. Also, because the token gets a zero importance score in the succeeding layer, i.e., $s^{(l+1)}(\mathrm{x}_i) = 0$, it is likely to be pruned again. Therefore, the soft pruning scheme is nearly identical in behavior to hard pruning, yet its differentiable form allows for backpropagation and gradient-based optimizations to make θ learnable. After (i) jointly training model parameters and thresholds on downstream tasks with the soft pruning scheme, (ii) we fix the thresholds and binarize the soft mask, and (iii) perform a follow-up fine-tuning of the model parameters. The pseudo-code for this three-step algorithm is given in Algorithm 1. Intuitively, the magnitude of gradient $d\tilde{M}^{(l)}(\mathrm{x}_i)/d\theta^{(l)}$ is maximized when the importance score $s^{(l)}(\mathrm{x}_i)$ is close enough to the threshold $\theta^{(l)}$ and becomes near zero elsewhere. Therefore, the threshold can be trained only based on the tokens that are about to be pruned or retained.

Regularization. It is not possible to learn θ without regularization, as the optimizer generally gets a better loss value if all tokens are present. As such, we add a regularization term to penalize the network if tokens are left unpruned. This is achieved by imposing an L1 loss on the masking operator $\tilde{M}$:

$$\mathcal{L}_{\text{new}} = \mathcal{L} + \lambda\mathcal{L}_{\text{reg}} \text{ where } \mathcal{L}_{\text{reg}} = \frac{1}{L}\sum_{l=1}^{L} ||\tilde{M}^{(l)}(\mathrm{x})||_1. \tag{3.12}$$

Here, $\mathcal{L}$ is the original loss function (e.g., cross-entropy loss), and λ is the regularization parameter. Larger values of λ result in higher pruning ratios. This regularization operator induces an additional gradient to the threshold:

$$\frac{d\mathcal{L}_{\text{reg}}}{d\theta^{(l)}} = \frac{1}{d\theta^{(l)}}||\tilde{M}^{(l)}(\mathrm{x})||_1 = \sum_{i=1}^{n} \frac{d\tilde{M}^{(l)}(\mathrm{x}_i)}{d\theta^{(l)}} \tag{3.13}$$

If there are more tokens near the threshold, then the gradient $d\mathcal{L}_{\text{reg}}/d\theta^{(l)}$ is larger. As a result, the threshold is pushed to a larger value, which prunes more tokens near the threshold boundary.

Chapter 4

Experiments

4.1 Experiment Setup

We implemented LTP on RoBERTa$_{\text{base}}$ [28] using HuggingFace's repo[1] and tested on (English) GLUE tasks [48] and SQuAD 2.0 [33]. For GLUE tasks [48], we use 6 tasks for evaluation including sentence similarity (QQP [17], MRPC [8], STS-B [3]), sentiment classification (SST-2 [40]), textual entailment (RTE [6]) and natural language inference (MNLI [52], QNLI [34]). For evaluating the results, we measure classification accuracy and F1 score for MRPC and QQP, Pearson Correlation and Spearman Correlation for STS-B, and classification accuracy for the remaining tasks on validation sets. For the tasks with multiple metrics (i.e., MRPC, QQP, STS-B), we report their average. For SQuAD 2.0 [33], which is a question and answering task, we measure F1 score for evaluating the results.

As mentioned in Section 3.3, the training procedure of LTP soft pruning followed by hard pruning. For soft pruning, we train both the model parameters and the thresholds on downstream tasks for 1 to 10 epochs, depending on the dataset size. We find it effective to initialize the thresholds with linearly rising values as described in 4.4 with a fixed threshold of the final layer. We search the optimal temperature T in a search space of {1, 2, 5, 10, 20}e-4 and vary λ from 0.001 to 0.4 to control the number of tokens to be pruned (and thus the FLOPs) for all experiments. We then fix the thresholds and perform an additional training with the hard pruning to fine-tune the model parameters only. More detailed hyperparameter settings are listed in Table 4.1 for GLUE and SQuAD 2.0.

We also compare LTP with the current state-of-the-art token pruning methods of SpAtten [49] and LAT [21] following the implementation details in their papers. SpAtten is trained based on the implementation details in the paper: the first three layers retain all tokens and the remaining layers are assigned with linearly decaying token retain ratio until it reaches the final token retain ratio at the last layer. We vary the final token retain ratio from 1.0 to -1.0 (prune all tokens for non-positive retain ratios) to control the FLOPs of SpAtten. For both LTP and SpAtten, we use learning rate of {0.5, 1, 2}e-5, except for the soft pruning stage of

[1] `https://github.com/huggingface/transformers/`

Table 4.1: Detailed hyperparameters for LTP training.

