TPrune: Efficient Transformer Pruning for Mobile Devices

JIACHEN MAO, Duke University, United States
HUANRUI YANG, Duke University, United States
ANG LI, Duke University, United States
HAI LI, Duke University, United States
YIRAN CHEN, Duke University, United States

The invention of Transformer model structure boosts the performance of Neural Machine Translation (NMT) tasks to an unprecedented level. Many previous works have been done to make the Transformer model more execution-friendly on resource-constrained platforms. These researches can be categorized into three key fields: Model Pruning, Transfer Learning, and Efficient Transformer Variants. The family of model pruning methods are popular for their simplicity in practice and promising compression rate and have achieved great success in the field of convolution neural networks (CNNs) for many vision tasks. Nonetheless, previous Transformer pruning works did not perform a thorough model analysis and evaluation on each Transformer component on off-the-shelf mobile devices. In this work, we analyze and prune transformer models at the line-wise granularity and also implement our pruning method on real mobile platforms. We explore the properties of all Transformer components as well as their sparsity features, which are leveraged to guide Transformer model pruning. We name our whole Transformer analysis and pruning pipeline as TPrune. In TPrune, we first propose Block-wise Structured Sparsity Learning (BSSL) to analyze Transformer model property. Then, based on the characters derived from BSSL, we apply Structured Hoyer Square (SHS) to derive the final pruned models. Comparing with the state-of-the-art Transformer pruning methods, TPrune is able to achieve a higher model compression rate with less performance degradation. Experimental results show that our pruned models achieve 1.16× - 1.92× speedup on mobile devices with 0% - 8% BLEU score degradation compared with the original Transformer model.

CCS Concepts: • **Computing methodologies** → *Neural networks*; **Machine translation**; **Regularization**; **Feature selection**; • **Computer systems organization** → **Embedded software**; • **Software and its engineering** → Software libraries and repositories.

Additional Key Words and Phrases: neural machine translation, neural networks, model pruning, embedded software, real-time system, mobile computing

ACM Reference Format:
Jiachen Mao, Huanrui Yang, Ang Li, Hai Li, and Yiran Chen. . TPrune: Efficient Transformer Pruning for Mobile Devices. 1, 1 (July), 22 pages.

Authors' addresses: Jiachen Mao, Duke University, Durham, United States, jiachen.mao@duke.edu; Huanrui Yang, Duke University, Durham, United States, huanrui.yang@duke.edu; Ang Li, Duke University, Durham, United States, ang.li630@duke.edu; Hai Li, Duke University, Durham, United States, hai.li@duke.edu; Yiran Chen, Duke University, Durham, United States, yiran.chen@duke.edu.

XXXX-XXXX//7-ART $15.00
https://doi.org/

1 INTRODUCTION

1.1 Motivation

In modern cyber-physical systems (CPS), the human-computer interaction plays an important role many Human-centered applications. As a result of that, Neural Machine Translation (NMT) acts as an input media for better human machine interface understanding in CPS. Transformer has emerged in the NMT tasks as a promising paradigm for sequence modeling [1]. Empowered by the stacking of attention and large feed forward layers, Transformer successfully achieves state-of-the-art performance in many NMT benchmarks. In recent years, the performance of many pre-trained attention-based models has been further enhanced by larger datasets as well as larger model sizes [2–4].

However, the ever-growing model size of the transformer models introduces tremendous memory consumption and computational cost, making the models hard to be deployed onto real-time CPS [5, 6]. Running the transformer models on remote servers may potentially cause latency, privacy, and security problems [7, 8]. Hence, efficiently executing transformer models on resource-constrained platforms becomes critical [9–12]. Among all the solutions, compressing the model size has been proven as an effective way to reduce the resource needs of transformer model execution. Current model compression techniques for Transformer models mainly fall into three categories – *Model Pruning*, *Transfer Learning*, and *Efficient Transformer Variants*.

Model Pruning method fine-tunes the original pre-trained model to force the weights [13–15] or activations [16–18] to be zeros as many as possible. A transformer model may be pruned at four different pruning granularity levels, say, layer-wise pruning [19], head-wise pruning [20, 21], line-wise pruning [22] and element-wise pruning [23].

Transfer Learning method aims at using the knowledge from a pre-trained large model to guide the training of a smaller model structure. In [24], Sanh *et al.* initially remove 1 out of every 2 layers of the teacher model to form the student model. Then, the student model is trained based on the KL-divergence of the logits from the student and teacher model. In [25], Sun *et al.* transfer the knowledge of the Transformer model using both the MSE of feature maps and the KL divergence of per-head self-attention distribution between the student and teacher models as the loss.

Efficient Transformer Variants method tries to substitute the original costly Transformer modules with more efficient operators. For example, in [26], Zhang *et al.* introduce average layer and gating layer to summarize history attention via a cumulative average operation over previous positions to replace the original self-attention scheme. In [27], Wu *et al.* introduce Long-Short Range Attention, which designates different heads for local heads modeling and long-distance relationship to broaden the functionalities of attention structure in Transformer.

Among all three model compression methods, model pruning is the most straightforward technique. Compared with transfer learning and efficient Transformer variants, model pruning neither requires extra human effort in model structure design nor needs to train the model from scratch.

1.2 Proposed Method

Our work falls into the category of model pruning for real-time cyber-physical system on mobile devices at line-wise pruning granularity, similar to [22]. Note that model pruning is orthogonal to other model compression methods. All the models, including original Transformer and Transformer variants, can be pruned with possibly accuracy tradeoff. Therefore, in this work, we compare our method with all the model pruning works on Transformer models. Different from these works where pruning is often performed heuristically, we thoroughly analyze all the Transformer components to find the best pruning scheme that can achieve a satisfactory sparsity-accuracy tradeoff. Moreover,

we quantitatively test the real inference speedup of our pruned Transformer models on resource-constrained mobile devices and report these first-hand results.

1.3 Contributions

In the CPS enabled by NMT, the security of the users and translation latency are two key challenges. In this work, we propose *TPrune*, a Transformer-based analyzing and pruning method for efficient deployment of Neural Machine Translation (NMT) applications on mobile devices. With *TPrune*, we are able to derive more efficient Transformer models for the local execution on embedding devices so as to enable better user-privacy and lower response latency in CPS. Specifically, the contributions of this work can be summarized as follows:

- We use Block-wise Structure Sparsity Learning (BSSL) to exploit the redundancy of the Transformer model through group lasso regularizer.
- We visualize the Transformer model pruned using BSSL and analyze the sparsity pattern of the pruned model. We then derive the pruning rules of each Transformer component.
- We extend Structured Hoyer Square (SHS) regularizer from Convolutional Neural Networks (CNNs) to Transformer models guided by the derived pruning rules.
- We evaluate the model performance, sparsity, and computational cost of the model pruned by *TPrune* on real mobile devices. We also compare the effectiveness of *TPrune* with other state-of-the-art works in the field of Transformer pruning [21] and pruning regularizer [14].

The remainder of our paper is organized as follows: In Section 2, we first give a brief introduction to Transformer architecture. Second, we profile the characteristics and computational costs of different neural network layers on mobile devices and present the challenge of Transformer deployments. Then, we summarize the related Transformer model pruning works and how these works related to our work. Finally, we illustrate the regularizers that are widely used for model pruning. In Section 3, we first present the objects of our analysis. Then, we introduce Block-wise Structure Sparsity Learning (BSSL) and use it to derive the pruning properties of Transformer models. In Section 4, we first adopt Structured Hoyer Square regularizer for Transformer line-wise pruning. Then, we discuss our pruning strategy of the Transformer model based on the guidance of BSSL-based analysis in Section 3. In Section 5, we give the experimental setup and results of our pruned Transformer models including the model sparsity, execution speedup, and overall performance of *TPrune*. Section 6 concludes the paper.

2 PRELIMINARY

2.1 Transformer Architecture

Fig. 1 depicts the architecture of Transformer model in [1]. Transformer is a sequence-to-sequence model which is composed of decoder (left) and encoder (right). Both decoder and encoder are made up of several identical Transformer blocks. The input and output of each block in Transformer have identical dimensions (d_{model}), which are the same as the embedding dimension (d_{embed}) of the input/output words. Following [1], in this work, we set d_{model} to 512. In each Transformer block, there exist two main components, which are Multi-Head Attention (MHA) and Feed Forward Network (FFN). The detail of these two components will be illustrated in Section 3. In [1], translation tasks are adopted in the experiment. Similarly, in this work, we also conducted our experiments on NMT English to German translation task.

