Learned Token Pruning for Transformers

Sehoon Kim∗,1, Sheng Shen∗,1, David Thorsley∗,2, Amir Gholami∗,1, Joseph Hassoun2, Kurt Keutzer1
1University of California, Berkeley, 2Samsung Semiconductor, Inc.
{sehoonkim, sheng.s, amirgh, keutzer}@berkeley.edu, {d.thorsley, j.hassoun}@samsung.com

Abstract—A major challenge in deploying transformer models is their prohibitive inference cost, which quadratically scales with the input sequence length. This makes it especially difficult to use transformers for processing long sequences. To address this, we present a novel Learned Token Pruning (LTP) method that reduces redundant tokens as the data passes through the different layers of the transformer. In particular, LTP prunes tokens with an attention score below a threshold value, which is learned during training. Importantly, our threshold based method avoids algorithmically expensive operations such as top-k token selection which are used in prior token pruning methods, and also leads to structured pruning. We extensively test the performance of our approach on multiple GLUE tasks and show that our learned threshold based method consistently outperforms the prior state-of-the-art top-k token based method by up to ∼2% higher accuracy with the same amount of FLOPs. Furthermore, our preliminary results show up to 1.4× and 1.9× throughput improvement on Tesla T4 GPU and Intel Haswell CPU, respectively, with less than 1% of accuracy drop (and up to 2.1× FLOPs reduction). Our code has been developed in PyTorch and has been open-sourced [1].

I. INTRODUCTION

Transformer-based deep neural network (DNN) models [46], such as BERT [10] and RoBERTa [30] have demonstrated ability to achieve state-of-the-art results in various Natural Language Processing (NLP) tasks, such as sentence classification and question answering. However, efficiently deploying these models has become increasingly challenging due to the large size of these models, the need for real-time inference, and the limited resources (energy, compute, memory) available.

The core mechanism at the heart of a transformer layer is multi-head self-attention. In this mechanism, each position in the input sequence attends to every other position to compute a new representation of the sequence. Because all positions attend to each other, the complexity of the computation increases quadratically with input sequence size and the ability to apply transformer models to long input sequences becomes limited.

One promising approach to address this is pruning, which 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 theoretically produce significant computational savings while preserving network accuracy. However, in practice, commodity neural network accelerators cannot efficiently exploit unstructured sparsity patterns. Despite not generating as much sparsity as unstructured methods, structured pruning methods are typically preferred in practice due to their relative ease of deployment to hardware.

The multi-head self-attention mechanism admits several structures for pruning; for example, head pruning [31] decreases the size of the model by decreasing the number of heads in the mechanism when they are not needed. Another, orthogonal, approach to pruning that we consider in this work is token pruning, which reduces the amount of 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 in order to make a successful inference.

Existing token pruning methods include SpAtten [49], which uses top-k pruning engine to select the k most important tokens in each transformer layer to preserve for later layers. A similar approach is taken in PoWER-BERT [17], which uses a learnable retention configuration that determines the number of tokens to preserve in each transformer layer. Both of these methods specify an exact number (or ratio) of tokens to be pruned at each layer during inference. This requires computing the top-k most (or least) important tokens in each layer for each inference, which is difficult to do efficiently without developing specialized hardware like in SpAtten.

Furthermore, a more fundamental issue with pruning
Fig. 1: Distributions of processed input sequence lengths from the validation sets for representative tasks in the GLUE benchmark: (a) MNLI; (b) QQP; (c) SST-2.

a fixed number of tokens per sequence (as done in top- \( k \) based methods [17, 49]) is that input sentences can have varying lengths. Figure 1 shows the distribution of input sequence lengths for several tasks in the GLUE benchmark. It is apparent that the distributions of sequence lengths vary both within tasks and across tasks. We would expect that we could prune tokens more aggressively for longer sentences and less aggressively for shorter sentences, instead of keeping a fixed number of tokens regardless of sequence length.