Stage	Hyperparam	GLUE	SQuAD 2.0
Soft pruning	epochs	1 - 10	1
	learning rate	2e-5	2e-5
	T	{1, 2, 5, 10, 20}e-4	{1, 10}e-4
	λ	0.001 - 0.2	0.001 - 0.4
	initial final threshold	0.01	0.003
Hard pruning	epochs	10	5
	lr	{0.5, 1, 2}e-5	{0.5, 1, 2}e-5

LTP where we use 2e-5. We follow the optimizer setting in RoBERTa [28] and use batch size of 64 for all experiments. LAT is trained using the same hyperparameter and optimizer setting in the paper except for the length drop probabilities: for more extensive search on more aggressive pruning configurations, we used 0.25, 0.3, 0.35, and 0.4 for the length drop probability instead of 0.2 in the original setting.

We use PyTorch 1.8 throughout all experiments. For CPU inference speed experiments, we use an Intel Haswell CPU with 3.75GB memory of Google Cloud Platform. For GPU inference speed experiments, we use an AWS p3.2xlarge instance that has a NVIDIA V100 GPU with CUDA 11.1.

An important issue in previous work [13, 21] is that *all* input sequences for a specific task are padded to the nearest power of 2 from the 99th percentile of the sequence lengths, and then the pruned performance is compared with the padded baseline. This results in exaggerated performance gain over the baseline. For instance, in [13], inputs from the SST-2 dataset are padded to 64, while its average sentence length is 26 (cf. Figure 1.1). With this approach, one can achieve roughly 2.5× speedup by just removing padding. As such, we avoid any extra padding of input sequences and all speedups and throughputs we report are compared with the unpadded baselines.

4.2 Performance Evaluation

Table 4.2 lists the accuracy and GFLOPs for LTP. We select a model for each downstream task that achieves the smallest GFLOPs while constraining the accuracy degradation from the baseline ($\text{RoBERTa}_{\text{base}}$) to be at most 1%. Using our method, sequence lengths in each layer can vary across different input sentences. Therefore, we report the averaged GFLOPs of processing all input sentences in the development set. As shown in the table, our method achieves speedup of 1.96× on average and up to 2.10× within 1% accuracy degradation.

Figure 4.1 plots the accuracy of LTP (blue lines) as well as the prior pruning methods (red lines for SpAtten and orange lines for LAT) with different FLOPs on GLUE tasks. LTP consistently outperforms SpAtten for all tasks with up to ~2% higher accuracy under the

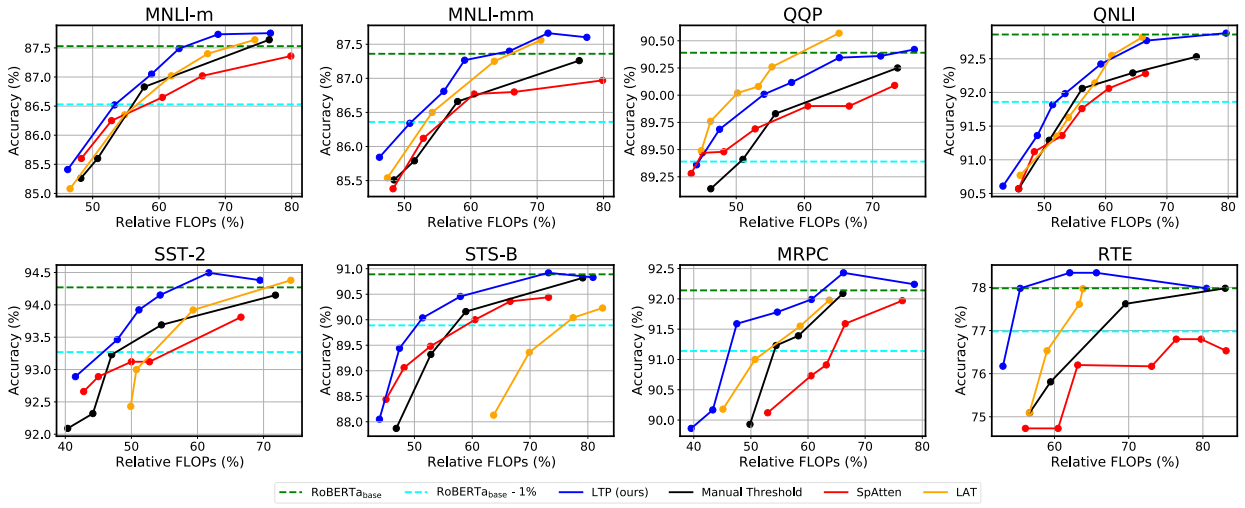


Figure 4.1: Performance of different pruning methods on GLUE tasks for different token pruning methods across different relative FLOPs, i.e., normalized FLOPs with respect to the the baseline model. Manual threshold assigns linearly raising threshold values for each layer instead of learning them. The performance of the baseline model without token pruning (RoBERTa$_{\text{base}}$) and the model with 1% performance drop (RoBERTa$_{\text{base}}$ - 1%) are dotted in horizontal lines for comparison.

Table 4.2: Detailed performance and efficiency comparison of LTP applied to RoBERTa$_{\text{base}}$.