2.2 Model Profiling on Mobile Devices

Although Transformer could be categorized as a deep neural network which is full of linear weight matrix that seems amiable to deep compression [13], the normalized weight magnitude

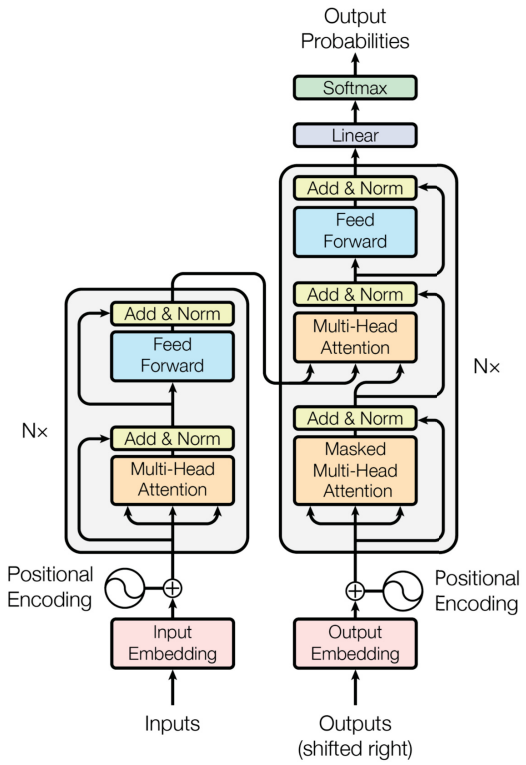

Fig. 1. Transformer Architecture in [1]

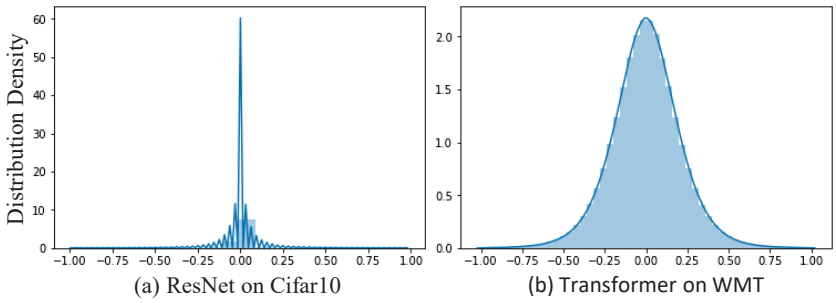

Fig. 2. Weight distribution of (a) ResNet and (b) Transformer

of Transformer models is much larger than that of CNNs due to the nature of the target tasks. Fig. 2 compares the weight distribution of Transformer on WMT dataset [28] and ResNet [29] on CIFAR-10 dataset [30]. Note that Transformer and ResNet are targeting at language processing and computer vision, respectively. For the ease of comparison, we visualize the weights from −1 to 1 for both models. Comparing with the ResNet in Fig. 2(b), the range of the distribution of the Transformer's weights is much wider, meaning that the weight magnitude of Transformer models is much larger than that of CNNs. Therefore, the pruning of Transformer faces many challenges compared to the pruning of CNNs, which has been proven in many previous works [20–23].

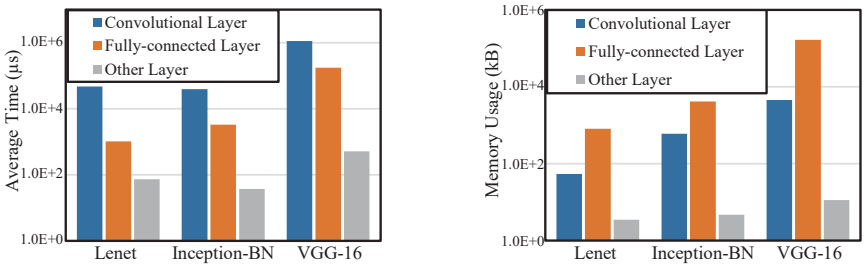

Fig. 3. Comparison between Fully-connected layers and Convolutional layers.

In order to understand the properties of different types of layers in deep neural networks, we deploy three representative models on Nexus5 [6, 31], which are Lenet [32], Inception-BN [29], and VGG-16 [33]. For each model, we obtained the breakdown of time consumption and memory cost of convolutional layers, fully-connected layers, and other layers. As shown in Fig. 3, fully-connected layers and convolutional layers consume most resources during the execution. More specifically, fully-connected layers are the most memory-intensive while convolutional layers are the most computation-intensive. For Transformer, both MHA and FFN are composed of fully-connected layers. Therefore, reducing the number of model parameters is critical to the deployment of Transformer on resource-constrained mobile devices.

2.3 Transformer Model Pruning

As mentioned in Section 1.1, many previous works have been done in the field of Transformer model compression. In the field of NMT tasks, besides efficient Transformer variance [26, 27, 34], Transformer pruning is the most popular topic for Transformer compression due to its simplicity and practicality [19–23]. These Transformer pruning works could be classified into the following types in terms of different pruning granularities.

Layer-wise pruning uses Transformer layers as the smallest pruning granularity. For example, in [19], Fan *et al.* propose LayerDrop, which selects sub-network of any depth from the original Transformer network during inference time. In LayerDrop, a drop rate is learned for each Transformer layer during training and only the highest-scored layers are adopted during testing.

Head-wise pruning uses attention heads in Transformer as the smallest pruning granularity. For example, in [20], Voita *et al.* apply gates on each head of MHA for pruning using stochastic relaxation. It prunes half of all heads in Transformer with less than 0.25 BLEU degradation. In [21], Michel *et al.* also mask the heads in MHA iteratively based on the proposed proxy score to evaluate the importance of each attention head.

Line-wise pruning uses rows or columns in Transformer layer matrices as the smallest pruning granularity. For example, in [22], Kenton *et al.* utilize auto-sizing to prune Transformer model weights with regularizer and proximal gradient descent. However, auto-sizing severely affects the performance of the pruned model. For example, the BLEU score drops from 27.9 to 20.9 when 20% of parameters are removed in Arabic to English translation task.

Element-wise pruning uses random weights as the smallest pruning granularity. For example, in Transformer.zip [23], Cheong *et al.* implement iterative magnitude pruning, which masks out the weights that are less than a certain threshold and iteratively retrain the model. Although element-wise pruning, in general, leads to a higher sparsity than other pruning methods, it achieves real speedup only under some specific hardware setting.

Moreover, BERT is a self-supervised approach to train a pre-trained model for many downstream language tasks with Transformer structure [2]. Many model compression works have also be done on BERT utilizing various technologies like transfer learning [24, 25], pruning [35–37], and quantization [38, 39].

TPrune falls into the category of line-wise pruning. In this work, we mainly focus on the speedup of the execution of NMT tasks of Transformer models using pruning methods when being deployed on resource-constrained platforms.

2.4 Model Pruning with Regularizer

When pruning a model, we could use a straightforward training loss: L_0 regularization of model weights. In such a case, L_0 represents the number of non-zero weights. However, L_0 is non-convex and indifferentiable. Hence, gradient-based methods are hard to apply to L_0 regularizer. In [40], Han *et al.* iteratively prune a fixed percentage of weights with the smallest magnitude. Such a heuristic method hardly achieves the optimal compression rate. As an alternative, L_1 regularizer is introduced. In [41], Liu *et al.* apply L_1 regularizer on weights to achieve element-wise sparsity. However, the drawback of L_1 is its scale-variance property, meaning that the gradient varies with respect to the scale of the original weight magnitude.

Structured pruning extends element-wise pruning to group-wise. By removing groups of weights from the original model weights, the memory usage locality can be preserved and the speedup on real computing systems can be achieved. In [14], Wen *et al.* group each column and row of the weight matrix using L_2 norm and minimize the L_1 of all these grouped weights. Similarly, in [22], Kenton *et al.* apply L_1 to the L_∞ of all the grouped weights, where the scale variance problem still holds. In [20], Gate method is proposed to apply on each head of MHA component in Transformer. The gate value is drawn independently from a parameterized concrete distribution. However, such a method incurs extra overheads in the gate approximation, making it less scalable.

2.5 Structured Hoyer Square (SHS)

In *TPrune*, we adopt Structured Hoyer Square (SHS) for structured Transformer model pruning [42]. Previously, the effectiveness of SHS on CNNs has been demonstrated in [42]. In this work, we will explore the challenges of applying SHS to Transformer models for sequence-to-sequence applications.

