# AutoTinyBERT: Automatic Hyper-parameter Optimization for Efficient Pre-trained Language Models

**Yichun Yin[1], Cheng Chen[2*], Lifeng Shang[1], Xin Jiang[1], Xiao Chen[1], Qun Liu[1]**
[1]Huawei Noah's Ark Lab
[2]Department of Computer Science and Technology, Tsinghua University
{yinyichun,shang.lifeng,jiang.xin,chen.xiao2,qun.liu}@huawei.com
c-chen19@mails.tsinghua.edu.cn

## Abstract

Pre-trained language models (PLMs) have achieved great success in natural language processing. Most of PLMs follow the default setting of architecture hyper-parameters (e.g., the hidden dimension is a quarter of the intermediate dimension in feed-forward sub-networks) in BERT (Devlin et al., 2019). Few studies have been conducted to explore the design of architecture hyper-parameters in BERT, especially for the more *efficient* PLMs with tiny sizes, which are essential for practical deployment on resource-constrained devices. In this paper, we adopt the one-shot Neural Architecture Search (NAS) to automatically search architecture hyper-parameters. Specifically, we carefully design the techniques of one-shot learning and the search space to provide an adaptive and efficient development way of tiny PLMs for various latency constraints. We name our method AutoTinyBERT[1] and evaluate its effectiveness on the GLUE and SQuAD benchmarks. The extensive experiments show that our method outperforms both the SOTA search-based baseline (NAS-BERT) and the SOTA distillation-based methods (such as DistilBERT, TinyBERT, MiniLM and MobileBERT). In addition, based on the obtained architectures, we propose a more efficient development method that is even faster than the development of a single PLM.

## 1 Introduction

Pre-trained language models, such as BERT (Devlin et al., 2019), RoBERTa (Liu et al., 2019) and XLNet (Yang et al., 2019), have become prevalent in natural language processing. To improve model performance, most PLMs (e.g. ELECTRA (Clark et al., 2019) and GPT-2/3 (Radford et al., 2019; Brown et al., 2020)) follow the default rule of hyper-parameter setting[2] in BERT to scale up their model sizes. Due to its simplicity, this rule has been widely used and can help large PLMs obtain promising results (Brown et al., 2020).

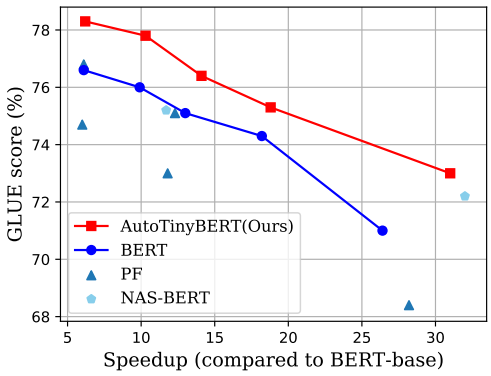


Figure 1: Inference speedup vs. GLUE scores. Under the same speedup constraint, our method outperforms both the default hyper-parameter setting of BERT (Devlin et al., 2019), PF (Turc et al., 2019)) and NAS-BERT (Xu et al., 2021). More details are in the Section 4.2.

In many industrial scenarios, we need to deploy PLMs on resource-constrained devices, such as smartphones and servers with limited computation power. Due to the expensive computation and slow inference speed, it is usually difficult to deploy PLMs such as BERT (12/24 layers, 110M/340M parameters) and GPT-2 (48 layers, 1.5B parameters) at their original scales. Therefore, there is an urgent need to develop PLMs with smaller sizes which have lower computation cost and inference latency. In this work, we focus on a specific type of *efficient* PLMs, which we define to have inference time less than 1/4 of BERT-base.[3]

*Contribution during internship at Noah's Ark Lab.

[1]Our code implementation and pre-trained models are available at https://github.com/huawei-noah/Pretrained-Language-Model .

[2]The default rule is $d^m = d^{q|k|v} = 1/4d^f$, which means the dimension of hidden vector $d^m$ is equal to the dimensions of query/key/value vector $d^{q|k|v}$ and a quarter of the intermediate size $d^f$ in feed-forward networks.

[3]We empirically find that being at least 4x faster is a basic requirement in practical deployment environment.



*Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, pages 5146–5157
August 1–6, 2021. 

Although, there have been quite a few work using knowledge distillation to build small PLMs (Sanh et al., 2019; Jiao et al., 2020b; Sun et al., 2019, 2020), all of them focus on the application of distillation techniques (Hinton et al., 2015; Romero et al., 2014) and do not study the effect of architecture hyper-parameter settings on model performance. Recently, neural architecture search and hyper-parameter optimization (Tan and Le, 2019; Han et al., 2020) have been widely explored in machine learning, mostly in computer vision, and have been proven to find better designs than heuristic ones. Inspired by this research, one problem that naturally arises is *can we find better settings of hyper-parameters*[4] *for efficient PLMs?*

In this paper, we argue that the conventional hyper-parameter setting is not best for efficient PLMs (as shown in Figure 1) and introduce a method to automatically search for the optimal hyper-parameters for specific latency constraints. Pre-training efficient PLMs is inevitably resource-consuming (Turc et al., 2019). Therefore, it is infeasible to directly evaluate millions of architectures. To tackle this challenge, we introduce the one-shot Neural Architecture Search (NAS) (Brock et al., 2018; Cai et al., 2018; Yu et al., 2020) to perform the automatic hyper-parameter optimization on efficient PLMs, named as AutoTinyBERT. Specifically, we first use the one-shot learning to obtain a big SuperPLM, which can act as *proxies* for all potential sub-architectures. *Proxy* means that when evaluating an architecture, we only need to extract the corresponding sub-model from the SuperPLM, instead of training the model from scratch. SuperPLM helps avoid the time-consuming pre-training process and makes the search process efficient. To make SuperPLM more effective, we propose practical techniques including the *head sub-matrix extraction* and *efficient batch-wise training*, and particularly limit the search space to the models with *identical layer structure*. Furthermore, by using SuperPLM, we leverage search algorithm (Xie and Yuille, 2017; Wang et al., 2020a) to find hyper-parameters for various latency constraints.

In the experiments, in addition to the pre-training setting (Devlin et al., 2019), we also consider the setting of task-agnostic BERT distillation (Sun et al., 2020) that pre-trains with the loss of knowledge distillation, to build efficient PLMs. Extensive results show that in pre-training setting, AutoTinyBERT not only consistently outperforms the BERT with conventional hyper-parameters under different latency constraints, but also outperforms NAS-BERT based on neural architecture search. In task-agnostic BERT distillation, AutoTinyBERT outperforms a series of existing SOTA methods of DistilBERT, TinyBERT and MobileBERT.

Our contributions are three-fold: (1) we explore the problem of how to design hyper-parameters for efficient PLMs and introduce an effective and efficient method: AutoTinyBERT; (2) we conduct extensive experiments in both scenarios of pre-training and knowledge distillation, and the results show our method consistently outperforms baselines under different latency constraints; (3) we summarize a fast rule and it develops an AutoTinyBERT for a specific constraint with even about 50% of the training time of a conventional PLM.