Task	Accuracy Metric		GFLOPs		Speedup
	RoBERTa	LTP	RoBERTa	LTP	LTP
MNLI-m	87.53	86.53	6.83	3.64	1.88×
MNLI-mm	87.36	86.37	7.15	3.63	1.97×
QQP	90.39	89.69	5.31	2.53	2.10×
QNLI	92.86	91.98	8.94	4.77	1.87×
SST-2	94.27	93.46	4.45	2.13	2.09×
STS-B	90.89	90.03	5.53	2.84	1.95×
MRPC	92.14	91.59	9.33	4.44	2.10×
RTE	77.98	77.98	11.38	6.30	1.81×
SQuAD 2.0	83.04	82.25	32.12	16.99	1.89×

same amount of FLOPs. Compared with LAT, LTP outperforms for all tasks except for QQP with up to ~2.5% higher accuracy for the same target FLOPs. For QQP alone, LTP achieves at most ~0.2% lower accuracy than LTP.

An important observation is that for SST-2 and STS-B where LTP (ours) outperforms

Table 4.3: Quantiles (Q1/Q2/Q3) and KL divergence of sentence lengths of training and evaluation datasets for GLUE tasks. KL divergence are measured after binning the sentence lengths into 20 bins for RTE, MRPC, and STS-B and 50 bins for the others.

Task	Quantiles (train)	Quantiles (eval)	KL Div.
MNLI-m	27/38/50	26/37/50	0.0055
MNLI-mm	27/38/50	29/39/51	0.0042
QQP	23/28/36	23/28/36	0.0006
QNLI	39/48/59	39/49/61	0.0092
SST-2	7/11/19	18/25/33	1.2250
STS-B	20/24/32	21/29/41	0.0925
MRPC	45/54/63	45/54/64	0.0033
RTE	44/57/86	42/54/78	0.0261

LAT with large margins, the sequence length varies greatly from the training dataset to the evaluation dataset as can be seen from the large KL-divergence in Table 4.3 and Figure 1.1 (b, c). On the other hand, for QQP, the only dataset that LAT slightly outperforms LTP (ours), the sequence length distribution of the training dataset is almost identical to that of the evaluation dataset as can be seen from the small KL-divergence in Table 4.3 and Figure 3.1 (a). This observation supports our claim in Section 1 and 2: LTP is robust to sequence length variations as it does not fix the pruning configuration unlike other methods using a single pruning configuration regardless of the input sequence length. This is important in practice because the sequence lengths during inference do not always follow the sequence length distribution of the training dataset as in SST-2 and STS-B. We make a further discussion in Section 4.3.

For SQuAD 2.0, we have similar results to GLUE. As can be seen in Table 4.2 and Figure 4.2 (Left), we obtain nearly identical F1 score to baseline at 0.58 relative FLOPs, and 1.89× speedup with less than 1% drop of F1 score. The SQuAD 2.0 dataset is divided into two subsets: the subset of examples where the answer to the question is included in the context text, and the subset that has no answer. In Figure 4.2 (Right), we further plot the results on each subset of the dataset (black and red for the one with and without answers, respectively). We see that the F1 score decreases for the subset with answers and increases for the subset without answers as we decrease the relative FLOPs. This is to be expected as the question answering head will predict no answer if the start and end points of the answer within the context cannot be determined due to high token pruning ratios. Thus, a careful setting of λ in Eq. 3.12 is necessary to balance the accuracy between the two subsets.

At last, we also highlight that LTP has an additional gain over the prior top-k based approaches by avoiding computationally inefficient top-k operations as further discussed in Section 4.7.

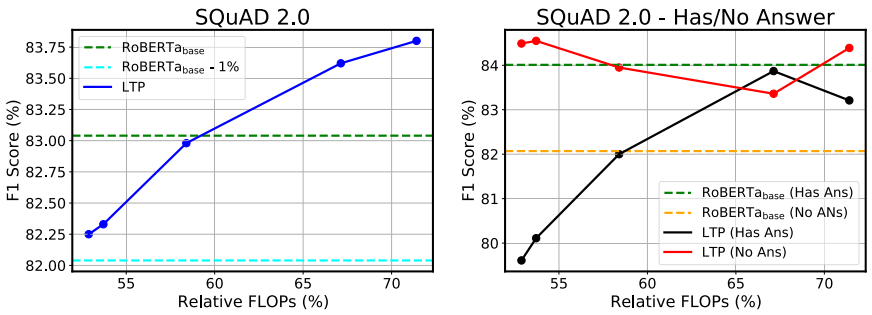


Figure 4.2: (Left) Performance of LTP on SQuAD 2.0 across different relative FLOPs with respect to the the baseline model on the full validation set. (Right) Performance of LTP on the subsets of the validation set, one with answers (Has Ans, black) and the other without (No Ans, red). The performance of the baseline model without token pruning (RoBERTa$_{\text{base}}$) and the model with 1% performance drop (RoBERTa$_{\text{base}}$ - 1%) are dotted in horizontal lines for comparison.

