Learn-to-Share: A Hardware-friendly Transfer Learning Framework Exploiting Computation and Parameter Sharing

Cheng Fu [1] Hanxian Huang [1] Xinyun Chen [2] Yuandong Tian [3] Jishen Zhao [1]

Abstract

Task-specific fine-tuning on pre-trained transformers has achieved performance breakthroughs in multiple NLP tasks. Yet, as both computation and parameter size grows linearly with the number of sub-tasks, such methods are increasingly difficult to adopt in the real world due to unrealistic memory and computation overhead on computing devices. Previous works on fine-tuning focus on reducing the growing parameter size to save storage cost by parameter sharing. However, compared to storage, the constraint of computation is a more critical issue with the fine-tuning models in modern computing environments; prior works fall short on computation reduction.

To enable efficient fine-tuning, we propose *LeTS*, a framework that leverages both computation and parameter sharing across multiple tasks. LeTS consists of two principles. First, LeTS decouples the computation dependency in traditional fine-tuning model by proposing a novel neural architecture to reuse the intermediate results computed from the pre-trained model and the input. Furthermore, we leverage differentiable neural architecture search to determine task-specific computation sharing scheme. Second, by treating the final weight parameters as a weight difference added to the pre-trained weight, we propose a novel early stage pruning approach to generate a mask at the beginning of fine-tuning. By combining these two principles, LeTS further reduces the computation demand by exploiting the sparsity feature of weight difference. Extensive experiments show that with 1.4% of extra parameters per task, LeTS reduces the computation by 49.5% on GLUE benchmarks with only 0.2% accuracy loss compared to the full fine-tuning method.

[1]University of San Diego [2]University of California, Berkeley [3]Facebook AI Research. Correspondence to: Cieua Vvvvv <c.vvvvv@googol.com>.

Proceedings of the 38th International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

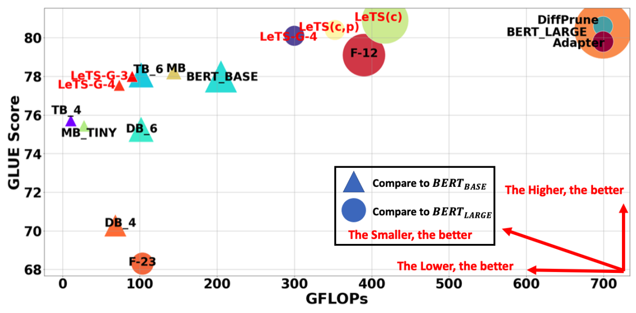


Figure 1: Roadmap of GLUE score (nine sub-tasks) v.s. total operations and parameters. The area of a point is proportional to the parameter size. *F-n* denotes freezing bottom n layers. *MB, TB, DB* represent MobileBERT, TinyBERT, DistillBERT respectively. LeTS outperforms other parameter-sharing methods in terms of computation and parameter efficiency. LeTS is orthogonal to model compression techniques (e.g. MB/TB/DB) as LeTS does not modify the pre-trained model for fine-tuning.

1. Introduction

Fine-tuning from pre-trained Transformers (Vaswani et al., 2017) has become the *de-facto* method for many NLP tasks, with performance breakthrough in various natural language understanding benchmarks (Devlin et al., 2019; Lan et al., 2020; Liu et al., 2019d; Joshi et al., 2020; Yang et al., 2019). Yet, the growing number of different NLP tasks arriving in stream makes this approach hard to integrate into real-world commercial products. The key bottlenecks lie in both **computation** and **storage** constraints. In particular, with conventional fine-tuning methods (Howard & Ruder, 2018; Wang et al., 2018), both single input processing latency and storage requirement grow linearly to the number of sub-tasks. This incurs an impractical computation, power, and storage overhead for a commercial product.

Both computation and storage constraints are critical to fine-tuning tasks. On the one hand, without much sacrifice in the quality of service, cloud computing vendors care more about the computation constraint to further improve the quality. The storage overhead can potentially be resolved by the advances in memory and storage technologies (Intel, 2019; Consortium, a;b), which enable the low-cost and large capacity data storage. On the other hand, in memorylimited devices (e.g, mobile), both constraints are critical to user experience. Most prior works focus on reducing the storage constrained across multiple sub-tasks by leveraging parameter-sharing. A multi-task learning (MTL) solution trains all the sub-tasks together (Liu et al., 2019b; Clark et al., 2019). However, it requires access to all the sub-tasks at the design time. Furthermore, MTL is not scalable with the increasing number of sub-tasks, as it is hard to balance the performance of multiple tasks and solve them equally well (Stickland & Murray, 2019).

Recently, Adapter (Houlsby et al., 2019) considers the new tasks in the fashion of arriving in stream which is more scalable compared to MTL. It proposes to add a task-specific building block between each attention layer and freeze the other parameters during fine-tuning. Recent works propose a differentiable pruning method (Guo et al., 2020) that achieves better results than Adapter. All the aforementioned parameter-efficient efforts cannot relieve the computation bottleneck in multi-task inference, because tuning the bottom layers will influence the computation results in the downstream layers. As such, re-computation is required.

Our goal is to resolve the computation and parameter constraints in multi-task evaluation. To achieve our goal, we propose **Le**arn-**t**o-**S**hare (*LeTS*), a new transfer-learning framework that exploits both computation- and parameter-sharing to reduce computation and storage demands, while preserving a high sub-task performance. The key contributions of LeTS are as follows:

(i) We propose a new fine-tuning architecture design space (Figure 2(a)). The output of each self-attention layer will be aggregated at the end using a pooling layer and a bidirectional LSTM (Huang et al., 2015) (Bi-LSTM) to obtain the final classification result. In this way, modifications on the bottom layers do not influence the downstream computation which enables concurrent execution inside the transformer. Many computations can be bypassed when the Bi-LSTM uses the already computed attention as input. Also, we identify that even more computations can be reduced by using *layer normalization approximation* (Sec. 3.2).

(ii) We design a differentiable neural architecture search (NAS) algorithm to find an optimal fine-tuning architecture for a sub-task. Specifically, NAS selects the input to each attention layer and the final pooling layer. When a computed result is selected as the layers' input, we can bypass many computations to achieve computation sharing. A new computation-aware loss function for our search space is proposed to search models that can reduce computation and preserve task accuracy.

(iii) We treat the obtained **f**ine-tuning model weights as the sum of **p**re-trained weights and weight difference (**δ**): $W^{\mathbf{f}} = W^{\mathbf{p}} + W^{\delta}$, and propose a novel early-stage pruning method to design W^{δ}. A weight mask to represent pruning is generated for W^{δ} at the beginning of the fine-tuning using a single batch training. Instead of randomly initialization W^{δ}, we use a task-specific gradient accumulation to initialize the W^{δ} to get a robust weight mask.

(iv) We systematically integrate (ii) and (iii) to generate fine-tuning models with high task performance and low-computation and storage cost. During NAS, a generated mask from (ii) on the trainable parameters can better characterize the model performance. Also, during the online prototyping, when the input and output of a given linear layer is already computed, the computation can be reduced into a sparse-matrix multiplication by leveraging the sparsity produced from (iii).

Our framework offers a holistic solution to designing efficient fine-tuning language models for different computing environments. Extensive experiments show that LeTS reduces computation cost by a large margin while achieving a *competitive sub-task accuracy*. More specifically, for computing and storage restricted platforms, LeTS yields 49.5% computation reduction by adding only 1.4% extra parameters per task while preserving the accuracy (-0.2% on average) of the fine-tuned BERT (Devlin et al., 2019) on GLUE benchmarks. For a computing environment with low-cost storage budget, LeTS can achieve 40.2% computation reduction with no accuracy loss (+0.3%). LeTS becomes more powerful in saving computations with the increasing amount of sub-tasks. For BERT$_{\text{BASE}}$, LeTS requires 7.2 GFLOPs[1] for every newly added task compared to 22.5 GFLOPs of a fine-tuned BERT$_{\text{BASE}}$.

LeTS is the first framework that considers both computation and storage efficiency in fine-tuning for multi-task NLP. Our work can be combined with model compression techniques (Lan et al., 2020; Sanh et al., 2019) to enable agile and efficient NLP evaluation.

2. Overview

In this section, we discuss the key design components in LeTS. The detailed design flow is shown in Algorithm 1.