To address the above challenges, in this work we propose a learned threshold based token pruning method which does not require any additional hardware support and only uses structured sparsity. Our contributions are:

- We propose a threshold based token pruning method which does not require collecting any extra statistics from the data and only needs a simple comparison operation to detect unimportant tokens. This is an important advantage over costly top- \( k \) based pruning methods, which require computing top- \( k \) tokens during the run-time. In particular, our empirical results show that a threshold operation can be up to 7.4 \( \times \) and 33.4 \( \times \) faster than a top- \( k \) operation for sequence length of 128 and 1024, respectively. See Section III-B for details.

- One drawback of the threshold based method is that it requires manually setting the threshold values for different blocks, which can vary for different NLP tasks. To address this, we additionally propose a Learned Token Pruning (LTP) strategy which fully automates our threshold pruning framework 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 right thresholds for different transformer blocks and adapted for different tasks. See Section III-C for details.

- We apply LTP to RoBERTa and extensively evaluate its performance on GLUE tasks. We show that LTP achieves 2.10\( \times \) FLOPs reduction compared to the baseline within only 1% accuracy degradation. This allows up to 1.4\( \times \) and 1.9\( \times \) throughput improvement on Tesla T4 GPU and Intel Haswell CPU, respectively. Additionally, we show that LTP consistently outperforms the state-of-the-art top- \( k \) based token pruning methods, achieving up to 2\% accuracy improvement for the same amount of FLOPs. See Section IV-B and IV-D for details.

II. RELATED WORK

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, 21, 24, 26, 27, 36, 45, 47, 50, 60]; (ii) knowledge distillation [23, 38, 42–44]; (iii) quantization [2, 4, 13, 15, 25, 40, 58, 59, 61]; and (iv) pruning. Pruning methods can be further categorized into unstructured pruning [12, 28, 33, 39, 54] and structured pruning [17, 18, 20, 29, 31, 49, 56, 57, 62].

In NLP, for unstructured pruning, the lottery-ticket hypothesis [16] has been explored for transformers in [7, 34, 55]. Recently, [63] pre-trained transformers can leverage pruning as an effective way to fine-tune on downstream tasks. [39] proposes movement pruning, which achieves significant performance improvements versus magnitude-based methods in high sparsity regimes by considering the weights changing during fine-tuning.

For structured pruning in NLP, [5, 32, 52] first explore the row and block-wise pruning for LSTM models [19]. [14, 37] use LayerDrop to train a model that will have adaptive depth during training and drop unimportant layers during fine-tuning. This work demonstrates that shallow but effective models can be extracted from the pre-trained transformers. [31] drops different attention heads in the multi-head attention component of the transformer.
and shows many attention heads can be redundant. [51] approximates the weight matrices in transformers using low-rank factorization and iteratively drops the rank-1 weight values during training. Recently, besides the transformer architecture, the input of the transformer is pruned in [17, 49]. This method is denoted as token-pruning, where only top-$k$ tokens are kept at each block. However, one important drawback of these methods is their reliance on the algorithmically inefficient top-$k$ operator. To address this, we propose a novel structured, token pruning method that does not require any top-$k$ calculation, and yet achieves higher accuracy at lower FLOPs. Furthermore, the only operation that our method requires is threshold (i.e., comparison) operation which is efficient and does not require any dedicated hardware logic.

### III. METHODOLOGY

In this section, we briefly 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 [46] 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 $W_{k}^{(h,l)}$, $W_{q}^{(h,l)}$, $W_{v}^{(h,l)} \in \mathbb{R}^{d_h \times d}$, $W_{o}^{(h,l)} \in \mathbb{R}^{d \times d_h}$, 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 essentially measures the pairwise importance of each token on every other token in the input:

$$MHA(x) = \sum_{h=1}^{N_h} \text{Att}_{W_{k,q,v,o}^{(h)}}(x),$$

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

$$\text{Att}_{W_{k,q,v,o}}(x) = \sum_{i=1}^{n} \text{softmax}\left(\frac{x^T W_{q} W_{k} x_i}{\sqrt{d}}\right) W_{v} x_i,$$