The key reason why we adopt Hoyer regularizer is because of its scale-invariant property, i.e., $R(\alpha X) = R(X)$, where Hoyer regularizer $R(X)$ equals:

$$R(X) = \frac{\sum_i |x_i|}{\sqrt{\sum_i x_i^2}}. \tag{1}$$

The minima structure of Hoyer regularizer is close to $L0$, which meets the intention of weight pruning. In order to align Equation (1)of scale $(0 - \sqrt{N})$ with $L0$ of range $(0 - N)$, Hoyer-Square (HS) regularizer is proposed for element-wise sparsity [42] as:

$$HS(W) = \frac{(\sum_i |w_i|)^2}{\sum_i w_i^2}. \tag{2}$$

In addition to scale-invariant, the HS regularizer is also differentiable. It has the same range and minima structure as the $L0$ norm. In the scenario of structured pruning, HS regularizer is turned into Structured Hoyer Square (SHS), which can be expressed as:

$$SHS(W) = \frac{(\sum_{g=1}^{G} ||w^{(g)}||_2)^2}{\sum_{g=1}^{G} ||w^{(g)}||_2^2}, \tag{3}$$

where $||w^{(g)}||_2$ denotes the $L2$ norm of group of weights $w^{(g)}$. Hence, Equation (3) is equivalent to:

$$SHS(W) = \frac{(\sum_{g=1}^{G} ||w^{(g)}||_2)^2}{||W||_2^2}, \tag{4}$$

3 ANALYSIS OF TRANSFORMER WITH BLOCK-WISE STRUCTURE SPARSITY LEARNING

3.1 Analysis of Model Components

As mentioned in Section 2, Transformer is mainly made up of two layer components, of which we want to decrease the parameter size:

Multi-Head Attention (MHA): Multi-head attention components take three matrices: Query (Q) of dimension d_q, Key (K) of dimension d_k, and Value (V) of dimension d_v as input and generate the transformed matrices of dimension d_o. In [1], d_q=d_k=d_v=d_o=d_{model}/n, where n represents the number of heads in MHA components, which is set to 8. Hence, d_q=d_k=d_v=d_o=d_{model}/h=512/8=64. For multi-head self-attention, Q, k, V are identical. Formally, MHA is calculated as:

$$\begin{aligned} MHA(Q,K,V) &= Concat(h_1,...,h_n)W^o, \\ where\ h_i &= Attn(QW_i^Q, KW_i^K, VW_i^V). \end{aligned} \tag{5}$$

In Equation (5), the attention function ($Attn$) equals:

$$Attn(Q,K,V) = softmax(QK^T/\sqrt{d_k})V. \tag{6}$$

As expressed in Equation (6), attention operation basically multiplies Q and K (scaled by $\sqrt{d_k}$), applies softmax on the results and multiplies it with V, where no model parameters are involved. Therefore, our pruning targets of MHA are W^Q,W^K,W^V, and W^o, where W^o is called the output transform matrix. Because Q, K, V matrices are of the same shape, W_i^Q, W_i^K, and W_i^V (i = [1...8]) could be combined into one weight matrix and computed in parallel. Here W^Q, W^K, W^V are of shape (d_{model}, d_{model}) and W^o is also of shape (d_{model}, d_{model}).

Fully-connected Feed-forward Network (FFN): Fully-connected Feed-forward Networks (FFN) take the output of MHA as input and consist of two subsequent fully connected layers with ReLU activation function in between. FFN is formulated as:

$$FFN(x) = max(0, xW_{ffn1} + b_1)W_{ffn2} + b_2, \tag{7}$$

where b_1 and b_2 are biases. Therefore, W_{ffn1} and W_{ffn2} are the pruning targets of FFN components, which are of shape (d_{model}, d_{ffn}) and (d_{ffn}, d_{model}), respectively. Following [1], d_{ffn} equals 2048, which is 4 times of d_{model}.

3.2 Analysis of Pruning Target

Although our pruning targets are fully-connected layers, there still exist some challenges in learning the compact structure of Transformer. Transformer is composed of encoder and decoder, which may have different layer properties due to their different functionalities. Moreover, the input dimension of W_{ffn1} denotes the dimension of the input word embedding (d_{embed}) while the output dimension represents the linear transformations (d_{ffn}). It is hard to prune the model without a full understanding of the underlined meanings of the network configurations. More specifically, in order to better prune a Transformer model, we need to answer several questions relevant to the properties of Transformer components:

- For each of our pruning targets (W^Q, W^K, W^V, W^o, W_{ffn1} and W_{ffn2}), should we prune it row-wise, or column-wise, or both row-wise and column-wise?
- Should we treat encoder and decoder equally during pruning?

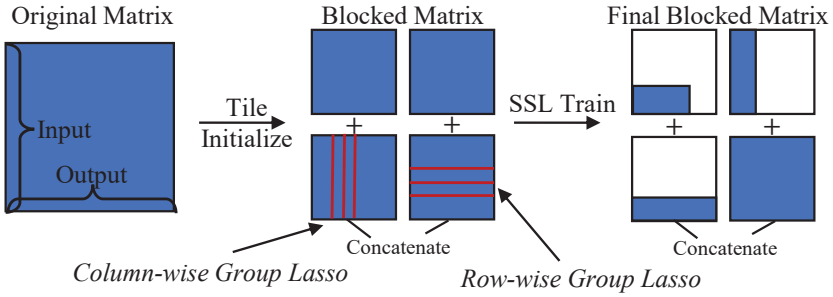

Fig. 4. Block-wise structured sparsity learning.

- Is the pruning properties the same for shallow layers and deep layers?
- Should W^Q, W^K, W^V be pruned with the same sparsity in MHA?

We will answer these questions in the following sections.

3.3 Block-wise Structured Sparsity Learning (BSSL)

In this work, we use Block-wise Structured Sparsity Learning (BSSL) to analyze the pruning properties of Transformer. In BSSL, we divide the original weight matrix into several sub-blocks with the same shape. We apply row-wise and column-wise group lasso penalty on each sub-block with the same weight decay. Row-wise and column-wise BSSL regularizer on a single sub-block of shape $(r,\ c)$ is formulated as:

$$L_{row}(W) = \sum_{i=1}^{r} \sqrt{\sum_{j=1}^{c} (W[i,j])^2} \tag{8}$$

and

$$L_{col}(W) = \sum_{i=1}^{c} \sqrt{\sum_{j=1}^{r} (W[j,i])^2}. \tag{9}$$

For an original weight matrix W, which is divided into $x \times y$ sub-blocks, the final training loss for a model with l layers can be defined as:

$$L = L_D + \lambda \sum_{i=1}^{l} \sum_{j=1}^{x} \sum_{k=1}^{y} (L_{row}(W_{[i,j,k]}) + L_{col}(W_{[i,j,k]})), \tag{10}$$

where λ is the weight decay, which controls the tradeoff between the loss on training data (L_D) and BSSL regularizer penalty. Here, $W_{[i,j,k]}$ denotes the sub-block with index $(j,\ k)$ in the original weight matrix of layer i.

Essentially, BSSL is a generalization of weight pruning with different granularities. For example, when the sub-block shape equals the original weight matrix shape, BSSL is line-wise pruning. When sub-block shape is $(1,\ 1)$, it becomes element-wise pruning. BSSL can be used to analyze the best pruning granularity of Transformer as follows:

- In order to understand which dimension of each weight matrix is more likely to be pruned, all sub-blocks of each layer have an identical size. As a result, the penalty on each row and column of the sub-block is the same.

- For W^Q, W^K, and W^V in MHA layer, the dimension of sub-block is equal to head width. In this way, we could understand how many heads are needed for each MHA component and what is the dimension that is required for each head.
- For W^o in MHA layer, we divide them into 2×2 sub-blocks. Each sub-block is of shape (256, 256).
- For W_{ffn1} and W_{ffn2} in FFN, we set the sub-block of the same shape as W^o. Therefore, W_{ffn1} and W_{ffn2} are divided into 2×8 sub-blocks and 8×2 sub-blocks, respectively.

Fig. 4 depicts the implementation procedure of BSSL analysis. For a target Transformer model, we first train an original full-size model. Then, we build a new Transformer graph, where each original layer matrix is replaced with tiled sub-blocks. The input dimension of the original layer matrix is split and fed into the tiled layer matrix and the output dimension of the tiled layer matrix is concatenated so as to be fed into the next layer. In this way, each layer in the original Transformer could be transformed into the block-wise layer matrix independent of the other layers. Because the combined shape of all the sub-blocks is the same as the original weight matrix, we initialize all the sub-blocks with the original weight values. After initialization, the block-wise Transformer models could achieve the same accuracy as the original model. Last, we apply row-wise and column-wise group lasso on each sub-block and perform the BSSL analysis. During training, rows and columns of each sub-block are expected to become zeros gradually. Finally, we visualize the sparsity properties of the model after BSSL analysis for a better understanding of Transformer weights redundancy.

3.4 Observation on BSSL Analysis

We apply BSSL on a Transformer model trained on WMT English-German dataset. The weight decay of BSSL regularizer $\lambda = 10^{-3}$ and the learning rate equals 10^{-2}, which is one-tenth of the original learning rate. In order to better understand the property of each layer and component of Transformer, we visualize the line-wise sparsity of the Transformer model after BSSL analysis. Following previous works [14, 42], the total number of the zero weights are determined by a pre-defined threshold, which is set to 10^{-4} in our case.