4.3 Robustness to Sequence Length Variation

In Section 4.2, we claim that LTP is more robust against sequence length variations from training time to evaluation time. Here, we make a more systematic analysis on this. Ideally, performance should be independent of sequence length. To quantitatively test the robustness of pruning methods against sequence length variations, we train LTP and LAT on QNLI and QQP, but only using the training examples whose sequence lengths are below the median length of the evaluation dataset. We then evaluate the resulting models using the evaluation examples with sequence lengths (i) below the median (˜Q2), (ii) between the median and the third quantile (Q2˜Q3), and (iii) above the third quantile (Q3˜) of the evaluation dataset. To make a fair comparison, we choose models from LTP and LAT that require similar FLOPs on ˜Q2.

The results are listed in Table 4.4. LTP consistently achieves better accuracy and FLOPs over different sequence lengths, even with those that are significantly longer than the training sequences. On the contrary, LAT shows significant accuracy degradation as longer sequences are over-pruned, which can be seen from the significant FLOPs reduction. In particular, LTP outperforms LAT by up to 16.44% and 9.20% on QNLI and QQP for the Q3˜ evaluation dataset.

4.4 Ablation Studies

Instead of learning thresholds, we can set them manually. Because manually searching over the exponential search space is intractable, we add a constraint to the search space by assigning linearly rising threshold values for each layer, similar to how SpAtten [49] assigns the token

Table 4.4: LTP and LAT trained with the sequences shorter than the median length, and evaluated with the sequences shorther than the median (˜Q2), between the median and the third quantile (Q2˜Q3), and longer than the third quantile (Q3˜) of the evaluation dataset. FLOPs are relative FLOPs (%) with respect to RoBERTa$_{base}$.

	Task	QNLI			QQP		
		˜Q2	Q2˜Q3	Q3˜	˜Q2	Q2˜Q3	Q3˜
LTP (ours)	Acc.	91.21	90.02	91.81	89.42	89.51	91.37
	FLOPs	55.89	55.60	56.02	55.18	56.29	58.01
LAT	Acc.	90.87	86.12	75.37	89.20	87.27	82.17
	FLOPs	56.21	46.55	35.89	55.17	46.61	34.14
Diff.	Acc.	+0.34	+3.90	+16.44	+0.22	+2.24	+9.20

retain ratios: given the threshold of the final layer $\theta^{(L)}$, the threshold for layer l is set as $\theta^{(L)}l/L$. We plot the accuracy and FLOPs of the manual threshold approach in Figure 4.1 as black lines. While this approach exhibits decent results on all downstream tasks, the learned thresholds consistently outperform the manual thresholds under the same FLOPs. This provides empirical evidence for the effectiveness of our threshold learning method.

4.5 Direct Throughput Measurement on Hardware

We directly measure throughputs on real hardware by deploying LTP on a NVIDIA V100 GPU and a Intel Haswell CPU. For inference, we completely remove the pruned tokens and rearrange the retained tokens into a *blank-free* sequence to have a latency gain. One consequence of adaptive pruning, however, is that each sequence will end up with a different pruning pattern and sequence length. As such, a naive hardware implementation of batched inference may require padding all the sequences in a batch to ensure that they all have the same length (i.e., the maximum sequence length in the batch), which results in a significant portion of computation being wasted to process padding tokens. To avoid this, we use NVIDIA's Faster Transformer[2] for GPU implementation that requires large batch sizes. This framework dynamically removes and inserts padding tokens during inference so that most of the transformer operations effectively skip processing padding tokens. This enables fast inference even with irregular pruning lengths of individual sequences. For the CPU implementation, we find naive batching (i.e., padding sequences to the maximum sentence length) enough for good throughput.

The measured throughput results are shown in Figure 4.3 for different batch sizes. For all experiments, relative throughput is evaluated 3 times on the randomly shuffled datasets.

[2] `https://github.com/NVIDIA/FasterTransformer`

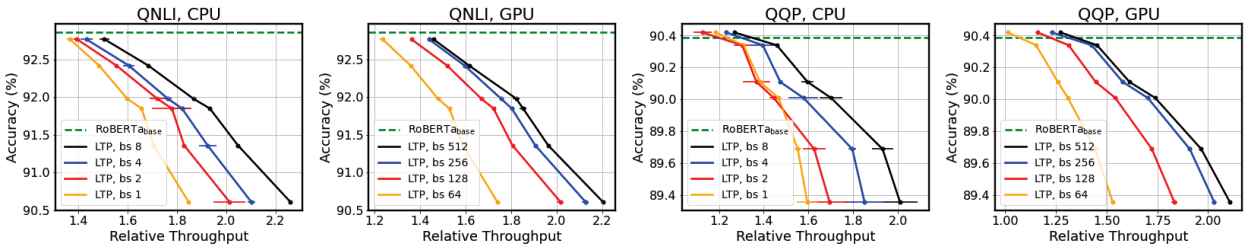


Figure 4.3: Relative throughput of LTP with respect to the baseline without token pruning (RoBERTa$_{\text{base}}$) with different batch sizes on Intel Haswell CPU and NVIDIA V100 GPU. The performance of RoBERTa$_{\text{base}}$ are dotted in horizontal lines.

