Learned Token Pruning for Transformers

Sehoon Kim[*,1], Sheng Shen[*,1], David Thorsley[*,2], Amir Gholami[*,1], Woosuk Kwon[1],
Joseph Hassoun[2], Kurt Keutzer[1]
[1]University of California, Berkeley, [2]Samsung Semiconductor, Inc.
{sehoonkim, sheng.s, amirgh, woosuk.kwon, keutzer}@berkeley.edu,
{d.thorsley, j.hassoun}@samsung.com

***Abstract*—Deploying transformer models in practice is challenging due to their inference cost, which scales quadratically with input sequence length. To address this, we present a novel Learned Token Pruning (LTP) method 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 value which 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 tasks and show that our method outperforms the prior state-of-the-art token pruning methods by up to ~2.5% higher accuracy with the same amount of FLOPs. In particular, LTP achieves up to 2.1× FLOPs reduction with less than 1% accuracy drop, which results in up to 1.9× and 2.0× throughput improvement on Intel Haswell CPUs and NVIDIA V100 GPUs, respectively. Furthermore, we demonstrate that LTP is more robust than prior methods to variations on input sentence lengths. Our code has been developed in PyTorch and has been open-sourced [1].**

I. Introduction

Transformer-based deep neural network (DNN) models [44], such as BERT [10] and RoBERTa [27] achieve state-of-the-art results in Natural Language Processing (NLP) tasks such as sentence classification. 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 is quadratic with input sequence size; thus the ability to apply transformer models to long input sequences becomes limited.

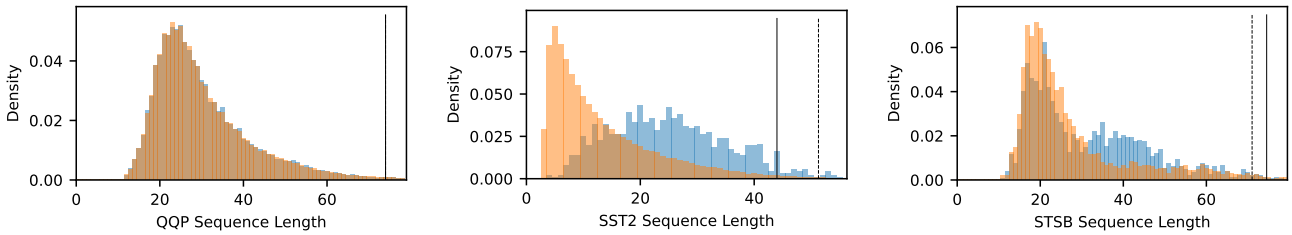
[*]Equal contribution.

Pruning is a popular technique to reduce the size of DNNs 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 pruning; for example, head pruning [28] decreases the size of the model by removing unneeded heads in each transformer layer. Another orthogonal approach to pruning that we consider here is *token pruning*, which reduces computation by progressively removing unimportant tokens in the sentence 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 [16] and Length-Adaptive Transformer (LAT) [23] search for a single token pruning configuration (e.g., sentence length for each layer) for an entire dataset. However, input sequence lengths can vary greatly within tasks and between training and validation sets (as in Figure 1), and thus a single pruning configuration can potentially under-prune shorter sequences or over-prune longer sequences.

In the other class, the token pruning method adjusts the configuration based on the input sequence. SpAtten [48] 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 [53] uses reinforcement learning (RL) to find a policy network that dynamically reduces the number of tokens based on the length and



(a) *QQP* **(b)** *SST-2* **(c)** *STS-B*

Fig. 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.*

content of the input sequence, but requires additional costly training for the method to converge.

Additionally, all of these methods depend 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.

In this work, 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. Our contributions are:

- We propose Learned Token Pruning (LTP), a threshold-based token pruning method, which only needs a simple comparison (i.e., threshold) operation to detect unimportant tokens. In addition, LTP fully automates the search for optimal pruning configurations by posing the problem as learning a binarized mask. In particular, we use a differentiable soft binarized mask which can be trained to learn the correct thresholds for different transformer layers and different tasks. (Section III-C)
- We apply LTP to RoBERTa and evaluate its performance on GLUE tasks. We show LTP achieves up to 2.10× FLOPs reduction compared to the baseline with less than 1% accuracy degradation. We also show that LTP outperforms SpAtten and LAT in most cases, achieving additional FLOPs reduction for the same drop in accuracy. (Section IV-B)
- We show that LTP is robust against sentence length variations. LTP exhibits consistent accuracy and FLOPs over different sentence length distributions, achieving up to 16.4% accuracy gap from LAT. (Section IV-C)
- We directly deployed and measured the throughput of LTP on both GPUs and CPUs. We achieve up to 1.93× and 1.97× throughput improvement on NVIDIA V100 GPU and Intel Haswell CPU as compared to unpruned FP16 baseline, respectively. (Section IV-E).
- We show that the inference costs can be further reduced by combining LTP with other compression methods such as quantization. Our experimental results demonstrate that quantized LTP models achieve up to 10× reduction in the number of bit operations (BOPs) [43] with $< 2\%$ accuracy drop compared to the unpruned FP16 baseline. (Section IV-F)

II. Related Work

Efficient NLP. 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 [8, 19, 22, 25, 26, 33, 42, 46, 49, 57]; (ii) knowledge distillation [21, 35, 39–41]; (iii) quantization [2, 4, 13, 24, 37, 55, 56, 58]; and (iv) pruning. Here, we focus only on pruning and briefly discuss the related work.