## 2 Preliminary

Before presenting our method, we first provide some details about the Transformer layer (Vaswani et al., 2017) to introduce the conventional hyper-parameter setting. Transformer layer includes two sub-structures: the multi-head attention (MHA) and the feed-forward network (FFN).

For clarity, we show the MHA as a decomposable structure, where the MHA includes $h$ individual and parallel self-attention modules (called heads). The output of MHA is obtained by summing the output of all heads. Specifically, each head is represented by four main matrices $\boldsymbol{W}_i^q \in \mathbb{R}^{d^m \times d^q/h}$, $\boldsymbol{W}_i^k \in \mathbb{R}^{d^m \times d^k/h}$, $\boldsymbol{W}_i^v \in \mathbb{R}^{d^m \times d^v/h}$ and $\boldsymbol{W}_i^o \in \mathbb{R}^{d^v/h \times d^o}$, and takes the hidden states[5] $\boldsymbol{H} \in \mathbb{R}^{l \times d^m}$ of the previous layer as input. The output of MHA is given by the following formulas:

$$
\begin{aligned}
&\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i = \boldsymbol{H}\boldsymbol{W}_i^q, \boldsymbol{H}\boldsymbol{W}_i^k, \boldsymbol{H}\boldsymbol{W}_i^v \\
&\mathrm{ATTN}(\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i) = \mathrm{softmax}(\frac{\boldsymbol{Q}_i {\boldsymbol{K}_i}^T}{\sqrt{d^{q|k}/h}})\boldsymbol{V}_i \\
&\boldsymbol{H}_i = \mathrm{ATTN}(\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i)\boldsymbol{W}_i^o \\
&\mathrm{MHA}(\boldsymbol{H}) = \sum_{i=1}^{h} \boldsymbol{H}_i,
\end{aligned}
\tag{1}
$$

where $\boldsymbol{Q}_i \in \mathbb{R}^{l \times d^q/h}$, $\boldsymbol{K}_i \in \mathbb{R}^{l \times d^k/h}$, $\boldsymbol{V}_i \in \mathbb{R}^{l \times d^v/h}$ are obtained by the linear transformations of $\boldsymbol{W}_i^q$, $\boldsymbol{W}_i^k$, $\boldsymbol{W}_i^v$ respectively. $\mathrm{ATTN}(\cdot)$ is the

[4] We abbreviate the phrase *architecture hyper-parameter* as *hyper-parameter* in the paper.

[5] We omitted the batch size for simplicity.

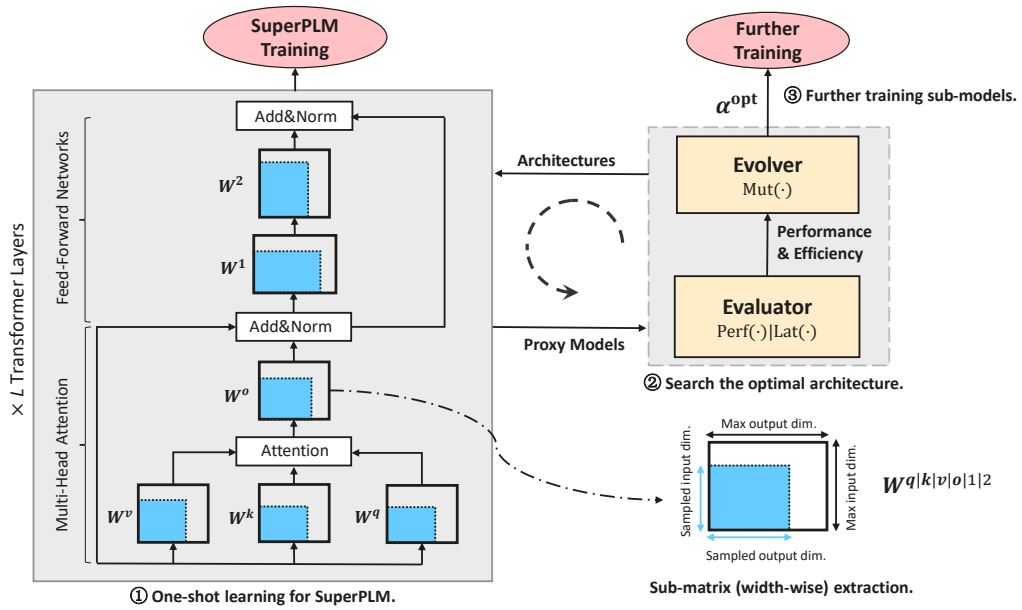


Figure 2: Overview of AutoTinyBERT. We first train an effective SuperPLM with one-shot learning, where the objectives of pre-training or task-agnostic BERT distillation are used. Then, given a specific latency constraint, we perform an evolutionary algorithm on the SuperPLM to search optimal architectures. Finally, we extract the corresponding sub-models based on the optimal architectures and further train these models.

scaled dot-product attention operation. Then output of each head is transformed to $\boldsymbol{H}_i \in \mathbb{R}^{l \times d^o}$ by $\boldsymbol{W}_i^o$. Finally, outputs of all heads are summed as the output of MHA. In addition, residual connection and layer normalization are added on top of MHA to get the final output:

$$\boldsymbol{H}^{\text{MHA}} = \text{LayerNorm}(\boldsymbol{H} + \text{MHA}(\boldsymbol{H})). \quad (2)$$

In the conventional setting of the hyper-parameters in BERT, all dimensions of matrices are the same as the dimension of the hidden vector, namely, $d^{q|k|v|o}$=$d^m$. In fact, there are only two requirements of $d^q$=$d^k$ and $d^o$=$d^m$ that must be satisfied because of the dot-product attention operation in MHA and the residual connection.

Transformer layer also contains an FFN that is stacked on the MHA, that is:

$$\boldsymbol{H}^{\text{FFN}} = \max(0, \boldsymbol{H}^{\text{MHA}}\boldsymbol{W}^1 + b_1)\boldsymbol{W}^2 + b_2, \quad (3)$$

where $\boldsymbol{W}^1 \in \mathbb{R}^{d^m \times d^f}$, $\boldsymbol{W}^2 \in \mathbb{R}^{d^f \times d^m}$, $b_1 \in \mathbb{R}^{d^f}$ and $b_2 \in \mathbb{R}^{d^m}$. Similarly, there are modules of residual connection and layer normalization on top of FFN. In the original Transformer, $d^f$=$4d^m$ is assumed. Thus, we conclude that the conventional hyper-parameter setting follows the rule of $\{d^{q|k|v|o}$=$d^m$, $d^f$=$4d^m\}$.