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

$$\text{FFN}(x_{\text{MHA}}) = \sigma(W_2(W_1 x_{\text{MHA}} + b_1)) + b_2,$$

$$x_{\text{out}} = \text{LN}(\text{FFN}(x_{\text{MHA}}) + x_{\text{MHA}}),$$

where $W_1, W_2, b_1$ and $b_2$ are the FFN parameters, and $\sigma$ is the activation function (which is typically GELU for BERT).
B. Threshold Token Pruning

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

$$A^{(h,l)}(x_i, x_j) = \text{softmax}(\frac{x^T W_q^T W_k x}{\sqrt{d}})_{(i,j)} \in \mathbb{R}. \quad (6)$$

The cost of computational complexity for computing the attention matrix is $O(d^2 n + n^2 d)$, which quadratically scales with sequence length. As such, the attention operation becomes a bottleneck when applied to large sentences. One promising method to address this is to 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 need to define a metric to determine unimportant tokens. One approach for doing this is to compute the importance score of token $x_i$ in layer $l$ [17, 49]:

$$s^{(l)}(x_i) = \frac{1}{N_h n} \sum_{h=1}^{N_h} \sum_{j=1}^{n} A^{(h,l)}(x_i, x_j). \quad (7)$$

Intuitively, the attention probability $A^{(h,l)}(x_i, x_j)$ can be interpreted as the normalized amount that all the other tokens $x_j$ attend to token $x_i$. Therefore, token $x_i$ can be considered important if it receives more attention from all tokens across all heads, which directly leads us to equation 7. The procedure of computing importance score from attention probability is illustrated in Figure 2 (Right).

In [17, 49], tokens are ranked according to their importance scores and are pruned based on a top-$k$ selection strategy. Specially, token $x_i$ will be pruned at layer $l$ if it has the important score $s^{(l)}(x_i)$ 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 and can be expensive without specialized hardware; we provide further empirical results showing this in Section IV-E.

To address this, 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 $M^{(l)} : \{1, \ldots, n\} \rightarrow \{0, 1\}$ which indicates whether a token should be kept or pruned:

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

In other words, the 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 the computation complexity in bottom layers (aka cascade token pruning). This process is illustrated in Figure 3 along with the expensive top-$k$ method.

C. Learnable Threshold for Token Pruning

A key concern with the naïve method above is how to determine the threshold values for each layer. Not only do the threshold values changes for different layers, they also vary for different tasks. This can be addressed by
making the masking function (i.e., $M$ in Eq. 8) learnable. However, there are several technical challenges that need to be considered. First, due to the binary nature of $M$ there is no gradient flow for the pruned tokens. Second, the $M$ operator is non-differentiable which prevents gradient flows into thresholds. Furthermore, note that we cannot apply the conventional Straight-Through Estimator (STE) \cite{bengio2013estimating} to estimate the gradients with respect to the thresholds. In addition, because the token gets near zero and thus has little impact on the threshold by a sufficient margin, its layer output activation $l$ becomes near zero and thus has little impact on the threshold value for layer $l$. That is to say, $s^{(l+1)}(x_i) = 0$, it is most likely to be pruned again. Therefore, the behavior of the soft pruning scheme is nearly identical to that of the hard pruning scheme, yet its differentiable form allows the use of backpropagation and gradient based optimizations to make $\theta$ learnable. After jointly training the model parameters and the 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.

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

$$
\tilde{M}^{(l)}(x_i) = \sigma \left( \frac{s^{(l)}(x_i) - \theta^{(l)}}{T} \right)
$$

where $T$ is a temperature parameter, and $\theta^{(l)}$ is the learnable threshold value for layer $l$. With a sufficiently large temperature value $T$, $\tilde{M}^{(l)}(x_i)$ will closely approximate the hard masking $M^{(l)}(x_i)$ 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 to the output activation of layer $l$. That is to say,

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

where $x^{(l)}_{\text{MHA}}$ is the output activation of MHA in layer $l$. If the importance score of token $x_i$ is below the threshold by a sufficient margin, its layer output activation becomes near zero and thus has little impact on the succeeding layer. In addition, because the token gets a zero importance score in the succeeding layer, i.e., $s^{(l+1)}(x_i) = 0$, it is most likely to be pruned again.