▪ **Motivation.** In a real-time multi-task evaluation, an input query is evaluated by many fine-tuned transformers at the same time. Each one focuses on one specific sub-task and some tasks may depend on the computation result from others. For instance, multiple tasks exist in document editing software (e.g., Google Doc or Microsoft Word), such as analyzing tone, checking grammar and then generating editing suggestions. Yet, the traditional fine-tuning method is extremely inefficient as the required computation and parameters grow linearly to the number of sub-tasks, which incurs the degraded quality of service and user experience. In this work, we aim to yield speedup through computation

[1] 1 GFLOPs = 1 billion floating-point operations

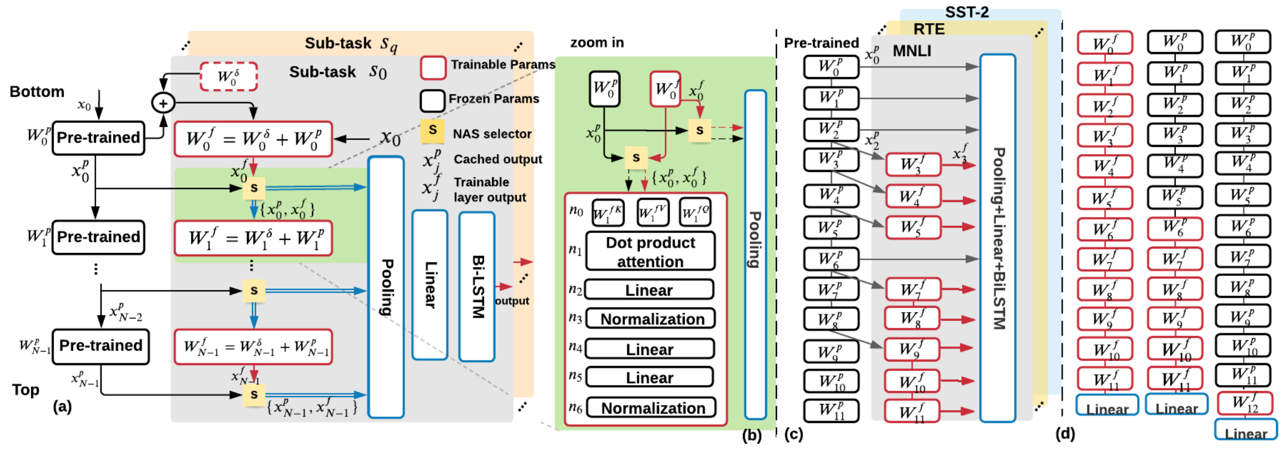


Figure 2: (a) The search space of LeTS for computation and parameter sharing. x_j^p can be reused by all the sub-tasks. For each sub-task, NAS selectors choose the inputs to the pooling layer and the next layer. W_j^δ is sparse (pruned by Delta-Pruning). (b) Zoom-in of a single self-attention layer. (c) An example model on BERT$_{\text{BASE}}$ produced by the search. (d) Standard fine-tuning architecture / Freezing bottom 6 layers / Appending 1 layer on top of the pre-trained model.

reuse for multi-task evaluation. Different from previous work (Xin et al., 2020) that uses layer output entropy to stop the execution earlier, LeTS can generate a guaranteed speedup that is not input-dependent.

▪ **Limitation of traditional fine-tuning procedures.** We observe three limitations that hinge the parallelization and computation sharing in traditional fine-tuned procedures: (1) The computation of an attention layer can only start execution when all its previous layers yield the results. (2) Any modification of the bottom layers change the subsequent computation, thus re-computation is required. (3) Although previous parameter-sharing work (Guo et al., 2020) can make W^δ sparse to reduce parameter growth in a sub-task, this sparsity cannot be exploited to reduce computation.

▪ **LeTS design.** Motivated by these observations, we propose a novel fine-tuning architecture that can reduce computation by reusing computed results. Also, the new architecture decouples the data dependency of different layers to enable speedup. The architectures can be formulated into a search space as shown in Figure 2(a). Given input query x_0, LeTS first caches all N attention layers' output ($x_j^p, j \in \{1...N\}$) computed from input query x_0 and pre-trained model W^p. For a given layer j in sub-task s, the input to the trainable layer W_j^f can be chosen from cached result x_{j-1}^p or the computed result x_{j-1}^f from the previous trainable layer. The attention output to the pooling layer can be chosen from (i) x_{j-1}^p or (ii) x_{j-1}^f. LeTS uses pooling and Bi-LSTM to aggregate the outputs from attention layers to generate the final result.

We use an example searched architecture in Figure 2(c) to illustrate the advantages of the new architecture :

(i) **Bypass self-attention layers.** When the cached result x_j^p is used by the final pooling layer and next trainable layer, the computation and parameters of the entire layer can be saved. This can be applied at layer W_j^p where $j \in \{0, 1, 2, 6\}$.

(ii) **Exploit the sparsity of W^δ.** LeTS can leverage the sparsity feature of W_j^δ. More specifically, when the input to the attention layer is x_j^p, LeTS computes $x_j^p \cdot W_j^\delta$ and adds it to a cached result. In Figure 2(c), when $j \in \{3, 4, 5, 7, 9\}$, the computation between x_j^p and W_j^{fK}, W_j^{fQ}, and W_j^{fV} (key/query/value parameter) can be reduced using this unstructured sparsity. Note that this sparsity matrix multiplication can be easily implemented under any popular machine learning libraries (Pytorch-Sparse; Tensorflow-Sparse).

(iii) **Bypass linear layers in self-attention.** The pooling layer extracts the first hidden vector of each layer's output as the aggregate representation. For W_j^f where $j \in \{3, 4, 5, 7, 9, 11\}$, only the first hidden vector of the output is used in the downstream computation; in this scenario, we move the pooling operation between n_3 and n_4 in Figure 2(b). As such, n_4 and n_5 in the self-attention layer can be reduced to a matrix-vector multiplication. The normalization layer (n_6) would be applied to only the pooling vector as an approximation for the original layer output (Sec. 3.2).

(iv) **Enable concurrent execution inside each transformer.** When the sub-tasks are dependent on each other and must be executed sequentially, the execution of our model can still be paralleled across computing devices inside each transformer. This is because the parameter tuning on the early layers do not necessarily influence the downstream computation anymore. In Figure 2(c), assuming all nine tasks share the same architecture, the execution time is determined by the critical path (W_0^p-to-W_8^p+ $9\times W_9^f$-to-W_{11}^f), which will be $9T + 3T \times 9$ (3.0× max speedup) compared to $12T \times 9$ of traditional BERT$_{\text{BASE}}$ fine-tuning (Figure 1(d)), assuming executing each attention layer takes time T.

A breakdown of the extra computation and parameter per task by leveraging (i)(ii)(iii) is shown in Figure 3. The computation and parameter overheads of the extra linear layers and Bi-LSTM are 0.01%/0.75% (Figure 3).

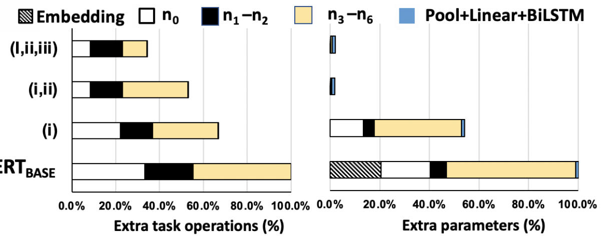


Figure 3: Extra computation and parameter breakdown leveraging (i) (ii) and (iii) for the example in Figure 2(c).

▪ **Neural architecture search for computation sharing.** The obtained architecture should achieve competitive sub-task accuracy with low extra computation operations. To address the above problem, we leverage a differentiable NAS algorithm with a computation-aware loss to reflect both computation cost and accuracy of a sub-task. (Sec. 3)

▪ W^{δ} **pruning.** Recent parameter-sharing approaches either add a new module between attention layers (Pfeiffer et al., 2020; Houlsby et al., 2019) or generate a weight mask simultaneously during fine-tuning using l_0 normalization (Guo et al., 2020). Yet, many works (Gale et al., 2019) have shown that l_0 regularization output is inconsistent for large-scale tasks. Also, the training parameters (i.e., weight mask and parameters) double during fine-tuning.

In this work, we treat the final fine-tuning weight W^f as the addition between pre-trained weight W^p and a weight difference (W^{δ}). By proposing an early-stage pruning approach, called *Delta-Pruning*, we compute the connection sensitivity of W^{δ}, which reveals the important connections in the W^{δ} for a given task (See Sec. 3.1). In this way, we can obtain the deterministic task-specific mask at the beginning of fine-tuning and use the generated mask to guide NAS.

3. Method

In this section, we detail the *Delta-Pruning* and Computation-aware neural architecture search algorithms in Algorithm 1.

3.1. Delta-Pruning in Early Stage

Delta-Pruning is motivated by SNIP (Lee et al., 2019) which targets to generate weight sparsity before training. We decompose the final fine-tuned weight (W^f) as Eq. (1).

$$W^f = W^{\mathrm{p}} + c \odot W^{\delta} \tag{1}$$

Here, $W^{\delta} \in \mathbb{R}^d$ is the fine-tuning weight difference, $\mathbf{c} \in \{0,1\}^d$ is the generated mask for W^{δ}. $\odot$ is an element-wise product. Given a task dataset $\mathcal{D}$, the goal of Delta-Pruning is to find mask c at the beginning of fine-tuning without interfering with the searching and final fine-tuning phase.