## 3 Methodology

### 3.1 Problem Statement

Given a constraint of inference time, our goal is to find an optimal configuration of architecture hyper-parameters $\alpha^{\text{opt}}$ built with which PLM can achieve the best performances on downstream tasks. This optimization problem is formulated as:

$$\begin{aligned} \alpha^{\text{opt}} &= \arg\max_{\alpha \in \mathcal{A}} \text{Perf}(\alpha, \theta_\alpha^*), \\ \text{s.t.} \quad \theta_\alpha^* &= \arg\min_{\theta} L_\alpha(\theta), \ \text{Lat}(\alpha) \leq T, \end{aligned} \quad (4)$$

where $T$ is a specific time constraint, $\mathcal{A}$ refers to the set of all possible architectures (i.e., combination of hyper-parameters), $\text{Lat}(\cdot)$ is a latency evaluator, $L_\alpha(\cdot)$ denotes the loss function of PLMs with the hyper-parameter $\alpha$, and $\theta$ is the corresponding model parameters. We aim to search an optimal architecture for efficient PLM ($\text{Lat}(\alpha) < 1/4 \times \text{Lat}(\text{BERT}_{\text{base}})$).

### 3.2 Overview

A straightforward way to get the optimal architecture is to enumerate all possible architectures. However, it is infeasible because each trial involves a time-consuming pre-training process. Therefore, we introduce one-shot NAS to search $\alpha^{\text{opt}}$, as shown in the Figure 2. The proposed method includes three stages: (1) the one-shot learning to obtain SuperPLM that can be used as the proxy for

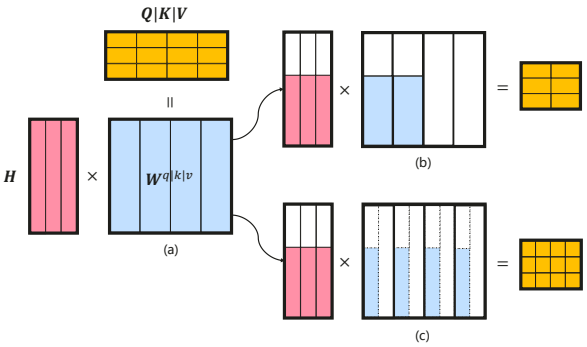


Figure 3: MHA sub-matrix extraction. (a) means that the original matrix operation where we take four heads and three hidden vectors as an example. White boxes refer to the un-extracted parameters. (b) means that we extract heads while keeping the dimension per head. (c) means that we extract parameters from each head while keeping the head number as the original matrix.

various architectures; (2) the search process for the optimal hyper-parameters; (3) the further training with the optimal architectures and corresponding sub-models. In the following sections, we first introduce the search space, which is the basis for the one-shot learning and search process. Then we present the three stages respectively.

### 3.3 Search Space

From the Section 2, we know that the conventional hyper-parameter setting is: $\{d^{q|k|v|o}{=}d^{m}, d^{f}{=}4d^{m}\}$, which is widely-used in PLMs. The architecture of a PLM is parameterized as: $\alpha = \{l^t, d^m, d^q, d^k, d^v, d^f, d^o\}$, which is subjected to the constraints $\{d^q = d^k, d^o = d^m\}$. Let $l^t$ denote the layer number and $d^*$ refer to different dimensions in the Transformer layer. We denote the search space of $l^t$ and $d^*$ as $\mathcal{A}_{l^t}$ and $\mathcal{A}_{d^*}$ respectively. The overall search space is: $\mathcal{A} = \mathcal{A}_{l^t} \times \mathcal{A}_{d^{m|o}} \times \mathcal{A}_{d^{q|k}} \times \mathcal{A}_{d^v} \times \mathcal{A}_{d^f}$.

In this work, we only consider the case of identical structure for each Transformer layer, instead of the non-identical Transformer (Wang et al., 2020a) or other heterogeneous modules (Xu et al., 2021) (such as convolution units). It has two advantages: (1) it reduces an exponential search space of $\mathcal{O}(\prod_{*} |\mathcal{A}_{d^*}|^{|\mathcal{A}_{l^t}|})$ to a linear search space of $\mathcal{O}(\sum_{*} |\mathcal{A}_{d^*}||\mathcal{A}_{l^t}|)$, greatly reducing the number of possible architectures in SuperPLM training and the exploration space in the search process. It leads to a more efficient search process. (2) An identical and homogeneous structure is in fact more friendly to hardware and software frameworks, e.g., Hugging Face Transformer (Wolf et al., 2020). With a few changes, we can use the original code to use AutoTinyBERT, as shown in Appendix A.

**Algorithm 1** Batch-wise training for SuperPLM

**Input:** All possible candidates $\mathcal{A}$; Training thread (GPU) number $N$; Large-scale unsupervised dataset $D$; Training epochs $E$. Sample times $M$ per batch. SuperPLM parameters ($\theta$).
**Output:** Trained SuperPLM ($\theta$)

```
for t = 1 → E do
    for batch in D do
        Divide batch into N sub_batches
        Distribute sub_batches to N threads
        Clear the gradients
        for m = 1 → M do
            Sample N sub-models from A
            Distribute sub-models to threads
            Calculate gradients in each thread
        end for
        Update the θ with the average gradients
    end for
end for
```

### 3.4 One-shot Learning for SuperPLM

We employ the one-shot learning (Brock et al., 2018; Yu et al., 2020) to obtain a SuperPLM whose sub-models can act as the proxy for PLMs trained from scratch. The configurations of SuperPLM in this work are $l^t$=8, $d^{m|q|k|v|o}$=768, and $d^f$=3072. In each step of the one-shot learning, we train several sub-models randomly sampled from SuperPLM to make their performance close to the models trained from scratch. Although the sampling/search space has been reduced to linear complexity, there are still more than 10M possible substructures in SuperPLM (the details are shown in the Appendix B). Therefore, we introduce an effective batch-wise training method to cover the sub-models as much as possible. Specifically, in parallel training, we first divide each batch into multiple sub-batches and distribute them to different threads as parallel training data. Then, we sample several sub-models on each thread for training and merge the gradients of all threads to update the SuperPLM parameters. We illustrate the training process in the Algorithm 1.