In Fig. 5, we present the sparsity of 6 encoder layers of Transformer (left part of Fig. 1) after BSSL analysis. For each encoder layer, there exist W^Q, W^K, W^V, W^o for MHA and W_{ffn1}, W_{ffn2} for FFN. The blocks in red are the sub-blocks containing no rows or columns that are full of zeros, i.e., the sparsity is 0% after BSSL pruning. All the other green blocks indicate that they are of certain sparsity, either row-wise, column-wise, or both after BSSL analysis. The areas of green denote the non-zero weights in the blocks and the white areas represent the weights that are zeros. In other words, the smaller area colored by green in a block, the larger the sparsity of that block is. For W_{ffn2}, we visualize it by its transpose (W^T_{ffn2}).From the visualization of Transformer encoder in Fig. 5, we observe that:

- All the weight matrices are pruned to certain extent for the encoder.
- W^Q, W^K, W^V, and W^{ffn1} should be pruned column-wise while W^o and W^{ffn2} should be pruned row-wise.

In Fig. 6, we present the sparsity of 6 decoder layers of Transformer (right part of Fig. 1) after BSSL analysis. For each decoder layer, there exist W^Q, W^K, W^V, W^o for both self-MHA and encoder-decoder MHA and W_{ffn1}, W_{ffn2} for FFN. From the visualization of Transformer encoder after BSSL in Fig. 6, we observe that:

- For both self-MHA and encoder-decoder MHA, W^Q, W^K, W^V are all pruned to some extent.
- W^o, W^{ffn1}, and W^{ffn2} hardly get any sparsity even both line-wise and column-wise group lasso regularizers are applied.

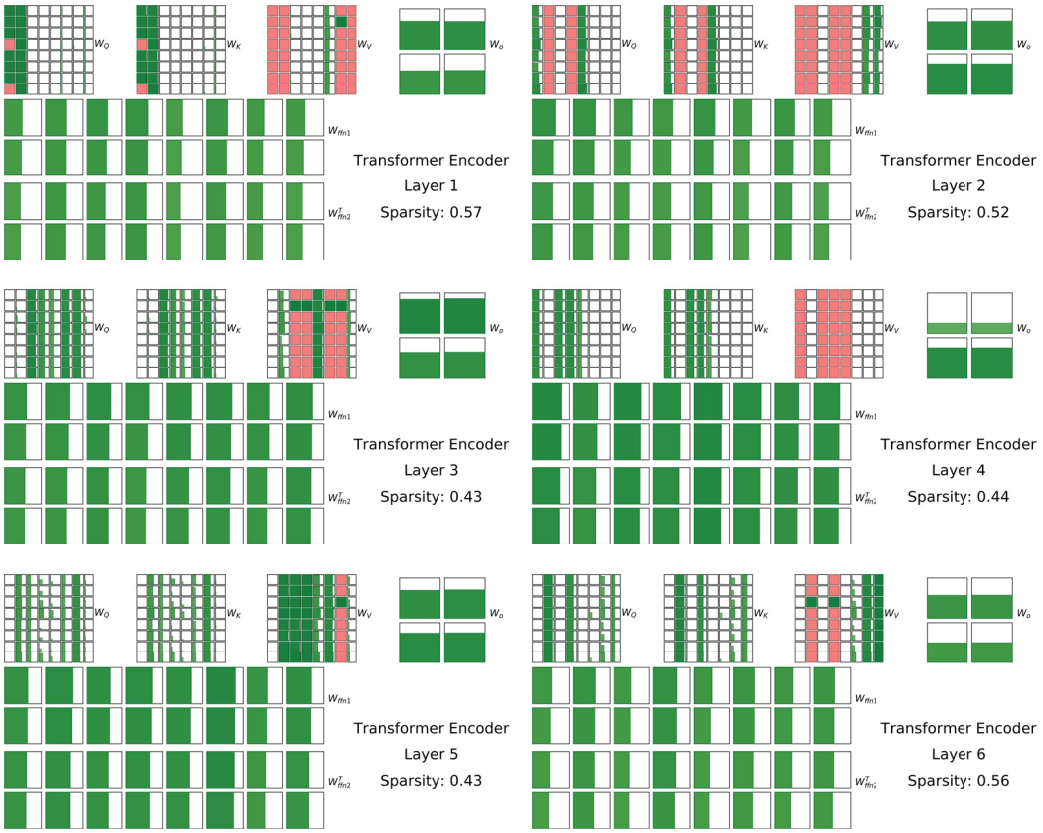

Fig. 5. Sparsity visualization of Transformer encoder with BSSL.

3.5 Conclusion of BSSL Analysis

Here, we summarize the properties derived from the BSSL analysis by looking into both the encoder and decoder of Transformer in Fig. 5 and Fig. 6, respectively.

First, with the help of BSSL analysis, we could find some properties of Transformer models, which have already been discovered in previous works on Transformer understanding [20, 43].

- Some heads in Fig. 5 and Fig. 6 are removed entirely, implying that only a subset of heads is important for translation.
- The column-wise sparsity of self-attention MHA in Fig. 5 is high (53.5% on average) while the sparsity of decoder-encoder MHA in Fig. 6 is much lower (22% on average), implying that the model prefers to prune encoder self-attention heads. In other words, decoder-encoder attention heads are more important.
- For the self-attention heads in Fig. 6, the heads in shallower layers (e.g., layer 1, 2) are retained more compared with deeper layers (e.g., layer 5, 6). For example, the column-wise sparsities of layers 1 and 2 are 13.3% and 23.3%, respectively while the column-wise sparsities of layers 5 and 6 are 36.7% and 93%, respectively.
- For the decoder-encoder attention heads in Fig. 6, more heads are retained in deeper layers (e.g., layer 5, 6). For example, the column-wise sparsities of layer 1 and 2 are 68% and 63%, respectively while 10% and 16.7% for layers 5 and 6, respectively.

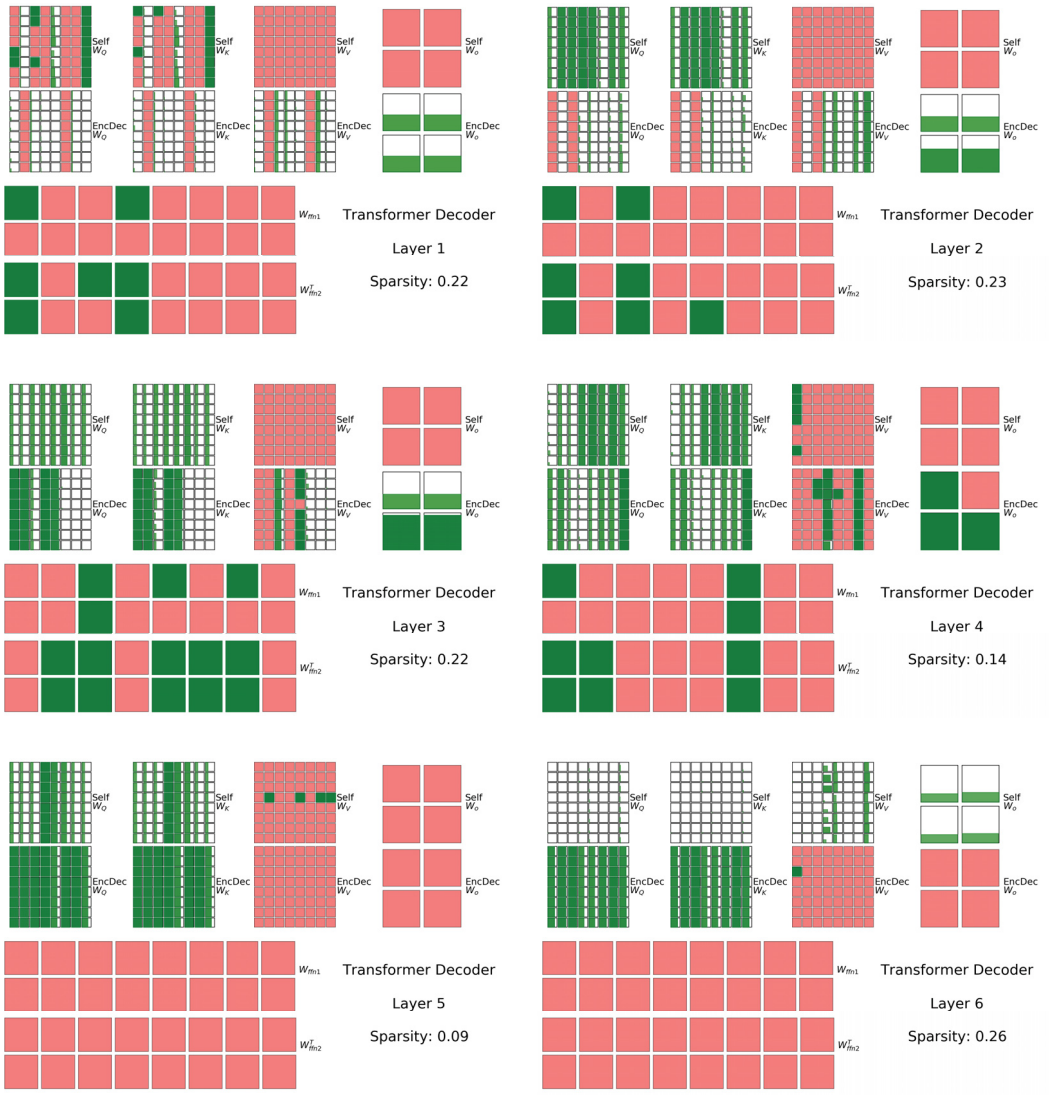

Fig. 6. Sparsity visualization of Transformer decoder with BSSL.