Algorithm 1 LeTS Design Flow.

```
input : Pre-trained model $W^{\mathrm{p}}$; Preserving parameter
    number $k$; Group restriction $G$ (Detailed in Sec. 4);
    Sub-task datasets $S = \{s_0, s_1, ..., s_q\}$.
output : Fine-tuning Policies $P_{\mathrm{out}}$ and Models $\mathcal{M}_{out}$.
 1: $\mathcal{M}_{out} \leftarrow \emptyset$, $P_{out} \leftarrow \emptyset$
 2: for $s_i$ in $S$ do
 3:     $W^{\tau} \leftarrow$ Generate_Search_Space$(W^{\mathrm{p}}, G)$
 4:     $c^{\tau} \leftarrow$ Delta_Pruning$(W^{\tau}, k, s_i)$ // $c^{\tau}$ is weight mask
 5:     $M_i, P_i \leftarrow$ Computation_Aware_Searching$(W^{\tau}, c^{\tau}, s_i)$
 6:     $c_i \leftarrow$ Delta_Pruning$(W^p, M_i, k, s_i)$
 7:     $M_i \leftarrow$ Final_Finetuning$(W^p, M_i, c_i, s_i)$
 8:     $M_{out} \leftarrow M_{out} \bigcup \{M_i\}$, $P_{out} \leftarrow P_{out} \bigcup \{P_i\}$
 9: end for
10: return $M_{out}, P_{out}$
```

Assuming the k parameters in W^{δ} is preserved, the constrained optimization problem can be described as Eq. (2):

$$\min_{\mathbf{c}, W^{\delta}} L(W^p + \mathbf{c} \odot W^{\delta}; \mathcal{D}) = \min_{\mathbf{c}, W^{\delta}} \frac{1}{n} \sum_{i=1}^{n} \ell(W^{\mathrm{p}} + \mathbf{c} \odot W^{\delta}; (x_i, y_i))$$
$$s.t. W^{\delta} \in \mathbb{R}^d, \mathbf{c} \in \{0,1\}^d, \ ||\mathbf{c}||_0 \leq k \tag{2}$$

Directly optimizing Eq. (2) using l_0 normalization will double the learnable parameters (Louizos et al., 2018) and is unstable for large-scale tasks (Gale et al., 2019). It is even more difficult to search l_0 masks together with architecture parameters in the DNAS algorithm (Sec 3.2). In this work, we intend to measure the effect of a connection e in W^{δ} on the loss function. Specifically, if removing W_e^{δ} does not show enough loss variation (ΔL_e), we set $c_e = 0$ to mask the gradient W^{δ} during training. Two challenges exist in computing ΔL_e: (i) Removing each connection in W_e^{δ} and check the variation in loss is computation-consuming. (ii) W_e^{δ} is unknown at the beginning of fine-tuning. A random initialization method cannot reflect the fully fine-tuned W^{δ}.

To resolve (i), we relax the binary constrain on c to a continuous space and compute the gradient of L with respect to c_e as g_e (Eq. (3)). Based on the intuition that the magnitude of derivative of c_e when $c_e = 1$ shows whether the parameter W_e^{δ} has a considerable effect on the loss or not, we use g_e to approximate ΔL_e for removing connection e in W_e^{δ}. As such, we define the connection sensitivity s_e for W_e^{δ} to be the g_e normalized by the sum of g_e in the network (Eq. (4)).

$$\Delta L_e(W^f; \mathcal{D}) \approx g_e(W^f; \mathcal{D}) = \left. \frac{\partial L(W_p + \mathbf{c} \odot W^{\delta}; \mathcal{D})}{\partial c_e} \right|_{\mathbf{c}=1} \tag{3}$$

$$s_e = \frac{|g_e(W^f; \mathcal{D})|}{\sum_{k=1}^{d} |g_k(W^f; \mathcal{D})|} \tag{4}$$

Then, assuming k parameters are pruned in W^{δ}, we generate mask $\mathbf{c}$ using a salient criterion computed from connection sensitivity s as Eq. (5):

$$c_e = \mathbb{1}[s_e - \widetilde{s}_k \geq 0],\ \forall e \in \{1...d\} \tag{5}$$

Here, $\widetilde{s}_k$ is the k-th largest element in the vector s and $\mathbb{1}[\cdot]$ is the indicator function.

To resolve (ii), we first learn the weight difference initialization by warm-up the fine-tuning using $\mathcal{D}$ for steps N_{steps} and get $\widetilde{W}^p$. We then approximate W^δ using a task-specific initialization as $\widetilde{W}^\delta = \widetilde{W}^p - W^p$. Our ablation study shows that using task-specific warm-up shows better results compared to random initialization as this accumulation of gradients can better reflect the final weight difference distribution.

3.2. Differentiable Neural Architecture Search for Computation Sharing

As discussed in Sec. 2, a promising task-specific fine-tuning architecture should yield low extra computation cost and high task accuracy. We formulate the selection of fine-tuning model as a bi-level non-convex optimization problem as shown in Eq. (6).

$$\min_{a \in \mathcal{A}} \min_{w_a} \mathcal{L}(a, w_a) \tag{6}$$

Here, $\mathcal{A}$ is a new search space proposed in LeTS, $a \in \mathcal{A}$ is a set of continuous variables in the NAS selectors that specifies a possible architecture, W_a is the selected fine-tuning architectures from the search space $\mathcal{A}$ given a. The loss function $\mathcal{L}$ penalizes both accuracy degradation as well as the increase of extra computations.

■ **Search space.** As shown in Figure 2(a), we decouple data dependency across layers by using a pooling layer, a linear layer, and a Bi-LSTM to aggregate all layers' output for final classification. The pooling layer uses the first hidden vector corresponding to the first token (i.e., [CLS] token) (Devlin et al., 2019) as the layer presentation. The pooling output vectors are then fed into a linear layer and a Bi-LSTM.

LeTS first builds up a stochastic super network W^τ for the searching phase. Before searching, we copy the weights from W^p to W^τ trainable layers and disable the gradient computation based on c^τ which is the weight difference mask obtained from the Delta-Pruning. Two decisions should be made in each attention layer W_j^τ: (i) the input to the trainable layer. It can be either the cached result (x_{j-1}^p) or the output from the previous trainable layer (x_{j-1}^f) (ii) the output to the pooling layer from layer j. It can either x_j^p or x_j^f. The total size of the search space would be 4^N where N is the layer number in the pre-training mode ($\approx 10^{15}$ for BERT$_{\text{LARGE}}$).

Two architecture selectors (s_{ij} , $i \in \{0, 1\}$) are used to decide (i) and (ii) in layer j respectively ($j \in 1...N$). Each s_{ij} is associated with an architecture vector $\mathbf{a}_{ij}$ (1-by-2). We relax the choice of the architecture selection to a Gumbel Softmax (Jang et al., 2016) over the two possible sources:

$$\overline{x_j^{in}} = [x_{j-1}^p; x_{j-1}^f] \cdot Gumbel(\mathbf{a}_{ij}) \tag{7}$$

Here, $\overline{x_j^{in}}$ is the input to the jth trainable layer in W^τ. [;] is a concatenation operation. *Gumbel* converts $\mathbf{a}_{ij}$ into a probability vector which is used to approximate discrete categorical selection. (Detailed in Appendix C) A temperature parameter T is associated with the *Gumbel* function to control its distribution. When T is high, $Gumbel(\mathbf{a}_{ij})$ becomes a continuous random variable and when T is low, $Gumbel(\mathbf{a}_{ij})$ is close to a discrete selection. During the search, we gradually lower T in *Gumbel* to guide NAS.

■ **Search algorithm and final fine-tuning architecture.** We alternatively update the two variables ($\mathbf{a}$ and W_a^τ under mask c^τ to solve the bi-level optimization problem in Eq. (6). More specifically, we leverage second-order approximation (Liu et al., 2019a) (Equations in Appendix C) to update $\mathbf{a}$ since: (i) The total parameters in $\mathbf{a}$ is not large ($\sim$100). As such, it is feasible to use the second-order approximation although it requires more the computation; (ii) second-order approximation can yield better solutions compared to gradient descent as shown in (Liu et al., 2019a).

When searching ends, we choose the final connectivity using $\mathbf{a}$ in jth layer. Precisely, the input connectivity is chosen as $x_{j-1}^t = \arg\max_{t \in \{p,f\}} a_{0jt}$ and the output to the pooling layer is $x_{j-1}^t = \arg\max_{t \in \{p,f\}} a_{1jt}$. Then, we apply an optimization on attention layers where s_{1j} chooses x_{j-1}^f and $s_{0,j+1}$ chooses x_j^p to reduce even more computation. We move the final pooling operation between n_3 and n_4 in Figure 2(b). In this way, the computation of the following linear layers (n_4, n_5) can be reduced to a matrix-vector multiplication. We refer to this method as *normalization approximation* as the variance and mean of the first latent vector from n_5 will approximate the mean and variance of the original n_5 output during layer normalization (n_6).