Given a specific hyper-parameter setting $\alpha = \{l^t, d^m, d^q, d^k, d^v, d^f, d^o\}$, we get a sub-model from SuperPLM by the depth-wise and width-wise extraction. Specifically, we first perform the depth-wise extraction that extracts the first $l^t$ Trans-

| Model | Speedup | SQuAD | SST-2 | MNLI | MRPC | CoLA | QNLI | QQP | STS-B | RTE | Score | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **AutoTinyBERT-S1** | 7.2× | 83.3 | 89.4 | 79.4 | 85.5 | 42.4 | 87.3 | 88.8 | 87.5 | 66.3 | **78.3** | **78.9** |
| BERT-S1 | 7.1× | 81.5 | 88.9 | 78.4 | 81.3 | 35.8 | 86.4 | 88.2 | 86.7 | 66.4 | 76.5 | 77.1 |
| PF-4L512D‡ (Turc et al., 2019) | 7.1× | 81.7 | 89.4 | 78.5 | 82.8 | 35.2 | 87.0 | 88.6 | 87.4 | 65.7 | 76.8 | 77.4 |
| PF-2L768D‡ (Turc et al., 2019) | 7.0× | 71.1 | 88.8 | 76.5 | 79.6 | 26.7 | 84.9 | 88.1 | 86.6 | 67.1 | 74.8 | 74.4 |
| **AutoTinyBERT-S2** | 15.7× | 78.1 | 88.2 | 76.8 | 82.8 | 35.5 | 85.4 | 87.8 | 86.5 | 68.2 | **76.4** | **76.6** |
| BERT-S2 | 14.8× | 77.6 | 87.5 | 76.5 | 79.6 | 32.8 | 84.4 | 87.0 | 86.6 | 66.4 | 75.1 | 75.4 |
| NAS-BERT$_{10}$† (Xu et al., 2021) | 12.7× | - | 88.6 | 76.0 | 81.5 | 27.8 | 86.3 | 88.4 | 84.3 | 68.7 | 75.2 | - |
| PF-2L512D‡ (Turc et al., 2019) | 12.8× | 69.2 | 87.1 | 74.7 | 76.9 | 23.2 | 84.4 | 87.0 | 86.0 | 64.9 | 73.0 | 72.6 |
| PF-6L256D‡ (Turc et al., 2019) | 13.3× | 77.0 | 87.6 | 76.4 | 80.3 | 33.2 | 85.7 | 86.7 | 86.0 | 64.9 | 75.1 | 75.3 |
| **AutoTinyBERT-S3** | 20.2× | 75.8 | 86.8 | 76.4 | 80.4 | 33.2 | 85.0 | 87.6 | 86.7 | 66.4 | **75.3** | **75.4** |
| BERT-S3 | 20.1× | 73.7 | 86.4 | 75.0 | 81.3 | 31.2 | 84.0 | 87.1 | 85.8 | 63.8 | 74.3 | 74.3 |
| **AutoTinyBERT-S4** | 31.0× | 71.9 | 86.5 | 74.2 | 81.9 | 17.6 | 84.6 | 86.5 | 85.9 | 66.7 | **73.0** | **72.9** |
| BERT-S4 | 31.3× | 69.5 | 85.5 | 73.9 | 76.9 | 15.9 | 83.9 | 85.9 | 85.3 | 61.0 | 71.0 | 70.9 |
| NAS-BERT$_5$† (Xu et al., 2021) | 32.0× | - | 84.9 | 74.2 | 80.0 | 19.6 | 83.9 | 85.7 | 82.8 | 67.0 | 72.3 | - |
| PF-6L128D‡ (Turc et al., 2019) | 28.2× | 63.6 | 84.6 | 72.3 | 78.6 | 0 | 83.3 | 83.8 | 84.5 | 65.7 | 69.1 | 68.5 |

Table 1: Comparison between AutoTinyBERT and baselines in pre-training setting. The results are evaluated on the dev set of GLUE benchmark and SQuADv1.1. We use the metric of Matthews correlation for CoLA, F1 for SQuADv1.1, Pearson-Spearman correlation for STS-B, and accuracy for other tasks. We report the average score excluding SQuAD (Score) in addition to the average score of all tasks (Avg.). The speedup is in terms of the $\text{BERT}_{\text{base}}$ inference speed and evaluated on a single CPU with a single input of 128 length. PF-xLyD, the x and y refer to the layer number and hidden dimension respectively. †denotes that the results are taken from (Xu et al., 2021) and ‡denotes that the results are obtained by fine-tuning the released models.

former layers from SuperPLM, and then perform the width-wise extraction that extracts bottom-left sub-matrices from original matrices. For MHA, we apply two strategies illustrated in Figure 3 : (1) keep the dimension of each head same as SuperPLM, and extract some of the heads; (2) keep the head number same as SuperPLM, and extract sub-dimensions from each head. The first strategy is the standard one and we use it for pre-training and the second strategy is used for task-agnostic distillation because that attention-based distillation (Jiao et al., 2020b) requires the student model to have the same head number as the teacher model.

### 3.5 Search Process

In the search process, we adopt an evolutionary algorithm (Xie and Yuille, 2017; Jiao et al., 2020a), where Evolver and Evaluator interact with each other to evolve better architectures. Our search process is efficient, as shown in the Section 4.4.

Specifically, Evolver firstly samples a generation of architectures from $\mathcal{A}$. Then Evaluator extracts the corresponding sub-models from SuperPLM and ranks them based on their performance on tasks of SQuAD and MNLI. The architectures with the high performance are chosen as the winning architectures and Evolver performs the mutation $\text{Mut}(\cdot)$ operation on the winning ones to produce a new generation of architectures. This process is conducted repeatedly. Finally, we choose several architectures with the best performance for further training. We use $\text{Lat}(\cdot)$ to predict the latency of the candidates to filter out the candidates that do not meet the latency constraint. $\text{Lat}(\cdot)$ is built with the method by Wang et al. (2020a), which first samples about 10k architectures from $\mathcal{A}$ and collects their inference time on target devices, and then uses a feed-forward network to fit the data. For more details of evolutionary algorithm, please refer to Appendix C. Note that we can use different methods in search process, such as random search and more advanced search, which is left as future work.

### 3.6 Further Training

The search process produces top several architectures, with which we extract these corresponding sub-models from SuperPLM and continue training them using the pre-training or KD objectives.

## 4 Experiment

### 4.1 Experimental Setup

**Dataset and Fine tuning.** We conduct the experiments on the GLUE benchmark (Wang et al., 2018) and SQuADv1.1 (Rajpurkar et al., 2016). For GLUE, we set the batch size to 32, choose the learning rate from {1e-5, 2e-5, 3e-5} and choose the epoch number from {4, 5, 10}. For SQuADv1.1, we set the batch size to 16, the learning rate to 3e-5 and the epoch number to 4. The details for all datasets are displayed in Appendix D.