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

**Input:** $M$: model finetuned on target downstream task

**Step 1:** Apply soft mask to $M$ and train both the thresholds and model parameters $\triangleright$ Soft Pruning

**Step 2:** Binarize the mask and fix the thresholds $\triangleright$ Hard Pruning

**Step 3:** Finetune the model parameters $\triangleright$ Hard Pruning

The pseudo-code for this three-step algorithm is given in Algorithm 1. Intuitively, the magnitude of gradient $d\tilde{M}^{(l)}(x_i)/d\theta^{(l)}$ is maximized when the importance score $s^{(l)}(x_i)$ 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 $\theta$ to prune the network without regularization, as the optimizer generally gets a better loss value if all the tokens are present. As such, we need to add a regularization term to penalize the network if many tokens are left unpruned. This can be achieved by imposing an L1 loss on the masking operator $\tilde{M}$:

$$
\mathcal{L}_{\text{new}} = \mathcal{L} + \lambda \mathcal{L}_{\text{reg}} \quad \text{where} \quad \mathcal{L}_{\text{reg}} = \frac{1}{L} \sum_{l=1}^{L} ||\tilde{M}^{(l)}(x)||_1. \tag{13}
$$
Table I: Detailed performance and efficiency comparison of LTP applied to RoBERTa\textsubscript{base}

| Model         | MNLI-\textsubscript{m} | MNLI-\textsubscript{mm} | QQP  | QNLI  | SST-2 | STS-B | MRPC | RTE  |
|---------------|-------------------------|--------------------------|------|-------|-------|-------|------|------|
| Accuracy      | RoBERTa\textsubscript{base} | 87.53                    | 87.36| 90.39 | 92.86 | 94.27 | 90.89| 92.14| 77.98|
| LTP           | 86.53                   | 86.37                    | 89.69| 91.98 | 93.46 | 90.03 | 91.59| 77.98|
| GFLOPs        | RoBERTa\textsubscript{base} | 6.83                     | 7.15 | 5.31  | 8.94  | 4.45  | 5.53 | 9.33 | 11.38|
| LTP           | 3.64                    | 3.63                     | 2.53 | 4.77  | 2.13  | 2.84  | 4.44 | 6.30 |

| Speedup       | LTP                     | 1.88×                     | 1.97× | 2.10× | 1.87× | 2.09× | 1.95× | 2.10× | 1.81× |

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

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

Note that if there are more tokens near the threshold, then the gradient $d\mathcal{L}_{\text{reg}}/d\theta^{(l)}$ will be larger. As a result, the threshold will be pushed to a larger value to prune more tokens from the threshold boundary.

IV. EXPERIMENTS

A. Experiment Setup

We implemented LTP on RoBERTa\textsubscript{base} [30] using HuggingFace’s transformers repo\footnote{https://github.com/huggingface/transformers/} and test on monolingual (English) GLUE tasks [48]. See A1 for the details of the datasets used. As mentioned in Section III-C, the training procedure of LTP consists of two separate stages: soft pruning that trains both the model parameters and the thresholds on downstream tasks, followed by hard pruning that fine-tunes the model parameters with the fixed thresholds. We also compare LTP with the current state-of-the-art top-$k$ based token pruning method of SpAtten [49] following the implementation details in their paper. We simply denote this baseline as top-$k$. See A2 for training details of LTP and the baseline. We use a Tesla T4 GPU and an Intel Haswell CPU with 3.75GB memory as target GPU and CPU hardware for latency measurement, respectively. We use PyTorch 1.8, CUDA 11.1 and cuDNN 8.0 for all experiments.