■ **Computation-aware loss function.** To consider both the task accuracy and computation cost, we define the loss function of LeTS's online searching phase as follows:

$$\mathcal{L} = CE(\mathbf{a}, W^\tau) \cdot \alpha log(E_{ops}(\mathbf{a}, W^\tau))^\beta \tag{8}$$

$CE(\mathbf{a}, W^\tau)$ is the cross-entropy loss given the architecture parameter $\mathbf{a}$ and the super net W^τ. α, β is the exponential factor that controls the magnitude of the operation terms. For the E_{ops}, we compute the expectation of operation over the architecture parameters $\mathbf{a}$:

$$E_{ops} = \sum \sum_j [Gumbel(\mathbf{a}_{0j}) \cdot Gumbel(\mathbf{a}_{1j})^T] \odot ops(W_j^\tau) \tag{9}$$

$ops(W_j^\tau)$ returns the number operations in layer j based on the selected combination of s_{0j} and s_{1j} into a 2-by-2 matrix. More specifically (See Figure 2(c)), (i) if both s_{1j} and s_{0j} select x_{j-1}^p as their input. then the computation of the entire layer j can be bypassed. (ii) If s_{0j} selects x_{j-1}^p and s_{1j} selects x_{j-1}^f as its input, then the computation between x_{j-1}^p and W_j^{fK}, W_j^{fK}, W_j^{fK} would become sparse-matrix multiplication by leveraging the sparse feature of W_j^δ. The number of operations is computed according to the sparsity ratio of W_j^τ. Beyond the above two cases, the computation number of a traditional self-attention layer is returned. Note that the *ops* return a matrix of constant values given W_j^τ. As such, the E_{ops} is differentiable to the architecture parameters $\mathbf{a}_{0j}$ and $\mathbf{a}_{1j}$ to search for computation-efficient models.

4. Evaluation

4.1. Experiment setup

■ **Datasets.** We evaluate LeTS on General Language Understanding Evaluation (GLUE) benchmark (Wang et al., 2018) consists of the following nine tasks: The Corpus of Linguistic Acceptability (CoLA). The Stanford Sentiment Treebank (SST-2). The Microsoft Research Paraphrase Corpus (MRPC). The Quora Question Pairs (QQP). The Semantic Textual Similarity Benchmark (STS-B). The Multi-Genre Natural Language Inference Corpus (MNLI) (We test on both matched domain MNLI$_m$ and mismatched domain MNLI$_{mm}$). The Stanford Question Answering Dataset (QNLI). The Recognizing Textual Entailment (RTE).

■ **Metrics.** We report Matthew's correlation for CoLA, Spearman for STS-B, F1 score for MRPC/QQP, and accuracy for MNLI/QNLI/SST-2/RTE, respectively. For computation efficiency, we report **max speedup** assuming the sub-tasks are dependent on each other. Also, we show **new FLOPs per task** over the FLOPs of a fine-tuned BERT and **total operations** that are required to compute the nine sub-tasks. For parameter efficiency, we report **total parameters** and **new parameters per task**.

■ **Baselines.** All previous parameter-sharing works are tested on BERT$_{\text{LARGE}}$ model (Devlin et al., 2019). We compare LeTS with the following baselines: (i) **Full fine-tuning** on BERT$_{\text{LARGE}}$ in a traditional way; (iii) **Adapter** (Houlsby et al., 2019). (iv) **DiffPruning** (Guo et al., 2020). (v) **BitFit** (Ravfogel & Goldberg), which fine-tunes only the bias parameters using a large learning rate. (See Sec. 5)

Also, we compare LeTS with model compression works, such as **DistillBERT** , **MobileBERT** (Sun et al., 2020) and **TinyBERT** (Jiao et al., 2020) which compressed the BERT$_{\text{BASE}}$ through knowledge distillation (Sec. 5). This comparison is conducted on BERT$_{\text{BASE}}$.

Note that all previous parameter-sharing works cannot reduce the computation overhead for multi-task evaluation. Thus, we build two extra baselines: (vi) We freeze bottom k self-attention layers and fine-tune the top layers. (vii) We append k new layers at the top of the pre-trained model and freeze the pre-trained weights (Figure 2(d)).

■ **LeTS design settings.** We leverage LeTS to design fine-tuning models for platforms with different computing/storage budgets: (i) We search task-specific architectures for each task in GLUE and fine-tuning it with the generated sparse mask (denoted as **LeTS-(p,c)**[2]). (ii) During the final fine-tuning, we also conduct an ablation study by removing the weight mask to achieve better accuracy (denoted as **LeTS-(c)**). This is suitable for computing platforms with a low-cost storage budget. (ii) To maximize the parallelism in a searched model, we decouple the attention layers into **g** groups (denoted as **LeTS-G-g**) and require the first layer in each group to use the cached inputs (x_{j-1}^p); thus the evaluation of different groups can be executed concurrently. Inside each group, we still apply DNAS to decide the connections.

■ **Hyperparameters.** The DNAS method takes 1 GPU day per task which is less than 0.5% of the pre-training cost of BERT$_{\text{LARGE}}$ (Devlin et al., 2019). The *max input length* for BERT is set to 128 to match previous baselines. Our pre-trained models and code base are from (Wolf et al., 2020). We use $N_{steps} = 100$ to initialize W^δ (Sec. 3.1). Inspired by (Ravfogel & Goldberg) that the bias terms requires a larger learning rate to achieve better fine-tuning results, we apply two optimizers with different learning rate scheduler to update the bias terms ($lr_b \in \{1e^{-3}, 5e^{-4}\}$) and other parts ($lr_w \in \{2e^{-5}, 1e^{-5}\}$) separately during the final fine-tuning. Details of other hyperparameters are shown in Appendix A. Training time and overheads are reported in Appendix B.

4.2. Results

■ **Comparison to baselines on GLUE dataset.** Our comparison with the baseline methods is shown in Table 1. LeTS-(c)/-(p,c) can achieve similar performance (+0.3%/-0.2% on average) to a fully fine-tuned BERT$_{\text{LARGE}}$ model while saving 40.2%/49.5% computation. With a more aggressive setting, LeTS-G-4 can reduce 57.0% computation (3.84× speedup) while matching the task performance of Adapters. In the meantime, LeTS-(p,c)/-G-4 only adds 1.4% parameters per task (including the Linear and Bi-LSTM layers), which is more parameter-efficient than Adapters. LeTS illustrates a trade-off between concurrent execution speedup and multi-task performance which is not done by previous works.

Compared to DistillBERT$_6$, MobileBERT, and TinyBERT$_6$ that reduce the total computation by 50.4%/28.3%/50.4% on BERT$_{\text{BASE}}$, LeTS-G-3/-G-4 shows 56%/62% computation reduction while preserving a high task performance (-0.4%/-0.7%) compared to the fine-tuned model. For fiercely compressed models (e.g., TinyBERT-4, MobileBERT$_{\text{TINY}}$,DistillBERT-4), they show large performance degradation (-2.3%/-2.6%/-7.7%) compared to the full fine-tuning model although saving more computations than LeTS. Also, LeTS shows the lowest parameter overhead (1.15×) compared to all compression models.

[2]**p** and **c** stand for parameter/computation-sharing

Compared to 'Freezing Bot-12' and 'Appending Top-1', LeTS also achieves better task performance (+1.4%/+8.4%) on average. That is because we relax the fine-tuning constraints on all the layers and aggregate the results for final classification. Also, when the number of sub-tasks increases, LeTS can save even more computations compared to Freezing Bot-12 (42.6% new FLOPs per task compare to 50%).

▪ **Comparison of Delta-Pruning with other parameter-sharing methods.** Tab. 2 shows the performance of *Delta-Pruning* compared to previous parameter-sharing works. To make a fair comparison, the result is tested using the original fine-tuning architecture. With the sparsity ratios restriction as 0.1%/0.25%/0.5% per task, LeTS achieves 2.4%/0.8/0.6% average performance increase compared to DiffPruning. This shows Delta-Pruning is effective to preserve task accuracy compared to the l_0 regularization method.

4.3. Sensitivity and Ablation study.

▪ **Varying the sparsity constraint on BERT$_{\text{LARGE}}$ model / Weight mask distribution.** We also conduct a sensitivity analysis using Delta-Pruning with various sparsity ratios (0.1%/0.25%/0.5%) across GLUE benchmarks (Table 2). Different tasks show different sensitivity with the growth of sparsity ratio. A better trade-off between accuracy v.s. sparsity ratio can be achieved through grid-search for each given task. Also, we show the distribution of weight masks for each layer varies across benchmarks (Figure 9). We hypothesize that when the tasks' inputs or outputs are related (e.g., QQP and QNLI both encode questions / MPRC and STS-B both generate similarity), they reveal similar mask distribution. This indicates that adding a uniform module (e.g., Adapter) between each layer for a task is sub-optimal.

▪ **Sensitivity to computation sharing ratio / Ablation to computation-aware loss function.** To shows the capability of LeTS in reducing computation while maintaining a high task accuracy, we perform NAS for 2 tasks multiple times (with different α, β in the loss function) and samples different architectures from the distribution. Figure 4 shows the results between extra operations v.s. task accuracy (on GLUE dev set). Freezing bot-k layers cannot preserve task accuracy with the increasing of k, while the architecture searched from LeTS does not show a large performance drop. LeTS also presents better task accuracy compared to the random sampled architectures, which shows that our DNAS algorithm can improve the quality of the searched model. When removing the computation-aware loss function, DNAS tends to select more trainable matrices to preserve task performance and the searched model cannot fully exploit computation sharing.

5. Related Work

Fine-tuning for transfer learning. Transfer a pre-trained model to a task of interest can be achieved by fine-tuning all the weights on that single task (Howard & Ruder, 2018).