**AutoTinyBERT.** Both the one-shot and further

| Model | Speedup | SQuAD | SST-2 | MNLI | MRPC¶ | CoLA | QNLI | QQP¶ | STS-B | RTE | Score | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Dev results on GLUE and dev result on SQuAD* | | | | | | | | | | | | |
| **AutoTinyBERT-KD-S1** | 4.6× | 87.6 | 91.4 | 82.3 | 88.5 | 47.3 | 89.7 | 89.9 | 89.0 | 71.1 | **81.2** | **81.9** |
| BERT-KD-S1 | 4.9× | 86.2 | 89.7 | 81.1 | 87.9 | 41.8 | 87.3 | 88.4 | 88.4 | 68.2 | 79.1 | 79.9 |
| MobileBERT$_{\text{Tiny}}$‡(Sun et al., 2020) | 3.6*× | 88.6 | 91.6 | 82.0 | 86.7 | - | - | - | - | - | - | - |
| **AutoTinyBERT-KD-S2** | 9.0× | 84.6 | 88.8 | 79.4 | 87.3 | 32.2 | 88.0 | 87.7 | 88.0 | 68.9 | **77.5** | **78.3** |
| BERT-KD-S2 | 9.8× | 82.5 | 87.8 | 77.9 | 86.5 | 31.5 | 86.9 | 87.6 | 87.4 | 66.4 | 76.5 | 77.2 |
| MiniLM-4L312D† (Wang et al., 2020b) | 9.8× | 82.1 | 87.3 | 78.3 | 83.6 | 26.3 | 87.1 | 87.3 | 86.3 | 62.4 | 74.8 | 75.6 |
| TinyBERT-4L312D†§ (Jiao et al., 2020b) | 9.8× | 81.0 | 87.8 | 76.9 | 77.9 | 22.9 | 86.0 | 87.7 | 83.3 | 58.8 | 72.7 | 73.6 |
| **AutoTinyBERT-KD-S3** | 10.7× | 83.3 | 88.3 | 78.2 | 85.8 | 29.1 | 87.4 | 87.4 | 86.7 | 66.4 | **76.2** | **77.0** |
| BERT-KD-S3 | 11.7× | 81.6 | 86.5 | 76.8 | 82.5 | 27.6 | 85.6 | 86.5 | 86.2 | 64.9 | 74.6 | 75.4 |
| **AutoTinyBERT-KD-S4** | 17.0× | 78.7 | 86.8 | 76.0 | 81.4 | 20.4 | 85.5 | 86.9 | 86.0 | 64.9 | **73.5** | **74.1** |
| BERT-KD-S4 | 17.0× | 77.4 | 85.7 | 75.4 | 80.3 | 18.9 | 85.0 | 85.9 | 84.7 | 63.1 | 72.4 | 72.9 |
| *Test results on GLUE and dev result on SQuAD* | | | | | | | | | | | | |
| **AutoTinyBERT-KD-S1** | 4.6× | 87.6 | 90.6 | 81.2 | 88.9 | 44.7 | 87.4 | 70.5 | 85.1 | 64.8 | **76.7** | **77.9** |
| BERT-3L-PKD‡ (Sun et al., 2019) | 4.1× | - | 87.5 | 76.7 | 80.7 | - | 84.7 | 68.1 | - | 58.2 | - | - |
| DistilBERT-4L‡ (Sanh et al., 2019) | 3.0× | 81.2 | 91.4 | 78.9 | 82.4 | 32.8 | 85.2 | 68.5 | 76.1 | 54.1 | 71.2 | 72.3 |
| TinyBERT-4L516D†§ (Jiao et al., 2020b) | 4.9× | 84.6 | 88.2 | 80.0 | 86.3 | 27.9 | 85.6 | 69.1 | 83.0 | 61.5 | 72.7 | 74.0 |
| MiniLM-4L516D† (Wang et al., 2020b) | 4.9× | 85.5 | 90.0 | 80.2 | 87.2 | 39.1 | 86.5 | 70.0 | 83.4 | 63.7 | 75.0 | 76.2 |
| MobileBERT$_{\text{Tiny}}$‡ (Sun et al., 2020) | 3.6*× | 88.6 | 91.7 | 81.5 | 87.9 | 46.7 | 89.5 | 68.9 | 80.1 | 65.1 | 76.4 | 77.8 |

Table 2: Comparison between AutoTinyBERT and baselines based on knowledge distillation. ‡ denotes that the results are taken from (Sun et al., 2020) and † means the models trained using the released code or the re-implemented code with ELECTRA$_{\text{base}}$ as the teacher model. ¶ means these tasks use accuracy for dev set and F1 for test set respectively. § denotes the task-agnostic TinyBERT without task-specific distillation. * means that the speedup is different from the (Sun et al., 2020), because it is evaluated on a Pixel phone and not on server CPUs. - means that the results are missing in the original paper. Other information refer to the Table 1.

training use BooksCorpus (Zhu et al., 2015) and English Wikipedia as training data. The settings for one-shot training are: peak learning rate of 1e-5, warmup rate of 0.1, batch size of 256 and 5 running epochs. Further training follows the same setting as the one-shot training except for the warmup rate of 0. In the batch-wise training algorithm 1, the thread number $N$ is set to 16, the sample times $M$ per batch is set to 3, and epoch number $E$ is set to 5. We train the SuperPLM with an architecture of $\{l^t$=8, $d^{m|q|k|v|o}$=768, $d^f$=3072$\}$. In the search process, Evolver performs 4 iterations with a population size of 25 and it chooses top three architectures for further training. For more details of the sampling/search space and evolutionary algorithm, please refer to Appendix B and C.

We train AutoTinyBERT in both ways of pre-training (Devlin et al., 2019) and task-agnostic BERT distillation (Sun et al., 2020). For task-agnostic distillation, we follow the first stage of TinyBERT (Jiao et al., 2020b) except that only the last-layer loss is used, and ELECTRA$_{\text{base}}$ (Clark et al., 2019) is used as the teacher model.

**Baselines.** For the pre-training baselines, we include PF (*Pre-training + Fine-tuning*, proposed by Turc et al. (2019)), BERT-S* (BERT under several hyper-parameter configurations), and NAS-BERT (Xu et al., 2021). Both PF and BERT-S* follow the conventional setting rule of hyper-parameters. BERT-S* uses the training setting: peak learning rate of 1e-5, warmup rate of 0.1, batch size of 256 and 10 running epochs. NAS-BERT searches the architecture built on the non-identical layer and heterogeneous modules. For the distillation baselines, we compare some typical methods, including DistilBERT, BERT-PKD, TinyBERT, MiniLM, and MobileBERT. The first four methods use the conventional architectures. MobileBERT is equipped with a bottleneck structure and a carefully designed balance between MHA and FFN. We also consider BERT-KD-S*, which use the same training setting of BERT-S*, except for the loss of knowledge distillation. BERT-KD-S* also uses ELECTRA$_{\text{base}}$ as the teacher model.

### 4.2 Results and Analysis

The experiment is conducted under different latency constraints that are from 4× to 30× faster than the inference of BERT$_{\text{base}}$. The results of pre-training and task-agnostic distillation are shown in the Table 1 and Table 2 respectively.