Besides the properties which have been discovered before, we indeed obtained more understanding of the sparsity visualization in Fig. 5 and Fig. 6 from the BSSL analysis:

- Even if we apply the block-wise penalty on Transformer, the final model trained by BSSL still achieves only line-wise sparsity in most cases. That is the main reason why we propose line-wise pruning for Transformer.
- W^Q, W^K are always pruned to the same column-sparsity level while W^V is pruned to a different sparsity level. This fact can be derived from Equation (6), where Q and K transpose is multiplied while V is related to the $softmax$ of the multiplication result.

- The dimension of FFN could be further compressed in Transformer encoder but not for Transformer decoder.
- All the weights are pruned either row-wise or column-wise because all the sub-blocks want to utilize the whole embedded dimension and hence, only the internal dimension could be pruned. In other words, the pruned dimension is always the internal dimension of each component (MHA and FFN).

Thanks to BSSL analysis, we are able to answer the question aforementioned in Section 3.2 about the properties of Transformer pruning. Those understandings of Transformer model help us to propose the pruning strategy in Section 4.

4 TRANSFORMER PRUNING

4.1 Line-wise Transformer Pruning

Based on our analysis, in *TPrune*, we prune the model line-wise to realize real speedup-up on mobile devices. In specific, we apply $L1$ on $L2$ norm of group weights to generate structured sparsity [14]. However, pruning using SSL has its disadvantage regarding the group scale. For example, consider adding SSL regularizer on a group of weights: W_g, where the $L2$ norm of this group of weights equals n_g. The corresponding gradient for single weight w_i in W_g equals wi/n_g. The gradients largely depend on the scale of the original weight value as well as the $L2$ norm of the group that the weight belongs to. As aforementioned in Section 2.5, in response to the scale-variant SSL, we adopt scale-invariant Structured Hoyer Square (SHS). In *TPrune*, each group of weights $w^{(g)}$ in Equation (4) of SHS regularizer is defined as a row or a column of Transformer layer matrices.

4.2 Pruning Strategy

Based on the pruning properties of Transformer derived from the BSSL analysis, we use SHS to perform line-wise pruning on our target Transformer model. The detailed pruning strategies are the follows:

- The visualization of BSSL in Fig. 5 shows that we can achieve high sparsity for all the layer components of Transformer. For Transformer encoder, we prune W^Q, W^K, W^V, W^{ffn1} in column-wise and prune W^o and W^{ffn2} in row-wise.
- The visualization of the BSSL in Fig. 6 shows that the sparsity for FFN and W^V in MHA is very limited. Therefore, for Transformer decoder, we only prune W^Q and W^K in column-wise.

Same as previous works [14, 42], the pruning procedure includes two steps: *pruning* and *fine-tuning*. In the pruning step, the model is trained with sparsity regularizer (e.g., SSL and SHS), aiming at removing the redundant weights as many as possible. In the fine-tuning step, the trained model from the pruning step is further fine-tuned without sparsity regularizer. The gradients are masked-out for all the zero weights during the fine-tuning step. In Section 5, we will compare our pruning strategy with naive pruning strategies.

5 EXPERIMENTS

5.1 Experimental Setup

Framework: We implement *TPrune* with TensorFlow [44], a widely used deep learning framework for training and testing on varies hardware platforms. More specifically, for the training and pruning of Transformer model, we use tensor2tensor (T2T) [45]. T2T is a TensorFlow library designed for machine learning research. It incorporates many datasets, pre-trained models, and pre-defined hyperparameters to accelerate model development. For the profiling of real-time execution of Transformer on mobile devices, we adopt TFLite Model Benchmark Tool, which is a TensorFlow

benchmark tool for embedded software on both IOS and android devices. It converts the original model structure to TFLite format targeting at embedding systems for profiling purposes.

Benchmarks and Settings: For the whole experiments, we use WMT English-German dataset provided by T2T to validate the effectiveness of *TPrune*. English-German dataset consists of around $32k$ tokens with shared source-target vocabulary. During the pruning step, the effective batch size is set to $6k$ tokens and trained for $40k$ steps. Following the tip in [46], the effective batch size and training steps are set larger than those in the pruning step, which is $16k$ tokens and $800k$ steps, respectively. The same as [1], we report the BLEU score as the metric to evaluate model performance and the testing dataset is newstest2014.

Transformer Model: In all the following experiments, we adopt the same model structure as the base Transformer structure in the original work [1]. The base Transformer model include 6 layers for both encoder and decoder. d_{model} and d_{embed} equal 512 and d_{ffn} equals 2048. Because this work is about Transformer model pruning, the final model sparsity is derived by only considering the weights of the main body of Transformer. The embedding matrix is not included. This is reasonable because the size of the weight matrix largely depends on the vocabulary size of the source and target languages. For the fairness of comparison, we start the model pruning from the official pre-trained Transformer model provided by T2T framework. The BLEU score of the original model on newstest2014 is 29.2.

Environment: In order to speed up the training, the models are trained under Linux system with 4 NVIDIA TITAN RTX GPUs in parallel. Therefore, the batch size of each GPU is $4k$ in order to reach $16k$ effective batch size. For time profiling on mobile devices, we test the Transformer models on 4 popular Android devices with different hardware configurations, i.e., Nexus 5, Pixel 2, Pixel 3, and LG G8 ThinQ. All the testing of Transformer are based on mobile CPUs.

5.2 Comparisons of Different Pruning Regularizers

Fig. 7 shows the sparsity and the corresponding BLEU score during the pruning step of the SHS and SSL regularizers, respectively. In Fig. 7, the weight decay λ is set as 10^{-5} and 10^{-4} for SSL and SHS regularizer, respectively. Because of the comparatively small λ, the pruned models after $400k$ steps still maintain a good performance.

As demonstrated in Fig. 7, under similar BLEU score after $400k$ steps training, the model pruned with SHS achieves a higher sparsity (8.16%) than the model pruned with SSL regularizer (2.2%). This is due to the reason that, when the pruning penalty λ is small, most of the weights are still non-zeros. In this case, SHS is more effective to remove the weights that are close to zeros while SSL

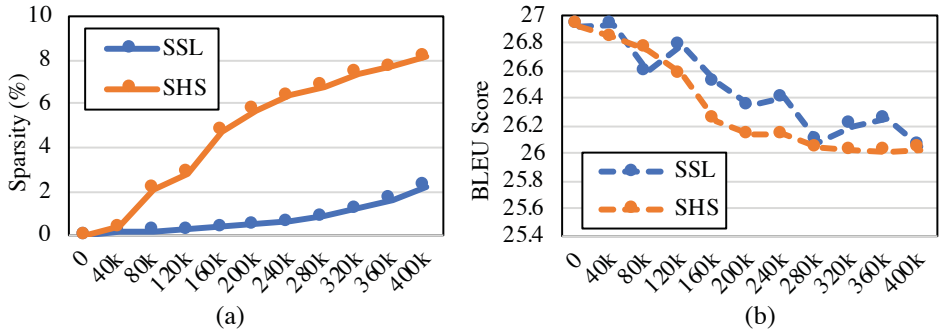

Fig. 7. Sparsity (a) and BLEU (b) between SSL ($\lambda = 10^{-5}$) and SHS ($\lambda = 10^{-4}$) with a small λ.

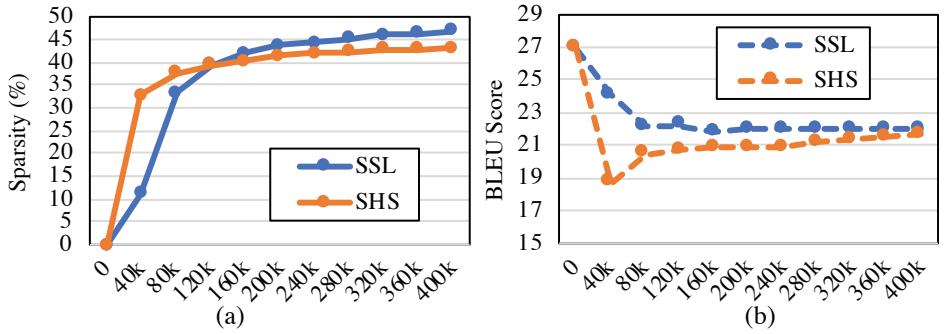

Fig. 8. Sparsity (a) and BLEU (b) between SSL ($\lambda = 5 \times 10^{-5}$) and SHS ($\lambda = 10^{-3}$) with a large λ.