A crucial point to mention is that we refrain from padding all input sentences to nearest powers of 2 as done in [17]. While adding extra padding definitely leads to good results table for speedup, it may not be a good metric to measure the effectiveness of a pruning method. For instance, the SST-2 task on average only includes 26 tokens for inputs, and thus padding the input to 64 as done in prior work would result in $\sim 2.5 \times$ pruning without a significant impact on the accuracy. As such, we set the sequence length of an individual input batch as the maximum length of the sentences it contains, thereby minimizing the number of extra token padding.

B. Performance Evaluation

Table I demonstrates the accuracy and GFLOPs for our learnable threshold token pruning method. We select a model for each downstream task that achieves the smallest GFLOPs while constraining the accuracy degradation from the baseline, i.e., RoBERTa\textsubscript{base}, to be at most 1 percentage point. Under our token pruning method, sequence lengths in each layer can vary across different input sentences. Therefore, here we report the averaged GFLOPs of processing all input sentences in development sets. As shown in the table, our token pruning method achieves speedup of 1.97× on average and up to 2.10× with a minimal accuracy degradation of <1%.

Figure 5 further demonstrates the accuracy of models with different FLOPs (blue lines). We also plot the accuracy of the top-$k$ pruning method for comparison (red lines). We see that our learnable threshold approach consistently outperforms the top-$k$ approach for all tasks by up to $\sim 2\%$ higher accuracy with the same amount of FLOPs. We particularly observe a noticeable performance gap around 60% of the relative FLOPs. We highlight that our learnable threshold approach not only achieves better accuracy than the top-$k$ approach, but it has additional gain from avoiding computationally inefficient top-$k$ operations, which will be further discussed in Section IV-E.

C. Ablation Studies

Instead of learning the thresholds, we can consider setting them manually. Because manually searching over the exponential search space is intractable, a feasible strategy is to add a constraint to the search space by assigning linearly raising threshold values for each layer,
Fig. 5: 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 base 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 is dotted in a horizontal line for comparison.

similar to the way that SpAtten [49] assigns the token retain ratios. That is to say, 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 5 as black lines. While the manual threshold approach exhibits decent results on all downstream tasks, the learned thresholds consistently outperforms the manual thresholds under the same FLOPs in most cases. This provides empirical evidence for the effectiveness of our threshold learning method.

D. Throughput Evaluation on Real Hardware

We also directly measure throughput improvement of LTP by evaluating its performance on Tesla T4 GPU and Intel Haswell CPU. We evaluate on the validation datasets of QQP and MRPC tasks, using the models obtained in IV-B for each task. We use both a randomly ordered dataset and a sorted dataset for evaluating LTP, where the sorting is performed based on the initial sequence lengths. Notice that we can expect a higher throughput using the sorted dataset as it dispatches batches with uniform sequence lengths which require a fewer number of extra token padding.

Table II demonstrates the throughput of LTP as compared to the baseline throughput of RoBERTa_base. As shown in the table, the effectiveness of LTP becomes more evident with larger batch sizes, thereby achieving throughput improvement of up to \( 1.39 \times \) and \( 1.41 \times \) on a GPU, and \( 1.79 \times \) and \( 1.86 \times \) on a CPU for QQP and MRPC, respectively. We attribute the cause of lower relative throughput at smaller batch sizes to hardware under-utilization, in which the gain from smaller matrix operations is marginal. In addition, we observe greater throughput improvement on a CPU, where it is more resource constrained and thus reducing the size of operations is more critical. With sorted datasets, LTP achieves more drastic throughput improvement of up to \( 1.81 \times \) and \( 1.57 \times \) on a GPU, and \( 3.10 \times \) and \( 2.42 \times \) on a CPU for QQP and MRPC, respectively. These numbers can be interpreted as the optimal relative throughput which can be achieved with more uniform sentence lengths.