We observe that in the settings of the pre-training and knowledge distillation, the performance gap of different models with similar inference time is obvious, which shows the necessity of architecture optimization for efficient PLMs. In the Table 1, some observations are: (1) the architecture optimization methods of AutoTinyBERT and NAS-BERT outperform both BERT and PF that use the default

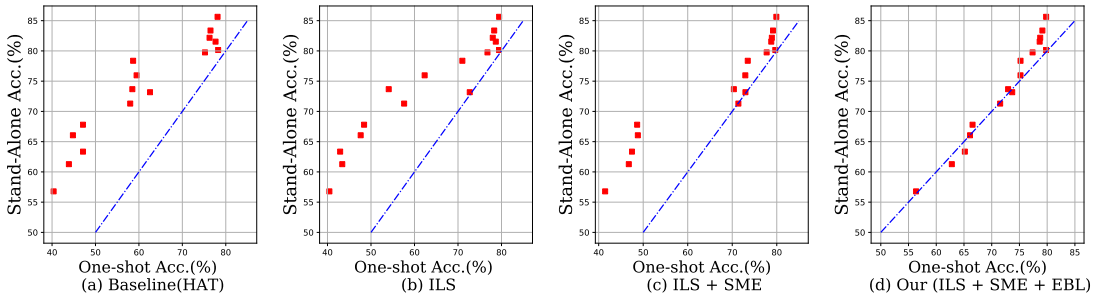


Figure 4: Ablation study of one-shot SuperPLM learning. Acc. means the average score on SQuAD and MNLI. The dashed line represents the function of y=x.

| Training Method | SQuAD F1(%) | MNLI Acc.(%) | Pairwise Acc.(%) |
|---|---|---|---|
| Stand-alone | 71.2 | 76.3 | 100 |
| Baseline (HAT) | 50.1 | 72.7 | 90.0 |
| ILS | 52.9 | 73.4 | 91.6 |
| ILS + SME | 59.5 | 74.1 | 94.2 |
| ILS + SME + EBL (Ours) | **70.5** | **74.4** | **96.7** |

Table 3: Ablation Study of SuperPLM. ILS, SME and EBL mean that the identical layer structure, MHA sub-matrix extraction and effective batch-wise training.

| Version | BERT | AutoTinyBERT | Speedup |
|---|---|---|---|
| *Pre-training* | | | |
| S1 | 4-512-2048-8-512 | 5-564-1054-8-512 | 7.1/7.2× |
| S2 | 4-320-1280-5-320 | 4-396-624-6-384 | 14.8/15.7× |
| S3 | 4-256-1024-4-256 | 4-432-384-4-256 | 20.1/20.2× |
| S4 | 4-192-768-3-192 | 3-320-608-4-256 | 28.4/27.2× |
| *Task-agnostic BERT Distillation* | | | |
| KD-S1 | 4-512-2048-12-516 | 5-564-1024-12-528 | 4.9/4.6× |
| KD-S2† | 4-312-1200-12-312 | 5-324-600-12-324 | 9.8/9.0× |
| KD-S3 | 4-264-1056-12-264 | 5-280-512-12-276 | 11.7/10.7× |
| KD-S4 | 4-192-768-12-192 | 4-256-480-12-192 | 17.0/17.0× |

Table 4: BERT and AutoTinyBERT architectures under the different speedup constraints. The architecture is formatted as "$l^t$-$d^{m|o}$-$d^f$-$h$-$d^{q|k|v}$". We assume that $d^{q|k} = d^v$ in the experiment for the training and search efficiency. † means that we use the structure of TinyBERT and do not strictly follow the conventional rule.

architecture hyper-parameters; (2) our method outperforms NAS-BERT that is built with the non-identical layer and heterogeneous modules, which shows that the proposed method is effective for the architecture search of efficient PLMs. In the Table 2, we observe that: (1) our method consistently outperforms the conventional structure in all the speedup constraints; (2) our method outperforms the classical distillation methods (e.g., BERT-PKD, DistilBERT, TinyBERT, and MiniLM) that use the conventional architecture. Moreover, AutoTinyBERT achieves comparable results with MobileBERT, and its inference speed is 1.5× faster.

### 4.3 Ablation Study of One-shot Learning

We demonstrate the effectiveness of one-shot learning by comparing the performance of one-shot model and stand-alone trained model on the given architectures. We choose 16 architectures and their corresponding PF models[6] as the evaluation benchmark. The pairwise accuracy is used as a metric to indicate the ranking correction between the architectures under one-shot training and the ones under stand-alone full training (Luo et al., 2019) and its formula is described in Appendix E.

We do the ablation study to analyze the effect of proposed identical layer structure (ILS), MHA sub-matrix extraction (SME) and effective batch-wise learning (EBL) on SuperPLM learning. Moreover, we introduce HAT (Wang et al., 2020a), as a baseline of one-shot learning. HAT focuses on the search space of non-identical layer structures. The results are displayed in Table 3 and Figure 4.

It can be seen from the figure that compared with stand-alone trained models, the HAT baseline has a significant performance gap, especially in small sizes. Both ILS and SME benefit the one-shot learning for large and medium-sized models. When further combined with EBL, SuperPLM can obtain similar or even better results than stand-alone trained models of small sizes and perform close to stand-alone trained models of big sizes. The results of the table show that: (1) the proposed techniques have positive effects on SuperPLM learning, and EBL brings a significant improvement on a challenging task of SQuAD; (2) SuperPLM achieves a high pairwise accuracy of 96.7% which indicates that the proposed SuperPLM can be a good proxy model for the search process; (3) the performance of SuperPLM is still a little worse than the stand-alone trained model and we need to do the further training to boost the performance.

[6]The first 16 models https://github.com/google-research/bert from 2L128D to 8L768D.

| Method | Speedup | Search Cost (GPU Hours) | Training Cost (GPU Hours) | Avg. |
|---|---|---|---|---|
| BERT-S5 | 9.9× | **0** | 580 | 76.1 |
| **AutoTinyBERT-S5** | **10.8**× | 150 | 870 | **77.9** |
| **AutoTinyBERT-Fast-S5** | 10.3× | 12 | **290** | 77.6 |

Table 5: Computation cost of different methods. AutoTinyBERT and AutoTinyBERT-Fast have 100 and 8 architectures (×1.5 V100 GPU hour) respectively to be tested in the search process. AutoTinyBERT performs further 5 epochs (×58 V100 GPU hours) training for top three architectures, BERT is trained from scratch with 10 epochs, and AutoTinyBERT-Fast does the further training for one architecture. We give more information including the model architectures and detailed scores of all tasks in the Appendix F.

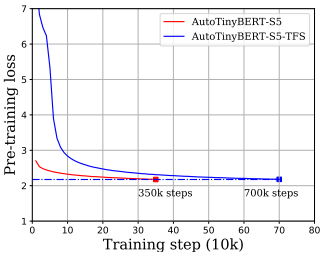


Figure 5: Learning curves of AutoTinyBERT and the stand-alone trained model. TFS means the model trained from scratch. AutoTinyBERT can save 50% training time compared with the model trained from scratch.

### 4.4 Fast Development of Efficient PLM

In this section, we explore an effective setting rule of hyper-parameters based on the obtained architectures and also discuss the computation cost of the development of efficient PLM. The conventional and new architectures are displayed in Table 4. We observe that AutoTinyBERT follows an obvious rule (except the S3 model) in the speedup constraints that are from 4× to 30×. The rule is summarized as: $\{1.6d^m \leq d^f \leq 1.9d^m,\ \ 0.7d^m \leq d^{q|k|v} \leq 1.0d^m\}$.