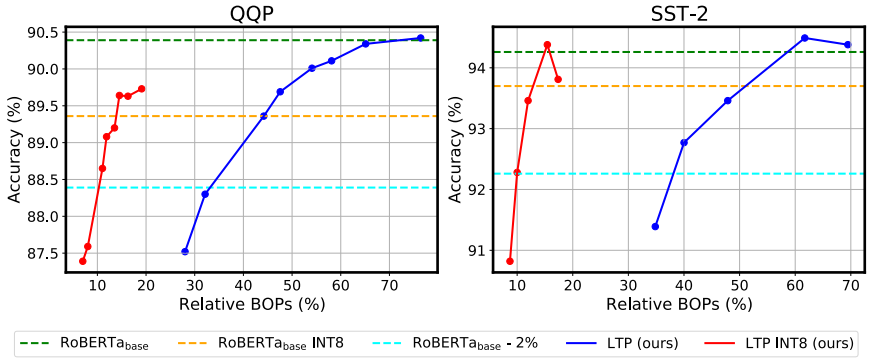


Figure 4.4: Accuracy and relative BOPs of the FP16 baselines and INT8 LTP models on QQP and SST-2 datasets. Note that FP16 unpruned RoBERTa$_{\text{base}}$ is used as the baseline. Thus, INT8 quantization of the models translates to 4× reduction in relative BOPs.

LTP achieves up to ∼1.9× and ∼2.0× thoughput improvement for QNLI and QQP on both CPU and GPU, as compared to the baseline. This is similar to the theoretical speedup inferred from the FLOPs reduction reported in Table 4.2. Importantly, the speedup of LTP increases with larger batch sizes on both CPU and GPU, proving effectiveness of LTP on batched cases.

4.6 LTP with Quantization and Knowledge Distillation

Here, we show that our token-level pruning method is compatible with other compression methods. In particular, we perform compression experiments by combining LTP with quantization and knowledge distillation [14] together. For quantization, we use the static uniform symmetric integer quantization method [12], which is easy to deploy in commodity hardware with minimal run-time overhead. All the model parameters are quantized to 8-bit integers, except for those of the embedding layer whose bit-width does not affect the inference

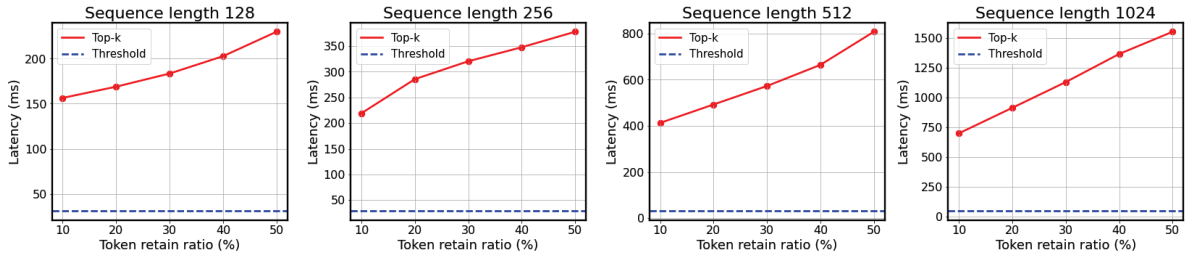


Figure 4.5: Wall-clock latency comparison between top-k operation and threshold operation on an Intel Haswell CPU for different sequence length across various token retain ratios. Note that the latency of a threshold operation is independent of sequence length.

speed. Afterwards, we apply knowledge distillation that helps recover accuracy for high compression ratios. We set the baseline RoBERTa$_{\text{base}}$ model as the teacher and the quantized LTP model as the student. We then distill knowledge from the teacher model into the student model through a knowledge distillation loss that matches the output logits of the classification layer and the output representations of the embedding layer in the teacher model to the counterparts in the student model. The training objective is a convex combination of the original loss and the knowledge distillation loss. As shown in Figure 4.4, we achieve up to 10× reduction in bit operations (BOPs) with less than 2% accuracy degradation as compared to FP16 RoBERTa$_{\text{base}}$ by combining quantization and knowledge distillation. The results empirically show the effectiveness of LTP with other compression methods.

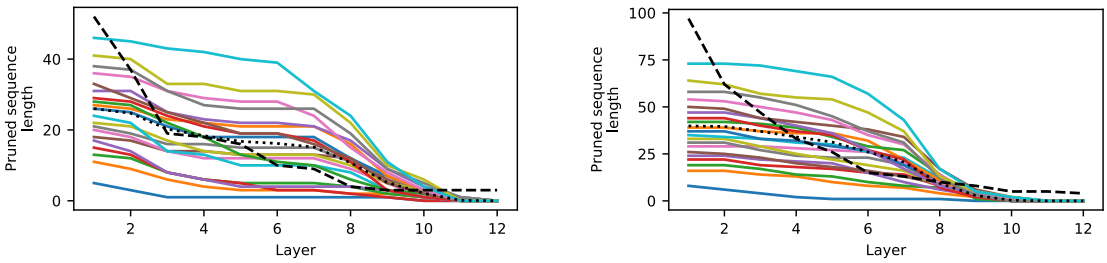
4.7 Discussions

Computation Efficiency Comparison

Here we compare the efficiency of top-k versus threshold operation. To do this, we use a batch size of 32 and average the latency over 1000 independent runs. For each sequence length, we test over five different token retain ratios from 10% to 50% (e.g., 10% token retain ratio is the case where we select top-k 10% of tokens from the input sequence).