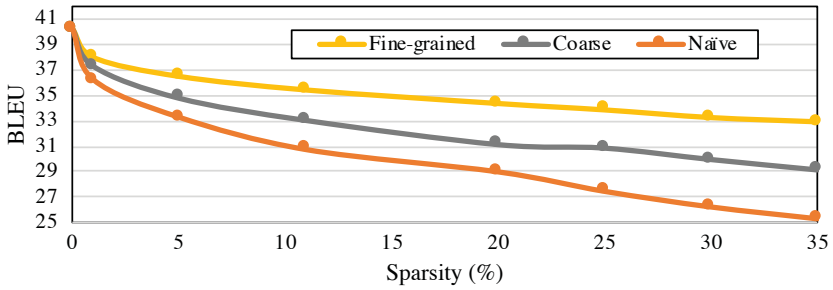

Fig. 9. BLEU on WMT development set with different line-wise pruning strategies under different sparsities.

applies almost equal penalties on all the weights, leading to less number of weights to be removed. Therefore, SHS outperforms SSL regularizer when we prefer a high-performance model with a low sparsity.

Fig. 8 shows the sparsity and the corresponding BLEU score of the models trained with larger weight decays, i.e., 5×10^{-5} and 10^{-3}, respectively, for SSL and SHS. Because of the higher λ setting, the pruned model sparsity reaches around 45% for both regularizers after $400k$ steps. The corresponding BLEU scores are 21.97 and 21.66, respectively, for SSL and SHS regularizer.

Different from the situation in Fig. 7, the performance of the SSL pruned model is similar to that of the SHS pruned model. Moreover, the larger λ settings of SSL and SHS regularizers lead to different performance trends in model pruning: the BLEU score of the SSL pruned model keeps decreasing when the sparsity of the model increases; the BLEU score of the SHS pruned model, however, first drops dramatically when the sparsity of the model increases (i.e., at $40K$ steps), and then gradually recovers. Such different trade-offs between the sparsity and the performance of SSL and SHS offer us unique design flexibility in Transformer pruning.

Table 1 summarizes the sparsity and the BLEU score of both SSL and SHS after $400k$ steps with a batch size of $8k$. We could find that SHS performs better with a low sparsity and SSL performs slightly better with a high sparsity. Mind that the performance may be further boosted by increasing the training steps.

Table 1. Sparsity and BLEU score after $400k$ steps pruning using SSL and SHS regularizers

Regularizer	λ	BLEU	Sparsity(%)
SSL	1×10^{-5}	26.05	2.21
	2×10^{-5}	24.46	26.33
	3×10^{-5}	23.31	38.23
	4×10^{-5}	22.53	43.61
	5×10^{-5}	21.97	46.78
SHS	1×10^{-4}	26.03	8.16
	2×10^{-4}	25.01	20.29
	3×10^{-4}	24.04	29.52
	4×10^{-4}	23.58	33.15
	5×10^{-4}	23.15	35.66
	1×10^{-3}	21.46	44.07

5.3 Evaluation of Pruning Strategies

In Fig. 9, we compare different pruning strategies of Transformer by presenting the BLEU score during the SHS training step on the WMT development set. Besides the adopted fine-grained pruning strategy (marked as "**Fine-grained**") that was discussed in Section 4, we add two more strategies for comparison:

- **Coarse Strategy**: In the fine-grained pruning strategy derived from the BSSL analysis, we treat the encoder and the decoder differently. In coarse strategy (marked as "**Coarse**"), we treat them the same as the pruning strategy for original encoder strategy.
- **Naive Strategy**: In naive strategy (marked as "**Naive**"), we simply apply both row-wise and column-wise penalties to all the target weight matrix. This strategy represents the baseline when we have no understanding of the Transformer structure properties.

As shown in Fig. 9, our proposed fined-grained pruning strategy always performs the best among all the strategies under any model sparsity. The naive strategy baseline performs the worst. For example, the BLEU score under 20% sparsity are 34.36, 31.13, and 28.98 for **Fine-grained**, **Coarse**, and **Naive**, respectively. The performance gap keeps increasing with the growth Transformer model sparsity, proving that our designed Fine-grained pruning strategy is effective in removing useless weights while keeping the model performance.

5.4 Evaluation of Layer-wise Sparsity

Fig. 10 and Fig. 11 present the layer-wise sparsities under different model sparsities for Transformer encoder and decoder, respectively. Due to the computation logic of Transformer, the following rules always hold:

- W^Q and W^K are always of the same column-wise sparsity.
- The column-wise sparsity of W^V and the row-wise sparsity of W^o are the same.
- The W^{ffn1}'s column-wise sparsity is the same as the W^{ffn2}'s row-wise sparsity.

Fig. 10 shows that for Transformer encoder, MHA is effectively pruned first. For example, when the model sparsity equals 9.76%, the sparsities of "W^Q, W^K" and "W^V, W^O" are 14.65% and 24.48%, respectively, while the sparsity of "W^{ffn1}, W^{ffn2}" is only 2.08%. Following the increase of model sparsity, the sparsities of all target layer types gradually become similar. For example, when the

model sparsity equals 45.07%, the sparsities for "W^Q, W^K", "W^V, W^O", and "W^{ffn1}, W^{ffn2}" are 79.07%, 79.52%, and 77.33%, respectively.

Because no regularizer penalty is applied to W^V, W^O, W^{ffn1}, and W^{ffn2} in Transformer decoder, Fig. 11 only shows the sparsities of W^Q and W^K in self-attention and encoder-decoder attention, respectively. The sparsities of the W^Q and W^K in encoder-decoder attention are always lower than that of the W^Q and W^K in self attention. This means that encoder-decoder attention is more important than self attention for Transformer encoder and lean to be preserved.

When comparing Fig. 10 with Fig. 11, we could find that under the model sparsities of 9.76% and 20.29%, the sparsities of W^Q and W^K in Transformer decoder is higher than that of encoder. For example, when the model sparsity equals 9.76%, the sparsity of "W^Q, W^K" in the encoder is 14.65% while the sparsity of "W^Q, W^K" in self-attention and encoder-decoder attention of decoder are 46.06% and 43.42%, respectively. This is because "W^{ffn1}, W^{ff2}" are not pruned for Transformer decoder. As a result of that, model redundancy of Transformer decoder is transferred to the MHA, leaving more pruning space for "W^Q, W^K" in the decoder.

5.5 Evaluation of Speedup on Mobile Devices

In this section, we evaluate the speedup of our pruned Transformer models on mobile devices. For all the profiling experiments on mobile devices, we set the warm-up runs as 5 times and the test runs as 80 times.

5.5.1 Impact of string length. Fig. 12 presents the execution time on Pixel 3 with different model sparsities and translation string lengths (denoted by "StnLen"). All the execution times in Fig. 12 are tested under 4 threads. As shown in Fig. 12, with the same sparsity, the execution time increases

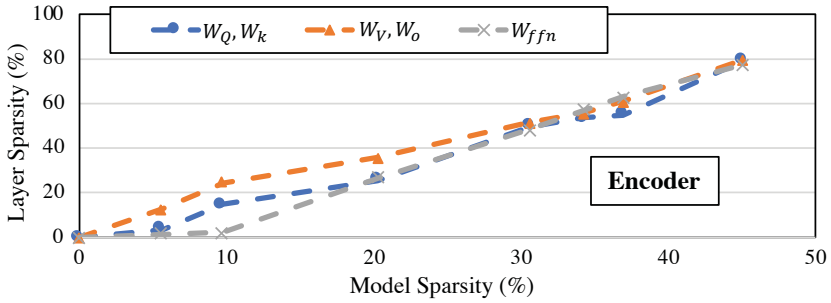

Fig. 10. Layer-wise sparsities of Transformer encoder under different model sparsities.

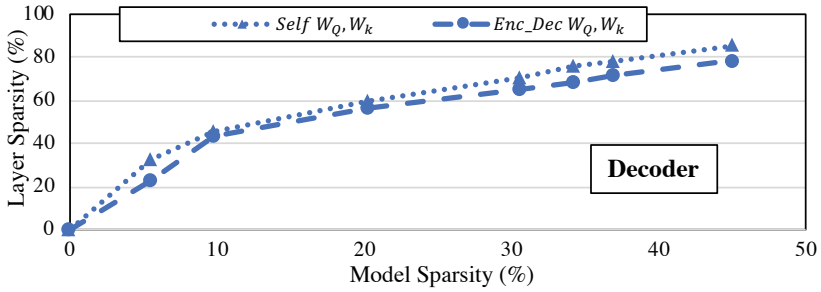

Fig. 11. Layer-wise sparsities of Transformer decoder under different model sparsities.