E. 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.,
Table II: Throughput (sentences/sec) of LTP applied to RoBERTa\textsubscript{base} processing the QQP and MRPC tasks on a Tesla T4 GPU and Intel Haswell CPU. LTP and LTP\textsuperscript{†} are evaluated using randomly ordered and sorted datasets, respectively. Notice that the sorted datasets allow the sequence lengths in a same batch to be uniform.

| Model        | GPU (Batch Size) | CPU (Batch Size) |
|--------------|------------------|------------------|
| LTP          | 120.3            | 7.5              |
| LTP\textsuperscript{†} | 118.6            | 10.6             |

Relative

| Throughput   | LTP              | LTP\textsuperscript{†} |
|--------------|------------------|------------------------|
| Throughput   | 0.87\times 1.02  | 0.86\times 1.04       |
| Throughput   | 1.02\times 1.20  | 1.04\times 1.26       |
| Throughput   | 1.38\times 1.39  | 1.66\times 1.81       |
| Throughput   | 1.17\times 1.70  | 1.87\times 2.23       |
| Throughput   | 1.76\times 1.79  | 2.47  \times 3.10     |
| Throughput   | 0.90\times 1.11  | 0.95\times 1.13       |
| Throughput   | 1.29\times 1.34  | 1.50\times 1.57       |
| Throughput   | 1.38\times 1.41  | 1.61\times 1.86       |
| Throughput   | 1.74\times 1.86  | 2.18\times 2.29       |

Fig. 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.

V. DISCUSSION

Figure 7 shows the relationship between the initial sequence length and the pruned sequence length for three layers within RoBERTa\textsubscript{base}. The dotted line shows the pruned sequence length for top-$k$ pruning of [49] when $k$ is chosen to equalize the number of FLOPs between the two approaches. Note that top-$k$ pruning of [49] does not prune any tokens for the first three layers. However, LTP automatically detects if tokens can be pruned for these layers. Furthermore, we can see that for later layers, LTP adapts the pruning to individual sequences. These adaptations allow our method to maintain greater accuracy than top-$k$ at an equal computation cost. See Section B for further discussion on the behavior of LTP.
VI. CONCLUSIONS

In this work, we present a Learned Token Pruning (LTP) method for BERT which is a fully automated structured pruning framework which only requires comparison of token importance with a static threshold value, which can be learned during training. The latter is achieved by introducing a differentiable soft binarized mask, which enables backpropagation to the pruning threshold values. Comparing to the state-of-the-art top-k based token pruning method, LTP outperforms it by up to 2% accuracy with the same amount of FLOPs. Extensive experiments on GLUE show the effectiveness of LTP that achieves up to 2.10× FLOPs reduction as compared to the baseline model within only 1% of accuracy degradation. Our preliminary (and not highly optimized) implementation shows up to 1.4× and 1.9× throughput improvement on a Tesla T4 GPU and an Intel Haswell CPU. Future works may include integrating LTP with orthogonal state-of-the-art structured pruning and quantization works for efficient BERT deployment.

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
[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] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. Learned step size quantization. arXiv preprint arXiv:1902.08153, 2019.

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

[13] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. arXiv preprint arXiv:1803.03635, 2018.
based on connection sensitivity. \textit{arXiv preprint arXiv:1810.02340}, 2018.

[29] Shaohui Lin, Rongrong Ji, Yuchao Li, Yongjian Wu, Feiyue Huang, and Baocchang Zhang. Accelerating convolutional networks via global & dynamic filter pruning. In \textit{IJCAI}, pages 2425–2432, 2018.

[30] 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. \textit{arXiv preprint arXiv:1907.11692}, 2019.

[31] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? \textit{arXiv preprint arXiv:1905.10650}, 2019.

[32] Sharan Narang, Eric Undersander, and Gregory Diamos. Block-sparse recurrent neural networks. \textit{arXiv preprint arXiv:1711.02782}, 2017.

[33] Sejun Park, Jaejo Lee, Sangwoo Mo, and Jinwoo Shin. Lookahead: a far-sighted alternative of magnitude-based pruning. \textit{arXiv preprint arXiv:2002.04809}, 2020.

[34] Sai Prasanna, Anna Rogers, and Anna Rumshisky. When BERT plays the lottery, all tickets are winning. \textit{arXiv preprint arXiv:2005.00561}, 2020.

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

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

[37] Hassan Sajjad, Fahim Dalvi, Nadir Durrani, and Preslav Nakov. Poor man’s BERT: Smaller and faster transformer models. \textit{arXiv preprint arXiv:2004.03844}, 2020.