With the above setting, we directly measure the latency of these two operations on an Intel Haswell CPU, and report the results in Figure 4.5. For top-k operation, there is a noticeable increase in latency when token retain ratios and sequence lengths become larger whereas this is not an issue for our threshold pruning method as it only requires a comparison operation. More importantly, top-k operation incurs a huge latency overhead that is up to 7.4× and 33.4× slower than threshold operation for sequence length of 128 and 1024, respectively.[3]

[3]The inefficiency of top-k is also further confirmed by [49], where they report only 1.1× speedup for GPT-2 without the top-k hardware engine that they developed.



(a) SST-2 (b) MNLI-m

Figure 4.6: Sample trajectories of pruned sequence length as the sequences are passed through model layers. For LTP, 20 samples were evenly selected from the sets after sorting by initial sequence length. (a) SST-2. (b) MNLI-m. The mean sequence length for LTP is shown by a black dotted line, and the LAT baseline is shown by a black dashed line. Parameters were selected so as to provide a 1% drop in accuracy from baseline for both methods.

Example Sequence Length Trajectories

Figure 4.6 shows how the pruned sequence length decreases for input sequences of varying lengths. For LAT, the token pruning configuration is fixed for all sequences in the dataset. In LTP, token pruning can be more or less aggressive depending on the sequence content and the number of important tokens in the sequence. On average, LTP calculates 25.86% fewer tokens per layer than LAT for MNLI-m and 12.08% fewer tokens for SST-2. For both LTP and LAT, the model has been trained to produce a 1% drop in accuracy compared to baseline.

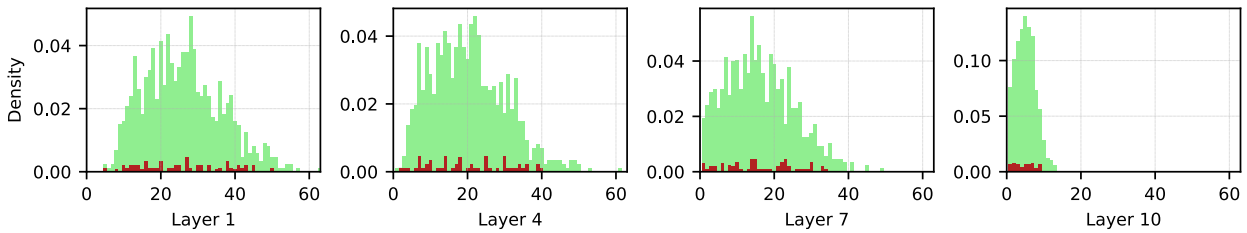
Unbiased Token Pruning for Various Sequence Length

Figure 4.7 shows the distributions of initial sequence lengths for sequences that are correctly classified and for sequences that are not. We see that for multiple tasks, there is no significant correlation between the length of the sequence and the accuracy of the pruned models. Importantly, this suggests that our method is not biased towards being more accurate on longer or shorter sequences.

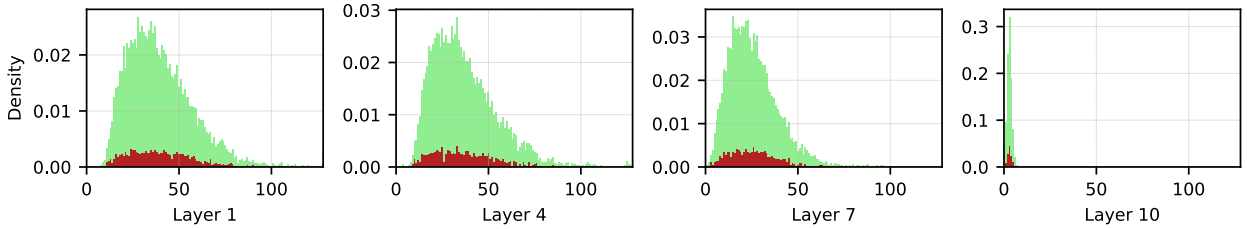
Comparison with TR-BERT on GLUE

Unlike LAT and SpAtten, TR-BERT [54] does not report results on the GLUE benchmark tasks described in the paper. We attempted to run TR-BERT on the GLUE tasks using the TR-BERT repo[4], but were unable to get the algorithm to converge to a high accuracy, despite varying the learning rate between 1e-6 and 1e-3 and the value of α, the parameter that defines the length penalty, over the search space of $\{0.01, 0.05, 0.1, 0.5, 1, 2, 5\}$. We also varied the

[4] `https://github.com/thunlp/TR-BERT`



(a) SST2



(b) MNLI-m

Figure 4.7: Histogram of pruned sequence length (x-axis) as the input sequence is processed through different transformer blocks. y-axis shows the relative count of sentences with the particular sequence length in x-axis. Green denotes input sequences that are correctly classified, and red denotes incorrect classifications.

number of training epochs based on the number of examples in each task's training set. The authors of TR-BERT note the convergence difficulties of RL learning while describing the algorithm in their paper.