Pruning. Pruning methods can be categorized into unstructured pruning and structured pruning. For unstructured pruning in NLP, the lottery-ticket hypothesis [14] has been explored for transformers in [7, 30, 54]. Recently, [59] leverages pruning as an effective way to fine-tune transformers on downstream tasks. [36] proposes movement pruning, which achieves significant performance improvements versus magnitude-based methods by considering the weights changing during fine-tuning. However, it is often quite difficult to efficiently deploy unstructured sparsity on commodity DNN accelerators for meaningful speedup.

For structured pruning in NLP, [5, 29, 51] explore row and block-wise pruning for LSTM models [18]. For transformer architectures, [12, 34] use LayerDrop

to train models that have adaptive depth during training and drop unimportant layers during fine-tuning. [28, 45] drop different attention heads in multi-head attention and show redundancies between attention heads. [50] applies low-rank approximation to guide structured weight matrix pruning.

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 [16], one of the earliest works, proposes to directly learn token pruning configurations. LAT [23] extends this idea by introducing LengthDrop, a procedure in which a model is trained with different pruning configurations, followed by an evolutionary search. This method has the advantage that the former training 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 only search a single token pruning configuration for the entire dataset. However, as shown in Figure 1, input sequence lengths vary greatly within tasks. As a consequence, 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 shift, where input sentences at inference time are longer than those in the training dataset [31].

In contrast, SpAtten [48] circumvents this issue by assigning a pruning configuration proportional to the input sentence 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 content. In addition, the pruning configurations are determined heuristically and thus can result in a suboptimal solution. Recently, TR-BERT [53] proposes to use reinforcement learning (RL) to learn a token selection policy network for each token at different layers. 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 prior 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 [48]. LTP, on the other hand, is based on a fully learnable threshold-based pruning strategy and thus is (i) adaptive to both input length and content, (ii) robust against sentence length variations, (iii) computationally efficient, and (iv) easy to deploy.

III. Methodology

In this section, we recap the BERT architecture as described in Section III-A. We then introduce our threshold token pruning method in Section III-B and demonstrate how to learn threshold values in Section III-C.

A. Background

BERT [10] consists of multiple transformer encoder layers [44] 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 is applied to input tokens of each layer, x, and measures the pairwise importance of each token on every other token in the input:

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

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

$$\mathrm{Att}_{\mathbf{W}_{k,q,v,o}}(\mathrm{x}) = \mathbf{W}_o \sum_{i=1}^{n} \mathbf{W}_v \mathrm{x}_i \mathrm{softmax}\Big(\frac{\mathrm{x}^T \mathbf{W}_q^T \mathbf{W}_k \mathrm{x}_i}{\sqrt{d}}\Big), \tag{2}$$

$$\mathbf{x}_{\mathrm{MHA}} = \mathrm{LN}\big(\mathrm{Att}_{\mathbf{W}_{k,q,v,o}}(\mathrm{x}) + \mathrm{x}\big), \tag{3}$$

where the last equation 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:

$$\mathrm{FFN}(\mathrm{x}_{\mathrm{MHA}}) = \sigma\big(\mathbf{W}_2(\mathbf{W}_1 \mathrm{x}_{\mathrm{MHA}} + b_1)\big) + b_2, \tag{4}$$

$$\mathbf{x}_{\mathrm{out}} = \mathrm{LN}\big(\mathrm{FFN}(\mathrm{x}_{\mathrm{MHA}}) + \mathrm{x}_{\mathrm{MHA}}\big), \tag{5}$$

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).

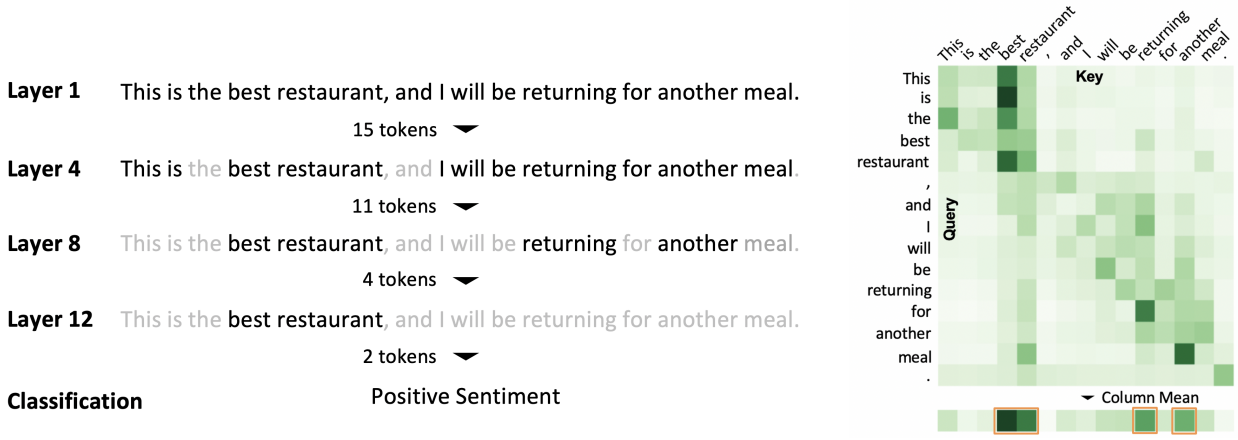


Fig. 2: *(Left) Schematic of token pruning for a sentiment analysis task. Unimportant tokens are pruned as the input sentence passes through the layers. (Right) An example of attention probability in a single head where a more important token receives more attention from other tokens. Thus each token's importance score is computed by taking the average attention probability it receives, which is computed by taking the column mean of the attention probability.*