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

[39] Victor Sanh, Thomas Wolf, and Alexander M Rush. Movement pruning: Adaptive sparsity by fine-tuning. \textit{arXiv preprint arXiv:2005.07683}, 2020.

[40] 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 \textit{AAAI}, pages 8815–8821, 2020.

[41] 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 \textit{Proceedings of the 2013 conference on empirical methods in natural language processing}, pages 1631–1642, 2013.

[42] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for bert model compression. \textit{arXiv preprint arXiv:1908.09355}, 2019.

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

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

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

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

[47] Apoorv Vyas, Angelos Katharopoulos, and François Fleuret. Fast transformers with clustered attention. \textit{Advances in Neural Information Processing Systems}, 33, 2020.

[48] 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. \textit{arXiv preprint arXiv:1804.07461}, 2018.

[49] Hanrui Wang, Zhekai Zhang, and Song Han. Spat-ten: Efficient sparse attention architecture with cascade token and head pruning. \textit{arXiv preprint arXiv:2012.09852}, 2020.

[50] Sinong Wang, Belinda Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. \textit{arXiv preprint arXiv:2006.04768}, 2020.

[51] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. \textit{arXiv preprint arXiv:1910.04732}, 2019.

[52] Wei Wen, Yuxiong He, Samyam Rajbhandari, Minjia Zhang, Wenhao Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning intrinsic sparse structures within long short-term memory. \textit{arXiv preprint arXiv:2004.06770}, 2020.
Adina Williams, Nikita Nangia, and Samuel R Bowman. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426*, 2017.

Xia Xiao, Zigeng Wang, and Sanguthevar Rajasekaran. Autoprun: Automatic network pruning by regularizing auxiliary parameters. In *Advances in Neural Information Processing Systems*, pages 13681–13691, 2019.

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.

Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9194–9203, 2018.

Shixing Yu, Zhewei Yao, Amir Gholami, Zhen Dong, Michael W Mahoney, and Kurt Keutzer. Hessian-aware pruning and optimal neural implant. *arXiv preprint arXiv:2101.08940*, 2021.

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.

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

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.

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.

Chenglong Zhao, Bingbing Ni, Jian Zhang, Qiwei Zhao, Wenjun Zhang, and Qi Tian. Variational convolutional neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2780–2789, 2019.

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 [48] for evaluation, which include sentence similarity (QQP [22], MRPC [11], STS-B [6]), sentiment classification (SST-2 [41]), textual entailment (RTE [9]) and natural language inference (MNLI [53], QNLI [35]). 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) 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-C with the threshold of the final layer 0.1. We search the optimal temperature $T$ in a search space of \{1e^{-4}, 2e^{-4}, 5e^{-4}, 1e^{-3}, 2e^{-3}\} and vary the $\lambda$ 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.

We train the top-$k$ baseline based on the implementation details in SpAtten [49] 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 the top-$k$ pruning method. For all experiments we use learning rate of \{5e^{-6}, 1e^{-5}, 2e^{-5}\}, except for the soft pruning stage where we use 2e^{-5}. We follow the optimizer setting in RoBERTa [30] and use batch size of 64 for all experiments.

B. Discussion

1) Unbiased Token Pruning for Various Sequence Length: Figure A.8 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.

2) Example Sequence Length Trajectories: Figure A.9 shows how the pruned sequence length decreases for input sequences of varying lengths. For top-$k$ pruning, the sequence length decreases linearly regardless of the initial sequence length or sequence content. In LTP, token pruning can be more or less aggressive depending on the sequence content and the number of important tokens in the sequence.
**Fig. A.8:** 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.

**Fig. A.9:** Sample trajectories of pruned sequence length as the sequences are passed through model layers. 20 samples were evenly selected from the sets after sorting by initial sequence length. (a) SST-2 with LTP. (b) MNLI-m with LTP. (c) SST-2 with top-k pruning. (d) MNLI-m with top-k pruning.