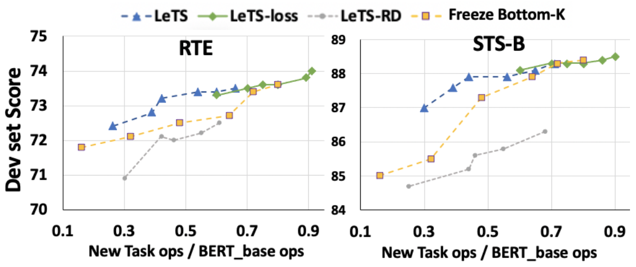


Figure 4: Sensitivity to computation sharing ratio (performed on BERT$_{\text{BASE}}$).

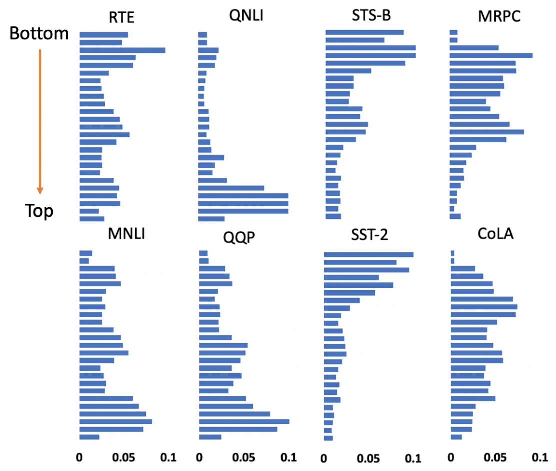


Figure 5: Distribution of LeTS's task-specific weight masks. (performed on BERT$_{\text{LARGE}}$)

Recent advances in text classification (Dai et al., 2019; Liu et al., 2019c; Joshi et al., 2020; Yang et al., 2019) have been achieved by fine-tuning a pre-trained transformer (Vaswani et al., 2017). However, it modifies all the weights of the network which is parameter inefficient for downstream tasks.

Multi-task learning. Multi-task learning (MTL) learns models on multiple tasks simultaneously and utilizes them across a diverse range of tasks (Caruana, 1997). MTL has been widely exploited using BERT and shows good performance on multiple text classification tasks (Liu et al., 2019b; Clark et al., 2019). In this work, we assume multiple tasks arrive in stream (i.e., online setting) and thus jointly training is not available as discussed in Sec. 1. Moreover, it is challenging to balance multiple tasks and solve them equally well in training (Stickland & Murray, 2019).

Parameter sharing for fine-tuning. Adapter is an alternation for parameter-efficient BERT models for online settings (Houlsby et al., 2019). It works well on machine translation (Bapna & Firat, 2019), cross-lingual transfer (Üstün et al., 2020), and task composition for transfer learning (Pfeiffer et al., 2020). These task-specific adapters are inserted between layers and cannot exploit the computation sharing because of the modification on the bottom layers. Recent work (Guo et al., 2020) use l_0 normalization to train a mask during fine-tuning for multi-task NLP. In this paper, we propose a novel method to prune weight difference based on SNIP (Lee et al., 2019) to condense the task-specific

Table 1: Comparison between LeTS and the baseline parameter-sharing works on BERT$_{\text{LRAGE}}$ using GLUE test set.

	Max Speedup [†]	New FLOPs Per Task% (Total FLOPs %)	Total Params %	New params Per Task	QNLI[‡]	SST-2	MNLI$_{m}$/$_{mm}$	CoLA	MRPC	STS-B	RTE	QQP	Avg
Full fine-tuning	1.00×	100% (900%)	9.00×	100%	92.7	**94.9**	86.7/85.9	60.5	89.3	86.5	70.1	**72.1**	**80.9**
Full fine-tuning [*]	1.00×	100% (900%)	9.00×	100%	**93.4**	94.1	**86.7/86.0**	59.6	88.9	86.6	71.2	71.7	80.6
Adapters (8-256)	1.00×	100% (900%)	1.32×	3.6%	90.7	94.0	84.9/85.1	59.5	89.5	86.9	**71.5**	71.8	80.4
Adapters (64)	1.00×	100% (900%)	1.18×	2.1%	91.4	94.2	85.3/84.6	56.9	89.6	**87.3**	68.6	71.8	79.8
Diff pruning	1.00×	100% (900%)	**1.05**×	**0.5%**	92.9	93.8	85.7/85.6	60.5	87.0	83.5	68.1	70.6	79.4
Diff pruning (struct.)	1.00×	100% (900%)	**1.05**×	**0.5%**	93.3	94.1	86.4/86.0	**61.1**	**89.7**	86.0	70.6	71.1	80.6
Freeze Bot-12	1.80×	50.0% (500%)	5.05×	45%	91.5	94.0	85.6/84.5	56.2	88.3	83.5	69.3	70.8	79.1
Freeze Bot-23	**6.75**×	**4.2% (138%)**	1.34×	3.7%	79.8	91.6	71.4/72.9	40.2	80.1	67.3	58.6	63.3	68.2
Append Top-1	**6.75**×	**4.2% (138%)**	1.34×	3.7%	82.1	91.9	75.7/74.6	43.4	83.4	81.2	59.8	66.6	72.1
LeTS-G-4 (p,c)	**3.84**×	**34.7% (387.3%)**	**1.13** ×	**1.4%(0.4%)**[$]	92.5	93.8	85.3/84.8	59.8	88.6	86.4	70.8	71.1	80.1
LeTS (c)	2.60×	51.9% (537.9%)	6.66×	62.9% (0.4%)	**92.9**	**94.5**	**86.4/86.0**	**61.1**	**89.0**	**86.8**	**71.6**	**71.7**	**80.9**
LeTS (p,c)	2.84×	42.6% (454.2%)	**1.13**×	**1.4%(0.4%)**[$]	92.6	94.2	85.5/85.1	60.4	88.9	86.5	71.4	71.1	80.4

[*] Fine-tuning results of our pre-trained BERT$_{\text{LARGE}}$ from huggingface (Wolf et al., 2020).
[‡] Besides DiffPruning and our results, previous works are reported on the old QNLI test set in the GLUE benchmark. The average is calculated without QNLI.
[†] When the sub-tasks are dependent on each other, max speedup can be achieved through concurrent execution. [$] The percentage in bracket is the parameter overhead of Bi-LSTM and linear layers.

Table 2: Sensitivity study to sparsity ratio constraint and comparison to parameter-sharing baselines on GLUE dev dataset.

	Total Params %	New params Per Task	QNLI	SST-2	MNLI$_{m}$/$_{mm}$	CoLA	MRPC	STS-B	RTE	QQP	Avg
Full fine-tuning	9.00×	100%	**93.5**	94.1	**86.5/87.1**	62.8	**91.9**	**89.8**	71.8	**87.6**	**85.0**
Diff-Pruning (struct.)	1.01×	0.1%	92.7	93.3	85.6/85.9	58.0	87.4	86.3	68.6	85.2	82.5
Diff pruning (struct.)	1.03×	0.25%	93.2	**94.2**	86.2/86.5	63.3	90.9	88.4	71.5	86.1	84.5
Diff pruning (struct.)	1.05×	0.5%	93.4	**94.2**	86.4/86.9	**63.5**	91.3	89.5	71.5	86.6	84.8
BitFit[*]	1.01×	0.08%	91.1	93.3	-	62.9	91.5	89.5	**75.1**	**87.6**	-
LeTS (p)	1.01×	0.1%	92.3	93.3	85.3/85.7	63.5	91.6	89.6	75.1	87.4	84.9
LeTS (p)	1.03×	0.25%	92.7	93.9	85.9/86.2	64.1	91.7	**89.8**	75.7	**87.6**	85.3
LeTS (p)	1.05×	0.5%	**92.9**	**94.0**	**86.4/86.2**	**64.4**	**91.9**	**89.8**	**75.8**	**87.6**	**85.4**

[*] BitFit does not report all the performance on GLUE dev set.

Table 3: Comparison between LeTS and the model compression works on BERT$_{\text{BASE}}$ using GLUE test set.

	Max Speedup	New FLOPs Per Task% (Total FLOPs %)	Total Params over BERT$_{\text{BASE}}$	Total Params	QNLI[*]	SST-2	MNLI$_{m}$/$_{mm}$	CoLA	MRPC	STS-B	RTE	QQP	Avg
Full-finetuning	1.00×	100% (900%)	9.00×	110M × 9	90.9	93.4	83.9/83.4	52.8	87.5	85.2	67.0	71.1	78.0
DistillBERT$_4$	3.00×	33.5% (300%)	4.27×	52.2M × 9	85.2	91.4	78.9/78.0	32.8	82.4	76.1	54.1	68.5	70.3
MobileBERT$_{\text{TINY}}$	8.55×	13.6% (123%)	1.24×	15.1M × 9	89.5	91.7	81.5/81.6	46.7	87.9	80.1	65.1	68.9	75.4
TinyBERT$_4$	9.40×	5.2% (46.8%)	1.19×	14.5M × 9	87.7	92.6	82.5/81.8	44.1	86.4	80.4	66.6	71.3	75.7
DistillBERT$_6$	2.00×	49.5% (446%)	5.48×	67.0M × 9	88.9	92.5	82.6/81.3	49.0	86.9	81.3	58.4	70.1	75.3
MobileBERT w/o OPT	1.78×	71.7% (645%)	2.07×	25.3M × 9	91.6	92.6	84.3/83.4	51.1	88.8	84.8	70.4	70.5	78.2
TinyBERT$_6$	2.00×	49.5% (446%)	5.48×	67.0M × 9	90.4	93.1	84.6/83.2	51.1	87.3	83.7	70.0	71.6	78.1
LeTS-G-3 (p,c)	2.83×	36.7% (397%)	1.15×	127M	90.4	92.2	82.8/81.8	50.5	88.3	84.6	70.0	70.3	77.6
LeTS-G-4 (p,c)	3.77×	27.6% (323%)	1.15×	127M	90.0	92.0	82.6/81.6	49.9	87.9	84.5	69.5	70.1	77.3

[*] The average is calculated without QNLI.

knowledge which achieves better parameter efficiency.