With the above rule, we propose a faster way to build efficient PLM, denoted as AutoTinyBERT-Fast. Specifically, we first obtain the candidates by the rule, and then select $\alpha^{\text{opt}}$ from the candidates. We observe the fact that the candidates of the same layer number seem to have similar shapes and we assume that they have similar performance. Therefore, we only need to test one architecture at each layer number and choose the best one as $\alpha^{\text{opt}}$.

To demonstrate the effectiveness of the proposed method, we evaluate these methods at a new speedup constraint of about 10× under the pre-training setting. The results are shown in Table 5. We find AutoTinyBERT is efficient and its development time is twice that of the conventional method (BERT) and the result is improved by about 1.8%. AutoTinyBERT-Fast achieves a competitive score of 77.6 by only about 50% of BERT training time. In addition to the proposed search method and fast building rule, one reason for the high efficiency of AutoTinyBERT is that the initialization of SuperPLM helps the model to achieve 2× the convergence speedup, as illustrated in Figure 5.

## 5 Related Work

**Efficient PLMs with Tiny sizes.** There are two widely-used methods for building efficient PLMs: pre-training and model compression. Knowledge distillation (KD) (Hinton et al., 2015; Romero et al., 2014) is the most widely studied technique in PLM compression, which uses a teacher-student framework. The typical distillation studies include DistilBERT (Sanh et al., 2019), BERT-PKD (Sun et al., 2019), MiniLM (Wang et al., 2020b), MobileBERT (Sun et al., 2020), MiniBERT (Tsai et al., 2019) and ETD (Chen et al., 2021). In addition to KD, the techniques of pruning (Han et al., 2016; Hou et al., 2020), quantization (Shen et al., 2020; Zhang et al., 2020; Wang et al., 2020c) and parameter sharing (Lan et al., 2019) introduced for PLM compression. Our method is orthogonal to the building method of efficient PLM and is trained under the settings of pre-training and task-agnostic BERT distillation, which can be used by direct fine-tuning.

**NAS for NLP.** NAS is extensively studied in computer vision (Tan and Le, 2019; Tan et al., 2020), but relatively little studied in the natural language processing. Evolved Transformer (So et al., 2019) and HAT (Wang et al., 2020a) search architecture for Transformer-based neural machine translation. For BERT distillation, AdaBERT (Chen et al., 2020) focuses on searching the architecture in the fine-tuning stage and relies on data augmentation to improve its performance. schuBERT (Khetan and Karnin, 2020) obtains the optimal structures of PLM by a pruning method. A work similar to ours is NAS-BERT (Xu et al., 2021). It proposes some techniques to tackle the challenging exponential search space of non-identical layer structure and heterogeneous modules. Our method adopts a linear search space and introduces several practical techniques for SuperPLM training. Moreover, our method is efficient in terms of computation cost and the obtained PLMs are easy to use.

## 6 Conclusion

We propose an effective and efficient method AutoTinyBERT to search for the optimal architecture hyper-parameters of efficient PLMs. We evaluate the proposed method in the scenarios of both the pre-training and task-agnostic BERT distillation. The extensive experiments show that AutoTinyBERT can consistently outperform the baselines under different latency constraints. Furthermore, we develop a fast development rule for efficient PLMs which can build an AutoTinyBERT model even with less training time of a conventional one.

## Acknowledgments

We thank all the anonymous reviewers for their valuable comments. We thank MindSpore[7] for the partial support of this work, which is a new deep learning computing framework.

## References

Andrew Brock, Theo Lim, JM Ritchie, and Nick Weston. 2018. Smash: One-shot model architecture search through hypernetworks. In *ICLR*.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *NeurIPS*.

Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. In *ICLR*.

Cheng Chen, Yichun Yin, Lifeng Shang, Zhi Wang, Xin Jiang, Xiao Chen, and Qun Liu. 2021. Extract then distill: Efficient and effective task-agnostic bert distillation.

Daoyuan Chen, Yaliang Li, Minghui Qiu, Zhen Wang, Bofang Li, Bolin Ding, Hongbo Deng, Jun Huang, Wei Lin, and Jingren Zhou. 2020. Adabert: Task-adaptive bert compression with differentiable neural architecture search. In *IJCAI*.

Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2019. Electra: Pre-training text encoders as discriminators rather than generators. In *ICLR*.

J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*.

[7]MindSpore. https://www.mindspore.cn/

Kai Han, Yunhe Wang, Qiulin Zhang, Wei Zhang, Chunjing Xu, and Tong Zhang. 2020. Model rubik's cube: Twisting resolution, depth and width for tinynets. *NeurIPS*.

Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *ICLR*.

Geoffrey E. Hinton, Oriol Vinyals, and J. Dean. 2015. Distilling the knowledge in a neural network. *ArXiv*, abs/1503.02531.

Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. 2020. Dynabert: Dynamic bert with adaptive width and depth. *NeurIPS*, 33.

Xiaoqi Jiao, Huating Chang, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020a. Improving task-agnostic bert distillation with layer mapping search. *arXiv preprint arXiv:2012.06153*.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020b. Tinybert: Distilling bert for natural language understanding. In *EMNLP: Findings*.

Ashish Khetan and Zohar Karnin. 2020. schubert: Optimizing elements of bert. In *ACL*.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. In *ICLR*.

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

Renqian Luo, Tao Qin, and Enhong Chen. 2019. Balanced one-shot neural architecture optimization. *arXiv preprint arXiv:1909.10815*.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. In *EMNLP*.

Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2014. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*.

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

Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Q-bert: Hessian based ultra low precision quantization of bert. In *AAAI*.

David So, Quoc Le, and Chen Liang. 2019. The evolved transformer. In *ICML*.

Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. 2019. Patient knowledge distillation for bert model compression. In *EMNLP-IJCNLP*.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. Mobilebert: a compact task-agnostic bert for resource-limited devices. In *ACL*.

Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*.

Mingxing Tan, Ruoming Pang, and Quoc V Le. 2020. Efficientdet: Scalable and efficient object detection. In *CVPR*.

Henry Tsai, Jason Riesa, Melvin Johnson, Naveen Arivazhagan, Xin Li, and Amelia Archer. 2019. Small and practical bert models for sequence labeling. In *EMNLP-IJCNLP*.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*.

Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. 2020a. Hat: Hardware-aware transformers for efficient natural language processing. In *ACL*.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020b. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers.

Ziheng Wang, Jeremy Wohlwend, and Tao Lei. 2020c. Structured pruning of large language models. In *EMNLP*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *EMNLP: System Demonstrations*.

Lingxi Xie and Alan Yuille. 2017. Genetic cnn. In *ICCV*.

Jin Xu, Xu Tan, Renqian Luo, Kaitao Song, Li Jian, Tao Qin, and Tie-Yan Liu. 2021. Task-agnostic and adaptive-size bert compression. In *openreview*.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. In *NeurIPS*.