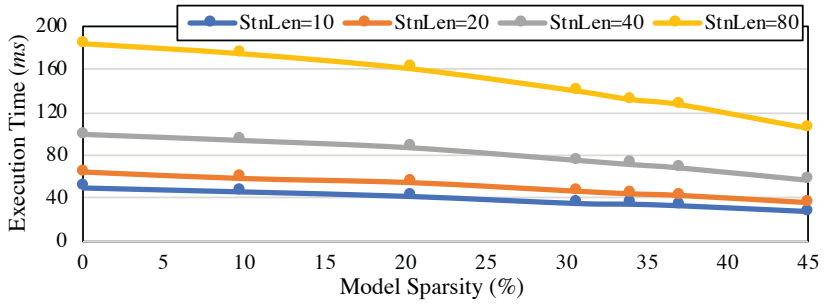

Fig. 12. Execution time with different string lengths.

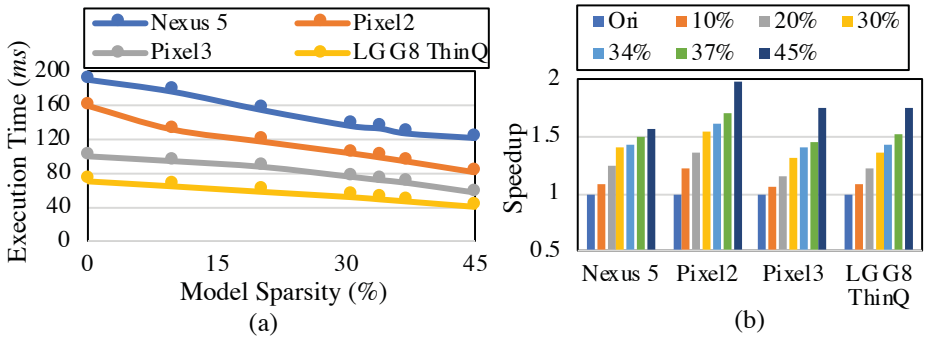

Fig. 13. Execution time (a) and Speedups (b) on different mobile devices.

with the string length. For example, for the original transformer model, the execution time increases from 50*ms* to 184*ms* when the string length increases from 10 to 80. Note that this result does not exactly follow the theoretical analysis in [1] where the computational cost of Transformer quadratically increases with the string length. Except the time complexity of the algorithm, many factors could affect the execution time of a Transformer model on a mobile device such as memory bandwidth, on-chip cache size, memory latency, etc. From Fig. 12, the execution time of the pruned model with a small string length still consumes a certain amount of latency. In this case, memory latency is the main reason that causes a large execution time even with short string length.

5.5.2 Impact of mobile devices. Fig. 13(a) compares the execution time of the Transformer model running on different popular mobile devices. All the results are tested under 4 threads with a string length of 40. As shown in the Figure, the computing performance of different mobile devices varies significantly. For example, when we execute the original models on Nexus 5, Pixel 2, Pixel 3, and LG G8 ThinQ, the corresponding execution times are 190*ms*, 159*ms*, 100*ms*, and 70*ms*, respectively. Nonetheless, increasing the sparsity of the model always helps to reduce the execution time: if we execute the Transformer model with a sparsity of 45%, the execution times of these tested mobile devices changes to 121*ms*, 81*ms*, 57*ms*, and 40*ms*, respectively. Fig. 13(b) demonstrates the normalized speedups of the Transformer models with different sparsities when compared with the original Transformer model on different mobile devices. When executed on Nexus 5, $1.07\times - 1.57\times$ speedups are achieved when the model sparsity grows from 10% to 45%. Pixel 2 achieves $1.2\times - 1.96\times$ speedups while Pixel 3 and LG G8 ThinQ achieve $1.07\times - 1.75\times$ speedups.

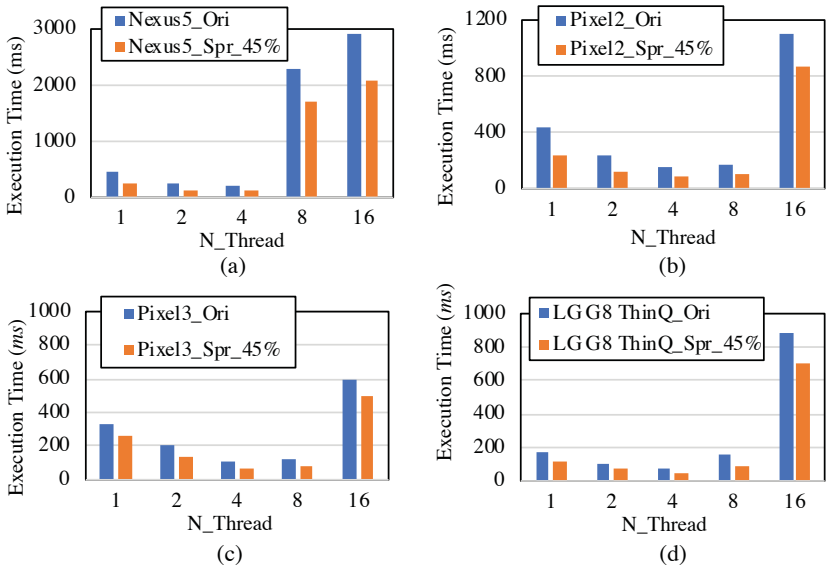

Fig. 14. Execution time with different number of threads on Nexus 5 (a), Pixel 2 (b), Pixel 3 (c) and LG G8 ThinQ (d).

5.5.3 Impact of multi-thread. Fig. 14 presents the impact of multi-threading on the execution time when executing Transformer models. All the execution time in Fig. 14 are tested with a string length of 40. The sparsity of the Transformer model is 45%. As presented in Fig. 14, the trends of the experimental results are similar across all the 4 mobile devices with different hardware configurations. Take Pixel 3 as an example, the speedups of running sparse Transformer model w.r.t. the original model are 1.6×, 1.5×, 1.57×, 1.63×, and 1.19×, respectively, for 1, 2, 4, 8, and 16 threads. Simply increasing the number of parallel threads does not always reduce the execution time: When the number of threads increases from 8 to 16, the execution time dramatically increases on both Pixel 2, Pixel 3, and LG G8 ThinQ due to the sharply increased model partitioning overheads. Moreover, as shown in Fig. 14(a), the execution time increases on Nexus 5 when the number of threads increases from 4 to 8. This is due to the limited CPU multi-processing capability of Nexus 5 when comparing with the other 3 tested mobile devices. In general, 4 parallel threads offer a good setup of executing the Transformer models (original and pruned) on all 4 tested mobile devices.

5.5.4 Speedup of pruned models. Fig. 15 shows the execution time when executing two pruned models with different sparsities. The blue line shows the results of the models pruned by our proposed SHS regularizer and the yellow line shows the results of head-wise pruning in previous works [20, 21]. All the results are tested with a string length of 40 and 4 parallel threads on Pixel 3. When the model sparsity changes from 4% to 28%, head-wise pruning outperforms SHS-based line-wise pruning because of the larger pruning granularity and simpler pruning strategy of head-wise pruning. Note that 28% is the highest sparsity that head-wise pruning can achieve as it is obtained by removing all the heads in Transformer model. As a comparison, SHS-based line-wise pruning can further prune the model sparsity to around 45%, resulting in 1.92× speedup finally. Note that the performance degradation of head-wise pruning is much worse than that of SHS-based line-wise pruning at the same sparsity, which will be discussed in the next section.

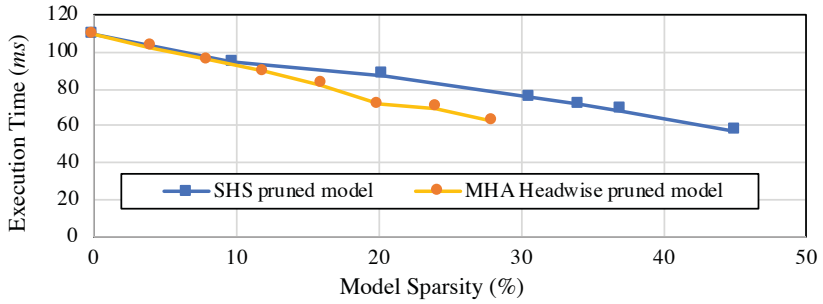

Fig. 15. Execution time when executing pruned models of different sparsity.

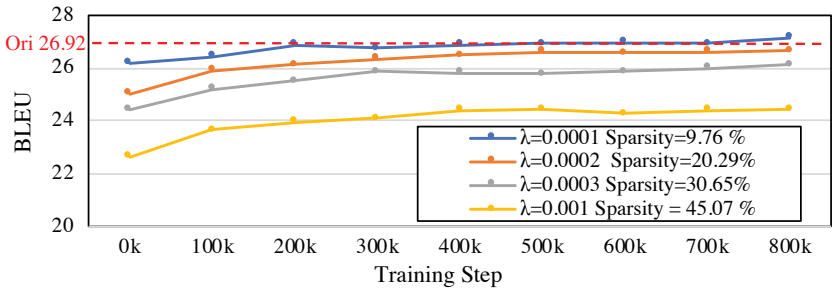

Fig. 16. BLEU score during the fine-tuning procedure.