B. Threshold Token Pruning

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

$$\mathbf{A}^{(h,l)}(\mathrm{x}_i, \mathrm{x}_j) = \mathrm{softmax}(\frac{\mathrm{x}^T \mathbf{W}_q^T \mathbf{W}_k \mathrm{x}}{\sqrt{d}})_{(i,j)} \in \mathbb{R}. \quad (6)$$

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 2 (Left).

For token pruning, we must define a metric to determine unimportant tokens. Following [16, 23, 48], we define the *importance score* of token x_i in layer l as:

$$s^{(l)}(\mathrm{x}_i) = \frac{1}{N_h} \frac{1}{n} \sum_{h=1}^{N_h} \sum_{j=1}^{n} \mathbf{A}^{(h,l)}(\mathrm{x}_i, \mathrm{x}_j). \quad (7)$$

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 7. The procedure for computing importance scores from attention probabilities is illustrated in Figure 2 (Right). Alternatively, [53] computes the token importance score using the variation in task loss when a token is not selected. In this work, we use the attentional score based approach.

In [16, 23, 48], 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 [48]; we provide empirical results showing this in Section B.

Instead, we introduce a new threshold-based token pruning approach which prunes tokens if their importance score is below a threshold denoted by $\theta^{(l)} \in \mathbb{R}$. Specifically, we define a pruning strategy by imposing a binary mask $M^{(l)}(\cdot) : \{1, \ldots, n\} \rightarrow \{0, 1\}$ which indicates whether a token should be kept or pruned:

$$M^{(l)}(\mathrm{x}_i) = \begin{cases} 1 & \text{if } s^{(l)}(\mathrm{x}_i) > \theta^{(l)}, \\ 0 & \text{otherwise.} \end{cases} \quad (8)$$

In other words, a token is pruned only if its importance score is below the threshold, and this 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 in the bottom layers.

C. 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

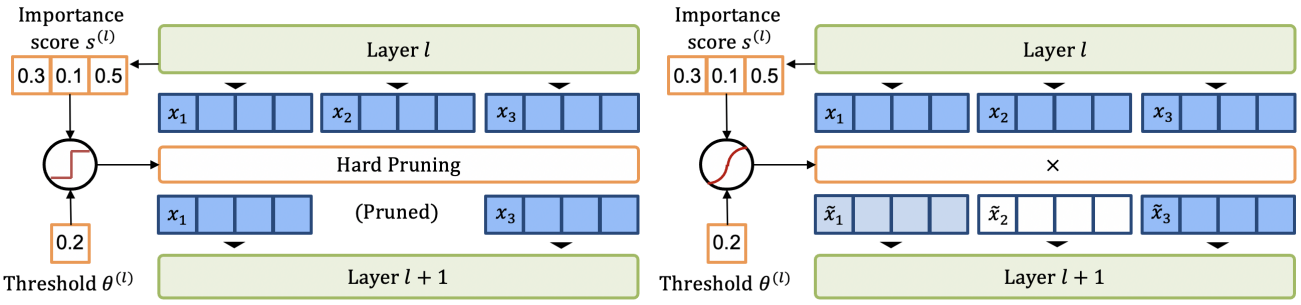


Fig. 3: *Different pruning strategies for threshold-based token pruning methods. (Left) Hard pruning uses a binary hard mask to select tokens to be pruned. (Right) Soft pruning replaces the binary mask with a differentiable soft mask.*

making the masking function (i.e., M in Eq. 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 thresholds. Furthermore, we cannot apply the conventional Straight-Through Estimator (STE) [3] to estimate gradients with respect to the threshold. 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.

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:

$$\tilde{M}^{(l)}(\mathrm{x}_i) = \sigma\left(\frac{s^{(l)}(\mathrm{x}_i) - \theta^{(l)}}{T}\right) \tag{9}$$

$$= \frac{1}{1+\exp(-(s^{(l)}(\mathrm{x}_i) - \theta^{(l)})/T)}, \tag{10}$$

where T is a temperature parameter, and $\theta^{(l)}$ is the learnable threshold value for layer l. With a sufficiently high temperature T, $\tilde{M}^{(l)}(\mathrm{x}_i)$ will closely approximate the hard masking $M^{(l)}(\mathrm{x}_i)$ in Eq. 8. Furthermore, instead of selecting tokens to be pruned or kept based on the hard mask of Eq. 8, we multiply the soft mask by the output activation of layer l. That is,

$$\tilde{\mathrm{x}}^{(l)}_{\text{out}} = \tilde{M}^{(l)}(\mathrm{x}^{(l)}) \cdot \mathrm{x}^{(l)}_{\text{out}} \tag{11}$$

$$= \tilde{M}^{(l)}(\mathrm{x}^{(l)}) \cdot \mathrm{LN}(\mathrm{FFN}(\mathrm{x}^{(l)}_{\text{MHA}}) + \mathrm{x}^{(l)}_{\text{MHA}}), \tag{12}$$

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 jointly training model parameters and thresholds on downstream tasks with the soft pruning scheme, we fix the thresholds, binarize the soft mask, and 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.