Chapter 5

Conclusions

In this thesis, we present Learned Token Pruning (LTP), a fully automated token pruning framework for transformers. LTP only requires comparison of token importance scores with threshold values to determine unimportant tokens, thus adding minimal complexity over the original transformer inference. Importantly, the threshold values are learned for each layer during training through a differentiable soft binarized mask that enables backpropagation of gradients to the threshold values. Compared to the state-of-the-art token pruning methods, LTP outperforms by up to ~2.5% accuracy with the same amount of FLOPs. Extensive experiments on GLUE and SQuAD show the effectiveness of LTP, as it achieves up to 2.10× FLOPs reduction over the baseline model within only 1% of accuracy degradation. Our preliminary (and not highly optimized) implementation shows up to 1.9× and 2.0× throughput improvement on an Intel Haswell CPU and an NVIDIA V100 GPU. Furthermore, LTP exhibits significantly better robustness and consistency over different input sequence lengths.

Bibliography

[1] Haoli Bai et al. "BinaryBERT: Pushing the Limit of BERT Quantization". In: *arXiv preprint arXiv:2012.15701* (2020).

[2] Aishwarya Bhandare et al. "Efficient 8-bit quantization of transformer neural machine language translation model". In: *arXiv preprint arXiv:1906.00532* (2019).

[3] Daniel Cer et al. "Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation". In: *arXiv preprint arXiv:1708.00055* (2017).

[4] Tianlong Chen et al. "The lottery ticket hypothesis for pre-trained BERT networks". In: *arXiv preprint arXiv:2007.12223* (2020).

[5] Rewon Child et al. "Generating long sequences with sparse transformers". In: *arXiv preprint arXiv:1904.10509* (2019).

[6] Ido Dagan, Oren Glickman, and Bernardo Magnini. "The PASCAL recognising textual entailment challenge". In: *Machine Learning Challenges Workshop*. Springer. 2005, pp. 177–190.

[7] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[8] William B Dolan and Chris Brockett. "Automatically constructing a corpus of sentential paraphrases". In: *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*. 2005.

[9] Angela Fan, Edouard Grave, and Armand Joulin. "Reducing transformer depth on demand with structured dropout". In: *arXiv preprint arXiv:1909.11556* (2019).

[10] Angela Fan et al. "Training with quantization noise for extreme model compression". In: *arXiv e-prints* (2020), arXiv–2004.

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

[12] Amir Gholami et al. "A survey of quantization methods for efficient neural network inference". In: *arXiv preprint arXiv:2103.13630* (2021).

[13] Saurabh Goyal et al. "Power-bert: Accelerating bert inference via progressive word-vector elimination". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 3690–3699.

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

[15] Lu Hou et al. "Dynabert: Dynamic bert with adaptive width and depth". In: *arXiv preprint arXiv:2004.04037* (2020).

[16] Forrest N Iandola et al. "SqueezeBERT: What can computer vision teach NLP about efficient neural networks?" In: *arXiv preprint arXiv:2006.11316* (2020).

[17] Shankar Iyer, Nikhil Dandekar, and Kornl Csernai. "First Quora Dataset Release: Question Pairs.(2017)". In: *URL https://data. quora. com/First-Quora-Dataset-Release-Question-Pairs* (2017).

[18] Xiaoqi Jiao et al. "Tinybert: Distilling bert for natural language understanding". In: *arXiv preprint arXiv:1909.10351* (2019).

[19] Angelos Katharopoulos et al. "Transformers are rnns: Fast autoregressive transformers with linear attention". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5156–5165.

[20] Ashish Khetan and Zohar Karnin. "schubert: Optimizing elements of bert". In: *arXiv preprint arXiv:2005.06628* (2020).

[21] Gyuwan Kim and Kyunghyun Cho. "Length-Adaptive Transformer: Train Once with Length Drop, Use Anytime with Search". In: *arXiv preprint arXiv:2010.07003* (2020).

[22] Sehoon Kim et al. "I-BERT: Integer-only BERT Quantization". In: *International conference on machine learning* (2021).

[23] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. "Reformer: The efficient transformer". In: *arXiv preprint arXiv:2001.04451* (2020).

[24] François Lagunas et al. "Block pruning for faster transformers". In: *arXiv preprint arXiv:2109.04838* (2021).

[25] Zhenzhong Lan et al. "Albert: A lite bert for self-supervised learning of language representations". In: *arXiv preprint arXiv:1909.11942* (2019).

[26] Bingbing Li et al. "Efficient transformer-based large scale language representations using hardware-friendly block structured pruning". In: *arXiv preprint arXiv:2009.08065* (2020).