Jiahui Yu, Pengchong Jin, Hanxiao Liu, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Thomas Huang, Xiaodan Song, Ruoming Pang, and Quoc Le. 2020. Bignas: Scaling up neural architecture search with big single-stage models. In *ECCV*.

Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. 2020. Ternarybert: Distillation-aware ultra-low bit bert. In *EMNLP*.

Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *ICCV*.

## A Code Modifications for AutoTinyBERT.

We modify the original code[8] to load AutoTinyBERT model and present the details of code modifications in the Figure B.1. We assume that $d^{q/k} = d^v$, and more complicated setting is that $d^v$ can be different with $d^{q/k}$, we can do corresponding changes based on the given modifications.

## B Search Space of Architecture Hyper-parameters.

We has trained two SuperPLMs with a architecture of $\{l^t$=8, $d^{m/q/k/v}$=768, $d^f$=3072$\}$ to cover the two scenarios of building efficient PLMs (pre-training and task-agnostic BERT distillation). The sampling space in the SuperPLM training is the same as the search space in the search process, as shown in the Table B.1. It can be inferred from the table that the search spaces of the pre-training setting and the knowledge distillation setting are about 46M and 10M, respectively.

| Variables | Search Space |
|---|---|
| *SuperPLM in Pre-training* | |
| $l^t$ | [1,2,3,4,5,6,7,8] |
| $d^{m/o}$ | [128,132,...,4$k$,...,764,768] |
| $d^f$ | [128,132,...,4$k$,...,3068,3072] |
| $h$ | [1,2,...,$k$,...,11,12] |
| $d^{q/k/v}$ | 64$h$ |
| *SuperPLM in Knowledge Distillation* | |
| $l^t$ | [1,2,3,4,5,6,7,8] |
| $d^{m/o}$ | [128,132,...,4$k$,...,764,768] |
| $d^f$ | [128,132,...,4$k$,...,3068,3072] |
| $h$ | [12] |
| $d^{q/k/v}$ | [180,192,...,12$k$,...,756,768] |

Table B.1: The search space for architecture hyper-parameters. We assume that $d^{q|k} = d^v$ in the experiment for the training and search efficiency.

## C Evolutionary Algorithm.

We give a detailed description of evolutionary algorithm in Algorithm 2.

## D Hyper-parameters for Fine-Tuning.

Fine-tuning hyper-parameters of GLUE benchmark and SQuAD are displayed in Table D.1. AutoTinyBERT and baselines follow the same settings.

| Tasks | Batch size | Learning rate | Epochs |
|---|---|---|---|
| SQuAD | 16 | 3e-5 | 4 |
| SST-2 | 32 | 2e-5 | 4 |
| MNLI | 32 | 3e-5 | 4 |
| MRPC | 32 | 2e-5 | 10 |
| CoLA | 32 | 1e-5 | 10 |
| QNLI | 32 | 2e-5 | 10 |
| QQP | 32 | 2e-5 | 5 |
| STS-B | 32 | 3e-5 | 10 |
| RTE | 32 | 2e-5 | 10 |

Table D.1: Hyper-parameters used for fine-tuning on GLUE benchmark and SQuAD.

## E Pairwise Accuracy.

We denote a set of architectures $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ as $\mathcal{A}_{eva}$ and evaluate SuperPLM on this set. The pairwise accuracy is formulated as bellow:

$$\frac{\sum_{\alpha_1 \in \mathcal{A}_{eva}, \alpha_2 \in \mathcal{A}_{eva}} 1_{f(\alpha_1) \geq f(\alpha_2)} 1_{s(\alpha_1) \geq s(\alpha_2)}}{\sum_{\alpha_1 \in \mathcal{A}_{eva}, \alpha_2 \in \mathcal{A}_{eva}} 1}, \tag{5}$$

where 1 is the 0-1 indicator function, $f(*)$ and $s(*)$ refer to the performance of one-shot model and stand-alone trained model respectively.

## F More details for Fast Development of efficient PLM.

We present the detailed results and architecture hyper-parameters for fast development of efficient PLM in Table F.1.

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

```
class BertSelfAttention(nn.Module):
   def __init__(self, config):
      ### Before modifications:
      self.attention_head_size = int(config.hidden_size /
          config.num_attention_heads)
      ### After modifications:
      try:
         qkv_size = config.qkv_size
      except:
         qkv_size = config.hidden_size

      self.attention_head_size = int(qkv_size / config.num_attention_heads)

class BertSelfOutput(nn.Module):
   def __init__(self, config):
      ### Before modifications:
      self.dense = nn.Linear(config.hidden_size, config.hidden_size)

      ### After modifications:
      try:
         qkv_size = config.qkv_size
      except:
         qkv_size = config.hidden_size

      self.dense = nn.Linear(qkv_size, config.hidden_size)
```

Figure B.1: Code Modifications to load AutoTinyBERT.

**Algorithm 2** The Evolutionary Algorithm

**Input**: the number of generations $T = 4$, the number of archtectures $\alpha$s in each generation $S = 25$, the mutation $\mathrm{Mut}(*)$ probability $p_m = 1/2$, the exploration probability $p_e = 1/2$.
Sample first generation $\mathbb{G}_1$ from $\mathcal{A}$, and Evoluator produces its performance $\mathbb{V}_1$.
**for** $t = 2, 3 \cdots, T$ **do**
    $\mathbb{G}_t \leftarrow \{\}$
    **while** $|\mathbb{G}_t| < S$ **do**
        Sample one architecture: $\alpha$ with a Russian roulette process on $\mathbb{G}_{t-1}$ and $\mathbb{V}_{t-1}$.
        With probability $p_m$, do $\mathrm{Mut}(*)$ for $\alpha$.
        With probability $p_e$, sample a new architecture from $\mathcal{A}$.
        Append the newly generated architectures into $\mathbb{G}_t$.
    **end while**
    Evaluator obtains $\mathbb{V}_t$ for $\mathbb{G}_t$.
**end for**
**Output**: Output the $\alpha^{opt}$ with best performance in the above process.

| Model | Speedup | SQuAD | SST-2 | MNLI | MRPC | CoLA | QNLI | QQP | STS-B | RTE | Score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| BERT-S5$_{4-384-1536-6-384}$ | 9.3× | 78.5 | 86.1 | 76.8 | 83.1 | 35.5 | 84.6 | 87.5 | 86.9 | 65.7 | 76.0 |
| AutoTinyBERT-S5$_{5-450-636-6-384}$ | 10.8× | 79.7 | 89.1 | 78.3 | 84.6 | 39.0 | 85.9 | 88.2 | 87.4 | 68.7 | 77.8 |
| AutoTinyBERT-Fast-S5$_{5-432-720-6-384}$ | 10.3× | 80.0 | 88.2 | 77.9 | 84.6 | 37.7 | 86.1 | 88.0 | 87.3 | 68.7 | 77.6 |

Table F.1: Detailed results for fast development of efficient PLM.