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

Regularization. It is not possible to learn θ to prune the network 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 $\tilde{M}$:

$$\mathcal{L}_{\text{new}} = \mathcal{L} + \lambda \mathcal{L}_{\text{reg}} \quad \text{where} \quad \mathcal{L}_{\text{reg}} = \frac{1}{L}\sum_{l=1}^{L} ||\tilde{M}^{(l)}(\mathrm{x})||_1. \tag{13}$$

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:

$$\frac{d\mathcal{L}_{\text{reg}}}{d\theta^{(l)}} = \frac{1}{d\theta^{(l)}}||\tilde{M}^{(l)}(\mathrm{x})||_1 \tag{14}$$

$$= \frac{1}{d\theta^{(l)}}\left(\sum_{i=1}^{n}\tilde{M}^{(l)}(\mathrm{x}_i)\right) = \sum_{i=1}^{n}\frac{d\tilde{M}^{(l)}(\mathrm{x}_i)}{d\theta^{(l)}}. \tag{15}$$

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.

IV. Experiments

A. Experiment Setup

We implemented LTP on RoBERTa$_{\text{base}}$ [27] using HuggingFace's transformers repo[1] and tested on monolingual (English) GLUE tasks [47]. As mentioned in Section III-C, the training procedure of LTP consists of two stages: soft pruning that trains both the model parameters and thresholds on downstream tasks, followed by hard pruning that fine-tunes the model parameters with fixed thresholds. We also compare LTP with the current state-of-the-art token pruning methods of SpAtten [48] and LAT [23] following the implementation details in their papers. TR-BERT [53] did not report results on GLUE tasks; we discuss in Section E. See A1 for the details of the datasets used, training process, baseline models, and experiment environments.

An important issue in previous work is that *all* input sentences for a specific task are padded to the nearest power of 2 [16, 23] from the 99th percentile of the sentence lengths, and then the pruned performance is compared with the padded baseline. This results in exaggerated performance gain over the baseline. For instance, in [16], inputs from the SST-2 dataset are padded to 64, while its average sentence length is 26 (cf. Figure 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 unpadded (or minimal padding for batching) baselines.

B. Performance Evaluation

Table I lists the accuracy and GFLOPs for the LTP method. We select a model for each downstream task that achieves the smallest GFLOPs while constraining the accuracy degradation from the baseline (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 token pruning method achieves speedup of 1.97× on average and up to 2.10× within 1% accuracy degradation.

Table I: *Detailed performance and efficiency comparison of* LTP *applied to RoBERTa$_{base}$.*

Task	Accuracy		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×

Figure 4 shows the accuracy of models with different FLOPs (blue lines). We plot the accuracy of the baseline pruning methods (red lines for SpAtten and orange lines for LAT). LTP consistently outperforms SpAtten for all tasks with up to ~2% higher accuracy with the same amount of FLOPs. We particularly observe a noticeable performance gap around 60% of the relative FLOPs.

Compared with LAT, LTP consistently outperforms all tasks (except for QQP and QNLI) with up to ~2.5% higher accuracy for the same target FLOPs. For QQP, LTP achieves at most ~0.2% lower accuracy than LTP. We attribute the good performance of LAT on QQP/QNLI to the similar sentence length distribution of the training dataset and the evaluation dataset. As shown in Table II, the training dataset and evaluation dataset of QQP/QNLI have highly similar sentence length distributions with relatively small KL divergence. In contrast, LAT exhibits relatively poor performance on SST-2 and STS-B in which KL divergence between the training dataset and the evaluation dataset is larger. We further explore the robustness against training and evaluation sentence length mismatch in Section IV-C. We also highlight that not only does LTP attain better solutions in accuracy-computation trade-off than top-k based methods, it has an additional gain from avoiding computationally inefficient top-k operations as further discussed in Section B.

C. Robustness to Length Variation

We test the robustness of pruning methods against variation in sentence lengths. Ideally, performance should be independent of sequence length. To test this robustness,