Hardware-aware NAS. Recent advances in NAS leverage differentiable methods by relaxing the selection of architectures in a continuous space to reduce the high search cost of RL-based NAS (Zoph & Le, 2016; Tan et al., 2019; Tan & Le, 2019). Previous differentiable NAS work (Wu et al., 2019; Liu et al., 2019a; Wan et al., 2020) mainly focus on computer vision tasks. In LeTS, we combine the searching algorithm of (Wu et al., 2019) and (Liu et al., 2019a) to resolve a unique problem in NLP. LeTS also presents a novel search space with a computation-aware loss to search model with high task accuracy and computation sharing ratio.

Model compression. Model pruning is another way to reduce a single model size and computation. (Gordon et al., 2020) prune weights in BERT based on magnitude, (Guo et al., 2019) use iteratively reweighted l_1 minimization, and (Lan et al., 2019) leverage cross-layer parameter sharing. Other works distill knowledge from a pre-trained model down to a smaller student model (Sanh et al., 2019; Sun et al., 2020; Jiao et al., 2020). Note that LeTS is orthogonal to all these methods, as we did not modify the pre-trained parameters. we leave this combination as future work.

6. Conclusion

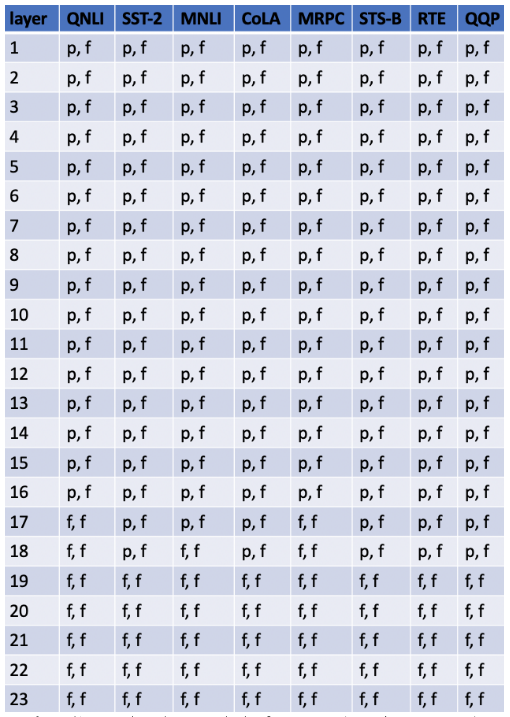
We propose LeTS, a transfer learning framework that achieves computation and parameter sharing when multiple tasks arriving in stream. LeTS proposes a novel architecture space that can reuse computed results to reduce computation. By leveraging NAS with a computation-aware loss function, LeTScan find models with high task performance and low computation overhead. By treating the fine-tuned weight as the sum of pre-trained weight and weight difference, we present a early stage pruning algorithm to compress weight difference without task performance decrease. The integration of the above novelty enables even more computation reduction by exploiting the sparsity of the weight difference. LeTS achieves better task performance compared to previous parameter-sharing only methods. Also, by leveraging computation sharing, LeTS engenders large computation reduction to enable scalable transfer learning.

A. Searched models / Explanation of metrics / Hyperparameters

In this section, we reveal the searched model of LeTS-(p,c) and LeTS-(c) of each sub-tasks searched on $\text{BERT}_{\text{LARGE}}$. Also, we detail the computation of our metrics used to characterize computation and parallelism (max speedup and new ops per task). At last, we report the hyperparameters used for searching/final fine-tuning.

A.1. Searched models for each task

The searched fine-tuning models on $\text{BERT}_{\text{LARGE}}$ for each sub-tasks of GLUE are shown in Fig. 6, p and f refer to the computed input from W^p_{j-1} and previous trainable layer output W^f_{j-1}. For each layer j, we report $(s_0,_s 1)$ where : 1) s_0 is the select input for the next trainable layer; the second is the input to the final aggregation layers (pooling+linear+Bi-LSTM discussed in Sec.2).



layer	QNLI	SST-2	MNLI	CoLA	MRPC	STS-B	RTE	QQP
1	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
2	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
3	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
4	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
5	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
6	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
7	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
8	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
9	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
10	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
11	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
12	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
13	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
14	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
15	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
16	p, f	p, f	p, f	p, f	p, f	p, f	p, f	p, f
17	f, f	p, f	p, f	p, f	f, f	p, f	p, f	p, f
18	f, f	p, f	f, f	p, f	f, f	p, f	p, f	p, f
19	f, f	f, f	f, f	f, f	f, f	f, f	f, f	f, f
20	f, f	f, f	f, f	f, f	f, f	f, f	f, f	f, f
21	f, f	f, f	f, f	f, f	f, f	f, f	f, f	f, f
22	f, f	f, f	f, f	f, f	f, f	f, f	f, f	f, f
23	f, f	f, f	f, f	f, f	f, f	f, f	f, f	f, f

Figure 6: Searched model for each given tasks using $\text{BERT}_{\text{LARGE}}$. (placeholder)

A.2. Metrics explanations

In Table 1 and Table 3, we report the *new operations per task* and *max speedup*. In this subsection, we detail the computation of these two metrics.

New operations per task We compute the new operations (FLOPs or Floating-point operations) introduced by the sub-task based on the searched result and weight mask. We normalize the number of new operations of a task s_i where $i \in 1\ldots,N$ using the total number of operations required for a single $\text{BERT}_{\text{LARGE}/\text{BASE}}$ (77.7 / 22.5 GFLOPs) as Eq. (10):

$$ops_{\%}(s_i) = \frac{ops(s_i)}{ops(\text{BERT}_{\text{LARGE}})} \times 100\% \tag{10}$$

This percentage $ops_{\%}$ indicates that you only need to extra $ops_{\%}$ new operations compared to adding an entire new transformer (100%) when adding the sub-task to your system.

New operations per task (%) reported in Table 1 and Table 3 is the average of $ops_{\%}$ for each GLUE sub-task:

$$new\_ops\_per\_task(s_i)(\%) = \frac{\sum_i^N (ops_{\%}(s_i))}{N} \tag{11}$$

Total operations (%) For LeTS, total operations of the nine tasks needs to add the overhead operations of computing pretrained weight and input:

$$Total\_ops(\%) = \frac{\sum_i^N (ops(s_i)) + overhead}{ops(\text{BERT})} \times 100\% \tag{12}$$

For example, the total operations of traditional fine-tuning method for the 9 GLUE tasks will be 9× of the operation of $\text{BERT}_{\text{LARGE}}$ (100%×9=900%) and the overhead/ops($\text{BERT}_{\text{LARGE}}$) is 0%. For freezing top-12 layers (Sec. 4), the new operations per task is 50% and the overhead/ops($\text{BERT}_{\text{LARGE}}$) is 50% [3], thus the total normalized operations would be 40%×9 + 50% = 500%. For LeTS, all the searched model does not take the computed results since layer 18-23 (Figure 6), as such the overhead/ops($\text{BERT}_{\text{LARGE}}$) would be 18/24≈ 75%. This overhead takes less portion with the increasing number of sub-tasks. Assuming the number of sub-task is N, the anticipated computation reduction v.s. the number of sub-tasks are visualized in Figure 7 for LeTS (p,c) and Freezing Top-12 on $\text{BERT}_{\text{LARGE}}$.

Max speedup When the sub-tasks are independent of each other, the user can leverage the computation reduction to achieve speedup. Yet, in many cases, the sub-tasks are chained together (i.e., data-dependent) and must be executed in order. In this scenario, LeTS's design space can yield fruitful speedup as we decouple the computation of different attention layers inside each transformer. We first identify the critical path (Hennessy & Patterson, 2011) (example computing max speedup for LeTS is showed in Figure 2(c) and Sec.2) of execution the 9 tasks. The max speedup in this case would be computed as:

$$max\_speedup = \frac{ops(BERT) \times N}{ops(critical\_path)} \tag{13}$$

[3]overhead is to compute the previous 12 layers between the pre-trained weights and input

Taking freezing top 12 layers as an example, ops(s_i)/ops(BERT$_{\text{LARGE}}$) = 50%, and the critical path would be overhead 50% and 50% computation time for each sub-task (max_speedup = 900%/500% = 1.8×).