[27] Zi Lin et al. "Pruning Redundant Mappings in Transformer Models via Spectral-Normalized Identity Prior". In: *arXiv preprint arXiv:2010.01791* (2020).

[28] Yinhan Liu et al. "RoBERTa: A robustly optimized bert pretraining approach". In: *arXiv preprint arXiv:1907.11692* (2019).

[29] Zejian Liu et al. "EBERT: Efficient BERT Inference with Dynamic Structured Pruning". In: *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. 2021, pp. 4814–4823.

[30] Paul Michel, Omer Levy, and Graham Neubig. "Are sixteen heads really better than one?" In: *arXiv preprint arXiv:1905.10650* (2019).

[31] Sai Prasanna, Anna Rogers, and Anna Rumshisky. "When BERT plays the lottery, all tickets are winning". In: *arXiv preprint arXiv:2005.00561* (2020).

[32] Ofir Press, Noah A Smith, and Mike Lewis. "Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation". In: *arXiv preprint arXiv:2108.12409* (2021).

[33] Pranav Rajpurkar, Robin Jia, and Percy Liang. "Know what you don't know: Unanswerable questions for SQuAD". In: *arXiv preprint arXiv:1806.03822* (2018).

[34] Pranav Rajpurkar et al. "SQuAD: 100,000+ questions for machine comprehension of text". In: *arXiv preprint arXiv:1606.05250* (2016).

[35] Aurko Roy et al. "Efficient content-based sparse attention with routing transformers". In: *Transactions of the Association for Computational Linguistics* 9 (2021), pp. 53–68.

[36] Hassan Sajjad et al. "On the Effect of Dropping Layers of Pre-trained Transformer Models". In: *arXiv preprint arXiv:2004.03844* (2020).

[37] Victor Sanh, Thomas Wolf, and Alexander M Rush. "Movement pruning: Adaptive sparsity by fine-tuning". In: *arXiv preprint arXiv:2005.07683* (2020).

[38] Victor Sanh et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv preprint arXiv:1910.01108* (2019).

[39] Sheng Shen et al. "Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT." In: *AAAI*. 2020, pp. 8815–8821.

[40] Richard Socher et al. "Recursive deep models for semantic compositionality over a sentiment treebank". In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1631–1642.

[41] Siqi Sun et al. "Patient knowledge distillation for bert model compression". In: *arXiv preprint arXiv:1908.09355* (2019).

[42] Zhiqing Sun et al. "Mobilebert: a compact task-agnostic bert for resource-limited devices". In: *arXiv preprint arXiv:2004.02984* (2020).

[43] Raphael Tang et al. "Distilling task-specific knowledge from BERT into simple neural networks". In: *arXiv preprint arXiv:1903.12136* (2019).

[44] Yi Tay et al. "Sparse sinkhorn attention". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 9438–9447.

[45] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[46] Elena Voita et al. "Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned". In: *arXiv preprint arXiv:1905.09418* (2019).

[47] Apoorv Vyas, Angelos Katharopoulos, and François Fleuret. "Fast transformers with clustered attention". In: *Advances in Neural Information Processing Systems* 33 (2020).

[48] Alex Wang et al. "GLUE: A multi-task benchmark and analysis platform for natural language understanding". In: *arXiv preprint arXiv:1804.07461* (2018).

[49] Hanrui Wang, Zhekai Zhang, and Song Han. "SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning". In: *arXiv preprint arXiv:2012.09852* (2020).

[50] Sinong Wang et al. "Linformer: Self-Attention with Linear Complexity". In: *arXiv preprint arXiv:2006.04768* (2020).

[51] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. "Structured pruning of large language models". In: *arXiv preprint arXiv:1910.04732* (2019).

[52] Adina Williams, Nikita Nangia, and Samuel R Bowman. "A broad-coverage challenge corpus for sentence understanding through inference". In: *arXiv preprint arXiv:1704.05426* (2017).

[53] Zhewei Yao et al. "MLPruning: A Multilevel Structured Pruning Framework for Transformer-based Models". In: *arXiv preprint arXiv:2105.14636* (2021).

[54] Deming Ye et al. "TR-BERT: Dynamic Token Reduction for Accelerating BERT Inference". In: *arXiv preprint arXiv:2105.11618* (2021).

[55] Ali Hadi Zadeh et al. "Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 811–824.

[56] Ofir Zafrir et al. "Q8BERT: Quantized 8bit bert". In: *arXiv preprint arXiv:1910.06188* (2019).

[57] Manzil Zaheer et al. "Big bird: Transformers for longer sequences". In: *arXiv preprint arXiv:2007.14062* (2020).

[58] Wei Zhang et al. "Ternarybert: Distillation-aware ultra-low bit bert". In: *arXiv preprint arXiv:2009.12812* (2020).

[59] Mengjie Zhao et al. "Masking as an efficient alternative to finetuning for pretrained language models". In: *arXiv preprint arXiv:2004.12406* (2020).