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

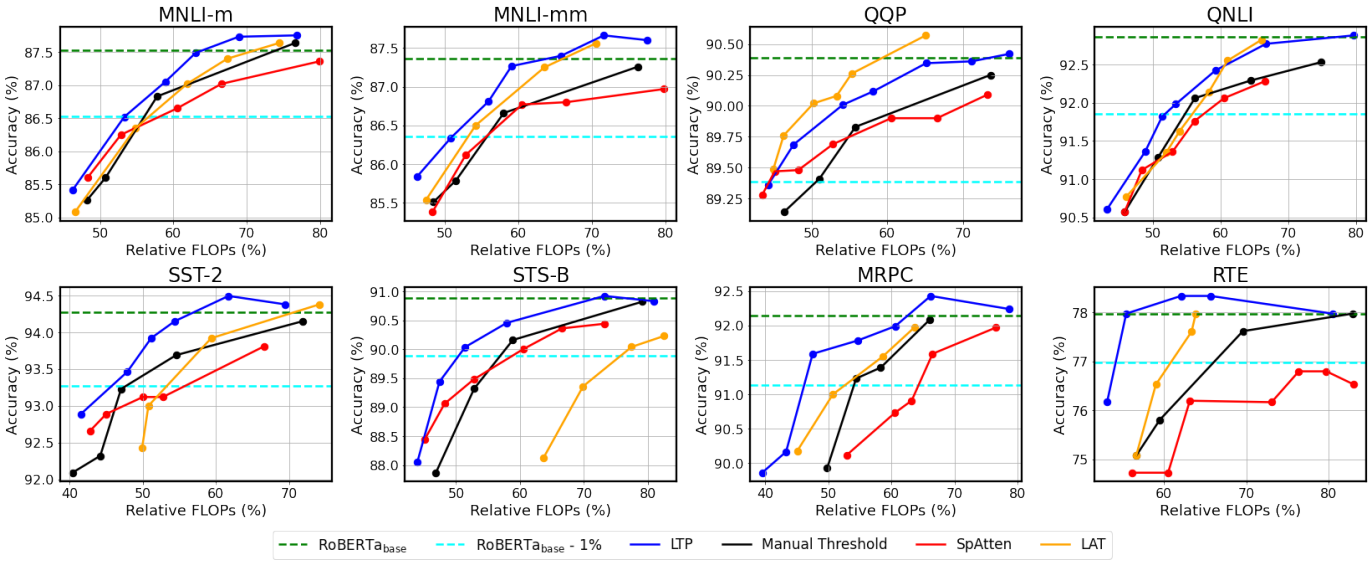


Fig. 4: *Performance of different pruning methods on GLUE tasks for different token pruning methods across different relative FLOPs, i.e., normalized FLOPs with respect to the the baseline model. Manual threshold assigns linearly raising threshold values for each layer instead of learning them. The performance of the baseline model without token pruning (RoBERTa$_{base}$) and the model with 1% performance drop (RoBERTa$_{base}$ - 1%) are dotted in horizontal lines for comparison.*

Table II: *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

we train both LTP and LAT on QNLI and QQP, where LAT shows a comparable or slightly better performance than LTP. Here, we only using training examples whose sentence lengths are below the median length of the validation dataset. We then evaluate the resulting models using validation examples with sequence lengths (1) below the median (~Q2), (2) above the median and below the third quantile (Q2~Q3), and (3) above the third quantile (Q3~) of the validation dataset. To make a fair comparison, we choose models from LTP and LAT that require similar FLOPs on ~Q2.

Table III: *LTP and LAT evaluated on different sentence lengths. Both are trained with training examples shorter than the median length of the evaluation dataset, and are tested on evaluation examples 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	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

The results are listed in Table III. LTP consistently achieves better accuracy/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 significant FLOP reduction. In particular, LTP outperforms LAT by up to 16.44% and 9.20% on QNLI and QQP for the ~Q3 evaluation dataset.

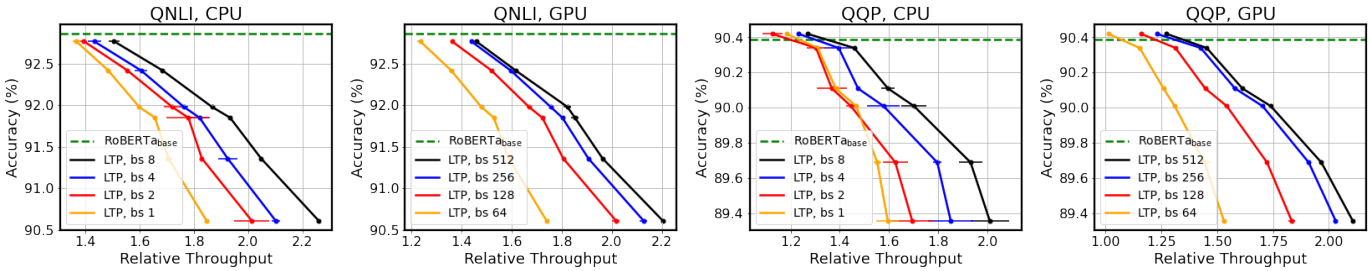


Fig. 5: *Relative throughput of* LTP *with respect to the baseline without token pruning (RoBERTa$_{base}$) with different batch sizes on Intel Haswell CPU and NVIDIA V100 GPU. The performance of RoBERTa$_{base}$ are dotted in horizontal lines.*

D. 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 [48] assigns the token retain ratios. That is, 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 as black lines. While the manual threshold 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.

E. Direct Throughput Measurement on Hardware

A consequence of adaptive pruning is that each sequence will end up with a different pruning pattern. As such, a naive hardware implementation of batched inference may require padding all the sentences in a batch to ensure that they all have the same length (i.e., the maximum sentence length in the batch). This results in a significant portion of computation being wasted processing padding tokens.

However, this issue is more or less implementation dependent, since the original unpadded sequences still result in dense matrix operations which can be efficiently deployed in hardware. To show this, we deployed LTP on both a NVIDIA V100 GPU and a Intel Haswell CPU, using the validation datasets of QNLI and QQP tasks. For the GPU implementation, we used NVIDIA's Faster Transformer[2], which dynamically removes and inserts padding tokens during inference so that most of the transformer operations effectively skip processing padding tokens. We implement LTP on top of Faster Transformer 4.0 for GPU experiments, enabling fast inference even with irregular pruning lengths of individual sentences. For the CPU implementation, we find naive batching (i.e., padding sentences to the maximum sentence length) enough for good throughput.