Note that both the freezing top-12 and original fine-tuning architecture is included in our search space. Yet, fine-tuning approach is computationally inefficient and freezing top-12 layers sacrifices the task performance a lot. LeTS pushes the Pareto frontier between task performance and sub-task computation overhead.

A.3. Final Fine-tuning Hyperparameters

Table 4 shows the hyperparameters for training our final searched model on GLUE tasks. For final testing, we select the model that achieves the best validation (dev set) result. We use two learning rate schedulers for the bias term in transformer and all other parameters (including the added Bi-LSTM and linear layer).

Another thing worth mention is that the *max input length* (l_m) in our evaluation is set to 128 to match previous baselines. With larger l_m (e.g., 512), the overhead of the aggregation layers / pretrained model and input computation would take even less portion to the overall computation cost. The computation reduction of LeTS will also be more explicit. That is because the computation complexity of transformer is proportional to the input length (l) $\mathcal{O}(l^2)$.

■ **Temperature scheduling and searching**

During the search, the initial temperature T in Eq. 14 is set to 4.0 and exponentially annealed by $exp(-0.065) \approx 0.936$ for every $\frac{1}{10}$ epoch. We use a early stop mechanism that terminate the searching phase when the selected model does not change for $\frac{1}{10}$ epochs. Because the model parameters start from pretrained BERT, the searching phase converges faster than traditional DNAS. For the loss function in Sec.3, E_{ops} is represented in billion operations. We set α to 0.5 and β to 0.5. The learning rate of alpha is initialized to $5e-4$ which is updated using an Adam optimizer (default optimizer settings in huggingface (Wolf et al., 2020)).

B. Gumbel Softmax and Second-order approximation

■ **Gumbel Softmax equations** The architecture parameters discussed in Sec.3.2 will be converted to a probability vector using Gumbel Softmax equations. Specifically, the architecture parameters $\mathbf{a}_{ij}$ associate with the ith selector in jth layer are computed as Eq. (14).

$$P_{a_{ij}} = Gumbel(a_{ijt}|\mathbf{a}_{ij}) = \frac{exp((a_{ijt} + g_{ijt})/T))}{\sum_t exp((a_{ijt} + g_{ijt})/T))} \quad (14)$$

Here, $g_{ijt} \sim$ Gumbel$(0,1)$ is a random noise following the Gumbel distribution. The output is a probability vector $P_{a_{ij}}$ (2×1).

■ **Second-order approximation equations**

As discussed in Sec.3.2, we iteratively update weight and architecture parameters of the super network (a and W^{τ}). The gradient of weight parameters (denote W^{τ} as W in appendix for simplicity) are updated using traditional gradient decent. And the gradient of architecture parameters (a) are computed by a second-order approximation. Specifically, we split the training dataset as into two parts (D1 takes 80%, D2 takes 20%). The gradient of the architecture parameters can be approximated as Eq. 15:

$$\nabla_a \mathcal{L}(W_a, a) \approx \nabla_\alpha \mathcal{L}_{D2}(W - \xi \nabla_W \mathcal{L}_{D1}(W,a), a) \quad (15)$$

Here, W_a is the final pre-trained model given architecture parameter a. To the l.h.s of Eq. 15 means we can train a for one step after fine-tuning the entire model. To reduce the search cost, the idea of differentiable NAS (DNAS) is to approximate the final W_a by adapting W using only a single training step (Liu et al., 2019a). To prevent the searched model from over-fitting to the training dataset, we split the training datasets (D1, D2) to update weight and architecture parameters, respectively. The Eq. 15 can be expanded into Eq. 16:

$$\nabla_a \mathcal{L}_{D2}(W', a) - \xi \nabla^2_{a,W} \mathcal{L}_{D1}(W,a) \nabla_{W'} \mathcal{L}_{D2}(W', a) \quad (16)$$

Where $W' = W - \xi \nabla_W \mathcal{L}_{D1}(W,a)$. The second term can be computed through a finite difference approximation. Assume ε is a small scalar and $W^{\pm} = W \pm \varepsilon \nabla_{W'} \mathcal{L}_{D2}(W', a)$. Then:

$$\nabla^2_{\alpha,w} \mathcal{L}_{D1}(w,\alpha) \nabla_{w'} \mathcal{L}_{D2}(w',\alpha) \approx \frac{\nabla_\alpha \mathcal{L}_{D1}(w^+,\alpha) - \nabla_\alpha \mathcal{L}_{D1}(w^-,\alpha)}{2\varepsilon} \quad (17)$$

To sum up, during the architecture parameter update, we first compute the $\nabla_a \mathcal{L}_{D1}(w^+,\alpha)$ and $\nabla_a \mathcal{L}_{D1}(w^-,\alpha)$ through two forward and backward pass and approximate the second term in Eq. 16 using Eq. 17. Due to the small size of $\mathbf{a}$ in the super network, the second order approximation is feasible and model selection is more accurate than using naive gradient decent ($\xi = 0$).

C. Detailed design flow of Delta-pruning / Ablation study of gradient-accumulation initialization

For all tasks we initialize the training for 10 epochs, which is larger than the original fine-tuning ($\sim 2.5\times$ longer fine-tuning time). This is because the additional Bi-LSTM takes more time to converge.

■ **GPU memory overhead.** LeTS do not explicitly require more memory during final fine-tuning (1.2× than traditional

Table 4: Hyperparameters for final fine-tuning. We use polynomial decay scheduler in transformer ((Vaswani et al., 2017)) on two learning rate.

	QNLI	SST-2	MNLI	CoLA	MRPC	STS-B	RTE	QQP
Epochs	7	7	7	10	10	10	10	7
Batch size	16	16	16	16	16	16	16	16
Learning Rate (bias terms)	5e-4	5e-4	5e-4	1e-3	1e-3	1e-3	1e-3	5e-4
Learning Rate (other parameters)	2e-5	2e-5	2e-5	2e-5	2e-5	2e-5	2e-5	2e-5
Warm-up steps	1986	1256	7432	420	350	720	350	28318

Algorithm 2 Delta-pruning

```
input  Pre-trained parameter $W^p$, offset parameter $W^\delta$, the
       desired $W^\delta$ sparsity constraint $k$, a mini batch training
       dataset $D_b$
output  Mask c
 1: Warn up fine-tuning $W$ for 100 epoch and get $\widetilde{W}^f$
 2: $W^\delta \leftarrow \widetilde{W}^f - W^p$, $c \leftarrow 1^d$
 3: Set trainable mask and perform one mini-batch training
    to get $\Delta L(W^f; \mathcal{D})$ (Eq.3 in the paper).
 4: for i in $\{1...d\}$ do
 5:     score $s_e = \frac{|g_e(W^f;\mathcal{D})|}{\sum_{k=1}^{d} |g_k(W^f;\mathcal{D})|}$
 6: end for
 7: Descending sort all the score $s$.
 8: for i in $\{1...d\}$ do
 9:     $c_i \leftarrow 1[s_i - \widetilde{s}_k \leq 0]$
10: end for
```

fine-tuning) as the pre-trained parameters is frozen (no gradient consumption) and the pruning mask is generated ahead of fine-tuning. For DiffPruning, the pruning mask is searched during fine-tuning. As such, it takes more GPU memory consumption ($\sim 2\times$) than traditional fine-tuning approach.

■ **Task-specific initialization and sparsity.** For initialize W^δ in Delta-Pruning (Sec.3.1), we show that using a warm-up method to accumulate gradients can generate a weight mask that represent the final fine-tuning results. We visualize the weight mask generated from final W^δ and gradient accumulation $\widetilde{W}^d elta$ (N_{steps}=100) for selected tasks with the sparsity ratio k=0.25% as shown in Figure 9.

We also conduct an ablation study by replacing the gradient accumulation initialization with random initialization and the visualization of the mask is shown in Figure 9. The random initialization method also decrease the average task performance on GLUE dev set by x.x% on average compared to the gradient accumulation method (x.x% as shown in Table 2).

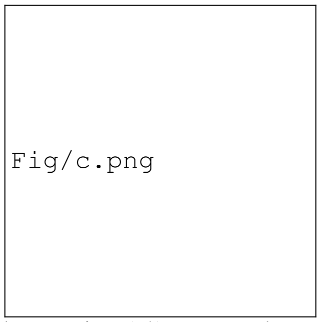


Figure 7: Total operation (%) v.s. number of sub-tasks for BERT-large.

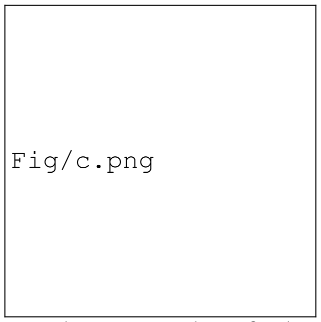


Figure 8: Max speedup v.s. number of sub-tasks for BERT-large.

References

Bapna, A. and Firat, O. Simple, scalable adaptation for neural machine translation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 1538–1548, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1165. URL https://www.aclweb.org/anthology/D19-1165.

Caruana, R. Multitask learning. *Machine learning*, 28(1): 41–75, 1997.