5.6 Evaluation of Sparsity-accuracy Trade-off

Fig. 16 shows how the BLEU score changes during the fine-tuning procedure of SHS pruning. The BLEU score of the original model is 26.92, which is marked as the red dash line in the Figure. Here we train all the models for 800k steps with 16k batch size. Fig. 16 shows that the SHS pruned model converges quickly when the pruned model sparsity is low. For example, the BLEU score of the model with a sparsity of 9.76% converges at 26.89 after only 200k fine-tuning steps. When the model sparsity increases to 45.07%, it takes approximately 400k steps for the BLEU score to converge at around 24.38. As shown in Fig. 16, when the model sparsity is below 30%, the fine-tuning step could always restore the performance back and make it close to the original level.

Table 2 summarizes the metrics of SHS pruned models with different sparsity's. Here the original model is used as the baseline. For comparison purposes, we also present the results of another state-of-the-art work with head-wise pruning [21]. For the fairness of the comparison, we port the head-wise pruning technology from original frameworks [21] to our T2T framework to ensure that all the used initial models and dataset are identical to our experiment. Moreover, because no speedup was tested in [21], all the speedup results here are tested by us on Pixel 3. As shown in Table 2, our pruned models demonstrate good performance with satisfactory speedup on mobile devices. Compared with the baseline model, the SHS pruned model achieves around 16% sparsity without any degradation in BLEU score. The corresponding speedup is 1.21× on Pixel 3. When the sparsity is increased to 20.29%, the BLEU score slightly degrades to 26.78, or 99.48% of the original performance. If we further increase the sparsity to 45%, we can obtain 1.92× speedup with a BLEU score of 24.78. As a comparison, head-wise Transformer pruning [21] incurs no performance degradation when the sparsity is 4.29%. The corresponding speedup is merely 1.05×. Increasing

Table 2. Summary of our line-wise pruning results and previous head-wise pruning results [21].

Dataset	Model	BLEU	BLEU Degradation (%)	Sparsity(%)	Speedup
WMT_EnDe	Baseline	26.92	0	0	1
	Model1	27.14	0	9.76	1.16
	Model2	26.93	0	15.63	1.21
	Model3	26.78	0.52	20.29	1.25
	Model4	26.14	2.9	30.65	1.44
	Model5	25.94	3.58	34.27	1.52
	Model6	25.63	4.79	36.93	1.59
	Model7	24.78	7.95	45.07	1.92
WMT_EnDe	Baseline	26.92	0	0	1
	[21]	26.92	0	4.29	1.05
	[21]	25.19	6.43	8.58	1.15
	[21]	20.74	22.96	17.14	1.33
	[21]	10.1	62.82	25.71	1.56

the sparsity of the pruning model will quickly degrade the model performance: when the sparsity becomes 25.71%, the BLUE score drops down to only 10.1. Even worse, the model could not be further pruned because of the limited heads in Transformer. In layer-wise pruning [19], 5% BLEU degradation is incurred with 50% sparsity, which achieves better performance-accuracy trade-off than *TPrune*. Fortunately, layer-wise pruning of Transformer is orthogonal to *TPrune*. Therefore, better performance is expected when combining these two pruning methods.

6 CONCLUSION

TPrune includes a Transformer model analysis method – Block-wise Structure Sparsity Learning (BSSL) and a line-wise pruning method using Structured Hoyer Square (SHS). In BSSL, we utilize structured sparsity learning on sub-blocks of Transformer weight matrix to analyze the model properties of Transformer. Based on the analysis derived from BSSL, we propose to use SHS regularizer to prune the Transformer model in either row-wise or column-wise for different Transformer components. Experimental results show that *TPrune* achieves a better trade-off between the accuracy and execution time of Transformer model than the prior-arts on some representative mobile devices.

ACKNOWLEDGMENTS

This work was supported in part by NSF CNS-1717657, CCF-1937435, CNS-1822085, and NSF IUCRC for ASIC memberships from Unisound etc.

REFERENCES

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[3] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 5754–5764. Curran Associates, Inc., 2019.

[4] 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. *CoRR*, abs/1907.11692, 2019.

[5] Jiachen Mao, Qing Yang, Ang Li, Hai Li, and Yiran Chen. Mobieye: An efficient cloud-based video detection system for real-time mobile applications. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[6] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1396–1401. IEEE, 2017.

[7] Kent W Nixon, Jiachen Mao, Juncheng Shen, Huanrui Yang, Hai Helen Li, and Yiran Chen. Spn dash-fast detection of adversarial attacks on mobile via sensor pattern noise fingerprinting. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6. IEEE, 2018.

[8] Fan Chen, Linghao Song, Hai Helen Li, and Yiran Chen. Zara: a novel zero-free dataflow accelerator for generative adversarial networks in 3d reram. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[9] J. Mao, Z. Qin, Z. Xu, K. W. Nixon, X. Chen, H. Li, and Y. Chen. Adalearner: An adaptive distributed mobile learning system for neural networks. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 291–296, Nov 2017.

[10] Bing Li, Wei Wen, Jiachen Mao, Sicheng Li, Yiran Chen, and Hai Helen Li. Running sparse and low-precision neural network: When algorithm meets hardware. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 534–539. IEEE, 2018.

[11] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 56–68. IEEE, 2019.

[12] Chuhan Min, Jiachen Mao, Hai Li, and Yiran Chen. Neuralhmc: an efficient hmc-based accelerator for deep neural networks. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 394–399, 2019.

[13] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[14] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2074–2082. Curran Associates, Inc., 2016.

[15] 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.

[16] Qing Yang, Jiachen Mao, Zuoguan Wang, and Hai Li. Dasnet: Dynamic activation sparsity for neural network efficiency improvement. *arXiv preprint arXiv:1909.06964*, 2019.

[17] Mengye Ren et al. Sbnet: Sparse blocks network for fast inference. In *CVPR*, pages 8711–8720, 2018.

[18] Mikhail Figurnov et al. Perforatedcnns: Acceleration through elimination of redundant convolutions. In *NIPS*, pages 947–955, 2016.

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

[20] 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.

[21] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? In *Advances in Neural Information Processing Systems*, pages 14014–14024, 2019.

[22] Kenton Murray, Jeffery Kinnison, Toan Q Nguyen, Walter Scheirer, and David Chiang. Auto-sizing the transformer network: Improving speed, efficiency, and performance for low-resource machine translation. *arXiv preprint arXiv:1910.06717*, 2019.

[23] Robin Cheong and Robel Daniel. transformers. zip: Compressing transformers with pruning and quantization. Technical report, Technical report, Stanford University, Stanford, California, 2019. URL https …, 2019.

[24] 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.

[25] 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.

[26] Biao Zhang, Deyi Xiong, and Jinsong Su. Accelerating neural transformer via an average attention network. *arXiv preprint arXiv:1805.00631*, 2018.

[27] Zhanghao Wu, Zhijian Liu, Ji Lin, Yujun Lin, and Song Han. Efficient transformer for mobile applicatoins. *International conference on learning representitive (ICLR)*, 2020.
[28] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.
[29] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.
[30] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. *Citeseer*, 2009.
[31] Jiachen Mao, Zhongda Yang, Wei Wen, Chunpeng Wu, Linghao Song, Kent W. Nixon, Xiang Chen, Hai Li, and Yiran Chen. Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns. In *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD '17, pages 751–756, Piscataway, NJ, USA, 2017. IEEE Press.
[32] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
[33] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
[34] Jie Hao, Xing Wang, Shuming Shi, Jinfeng Zhang, and Zhaopeng Tu. Multi-granularity self-attention for neural machine translation. *arXiv preprint arXiv:1909.02222*, 2019.
[35] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. *arXiv preprint arXiv:1910.04732*, 2019.
[36] JS McCarley. Pruning a bert-based question answering model. *arXiv preprint arXiv:1910.06360*, 2019.
[37] Fu-Ming Guo, Sijia Liu, Finlay S Mungall, Xue Lin, and Yanzhi Wang. Reweighted proximal pruning for large-scale language representation. *arXiv preprint arXiv:1909.12486*, 2019.
[38] 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. *arXiv preprint arXiv:1909.05840*, 2019.
[39] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. *arXiv preprint arXiv:1910.06188*, 2019.
[40] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.
[41] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 806–814, 2015.
[42] Huanrui Yang, Wei Wen, and Hai Li. Deephoyer: Learning sparser neural network with differentiable scale-invariant sparsity measures. *arXiv preprint arXiv:1908.09979*, 2019.
[43] Tobias Domhan. How much attention do you need? a granular analysis of neural machine translation architectures. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1799–1808, 2018.
[44] Martín Abadi et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.
[45] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018.
[46] Martin Popel and Ondřej Bojar. Training tips for the transformer model. *The Prague Bulletin of Mathematical Linguistics*, 110(1):43–70, 2018.