The measured throughput results are shown in Figure 5 for different batch sizes. For all experiments, relative throughput is evaluated 3 times on the randomly shuffled datasets. LTP achieves up to 1.83× and 1.82× higher throughput for QNLI, and 1.93× and 1.97× improvement for QQP on CPU and GPU, as compared to the baseline, respectively, with less than 1% accuracy degradation. This is similar to the theoretical speedup inferred from the reported FLOPs reduction in Table I. Importantly, the speedup of LTP increases with larger batch sizes on both CPU and GPU, proving effectiveness of LTP on batched cases.

F. Quantization and Pruning

Here, we show that our token-level pruning method is compatible with other compression methods. We performed compression experiments by combining pruning, quantization, and knowledge distillation [17] together. In particular, we quantized the NN weights to 8 bits, and applied knowledge distillation which is helpful for improving accuracy for high compression ratios. The detailed results are reported in Section C, where we achieve up to 10× reduction in bit operations (BOPs) with less than 2% accuracy degradation as compared to FP16 RoBERTa$_{base}$.

V. Conclusions

In this work, 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. Importantly, the threshold values are learned

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

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 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 a NVIDIA V100 GPU. Furthermore, LTP exhibits significantly better robustness and consistency over different input sequence lengths.

REFERENCES

[1] https://github.com/kssteven418/ltp, 2021.

[2] Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jing Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. Binarybert: Pushing the limit of bert quantization. *arXiv preprint arXiv:2012.15701*, 2020.

[3] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[4] Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram Saletore. Efficient 8-bit quantization of transformer neural machine language translation model. *arXiv preprint arXiv:1906.00532*, 2019.

[5] Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. Efficient and effective sparse lstm on fpga with bank-balanced sparsity. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 63–72, 2019.

[6] Daniel Cer, Mona Diab, Eneko Agirre, Inigo Lopez-Gazpio, and Lucia Specia. Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation. *arXiv preprint arXiv:1708.00055*, 2017.

[7] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained BERT networks. *arXiv preprint arXiv:2007.12223*, 2020.

[8] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[9] Ido Dagan, Oren Glickman, and Bernardo Magnini. The pascal recognising textual entailment challenge. In *Machine Learning Challenges Workshop*, pages 177–190. Springer, 2005.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

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

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

[13] Angela Fan, Pierre Stock, Benjamin Graham, Edouard Grave, Rémi Gribonval, Hervé Jégou, and Armand Joulin. Training with quantization noise for extreme model compression. *arXiv e-prints*, pages arXiv–2004, 2020.

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

[15] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[16] Saurabh Goyal, Anamitra Roy Choudhury, Saurabh Raje, Venkatesan Chakaravarthy, Yogish Sabharwal, and Ashish Verma. PoWER-BERT: Accelerating BERT inference via progressive word-vector elimination. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3690–3699. PMLR, 13–18 Jul 2020.

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

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] Forrest N Iandola, Albert E Shaw, Ravi Krishna, and Kurt W Keutzer. Squeezebert: What can computer vision teach nlp about efficient neural networks? *arXiv preprint arXiv:2006.11316*, 2020.

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

[21] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.

[22] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pages 5156–5165. PMLR, 2020.

[23] Gyuwan Kim and Kyunghyun Cho. Length-adaptive transformer: Train once with length drop, use anytime with search. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 6501–6511. Association for Computational Linguistics, August 2021.

[24] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. I-bert: Integer-only bert quantization. *International conference on machine learning*, 2021.

[25] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[26] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[27] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

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

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

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

[31] Ofir Press, Noah A Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. *arXiv preprint arXiv:2108.12409*, 2021.

[32] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[33] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

[34] Hassan Sajjad, Fahim Dalvi, Nadir Durrani, and Preslav Nakov. Poor man's BERT: Smaller and faster transformer models. *arXiv preprint arXiv:2004.03844*, 2020.

[35] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

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

[37] 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. In *AAAI*, pages 8815–8821, 2020.

[38] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y 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*, pages 1631–1642, 2013.

[39] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for bert model compression. *arXiv preprint arXiv:1908.09355*, 2019.

[40] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. *arXiv preprint arXiv:2004.02984*, 2020.

[41] Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. Distilling task-specific knowledge from BERT into simple neural networks. *arXiv preprint arXiv:1903.12136*, 2019.

[42] Yi Tay, Dara Bahri, Liu Yang, Donald Metzler, and Da-Cheng Juan. Sparse sinkhorn attention. In *International Conference on Machine Learning*, pages 9438–9447. PMLR, 2020.

[43] Mart van Baalen, Christos Louizos, Markus Nagel, Rana Ali Amjad, Ying Wang, Tijmen Blankevoort,

and Max Welling. Bayesian bits: Unifying quantization and pruning. *Advances in neural information processing systems*, 2020.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[45] 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. *arXiv preprint arXiv:1905.09418*, 2019.

[46] Apoorv Vyas, Angelos Katharopoulos, and François Fleuret. Fast transformers with clustered attention. *Advances in Neural Information Processing Systems*, 33, 2020.

[47] 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. *arXiv preprint arXiv:1804.07461*, 2018.