Clark, K., Luong, M.-T., Khandelwal, U., Manning, C. D., and Le, Q. V. BAM! born-again multi-task networks for natural language understanding. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 5931–5937, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1595. URL https://www.aclweb.org/anthology/P19-1595.

Consortium, C. Compute Express Link, a. URL https://www.computeexpresslink.org/.

Consortium, G.-Z. The Gen-Z Consortium, b. URL https://genzconsortium.org/.

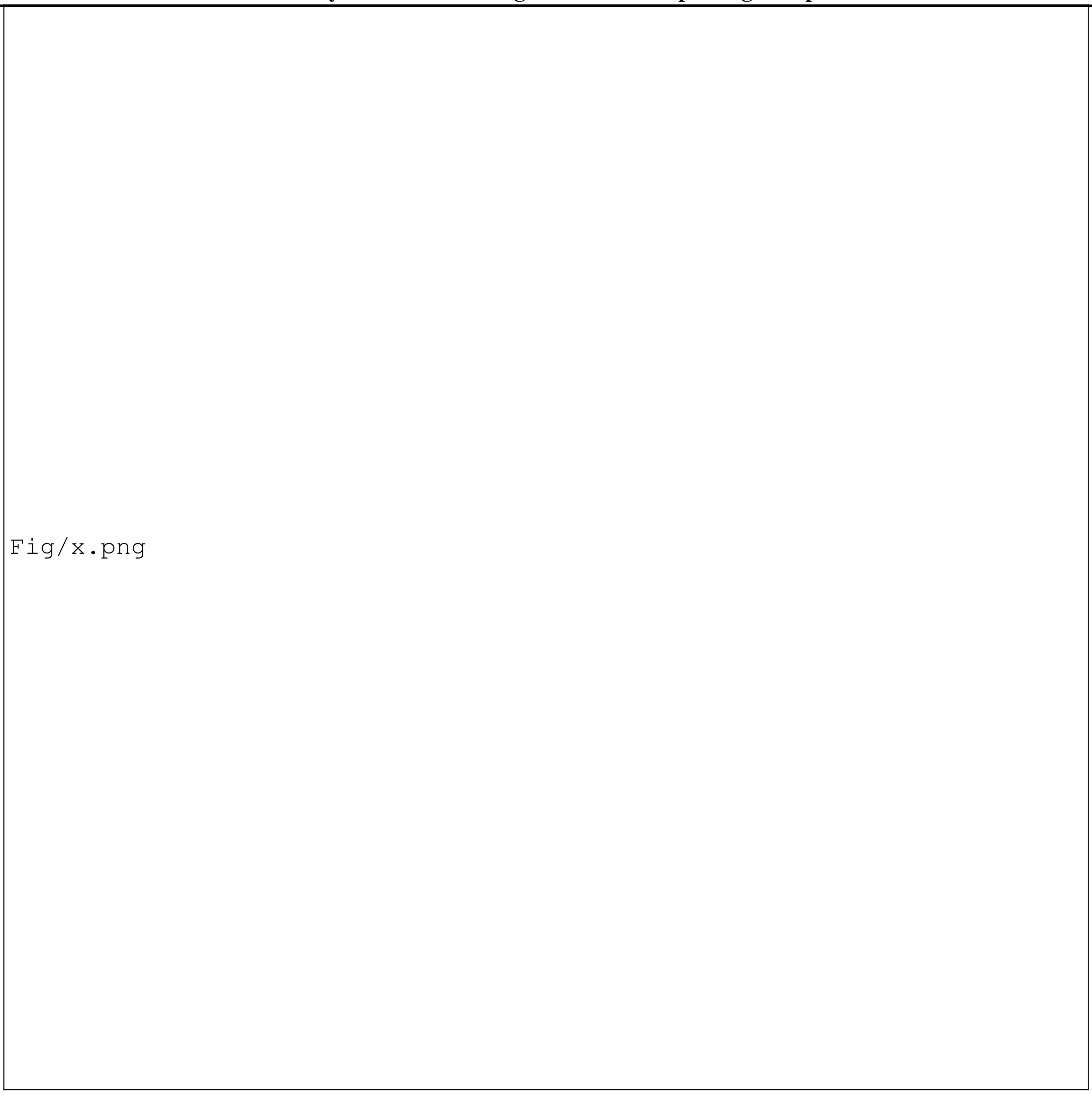


Figure 9: Visualization of final fine-tuning weight and the gradient accumulation for 100 steps. We show only the middle ($6th$) layer on selected tasks.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q., and Salakhutdinov, R. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 2978–2988, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1285. URL https://www.aclweb.org/anthology/P19-1285.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://www.aclweb.org/anthology/N19-1423.

Gale, T., Elsen, E., and Hooker, S. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

Gordon, M., Duh, K., and Andrews, N. Compressing bert: Studying the effects of weight pruning on transfer learning. pp. 143–155, 01 2020. doi: 10.18653/v1/2020.repl4nlp-1.18.

Guo, D., Rush, A. M., and Kim, Y. Parameter-efficient transfer learning with diff pruning, 2020.

Guo, F.-M., Liu, S., Mungall, F. S., Lin, X., and Wang, Y. Reweighted proximal pruning for large-scale language representation. *arXiv preprint arXiv:1909.12486*, 2019.

Hennessy, J. L. and Patterson, D. A. *Computer architecture: a quantitative approach*. Elsevier, 2011.

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pp. 2790–2799. PMLR, 2019.

Howard, J. and Ruder, S. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 328–339, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1031. URL https://www.aclweb.org/anthology/P18-1031.

Huang, Z., Xu, W., and Yu, K. Bidirectional LSTM-CRF models for sequence tagging. *CoRR*, abs/1508.01991, 2015. URL http://arxiv.org/abs/1508.01991.

Intel. Intel® Optane™ DC Persistent Memory, 2019. URL https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Jiao, X., Yin, Y., Shang, L., Jiang, X., Chen, X., Li, L., Wang, F., and Liu, Q. TinyBERT: Distilling BERT for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 4163–4174, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.372. URL https://www.aclweb.org/anthology/2020.findings-emnlp.372.

Joshi, M., Chen, D., Liu, Y., Weld, D. S., Zettlemoyer, L., and Levy, O. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics*, 8:64–77, 2020.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1eA7AEtvS.

Lee, N., Ajanthan, T., and Torr, P. SNIP: SINGLE-SHOT NETWORK PRUNING BASED ON CONNECTION SENSITIVITY. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=B1VZqjAcYX.

Liu, H., Simonyan, K., and Yang, Y. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019a. URL https://openreview.net/forum?id=S1eYHoC5FX.

Liu, X., He, P., Chen, W., and Gao, J. Multi-task deep neural networks for natural language understanding. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4487–4496, Florence, Italy, July 2019b. Association for Computational Linguistics. doi: 10.18653/v1/P19-1441. URL https://www.aclweb.org/anthology/P19-1441.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019c.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019d. URL http://arxiv.org/abs/1907.11692.

Louizos, C., Welling, M., and Kingma, D. P. Learning sparse neural networks through l_0 regularization. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=H1Y8hhg0b.

Pfeiffer, J., Rücklé, A., Poth, C., Kamath, A., Vulić, I., Ruder, S., Cho, K., and Gurevych, I. AdapterHub: A framework for adapting transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 46–54, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.7. URL https://www.aclweb.org/anthology/2020.emnlp-demos.7.

Pytorch-Sparse. Pytorch Sparse Library. URL https://pytorch.org/docs/master/sparse.html.

Ravfogel, E. B.-Z. S. and Goldberg, Y. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019. URL http://arxiv.org/abs/1910.01108.

Stickland, A. C. and Murray, I. Bert and pals: Projected attention layers for efficient adaptation in multi-task learning. In *International Conference on Machine Learning*, pp. 5986–5995. PMLR, 2019.

Sun, Z., Yu, H., Song, X., Liu, R., Yang, Y., and Zhou, D. MobileBERT: a compact task-agnostic BERT for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2158–2170, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.195. URL https://www.aclweb.org/anthology/2020.acl-main.195.

Tan, M. and Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pp. 6105–6114. PMLR, 2019.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

Tensorflow-Sparse. Tensorflow sparse matrix API. URL https://www.tensorflow.org/api_docs/python/tf/sparse/sparse_dense_matmul.

Üstün, A., Bisazza, A., Bouma, G., and van Noord, G. UDapter: Language adaptation for truly Universal Dependency parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2302–2315, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.180. URL https://www.aclweb.org/anthology/2020.emnlp-main.180.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Wan, A., Dai, X., Zhang, P., He, Z., Tian, Y., Xie, S., Wu, B., Yu, M., Xu, T., Chen, K., Vajda, P., and Gonzalez, J. E. Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pp. 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-5446. URL https://www.aclweb.org/anthology/W18-5446.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Huggingface's transformers: State-of-the-art natural language processing, 2020.

Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10734–10742, 2019.

Xin, J., Tang, R., Lee, J., Yu, Y., and Lin, J. Deebert: Dynamic early exiting for accelerating bert inference. *arXiv preprint arXiv:2004.12993*, 2020.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.