[48] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. *arXiv preprint arXiv:2012.09852*, 2020.

[49] Sinong Wang, Belinda Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

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

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

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

[53] Deming Ye, Yankai Lin, Yufei Huang, and Maosong Sun. TR-BERT: Dynamic token reduction for accelerating BERT inference. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5798–5809. Association for Computational Linguistics, June 2021.

[54] Haonan Yu, Sergey Edunov, Yuandong Tian, and Ari S Morcos. Playing the lottery with rewards and multiple languages: lottery tickets in rl and nlp. *arXiv preprint arXiv:1906.02768*, 2019.

[55] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 811–824. IEEE, 2020.

[56] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8BERT: Quantized 8bit bert. *arXiv preprint arXiv:1910.06188*, 2019.

[57] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *arXiv preprint arXiv:2007.14062*, 2020.

[58] Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. Ternarybert: Distillation-aware ultra-low bit bert. *arXiv preprint arXiv:2009.12812*, 2020.

[59] Mengjie Zhao, Tao Lin, Fei Mi, Martin Jaggi, and Hinrich Schütze. Masking as an efficient alternative to finetuning for pretrained language models. *arXiv preprint arXiv:2004.12406*, 2020.

APPENDIX

A. Experimental Details

1) Datasets: We use eight monolingual (English) GLUE tasks [47] for evaluation, which include sentence similarity (QQP [20], MRPC [11], STS-B [6]), sentiment classification (SST-2 [38]), textual entailment (RTE [9]) and natural language inference (MNLI [52], QNLI [32]). There are 364k, 4k, 6k, 67k, 3k, 392k, 105k training examples, respectively. 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.

2) Environments: We use PyTorch 1.8 for training LTP on various GPUs. 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.

3) Training details: The training procedure of LTP consists of two separate stages: 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 IV-D with the threshold of the final layer 0.01. We search the optimal temperature T in a search space of $\{1e-4, 2e-4, 5e-4, 1e-3, 2e-3\}$ and vary the λ from 0.001 to 0.2 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 10 epochs of training with the hard pruning to fine-tune the model parameters only.

SpAtten was 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 $\{5e-6, 1e-5, 2e-5\}$, except for the soft pruning stage of LTP where we use $2e-5$. We follow the optimizer setting in RoBERTa [27] and use batch size of 64 for all experiments.

LAT was 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.

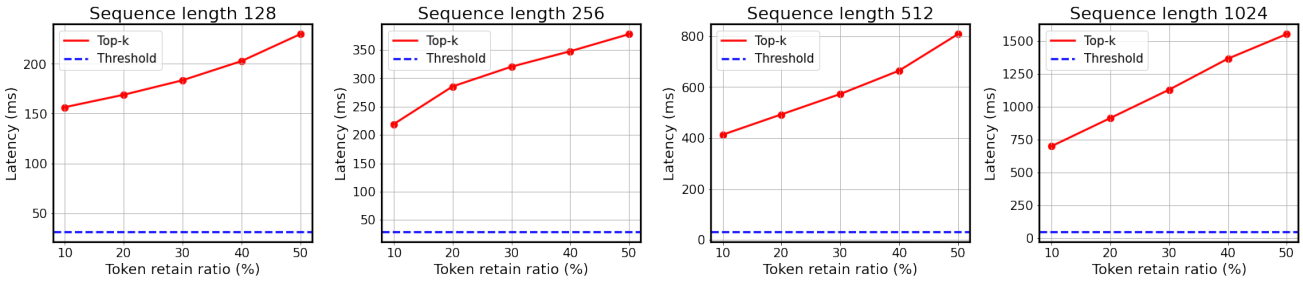


Fig. A.6: *Wall-clock latency comparison between top-k operation and threshold operation on an Intel Haswell CPU for different sequence length across various token retain ratios. Note that the latency of a threshold operation is independent of sequence length.*

B. 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. Furthermore, 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 10% of tokens from the input sentence by top-k).

With the above setting, we directly measure the latency of these two operations on an Intel Haswell CPU, and report the results in Figure A.6. 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]

C. LTP with Integer Quantization

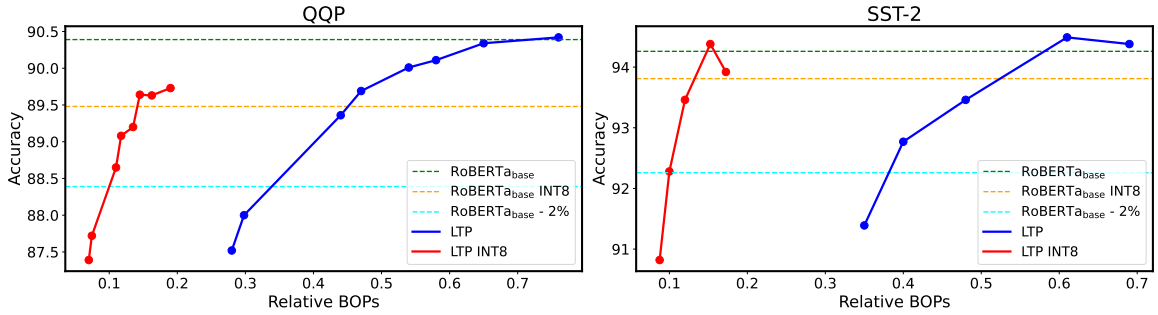


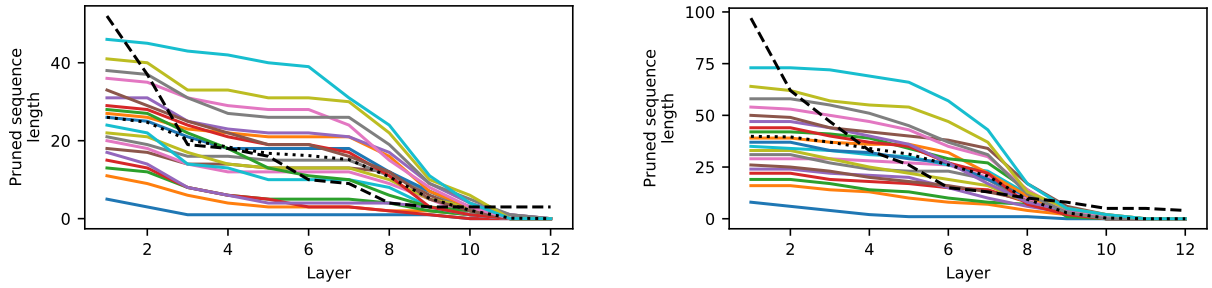
Fig. A.7: *Accuracy and relative BOPs of the FP16 and INT8* LTP *models on the QQP and SST-2 dataset. Note that FP16 unpruned RoBERTa$_{base}$ is used as the baseline. Thus, INT8 quantization of NN translates to* 4× *reduction in relative BOPs.*

1) Quantization method: We use the static uniform symmetric integer quantization method [15], which is easy to deploy in commodity hardware and incurs minimal run-time overhead. All the NN parameters are quantized to 8-bit integers, except for those of the embedding layer whose bitwidth does not affect the inference speed.

2) Knowledge distillation method: We set the RoBERTa$_{base}$ model as the teacher and the quantized LTP model as the student. The knowledge of the teacher model is distilled from both the output logits of the classification layer and the output representations of the embedding layer. The training objective is a convex combination of the original loss and the knowledge distillation loss.

3) Results: Figure A.7 shows the accuracy and relative BOPs of the FP16 and INT8 LTP models. We set FP16 RoBERTa$_{base}$ as the baseline because it is well known that FP32 classification models can be fit to FP16 with negligible accuracy degradation. The result demonstrates that LTP is robust to quantization methods, leading to up to 10× BOPs reduction with $< 2\%$ accuracy drop compared to the unpruned baseline.

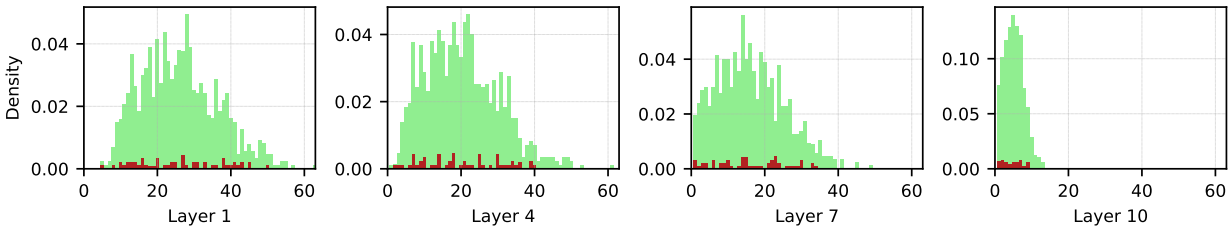
D. Discussion



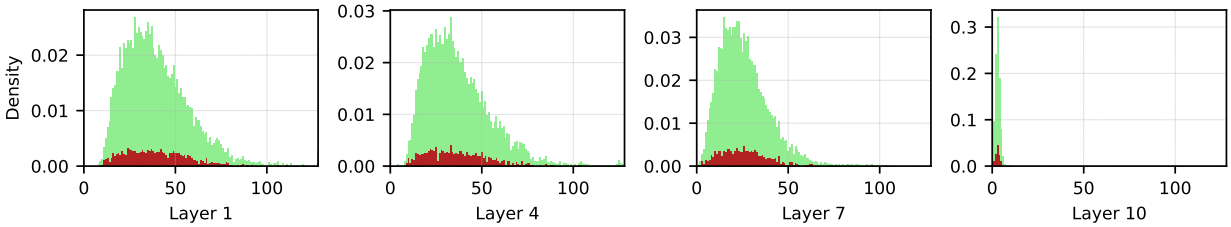
(a) *SST-2* **(b)** *MNLI-m*

Fig. A.8: *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.*

[3]The inefficiency of top-k is also further confirmed by [48], where they report only 1.1× speedup for GPT-2 without the top-k hardware engine that they developed.



(a) *SST2*



(b) *MNLI-m*

Fig. A.9: *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.*

1) Example Sequence Length Trajectories: Figure A.8 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.

2) Unbiased Token Pruning for Various Sequence Length: Figure A.9 shows the distributions of initial sequence lengths for sequences that were correctly classified and for sequences that were 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.

E. Comparison with TR-BERT on GLUE

Unlike LAT and SpAtten, TR-BERT [53] did not report results on the GLUE benchmark tasks described in Section A3. 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 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.

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